MasPar MP-1
System Overview and MPPE Manuals

MasPar System Overview
MPPE Quick Reference
MPPE User Guide
MasPar Commands Reference Manual

Part Number 9300-9001-00, Revision A8
MasPar Computer Corporation
MasPar System Overview

Part Number 9300-0100, Rev. A6
November 1992

MasPar Computer Corporation
MasPar System Overview
Document Part Number 9300-0100, Revision A6
November 1992

This manual gives a brief overview of the hardware and software products that are comprised in the MasPar Computer System.

MasPar Computer Corporation makes no representation or warranties regarding the contents of this document and reserves the right to revise this document or make changes in the specifications of the product described herein at any time without obligation to notify any person of such revision or change.

Copyright © 1990, 1991, 1992 MasPar Computer Corporation, Sunnyvale, California

All Rights Reserved
Printed in the United States of America

This manual, its format, and the hardware, software, and microcode described in it are copyrighted by MasPar Computer Corporation and may not be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the express written permission of MasPar Computer Corporation.

Restricted Rights Legend

The software and microcode documented in this manual has been developed at private expense and is not in the public domain. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (c)(1)(i) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

MasPar and the MasPar logo are registered trademarks of MasPar Computer Corporation.
MasPar Programming Environment, MasPar Fortran, MasPar Programming Language, MasPar Mathematics Library, MasPar Image Processing Library, MasPar Parallel Disk Array, MasPar Data Display Library, MPPE, and MPL are trademarks of MasPar Computer Corporation.
UNIX is a registered trademark of the American Telephone and Telegraph Company.
DEC, DECstation, and ULTRIX are trademarks of Digital Equipment Corporation.
Motif is a trademark of The Open Software Foundation, Inc.
X Window System is a trademark of the Massachusetts Institute of Technology.
AVS is a trademark of AVS, Inc. NFS is a trademark of Sun Microsystems.

MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale, California 94086
408-736-3300
408-736-9560 (FAX)
# Table of Contents

## Chapter 1. Introducing the MasPar System

- Data Parallel Terminology .................................................. 1-2
- The MasPar Hardware ...................................................... 1-3
  - Optional Hardware ................................................... 1-4
  - MPIOC ................................................................. 1-4
  - VME Interface ....................................................... 1-4
  - Disk Subsystem ...................................................... 1-4
- The MasPar Software ..................................................... 1-5
  - MPPE ................................................................. 1-5
  - Languages ............................................................ 1-5
  - Libraries ............................................................. 1-6
  - Other Tools .......................................................... 1-6

## Chapter 2. The MasPar Hardware

- MP-1 Models ............................................................... 2-3
- MP-2 Models ............................................................... 2-3
- The Front End ............................................................ 2-4
- The DPU ................................................................. 2-4
  - The Array Control Unit ........................................... 2-4
  - PE Array ............................................................. 2-5
  - PE Communications ............................................... 2-5
    - ACU to PE Communication ..................................... 2-6
    - PE to PE Communication ...................................... 2-6
- VME Interface ........................................................... 2-7
- MPIOC ................................................................. 2-7
  - MPIOC Overview ................................................... 2-7
Chapter 3. MasPar Software 3-1

UNIX Operating System .................................. 3-2
MasPar Programming Environment (MPPE) .................. 3-2
Using a Programming Environment ....................... 3-3
Programming with the MasPar Languages ................. 3-4
  Programming Models .................................. 3-5
  MasPar Programming Language (MPL) .................. 3-5
  MasPar Fortran ...................................... 3-6
  MasPar VAST-2 ...................................... 3-7
 Compiler Drivers ....................................... 3-8
 Libraries ............................................. 3-9

Appendix A. Documentation List A-1

Programming Environment ................................ A-1
MPL ...................................................... A-1
MasPar Fortran ......................................... A-2
Libraries .............................................. A-2
Hardware Documentation ................................ A-2

Appendix B. Technical Specifications B-1

Appendix C. Glossary C-1

Index
Chapter 1

Introducing the MasPar System

The MasPar® System integrates a *massively parallel* computer with a high-productivity *data parallel* software development environment. This manual provides a technical introduction to the elements of the MasPar system.

This chapter gives you an overview of what data-parallel computing is, and provides a general description of the hardware and software in the MP-1 system.

Chapter 2 gives a summary of the hardware components.

Chapter 3 provides a description of the software.

Appendix A lists all of the documentation available for the MasPar system.

Appendix B contains the technical specifications of the MasPar system.

Appendix C is a glossary of terms and acronyms.
Data Parallel Terminology

To understand the technology of the MasPar system, you first need to be familiar with the terms used in parallel computing, and the terms and acronyms used by MasPar. Additional terms and acronyms may be found in Appendix C.

**sequential computing**
This is referred to in the data-parallel world as *conventional* computing. In a sequential system, one instruction is given to the computer’s single CPU and that instruction is carried out before the next instruction is given.

**CPUs, processors, and processor elements**
A sequential computer system has a CPU; parallel computers have multiple *processors* in addition to a CPU. The MasPar system uses the term Processor Elements, or PEs, to refer to the processors in the MP-1 machine.

**parallel computing**
This is teaming processors together to solve problems in seconds that might take hours or even days on a conventional (sequential) system. The basic idea is that if one processor is fast, 10 working together would be faster, and 1,000 would be superfast.

**massively parallel**
This term describes parallel computing systems that have more than 1,000 processors. (However, some massively parallel MIMD systems have as few as 64 processors.) The MP-1 is massively parallel.

**SIMD and MIMD**
SIMD and MIMD are different parallel architectures. SIMD is an acronym for Single Instruction, Multiple Data. SIMD systems are used where there is a large data set to be processed and a single processor for each item in the data set. The same set of instructions is applied to all the items in the data set. MasPar computers are considered ASIMD systems (Autonomous SIMD). MIMD is an acronym for Multiple Instruction, Multiple Data. MIMD systems are used where there are separate, independent problems to solve. Each problem is assigned to a separate processor (or group of processors), and all the problems are solved simultaneously.

**data parallel computing**
Data parallel is a programming style used on massively parallel computers that have a data parallel architecture. To program in a data parallel manner means that you use the parallelism inherent in the problem data to take advantage of many parallel processors.

**DPU**
DPU is an acronym for Data Parallel Unit. This part of the MasPar System does all the parallel processing. The DPU contains the PEs, the Array Control Unit (ACU), and mechanisms for communications between the PEs and between the PEs and the ACU.

**ACU**
The Array Control Unit, or ACU, is a processor that has its own data and instruction
memory. Located in the DPU, its purpose is to control the PEs and perform operations on scalar data. The ACU also communicates with the front end; PEs do not communicate directly with the front end.

**PE Array**
In the MasPar system, the processor elements are arranged in a two-dimensional matrix called the PE Array.

**cluster**
In the PE Array, each non-overlapping matrix of a 4 by 4 array of PEs is a cluster.

**parallel variable or plural variable**
A parallel variable is replicated exactly (except for its values) in each PE. The variables are called ‘‘plural’’ variables in MPL.

**singular variable**
This is any variable that is *not* a parallel variable. Singular variables reside on the ACU or the front end, not in the PE array.

**The MasPar Hardware**

The MasPar hardware consists of at least two physical boxes: a *front end* that includes a graphics workstation, and a *Data Parallel Unit* (DPU). (You can also have an optional parallel disk array; see page 1-4.) The following figure shows how the system might look when it is installed in your office. The box under the monitor is the front end processor, and the box beside the desk is the DPU.
The front end
The front end hardware is made up of a processor, a monitor, a keyboard, and a mouse. The front end hardware provides you with a complete work environment running standard UNIX operating system software (DEC ULTRIX).

The DPU
The DPU is the part of the MasPar system that performs all the parallel processing. The DPU hardware has two major parts: the PE Array, which performs the parallel calculations, and the ACU, which controls the interactions between the front end and the PE Array. The DPU also contains communications mechanisms for PE to PE and PE to ACU communications; see page 2-5.

For information on the differences between the different models of the MasPar system, see page 2-3.

Optional Hardware
The optional MasPar hardware pieces make up an I/O subsystem that allows data to be moved into or out of the PEs at a high speed. The I/O subsystem is customizable, depending on the devices you select.

Currently the I/O subsystem is composed of an optional Parallel VME board, optional VME 6U and 9U adapter boards, a high speed I/O Controller, and an optional disk subsystem, the MasPar Parallel Disk Array (MPDA), which provides bulk data storage.

MPIOC
The MasPar® Input/Output Channel (MPIOC) is a high-speed bus used for transfers between I/O devices such as disk array controllers and I/O RAM. The MPIOC uses 64 bidirectional data lines.

VME Interface
The MasPar VME interface, along with the VME6U and 9U adapters, lets you attach a wide range of off-the-shelf VME cards to the MasPar system. These include frame buffers, frame grabbers, signal processing devices, disk controllers, and even custom VME cards. The routines in the standard MPL programming library let you write custom drivers or applications that access the VME interface from user space. The MasPar VME interface architecture lets you choose either the regular MasPar VME Interface or the MasPar Parallel VME Interface. (See page 2-7 for more details.)

Disk Subsystem
The disk subsystem consists of the MasPar Parallel Disk Array (MPDA), a scaleable I/O device for bulk data storage. The MPDA consists of a disk array controller, a parallel disk array, and a VME interface. The disk array controller provides a high-speed data
connection to the disk array. Architecturally, the disk array is like a conventional BSD 4.2 UNIX file system.

The disk array is comprised of 5-1/4" ESDI disk drives, arranged in banks of 4 or 8 data disks, with a parity disk for each bank. There is an optional hot standby disk in bank 1 and a maximum of two banks in an MPDA.

The MasPar Software

The MasPar software system includes the top-level ULTRIX user interface on the front end, high performance programming languages, an interactive programming and debugging environment, and other tools and libraries. The MasPar languages have features that enable you to use the DPU effectively. The programming environment includes its own on-line help, utilities, and an interactive window environment for analyzing, debugging, and profiling your programs. Libraries are available to aid in the development of specific applications.

MPPE

The MasPar Programming Environment (MPPE) is an interactive set of tools for debugging and optimizing your MasPar-compiled programs. MPPE simplifies your development effort by providing a graphical and intuitive environment for your work. You can use MPPE to:

- step through your program
- inspect parallel data values
- graphically display usage of the data processing elements in the DPU
- graphically analyze the amount of data being moved from the front end to the DPU (only applies to MasPar Fortran programs)

You can also have MPPE automatically start up when your program has a fault.

Languages

Two languages are currently available from MasPar: the MasPar Programming Language (MPL) and MasPar Fortran. MPL is MasPar's lowest level programming language; it provides programmers direct access to the DPU. MPL programmers must have a good understanding of the DPU architecture. MasPar Fortran is MasPar's implementation of Fortran 90, including the array extensions and intrinsic set. MasPar Fortran programmers do not need as detailed a knowledge of the DPU as MPL programmers.

The MasPar software allows you to choose the degree of interaction you want to have with the architecture. If you program in MasPar Fortran, for example, the architecture
isn’t as evident as when programming in MPL. In other words, one advantage of programming in MasPar Fortran is that the MasPar Fortran optimizing compiler automatically generates code for both the front end and the DPU. In contrast, if you use MPL, you can have explicit control of the PE array. So, the amount of direct control you want influences the choice of programming languages.

Program structure and instructions that do not involve parallel variables are the same in the MasPar languages as they are in conventional C or Fortran, and instructions still execute serially. The difference is that the MasPar languages enable you to use the parallel data processors efficiently. You (or the compiler) can declare parallel variables and then you can perform operations on them the same way you perform the same operations on singular variables. For example, in MasPar Fortran you can add the respective elements of two arrays and assign the sum to the respective elements of a third array in one statement, without coding a loop.

Libraries

The MasPar software system includes several libraries that provide common routines that you might want to use in your data parallel programs. These libraries are easy to access and fully integrated with the MasPar languages.

• the MasPar Mathematics Library (MPML)
• the MasPar Data Display Library (MPDDL)
• the MasPar Image Processing Library (MPIPL)

MPDDL is part of the standard MasPar system software. MPIPL and MPML are optional products. See page 3-9 for more information on each of these libraries.

Other Tools

The MasPar system includes command-line tools such as compiler drivers, utilities for managing libraries and checking the DPU job queue, and man pages. These command-line tools can greatly increase your efficiency. The following list contains several command line tools that you may use on a regular basis. The MasPar tools are discussed in detail in the System Administration Manual, the MasPar Commands manual, and the MasPar Fortran manual set.

`mpq` allows you to check the number of jobs that are currently accessing the DPU, to give you a better idea of how soon your job will have its share of the DPU time.

`mpvast` MasPar Fortran conversion utility; translates f77 code to MasPar Fortran code.

`mpdecl` MasPar declaration help

`mpdc` parallel calculator

`mpi` MasPar machine information
pcmp       parallel file compare

dpumanager
manage the dpu queue, set time and job limits (dpumanager is a daemon)
Introducing the MasPar System
Chapter 2

The MasPar Hardware

This chapter describes the hardware elements of the MasPar system in detail. (For even more detail on the architecture of the MasPar system, see the MasPar MP-1 Architecture Specification.)
The MasPar system architecture consists of four subsystems:

- the front end machine running ULTRIX, with standard UNIX I/O
- the high-speed I/O subsystem
- the PE array
- the ACU

Physically, the last three are all included in the DPU box. These subsystems work together transparently; all you see is the top-level UNIX-style environment. The following sections describe the hardware configurations in detail; for tables of hardware specifications, see Appendix B.
MP-1 Models

The MasPar first generation system includes two series of DPUs: the MP 1100 series and the MP 1200 series. Each model has at least 1,024 parallel data processor elements. The main difference between these two series is that the 1200 series can have more processor elements—more PEs means more data memory and higher performance. MP 1100 series machines have 6 I/O slots, and MP 1200 series machines have 16 I/O slots, of which 5 and 14 of the slots, respectively, are available. The MP 1200 series is larger and requires more power than the MP 1100 series.

The number of PEs is expandable in both MP 1100s and MP 1200s. There are 1,024 PEs on each PE board. The MP 1100 machines can be configured with 1, 2, or 4 PE boards. The MP 1200 machines can be configured with 1, 2, 4, 8, or 16 PE boards. If they have the same number of PEs, MP 1100 machines are functionally the same as MP 1200 machines.

Both the 1100 and 1200 models have a footprint of less than 7 square feet and fit into a standard office environment. Running on 110 or 220 volt AC power (depending on your configuration), they need no special facilities.

<table>
<thead>
<tr>
<th>1100 Series:</th>
<th>1200 Series:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enclosure</strong></td>
<td><strong>Enclosure</strong></td>
</tr>
<tr>
<td>42&quot; H x 18.5&quot; W x 32.5&quot; D</td>
<td>55.5&quot; H x 23.5&quot; W x 32.5&quot; D</td>
</tr>
<tr>
<td>&lt;500 lbs</td>
<td>&lt;800 lbs</td>
</tr>
<tr>
<td><strong>Power requirements</strong></td>
<td><strong>Power requirements</strong></td>
</tr>
<tr>
<td>110V/220V 15/10A</td>
<td>220V/60Hz 30A</td>
</tr>
<tr>
<td>50Hz or 60Hz</td>
<td>50Hz or 60Hz</td>
</tr>
<tr>
<td>300 - 1,200W, depending on</td>
<td>800 - 3000W, depending on</td>
</tr>
<tr>
<td>configuration</td>
<td>configuration</td>
</tr>
</tbody>
</table>

MP-2 Models

The MasPar® MP-2 is our second generation massively parallel processing system. Each model has at least 1,024 simple, parallel, data processor elements. The MasPar system consists of two boxes: a front-end and a DPU. Optionally, a disk array subsystem in its own cabinet is available.

The MP-2 includes MP 2200 Series systems. MP 2200 Series systems support 1, 2, 4, 8, or 16 PEB2s and have 15 I/O slots. A PEB2 array board contains 1024 4-bit processor elements.

The model number is derived from the series number and the number of PEB2s, as shown in the following table.
<table>
<thead>
<tr>
<th>Model Number</th>
<th>Family</th>
<th>Processors</th>
<th>Memory</th>
<th>I/O Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP 2201A</td>
<td>2200</td>
<td>1,024</td>
<td>64</td>
<td>15</td>
</tr>
<tr>
<td>MP 2202A</td>
<td>2200</td>
<td>2,048</td>
<td>128</td>
<td>15</td>
</tr>
<tr>
<td>MP 2204A</td>
<td>2200</td>
<td>4,096</td>
<td>256</td>
<td>15</td>
</tr>
<tr>
<td>MP 2208A</td>
<td>2200</td>
<td>8,192</td>
<td>512</td>
<td>15</td>
</tr>
<tr>
<td>MP 2216A</td>
<td>2200</td>
<td>16,384</td>
<td>1,024</td>
<td>15</td>
</tr>
</tbody>
</table>

The Front End

The front end for this release is a Digital Equipment Corporation (DEC) DECstation 5000. The front end includes a monitor where you can work, or you can communicate with the front end over a network. The front end includes Ethernet hardware and a minimum of 24 megabytes of memory.

Machines that can be used on the network to communicate with the front end include any workstation or terminal running an implementation of UNIX, such as DECstation 5000s, Sun SPARCstations/Sun4s, and most X terminals. To run MPPE, you need to be running an X window system, such as Motif.

Depending on how your system administrator sets up your MasPar system, up to 16 users at a time can communicate with the DPU. Sophisticated job switching techniques allow all of the users to share the DPU, with each job running for a period of time before it is switched out.

The DPU

The DPU is that part of the MasPar system that does all the parallel processing. It consists of the Array Control Unit, a PE array of at least 1,024 processor elements (up to 16,384), and sophisticated communications mechanisms.

The Array Control Unit

The ACU controls the PE array and controls interactions between the front end, the PE array, and the I/O subsystem. The ACU also performs operations on the scalar data in your program. The ACU is a 12-MIPS control processor with a data register set, data memory, and instruction memory. For the MP-1, it has thirty-two 32-bit registers, 128 kilobytes of data memory, and 1 megabyte of instruction memory RAM that expands to 4 gigabytes of virtual instruction memory. For the MP-2, it has a control processor, fifty 32-bit registers, 512 kilobytes of data memory, 4 megabytes of instruction memory RAM, and 4 gigabytes of virtual instruction address space that is managed as 4096-byte pages.
**PE Array**

The PE array is where all parallel data is processed. Variables that are declared (by you or by the compiler) to be parallel variables are allocated on the PEs. Each PE receives the same instruction simultaneously from the ACU. The PEs then execute the instruction on these parallel variables. In this way, PEs are used to represent different data points in the same problem. For example, PEs can be used to represent the devices in a circuit, the pixels in an image, or the molecules in a fluid.

Each PE is a 4-bit load/store arithmetic processor with dedicated register space and RAM. Each PE has a 1.8-MIPS processor, forty 32-bit registers, and 16 or 64 kilobytes of RAM.

The PEs are arranged in a two-dimensional matrix called the PE array. A system has 1K, 2K, 4K, 8K, or 16K PEs (see the section “MP-1 Models” in Chapter 1). These PEs are arranged in a matrix that either has an equal number of columns and rows or has twice as many columns as rows.

![PE Cluster Diagram](image)

Each non-overlapping square matrix of 16 PEs in the PE array is called a cluster. A 1K PE array is made up of 64 4x4 PE clusters, a 2K PE array has 128 clusters, and so forth. It’s important to understand the layout of the clusters when you are programming in MPL, so you can use the best method of communication between PEs.

**PE Communications**

There are two types of PE communications:

- communication between the PEs and the ACU
- communication between two PEs in the PE array

Communication between the PEs and the ACU takes place over a special ACU/PE bus. Communication between PEs in the PE array can be via X-Net or the Global Router. If you program in MPL, you must specifically indicate which method of communication to use; see the MPL and MasPar Fortran manuals for more information.
All X-net and Global router transmissions are parity checked, so there are no lost messages. This means that you don’t need to put any code into your program that performs software message checking.

**ACU to PE Communication**

When communication is from the ACU to the PEs, the ACU is said to broadcast instructions and data to the PEs. Regardless of the number of PEs, it takes the same amount of time to communicate to one PE as it does to communicate to all the PEs.

When the PEs all simultaneously communicate a piece of data to the ACU, the values of the parallel variable are logically reduced (global ORed) to one value in the singular ACU variable.

**PE to PE Communication**

PE to PE communication can be on a straight line, or any one PE can communicate with any other PE in any location in the array. Connected PEs can both send and fetch information.

Communication between two arbitrary PEs uses the Global Router. The straight-line communications are called X-Net communications, and they go in the following directions: north, northeast, east, southeast, south, southwest, west, and northwest.

The main difference between X-Net and Global Router communications is that the X-Net has a built-in direction of communication, while the Global Router does not. In general, X-Net communications are significantly faster than Global Router communications, but Global Router communications are more general purpose. Global Router communications may actually be faster than X-Net when communicating across a large number of PEs, or when communicating with PEs that aren’t in a direct line.
In MPL, you can choose X-Net or Global Router communications explicitly by using different language constructs. In other MasPar languages, the communications mechanisms are transparent to the user.

**VME Interface**

Data transfer between the MasPar system and hardware I/O products is available through the VMEbus interface (as well as through the Global Router). VME transfers are accomplished through one of two interfaces. The vmeaccess routines allow scalar and parallel access to the VME bus address space. Their procedural interface is guaranteed to work across various system software and hardware changes.

The VME file system also offers a procedural interface with the additional benefit of support for the various file descriptor-based interfaces (such as p_read, read2dcs, and so on). In the 3.0 release, the vmeaccess interfaces are built on top of the VME file system. If a PVME option (or an IOCTL option) is available, parallel transfers will take the fastest path between the target device and PMEM.

For details on VME data accessing and addressing modes, see Appendix B.

Specially crafted I/O library routines in the MasPar VME I/O library let you write custom drivers or applications that access the VME interface. VME access routines are described in detail in the *MPL Reference Manual* and the *MPL User Guide*.

**MPIOC**

**MPIOC Overview**

The MasPar® Input/Output Channel (MPIOC) is a high-speed bus used for transfers between I/O devices such as disk array controllers and I/O RAM. The MPIOC uses 64 bidirectional data lines.

The MPIOC supports three types of boards:

- The I/O Channel Controller (IOCTL). The IOCTLR performs all arbitration and transaction initiation functions for the MPIOC.

  The IOCTLR also interfaces to the VMEbus and to the MasPar global router network that connects to the processor elements (PEs).

  The IOCTLR has an I/O buffer of 8 MBytes of parity-checked memory.

- The I/O RAM modules (IORAMs). IORAMs act as a data buffer between the DPU and the MPIOC. The buffer has either 128 or 32 MBytes of Error Correcting Code (ECC)-protected memory, depending on the IORAM module model.
Optional I/O devices. I/O devices get data into or out of either the DPU or another device.

**MPDA**

The MPDA provides up to 22 gigabytes of formatted capacity and up to 10.5 megabytes per second of sustained disk I/O. The architecture of the MPDA is ideal for a wide range of data intensive applications, including scientific visualization, remote sensing, data acquisition, image processing, geographic database management, and seismic data analysis.

The MPDA uses RAID-3 (Redundant Arrays of Inexpensive Disks) architecture. The disk drive array transfers data in parallel with one redundant drive functioning as a parity check disk. The array works as one large virtual drive. Parity is checked using the XOR of data on a bank of four drives.

The MPDA treats the entire disk array bank as a single, logical storage device. You can create individual files of up to 2 gigabytes (the maximum size that UNIX supports—the MasPar file system allows files up to 1 terabyte), or many smaller files stored across multiple disks. Each formatted data disk provides 682.92 megabytes of storage, with up to 21.37 gigabytes of formatted capacity in a 16-disk configuration (if using 1.5 gigabyte drives) on 2 separate banks (10.67 per bank).

The MPDA can withstand a disk failure without any data loss or interruption of work because the parity disk allows one drive to fail without any data loss. For added protection, a hot standby disk drive option is available to withstand two separate disk failures without data loss. In addition, there is a 48-bit error correction code (ECC) on each disk and “on the fly” error recovery for the entire array. If a disk drive fails or errors in data appear, these levels of fault tolerance operate automatically and are transparent to the user.

You can replace a failed disk during system operation, and the MPDA minimizes performance degradation on active disks while reconstructing data on the new disk.

Disk-drive replacement procedures and configuration upgrades require a minimum amount of work. Each disk is in a modular disk canister, which is easily installed or removed. Often the modules in one configuration can be reused in a larger system, and most modules do not have to be moved from their current positions.

The MPDA requires memory on the the PVME I/O controller board with 8 megabytes of storage. The MPDA uses the VME bus to transfer data to the MP-1. Employing a configurable I/O RAM as a buffer, the PVME I/O controller provides direct access to the PE Array at rates of up to 64 megabytes per second.

A diagnostic port allows for debugging. The front panel lightpipe provides status for both system power and individual disks.
The MPDA is a standard NFS mountable file system that can be accessed from any NFS-capable system on the network, so the MPDA can be connected to systems other than the MasPar system.
Chapter 3

MasPar Software

This chapter details the programming environment, languages, and tools available on the MasPar system:

- the UNIX operating system
- the MasPar Programming Environment (MPPE)
- MPL
- MasPar Fortran
- associated compilers and other tools
UNIX Operating System

ULTRIX, DEC's implementation of UNIX, gives users access to hundreds of utilities and applications.

Running on top of ULTRIX is OSF Motif, the recommended windowing system for users of MPPE. Motif gives users a standard windowing environment, complete with icons, menus, and re-sizeable windows.

MasPar Programming Environment (MPPE)

The MasPar Programming Environment is an integrated graphical environment for developing and debugging your programs. It simplifies your work by providing a powerful source-level debugger, a visual interface for analyzing machine use and program history, and an interactive profiler to help you optimize your programs. MPPE runs in any X window environment, but Motif is recommended.

MPPE works on the MasPar system in the manner of a client-server process. A server process is set up to receive requests initiated by the client. In the case of the MasPar system, you work in the client process to debug or optimize your code, and when you execute an instruction that requires the use of the DPU (such as stepping through a program), the client passes the request to the server portion of MPPE, and the PEs are engaged. This makes for a more efficient data-parallel debugging process, since the PEs are only accessed when they are directly needed.

You can use MPPE to debug any program that was compiled with one of the MasPar compilers. It does not have to be a program that uses the DPU—you can write a program
that runs completely on the front end. However, MPPE has some features that are
provided especially for debugging data-parallel programs running on the MasPar parallel
processor. For example, you can display the values of parallel variables. (See the
following section on visualization tools.) A program that was compiled with one of the
MasPar compilers cannot be debugged using any other debugging tool (other than print
statements).

MPPE allows you to control and observe a program that is executing in the MasPar
parallel processing system. The Execution window shows all the source code, even if not
yet executed, as well as both global and local data. You can set and modify breakpoints,
step through the program, inspect data values, and profile all or part of your program.
Most MPPE functions can be performed by simply selecting that function from a menu
with the mouse.

You can also invoke and display MPPE remotely, using a machine that is not directly
connected to a DPU. Although the program you are debugging must run on the front-end
machine that is directly connected to the DPU, you can connect to the MasPar system
over the network and have your own workstation or terminal run (or echo) the process.
In most cases, this may increase your performance; for example, if the work you are
doing in MPPE involves very little use of the DPU, you may benefit from working on a
networked machine. See the System Administration manual for more information.

Using a Programming Environment

The standard workstation-based programming tools provided with the DECstation front
end adequately support editing, compilation, and building of programs. But workstation
tools do not adequately support controlled execution, debugging, profiling, or viewing of
parallel data.

MPPE can help you in developing, implementing, debugging, tuning, and maintaining
data parallel programs. A robust programming environment is especially important when
working with massively parallel computers because there are new language semantics
and architectural behaviors to model and control.

A massively parallel programming environment must:
- help you debug programs
- help you understand data parallel algorithms
- aid your performance tuning

MPPE does all of the the above, but more importantly, it does the following:
- Does not force you to make debugging decisions at compile time. Programs
  compiled using standard levels of optimization are handled correctly within
  MPPE—you aren’t forced to set flags at compile time in order to use MPPE
effectively.
- Gives truthful debugging of optimized code. MPPE can give good results
  even with optimized code. If MPPE can treat the code as though it was
unoptimized, it will do so; in addition, if it can’t work with the code, it will tell you so.

- Helps you visualize massively parallel data instead of forcing you to deal purely with numbers. Pattern matching is easier when done visually, especially with the huge data sets used in massively parallel programs.

- Enhances your productivity by providing a graphical user interface. Being able to see the source code, the stack back trace, and the values of active variables at the same time makes debugging easier—and being able to ‘‘point and click’’ to perform an action is a great time saver.

- Keeps displays simple but allows you to magnify details so you can see them in many different contexts. Basic information about your program is presented succinctly, but in-depth information is quickly available at the click of a button. This is true for types and values of variables, detailed profile information, and visualized data.

- Animates debugging steps so you can dynamically monitor your programs progress. You can see each line as it executes—this means you can not only monitor execution flow, but also monitor patterns in changing variable values.

Programming with the MasPar Languages

Parallel programming is, in its most basic sense, taking a large problem, breaking the problem into pieces, and giving each piece to a processor or group of processors. Quite often, you don’t even need to tell which piece goes with which processor.

Before you write any code for your parallel program, you need to identify what parts of your program benefit from being non-serialized. To determine what parts will benefit from being non-serialized, you can look for the following characteristics:

- what results you want from the program
- the list of tasks you expect the program to accomplish

If your results are something that can be divided into many separate components, then the program may benefit from being parallel. The same goes for the list of tasks—if each task requires the processing of large amounts of data, then this program will also benefit from parallelism.

For the MasPar system, the part of a program that would benefit the most from being parallel is any problem where you want to do exactly the same operation on thousands and thousands of different pieces of data. For example, each PE could represent one piece of a car in a crash test, or one entry in a large database, or one pixel in an image.

Depending on the type of problem you are working on, you need to choose an appropriate language to use. Both MPL and MasPar Fortran have features that may make one a better choice than the other. The following sections describe the programming models you might use, and describe the two languages.
Programming Models

The MasPar languages are based on familiar, conventional languages and well-known standards. The extensions to support parallel programming fit into the familiar syntax, so a new programming style is not required. But you do need to learn to think about your program in a different manner. In other words, you may need to learn a new programming model in order to take full advantage of the features in the MasPar system.

For example, not all code can or should be executed on the DPU—the fast scalar processor on the front end can play an important part in your data parallel program. You can use a mixture of scalar processing on the front end and data parallel processing on the DPU in several different ways:

- The synchronous programming model. The program consists of two separate but tightly integrated parts, one written for the front end and one for the DPU. In many programs, the front end sends a routine to the DPU, the DPU routine completes, and then the program returns to the front end. MPL is most commonly used to achieve this model.

- The asynchronous programming model. Works like the synchronous model except the front end doesn’t wait for the DPU to finish its routine before calling for another routine. This model may have a performance gain, but is much harder to write; however, there are some interactive applications that may require it.

- The unified programming model. This model consists of a single program compiled by a single compiler that generates code for both the front end and the DPU simultaneously. In MasPar Fortran, any synchronization is handled explicitly by the compiler-generated code.

MasPar Programming Language (MPL)

MPL is the lowest level programming language that MasPar supports. It is based on ANSI C, and it can be used for DPU system programming similar to the way C can be used on conventional machines.

Most ANSI C language features are supported by the MPL compiler. Most programs that compile under ANSI C will also compile under MPL. See the MPL manuals for details.

In addition to supporting ANSI C language features, the following statements, keywords, and library functions have been added to support data parallel programming:

- A new keyword, plural, distinguishes between two independent address spaces. Variables defined using the keyword plural are allocated identically on each PE in the PE array. Variables defined without using the keyword plural are singular variables and are allocated on the ACU.

- Plural expressions are supported. All arithmetic and addressing operations are supported for plural data types. For example, you can say "k = i + j", where k, i, and j are plural types.
- Data parallel control statement semantics are implemented. Data parallel control flow also controls the active set (defined below).

- Data parallel communications are implemented. New constructs, xnet and router, make data transfer in the PE array explicit.

- Dynamic auto arrays allow you to develop functions that can run on any size DPU.

An important concept in data parallel programming is that of the active set. The active set is the set of PEs that is used by your program at any given time during execution. This set is defined by conditional tests in your program. In MPL, the size of the active set can be no larger than the physical size of the PE array. Plural data operations in an MPL program apply uniformly to all active PEs.

Using MPL, it is possible to write a program that executes entirely in the DPU. Scalar code is executed in the ACU and parallel code is executed in the PEs under the control of the ACU. But executing entirely in the DPU is not practical for most real applications because the ACU is not as powerful in scalar computation as, for example, a DECstation 5000. Some operations, such as char and short memory operations and floating-point arithmetic operations, are not particularly fast on the ACU. See the MPL Reference Manual for more information.

The recommended model for programming in MPL is to recode the appropriate parts of existing applications using MPL to execute in a data-parallel way. Then, call these MPL subroutines from the front-end program. The basic steps are as follows:

1. Start with a conventional program that works on the front end.

2. Examine this program to determine which variables should be declared to be plural variables so they will be operated on in a data-parallel way.

3. Incrementally replace the code identified in step 2 with MPL subroutines; then call these subroutines from the front-end program.

See the MPL User Guide for more information.

MasPar Fortran

MasPar Fortran, an optional MasPar product, is based on FORTRAN 77 with extensions from DEC's VAX Fortran and the new Fortran 90 ANSI standard. The most important of these extensions are the Fortran 90 array statements, with which you can take advantage of data parallelism. These extensions include array notation and expressions, arrays as first-class objects, array sectioning, vector-valued subscripts, array constructors, control structures, a WHERE construct, and double-complex data types. Additionally, many new intrinsic functions have been added.

These Fortran 90 features assist you in easily accessing the data-parallel hardware of the MasPar system. The MasPar Fortran compiler determines which parts of your program should be allocated to the DPU and which should be allocated to the front end. Because the MasPar Fortran compiler determines where and how to compile and execute your
program, you do not need a detailed knowledge of how the data-parallel hardware works. However, some knowledge of data-parallel processing will help you write MasPar Fortran programs that use the MasPar hardware most efficiently. (Also, you can specify DPU data if you want to use special mapping directives.)

MasPar Fortran supports FORTRAN 77 as a subset. FORTRAN 77 programs that meet certain restrictions can be run “as is” on the MasPar system. However, such programs will not make use of the parallel-processing capabilities of the DPU. Although some DPU housekeeping code is generated, when you compile a FORTRAN 77 program most of the code will be scalar front-end code. As a result, when the program is executed, it will execute at the speed of the front end. To use the DPU effectively, you need to modify certain aspects of your program. By using Fortran 90 array statements (which are compiled into DPU code), you can take advantage of the power afforded by the data-parallel hardware.

These features are described fully in the MasPar Fortran Reference Manual and the MasPar Fortran User Guide. The MasPar VAST II manual describes how to convert FORTRAN 77 programs to MasPar Fortran.

**MasPar VAST-2**

MasPar VAST-2 enables Fortran programmers to increase their productivity dramatically by automatically translating application source code written in FORTRAN 77 to MasPar Fortran (Fortran 90-based) or from MasPar Fortran to FORTRAN 77. In addition to accelerating MasPar Fortran applications development, MasPar VAST-2 also gives you maximum flexibility in choosing the language (FORTRAN 77 or MasPar Fortran) for maintaining source code while preserving portability between scalar systems and MasPar’s massively parallel systems.

MasPar’s in-house users of the VAST-2 programming tool have experienced dramatic increases (sometimes as much as 100 percent) in programmer productivity. The key issue is the level of parallelism that is already defined in the application’s algorithm. If there is a high level of parallelism, MasPar VAST-2 delivers MasPar Fortran code which is ready to compile and run. If there is little implied parallelism, then at a minimum, VAST-2 converts existing DO loops into Fortran 90 array syntax. Thus, the expected productivity gains depend on the structure of your application.

The combination of MasPar VAST-2, MasPar Fortran, the MasPar Programming Environment (MPPE), and the MasPar libraries provides the industry’s leading data-parallel Fortran development environment.

After processing a Fortran program through MasPar VAST-2, two output files are created. One is a translated version of the input FORTRAN 77 program containing array syntax in place of the loops (i.e., a MasPar Fortran source file). This source file is ready for the MasPar Fortran compiler. The other is a listing of the input program with diagnostic comments, indicating which loops—if any—were not translated to array syntax and why. This list serves as a learning tool for future MasPar Fortran programming.
MasPar VAST-2 is both an optimizer (speeding up a program by exposing the implicit parallelism it contains) and a translator (changing the form of a program to a different dialect). Your goal—translation or optimization—determines which features of MasPar VAST-2 you want to enable.

MasPar VAST-2 translation features include:
- Conversion of DO and IF loops into array syntax.
- Recognition of most Fortran 90 intrinsics, including reductions, transcendental, and transformations.
- In-line expansion of subroutines and functions.
- Recognition of Fortran 90 matrix multiply and vector-matrix multiply intrinsics.

MasPar VAST-2 contains sophisticated data dependency analysis of loops which:
- Ensures safe translation of loops
- Maximizes the parallel execution of partially dependent loops
- Re-orders array references to increase efficiency

MasPar VAST-2 has built-in features for detecting and avoiding inefficient movement of data between the Data Parallel Unit (DPU) and the front end—thus increasing code performance. These features include:
- Detecting data movement caused by arrays that are used in both array syntax and scalar loops.
- Avoiding use of array syntax for arrays which should be processed on the front end because they are too small to benefit from the DPU.
- Splitting COMMON blocks into separate include files and into separate scalar and array COMMON blocks.
- Removing EQUIVALENCE statements which prevent parallel execution by MasPar Fortran.
- Emitting warning messages when arrays are dimensioned poorly for mapping onto the DPU.

**Compiler Drivers**

The MasPar system includes the following compiler drivers: `mpl_f77`, `mpl_cc`, `mpl`, and `mpfortran`.

- `mpl` Compiles and links code that contains MPL modules. Compiles and links MPL and MasPar assembler.
- `mpfortran` Compiles and links code that contains only MasPar Fortran modules. Compiles and links MasPar Fortran and MasPar assembler.
mpl_f77  Compiles and links code that contains some f77 modules and some
MPL modules. The mpl_f77 driver calls the f77 compiler. Note that
f77 does not automatically come with your DECstation. You must
purchase the f77 compiler and documentation separately from DEC.

mpl_cc  Compiles and links code that contains some ULTRIX C modules and
some MPL modules. The mpl_cc driver calls the ULTRIX cc
compiler.

Libraries

The MasPar system includes the following libraries:

- The MasPar Mathematics Library (MPML) is a set of routines that allows
you to perform various data-parallel mathematical operations. The library
includes block, vector, and matrix linear algebra routines, FFT routines, and
equation solvers. Most of the FFT routines and equation solvers are callable
from both MPL and MasPar Fortran. The other routines are callable only
from MPL. See the MPML Reference Manual.

- The MasPar Data Display Library (MPDDL) is a set of routines that helps
you take data from the PE array and display it as an image in an X window
or in a hardware framebuffer. These routines are callable from both MPL
and MasPar Fortran. See the MPDDL Reference Manual.

- The MasPar Image Processing Library (MPIPL) is a set of routines that
allows you to perform various operations on images stored in the PE array.
To display the images on a screen, you can use these routines in conjunction
with the MPDDL routines. See the MPIPL Reference Manual.

MPDDL is part of the standard system software. MPIPL and MPML are optional
products.
Appendix A

Documentation List

You may find the following publications useful.

Programming Environment

The following software manuals came with your system; consult them for more detailed information on the MasPar software programming environment:

- **MPPE User Guide**, PN 9305-0000
  - Explains how to use the MPPE window interface and how to use the tools in the MPPE.

- **MPPE Quick Reference**, PN 9305-0300
  - Provides a quick reference to the most used features of MPPE.

- **MasPar Commands Reference**, PN 9300-0300
  - Provides the syntax and usage of MasPar commands such as compilers and library management tools.

MPL

The following books are available for the MPL compiler that comes with the MasPar system:

- **MPL Reference Manual**, PN 9302-0001
  - Explains MPL keywords, statements, and library routines.
MPL User Guide, PN 9302-0101
Explains how to write a program using MPL. Gives examples and discusses performance and porting issues.

MasPar Fortran

The following books are available for the optional MasPar Fortran compiler:

MasPar Fortran Reference Manual, PN 9303-0000
Provides a complete reference to the syntax and usage of MasPar Fortran. Describes all intrinsics functions. Highlights the differences between MasPar Fortran and FORTRAN 77.

MasPar Fortran User Guide, PN 9303-0100
Explains how to convert a FORTRAN 77 program to MasPar Fortran, compile a program, and call other languages from within MasPar Fortran.

Libraries

The following books describe optional MasPar libraries:

MPDDL Reference Manual, PN 9302-0200
Provides a complete reference to the MasPar Data Display Library.

MPML Reference Manual, PN 9302-0400
Provides a complete reference to the MasPar Mathematics Library.

MPIPL Reference Manual, PN 9302-0300
Provides a complete reference to the MasPar Image Processing Library.

Hardware Documentation

The following architecture and hardware manuals came with your system; consult them for more detailed information on the MasPar hardware:

Site Prep Guide, PN 9300-5000
Gives the physical requirements of the MP-1 system and the MPDA, such as voltage and space requirements.

Installation and Service Manual, PN 9300-9023
 Tells how to install the hardware, run and interpret diagnostics, add processor element boards, and remove and replace FRUs.

MasPar MP-1 Architecture Specification, PN 9300-5001
Discusses the system architecture, theory of operations, register usage, the assembler, and more.
The following books describe optional hardware products:

**MasPar VME 6U Adapter Manual, PN 9300-5007**
Describes the MasPar VME 6U Adapter, its installation, configuration, and operation.

**MasPar VME 9U Adapter Manual, PN 9300-9019**
Describes the MasPar VME 9U Adapter, its installation, configuration, and operation.

**MasPar Parallel VME Manual, PN 9300-9018**
Describes the MasPar Parallel VME board, its installation, configuration, and operation.

**MasPar 3000-Series Parallel Disk Array Manual, PN 9300-5005, and MasPar 4000-Series Parallel Disk Array Manual, PN 9300-9040**
These books describe the two versions of the MasPar Parallel Disk Array (MPDA), its installation procedures, configuration, and operation. (System Administration information for the MPDA is in the *System Administration* manual.)
Appendix B

Technical Specifications

This appendix provides tables of technical information about the MasPar hardware, including:

- MP-1 series specifications
- MP-2 series specifications
- MPDA specifications
- Input/Output specifications
## MP-1 Series

### Number of Processors

<table>
<thead>
<tr>
<th></th>
<th>MP-1101</th>
<th>MP-1102</th>
<th>MP-1104</th>
<th>MP-1208</th>
<th>MP-1216</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP-1201</td>
<td>1,024</td>
<td>2,048</td>
<td>4,096</td>
<td>8,192</td>
<td>16,384</td>
</tr>
</tbody>
</table>

### Peak System Performance

- 32-bit integer MIPS: 1,600, 3,200, 6,400, 13,000, 26,000
- 64-bit integer MIPS: 800, 1,600, 3,200, 6,400, 13,000
- 32-bit floating point MFLOPS: 75, 150, 300, 600, 1,200
- 64-bit floating point MFLOPS: 34, 69, 138, 275, 550

### Memory

- Capacity (MBytes): 16 or 64, 32 or 128, 64 or 256, 128 or 512, 256 or 1,024
- Direct bandwidth (in MBytes per second): 690, 1,380, 2,760, 5,500, 11,000
- Indirect bandwidth (in MBytes per second): 250, 500, 1,000, 2,000, 4,000

### Registers

- Capacity (64-bit registers, 20 per PE): 20,480, 40,960, 81,920, 163,840, 327,680
- Bandwidth (GBytes per second): 6.4, 12.8, 25.6, 51.2, 102.4

### Inter-Processor Communications: 2D Torus Grid with 8-way nearest neighbor communications

- Bandwidth (MBytes per second): 1,400, 2,850, 5,700, 11,500, 23,000

### Global Router

- Connect time (μ-seconds): 3, 3, 3, 3, 3
- Bandwidth: send (MBytes per second): 80, 160, 320, 640, 1,300
- Bandwidth: fetch (MBytes per second): 80, 160, 320, 640, 1,300
## Technical Specifications

<table>
<thead>
<tr>
<th>Input/Output System</th>
<th>MP-1101</th>
<th>MP-1102</th>
<th>MP-1104</th>
<th>MP-1208</th>
<th>MP-1216</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MP-1201</td>
<td>MP-1202</td>
<td>MP-1204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I/O Controller RAM capacity (MBytes)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Processor-I/O Controller bandwidth peak (in MBytes per second)</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Maximum IORAM capacity (MBytes)</td>
<td>384/1,024</td>
<td>384/1,024</td>
<td>384/1,024</td>
<td>1,024</td>
<td>1,024</td>
</tr>
<tr>
<td>Maximum processor-IORAM bandwidth peak (in MBytes per second)</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1,024</td>
</tr>
<tr>
<td>MasPar I/O channel peak speed (MBytes per second)</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>MasPar Parallel VME peak speed (MBytes per second)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Number of I/O slots</td>
<td>5/14</td>
<td>5/14</td>
<td>5/14</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>
# MP-2 Series

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>MP-2201</th>
<th>MP-2202</th>
<th>MP-2204</th>
<th>MP-2208</th>
<th>MP-2216</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1,024</td>
<td>2,048</td>
<td>4,096</td>
<td>8,192</td>
<td>16,384</td>
</tr>
</tbody>
</table>

## Peak System Performance

- 32-bit integer MIPS: 4,250, 8,500, 17,000, 34,000, 68,000
- 64-bit integer MIPS: 2,100, 4,250, 8,500, 17,000, 34,000
- 32-bit floating point MFLOPS: 400, 800, 1,600, 3,200, 6,300
- 64-bit floating point MFLOPS: 150, 300, 600, 1,200, 2,400

## Memory

- Capacity (in MBytes): 64, 128, 256, 512, 1,024
- Direct bandwidth (in MBytes per second): 1,250, 2,500, 5,000, 10,000, 20,000
- Indirect bandwidth (in MBytes per second): 500, 1,000, 1,950, 3,900, 7,800

## Registers

- Capacity (64-Bit Registers): 20,480, 40,960, 81,920, 163,840, 327,680
- Bandwidth (in GBytes per second): 50.0, 100.0, 200.0, 400.0, 800.0

## Inter-Processor Communications: 2D Torus grid with 8-way nearest neighbor communications

- Bandwidth (in MBytes per second): 1,250, 2,500, 5,000, 10,000, 20,000

## Global Router

- Connect time (μ-seconds): 3.4, 3.4, 3.4, 3.4, 3.4
- Bandwidth: send (in MBytes per second): 80, 160, 320, 640, 1,300
- Bandwidth: fetch (in MBytes per second): 80, 160, 320, 640, 1,300
## Technical Specifications

<table>
<thead>
<tr>
<th>Input/Output System</th>
<th>MP-2201</th>
<th>MP-2202</th>
<th>MP-2204</th>
<th>MP-2208</th>
<th>MP-2216</th>
</tr>
</thead>
<tbody>
<tr>
<td>• I/O Controller RAM capacity in MBytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>• Processor-I/O Controller bandwidth peak in MBytes per second</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>• Maximum IORAM capacity in MBytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1,024</td>
<td>1,024</td>
<td>1,024</td>
<td>1,024</td>
<td>1,024</td>
</tr>
<tr>
<td>• Maximum PE-IORAM bandwidth peak in MBytes per second</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1,024</td>
</tr>
<tr>
<td>• MasPar I/O Channel peak speed in MBytes per second</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>• MasPar Parallel VME peak speed in MBytes per second</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>• Number of I/O slots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>
UNIX Subsystem

The UNIX subsystem for this release is a DECstation 5000/240HX.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit integer MIPS</td>
<td>42.9</td>
</tr>
<tr>
<td>Specmarks</td>
<td>32.4</td>
</tr>
<tr>
<td>32-bit floating point MFLOPS</td>
<td>10.8</td>
</tr>
<tr>
<td>64-bit floating point MFLOPS</td>
<td>6.0</td>
</tr>
<tr>
<td>Memory standard (MBytes)</td>
<td>24 or 32</td>
</tr>
<tr>
<td>Memory maximum (MBytes)</td>
<td>480</td>
</tr>
<tr>
<td>Disk standard (MBytes)</td>
<td>665</td>
</tr>
<tr>
<td>CD-ROM drive (MBytes)</td>
<td>600</td>
</tr>
</tbody>
</table>
MPDA Specifications

The following table contains information on the physical characteristics of the MPDA.

<table>
<thead>
<tr>
<th></th>
<th>DA4004A</th>
<th>DA4016A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensions</td>
<td>55.5&quot; H x 23.5&quot; W x 32.5&quot; D</td>
<td>55.5&quot; H x 23.5&quot; W x 32.5&quot; D</td>
</tr>
<tr>
<td>Maximum weight</td>
<td>&lt; 900 lbs</td>
<td>&lt; 900 lbs</td>
</tr>
<tr>
<td>Line voltage</td>
<td>220V or 240V 8A/60Hz or 50Hz</td>
<td>220V or 240V 16A/60Hz or 50Hz</td>
</tr>
<tr>
<td>Maximum power</td>
<td>2000W</td>
<td>3900W</td>
</tr>
<tr>
<td>Sound</td>
<td>&lt; 59 dBA</td>
<td>&lt; 59 dBA</td>
</tr>
<tr>
<td>Operating temperature</td>
<td>10 - 40°C (50 -104° F)</td>
<td>10-40°C (50-104°F)</td>
</tr>
<tr>
<td>Footprint</td>
<td>5.3 square feet</td>
<td>5.3 square feet</td>
</tr>
<tr>
<td>Relative humidity</td>
<td>10-80%</td>
<td>10-80%</td>
</tr>
<tr>
<td>(non-condensing)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The MPDA System must be physically attached to the MP-1 or MP-2, requiring enough floorspace to accommodate both cabinets.
Input/Output Specifications

The following table describes the specifications for the MasPar Input/Output Subsystem.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Controller Capacity (MBytes)</td>
<td>8</td>
</tr>
<tr>
<td>PE-I/O Controller BW Peak (MBytes/sec)</td>
<td>64</td>
</tr>
<tr>
<td>Maximum IORAM Capacity</td>
<td>1,024</td>
</tr>
<tr>
<td>Maximum PE-IORAM BW Peak (MBytes/sec)</td>
<td>64 to 1,024</td>
</tr>
<tr>
<td>I/O Channel Peak Speed (MBytes/sec)</td>
<td>200</td>
</tr>
<tr>
<td>Parallel VME Peak Speed (MBytes/sec)</td>
<td>16</td>
</tr>
<tr>
<td>Number of I/O slots</td>
<td>14</td>
</tr>
</tbody>
</table>
Appendix C

Glossary

This appendix contains definitions of many of the terms and abbreviations used in the MasPar documentation.

active set
This term applies only to MPL. The active set is the group of PEs that are enabled (i.e. haven’t been disabled) at any given time during execution. The size of the active set can be no larger than the physical size (number) of PEs.

ACU
Array Control Unit. The ACU is a processor that has its own data and instruction memory. Located in the DPU, its purpose is to control the PEs and perform operations on scalar data. The ACU sends data and instructions to each PE simultaneously. The ACU also communicates with the front end and the PEs; PEs do not communicate directly with the front end.

cluster
In the PE Array, each non-overlapping 4 by 4 array of PEs (16 PEs) is a cluster.

coarse grained
A coarse-grained parallel system is a system with many complex processors.

data parallel
A machine that has parallel data processors acting in unison on a single instruction at a time but on different data pvalues.

data visualizer
This MPPE window graphically displays the values of parallel variables.

DPU
Data Parallel Unit. This part of the hardware does all the parallel processing. The DPU contains the PEs, the Array Control Unit (ACU), and mechanisms for communications between the PEs and between the PEs and the ACU.
fine grained
A fine-grained parallel system is a system with many relatively simple processors. Such a system is usually a SIMD system. See grain size.

flat profile
A profile in which profile ticks are credited to a routine or statement only when it is actually executing. See "profile" below.

front end
A front end machine is a processor that runs an implementation of a UNIX operating system on a graphics workstation with a windowing capability similar to the X Window System. A front end machine is directly connected to a MasPar DPU. In this release, the front end is a DECstation 5000 with ULTRIX and DECwindows.

front-end-only program
This is a program that has been compiled with a MasPar compiler but that does not call any MPL code or any other parallel code. You may have such programs when you begin to port an existing program to the MasPar system or when you begin to write a new program for the MasPar system. There may also be times when you choose not to call the parallel modules of your program for debugging purposes.

Global Router
The Global Router manages simultaneous communication between any two arbitrary PEs, or between PEs and I/O.

grain size
Describes the relative size of a computer's individual processors. If there are very many small processors, or there are multiple computational elements within one processor, then the parallelism is called fine-grained. The MP-1 is a fine-grained system with many small processors. A coarse-grained system has many complex processors, and is usually an MIMD system.

hierarchical profile
A profile in which profile ticks are credited to a routine or statement not only when it is executing but when any flat-profiled routine calls directly or indirectly are executing.

inspector
This tool within the MPPE displays data values, types, and addresses.

machine visualizer
This tool displays the state of the PE array.

massively parallel
This term describes parallel computing systems that have more than 1,000 processors. The MP-1 is massively parallel.

MIMD
An acronym for Multiple Instruction, Multiple Data. MIMD systems are used where there are separate, independent problems to solve. Each problem is assigned to a separate processor (or group of processors), and each problem is solved simultaneously.

MPL
MasPar Programming Language. For more information, see the MPL User Guide and the MPL Reference Manual.

MPPE
Acronym for MasPar Programming Environment. MPPE is a window-based tool that allows you to control the state of
parallel computing

This is teaming processor elements together to solve problems in seconds that might take hours or even days on a conventional (sequential) system. The basic idea is that if one processor is fast, 10 working together would be faster, and 1,000 would be superfast.

parallel variable

A parallel variable is replicated exactly (except for its values) in each PE. The variables are called "plural" variables in MPL.

PE

Processor Element. Each PE is a load/store arithmetic processing element with dedicated register space and RAM. Each PE receives the same instruction simultaneously from the ACU. PEs that are enabled then execute the instruction on variables that reside on the PEs. These variables are called parallel variables. The set of PEs that is enabled at any given time during program execution is called the active set. In MPL, any variable that is declared to be a PE variable is replicated exactly, except for its value, in every PE.

PE array

Processor Element array. In the MasPar system, the PEs are arranged in a two-dimensional matrix of at least 1,024 PEs (up to 16,384 PEs).

processors

A sequential computer system has a CPU; parallel computers have multiple processors. The MasPar system uses the term Processor Elements, or PEs, to refer to the processors in the MP-1 machine.

profile

A list of statements or routines showing the number of profile ticks collected for each statement or subroutine. Two profile ticks are generated at every clock tick, one for the FE and one for the ACU.

sequential computing

This is referred to in the data-parallel world as conventional computing. In a sequential system, one instruction is given to the computer's single CPU and that instruction is carried out before the next instruction is given.

SIMD

Acronym for Single Instruction, Multiple Data. SIMD is a type of hardware. SIMD systems are used where there is a large data set to be processed and a single processor for each item in the data set. The same set of instructions are applied to all the items in the data set. MasPar computers are considered ASIMD systems (asynchronous SIMD).

singular variable

This is any variable that is not a parallel variable. Singular variables reside on the ACU or the Front End, not in the PE array. In MPL, singular variables can be used in programs for parallel processors, but they do not take advantage of parallelism. (There is no way to declare a variable that is allocated on only one PE in the PE Array.)
tick

Ticks provide a relative measure of the amount of time your program spends executing different statements and subroutines. See the discussions of the Statement Profiles bar, the Statement Profiles window, and the Routine Profiles window. Each time your program receives a UNIX clock interrupt, a tick is accumulated by the statement and routine that are currently being executed. Specifically, one tick is accumulated by the front end statement and one tick is accumulated by the ACU statement. In a synchronous program, these two are the same statement. In an asynchronous program, these two may be different statements.

Because ticks are accumulated using a sampling procedure (UNIX clock interrupts), a particular statement or routine may accumulate a slightly different number of ticks in different runs of the program. If you run the same program on two different MP-1 front end machines that have different clock periods, then the number of ticks accumulated will be different in each run but the relative number of ticks accumulated by each statement and each routine will be approximately the same.

toroidal wrap

This is the method the MasPar system uses to model the layout of the PEs in the PE array for X-net communications. For purposes of communicating between PEs in the PE array, the east edge of the physical array wraps around to the west edge, and the south edge of the physical array wraps around to the north edge.

visualize

Display in a graphical format, as opposed to a text format.

X-Net communications

Communications between any PE and any other PE in the same array that lies on a straight line from the original PE in one of the following directions: north, northeast, east, southeast, south, southwest, west, and northwest.
Active set 3-6, C-3
   definition of C-1
ACU 1-4, 2-4
   definition of 1-3, C-1
Array control unit
   See ACU
Asynchronous programming model 3-5

Broadcast 2-6

Cluster 2-5
Coarse grained C-1
Communications 2-5
   ACU—PE bus 2-6
   Global Router 2-6
   X-Net 2-6
Compiler drivers 1-6
Compilers 3-8
Control parallel
   definition of 1-2, C-2

Data parallel C-1
   definition of 1-2
Data Parallel Unit
   See DPU
Data storage 2-8
Data visualizer C-1
Debugger 1-5
Declaring variables 1-6, 2-5, 3-5
DECwindows C-2
Disk array

See MPDA
Disk subsystem 1-4
   See also MPDA
Documentation
   hardware A-2
   Software A-1
DPU 1-3, 1-4, 2-4
   definition of 1-2, C-1

Fine grained C-2
Front end 1-3, 2-4
   definition of C-2
Front-end-only programs C-2

Global router 3-6
Grain size
   definition of C-2

I/O 1-4, 2-7
I/O Channel Controller
   See IOCTLR
I/O RAM module
   See IORAM
I/O subsystem 1-4
Inspector C-2
IOCTLR 2-7
IORAM 2-7

Keyword plural 3-5

Machine visualizer C-2
MasPar Fortran 1-6, 3-6
Index

MasPar System
  illustration of  1-3
Massively parallel
  definition of  C-2, 1-2
MIMD  1-2, C-1, C-2
  See also  Control parallel
Monitor  1-4
MP  2200  2-3
MP-1 models
  MP 1100  2-3
  MP 1200  2-3
MP-2
  overview  2-3
MPDA  1-4
MPF
  See  MasPar Fortran
MPIOC  1-4, 2-7
  type code  2-7
MPL  1-5, 2-7, 3-5, 3-6, C-2
MPPE  1-5, C-2

Parallel computing
  definition of  1-2, C-3
Parallel variables  C-3
  definition of  1-3, C-3
PE  2-5, C-3
  definition of  1-2, C-3
PE array  1-4, 2-4, 2-5, C-3
  definition of  1-3, C-3
PE cluster
  definition of  1-3, C-1
Plural variables
  definition of  1-3, C-3
Processor element
  See  PE
Processor element control unit
  See  ACU
Processors  1-2, C-3
  See  PE
Profiling  C-4
Programming environment  1-5
  See also  MPPE
Programming models
  asynchronous  3-5
  synchronous  3-5
  unified  3-5

Sequential computing
  definition of  1-2, C-3
SIMD  1-2, 3-5, C-2, C-3
  See also  Data parallel
Singular variables
  definition of  1-3, C-3

Synchronous programming model  3-5

Ticks  C-4
Toroidal wrap  C-4
Type Code  2-7

ULTRIX  C-2
Unified programming model  3-5

Visualize  C-4
VME adapter  1-4, 2-8

X-net communications  3-6, C-4
MasPar Programming Environment (MPPE)

User Guide

Software Version 3.0

Document Part Number: 9305-0000
Revision: A8
July 1992

MasPar Computer Corporation
MPPE User Guide. Part Number 9305-0000, Revision A8, July 1992

MasPar Computer Corporation makes no representation or warranties regarding the contents of this document and reserves the right to revise this document or make changes in the specifications of the product described herein at any time without obligation to notify any person of such revision or change.

Copyright © 1990, 1991, 1992 MasPar Computer Corporation, Sunnyvale, California

All Rights Reserved
Printed in the United States of America

This manual, its format, and the hardware, software, and microcode described in it are copyrighted by MasPar Computer Corporation and may not be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the express written permission of MasPar Computer Corporation.

Restricted Rights Legend:
The software and microcode documented in this manual has been developed at private expense and is not in the public domain. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (c)(1)(i) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

MasPar and the MasPar logo are registered trademarks of MasPar Computer Corporation. MasPar Programming Environment, MasPar Fortran, MasPar Programming Language, MasPar Data Display Library, MasPar Image Processing Library, MasPar Mathematics Library, and MasPar Parallel Disk Array are trademarks of MasPar Computer Corporation.
DEC, DEC AVS, DEChet, DECwindows, DEChation, ULTRIX, VAX, and VAXstation are trademarks of Digital Equipment Corporation.
UNIX is a registered trademark of the American Telephone and Telegraph Company.
X Window System is a trademark of the Massachusetts Institute of Technology.
Motif is a trademark of the Open Software Foundation, Inc.
NFS is a trademark of Sun Microsystems.
AVS is a trademark of Advanced Visual Systems, Inc.
PostScript is a registered trademark of Adobe Systems Incorporated.
Sun, SunOS, Sun-4, SPARC, and SPARCstation are trademarks of Sun Microsystems, Inc.

MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale, California 94086
1-800-526-0916
1-408-736-9560 (FAX)
MPPE is the MasPar Programming Environment, a set of visually-oriented tools that help you to debug, understand, and optimize your data-parallel programs. You can use MPPE to look in detail at the behavior of a program as it executes, observe or change the values of variables in a program without recompiling, or monitor the performance of a program.

NOTE: The illustrations in this manual show MPPE running under the OSF Motif™ window manager. If you are using a different window manager, your screens may appear different, especially if your window manager doesn’t use window titles.

How To Use This Manual

This manual is organized to work as both a user’s guide and as a learning tool.

- Chapter 1 "Introducing MPPE" provides an introduction to MPPE and explains how MPPE fits into the MasPar® system.

- Read Chapter 2 "Starting An MPPE Session" to learn how to start MPPE and select a program to run. Chapter 2 also includes information on starting MPPE automatically, starting on a program fault, and using MPPE over the Internet.

- For an overview of the windows and options available in the MPPE interface, turn to Chapter 3 "Exploring the MPPE Interface." Chapter 3 also explains how to access and use the online Help.

- Chapter 4 "Opening and Executing Programs" tells you how to select a program you want to monitor or debug, how to move around in your code without executing it, and how to control your program execution through go to, skip, and step functions.
• Chapter 5 "Working With Breakpoints" tells you how to set and remove breakpoints and conditional breakpoints on lines and routines.

• Chapter 6 "Inspecting, Evaluating, and Visualizing Data" explains how to evaluate expressions, examine the contents of variables, and how to visualize the data in Processor Elements.

• Chapter 7 "Profiling" tells you how to view the compile-time and execution-time profiles of your programs.

• Chapter 8 "Resource and Variable History" describes looking at the record of how the contents of your variables change in relation to each other through time. It also discusses how your program uses DPU resources.

• Appendix A "Setting User Preferences" describes the settings and parameters used to customize MPPE.

• Appendix B "Invoking MPPE" tells you how to set environment variables for MPPE, and how to invoke and display MPPE on machines other than the MP-1 front end machine.

• Appendix C "Connecting to AVS" describes how to use the Application Visualization System (AVS) from within MPPE.

• Appendix D "Troubleshooting" discusses some common situations that could cause problems when you run your program in MPPE, and tells you how to contact MasPar customer support with your comments and questions.

**Typographic Conventions**

The following conventions are used throughout this book:

**This is used:**

- *text in italics* to show program variables, new terms
- **text in typewriter font** user input, items the user selects, examples, path names, command names
- Initial Caps on text names of tools, windows, panes, and menus
- "text in quotation marks" literals
## Terminology

The following terms are used throughout this manual:

<table>
<thead>
<tr>
<th>Term:</th>
<th>Meaning:</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>Move the cursor to the item you want and <strong>click</strong> the left mouse button once to select an item. Also, to select a menu item, <strong>middle-hold</strong> and <strong>drag</strong> the cursor to the option or feature you want, then release the mouse button to select the option.</td>
</tr>
<tr>
<td>click, left-click, right-click, and so on</td>
<td>Press the indicated mouse button once and then release it; as in, &quot;click the left mouse button on....&quot; or &quot;left-click the Execute button....&quot; Buttons, check boxes, and yes/no responses to dialog boxes are all clicked.</td>
</tr>
<tr>
<td>hold, middle-hold, left-hold, and so on</td>
<td>Press the indicated mouse button and hold it down until you’ve moved the mouse to the item you want to select.</td>
</tr>
<tr>
<td>double-click</td>
<td>Click the indicated mouse button twice in rapid succession on the same item or option.</td>
</tr>
<tr>
<td>drag</td>
<td>Move the mouse while holding down the indicated button (usually the left button). Used mostly for moving windows or selecting items on a menu.</td>
</tr>
</tbody>
</table>

For a glossary of common MasPar and industry terms, see the *MasPar System Overview* manual.

## For More Information

You may want to refer to the following manuals for more information.

<table>
<thead>
<tr>
<th>For: Background information on the MasPar MP-1 system and its architecture</th>
<th>See:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><em>MasPar System Overview.</em></td>
</tr>
<tr>
<td></td>
<td>PN 9300-0100.</td>
</tr>
<tr>
<td></td>
<td><em>MasPar Architecture Specification.</em></td>
</tr>
<tr>
<td></td>
<td>PN 9300-5001.</td>
</tr>
<tr>
<td>Information on MasPar system commands</td>
<td><em>MasPar Commands.</em></td>
</tr>
<tr>
<td></td>
<td>PN 9300-0300.</td>
</tr>
<tr>
<td>A quick overview of the most used features of MPPE.</td>
<td><em>MPPE Quick Reference.</em></td>
</tr>
<tr>
<td></td>
<td>PN 9305-0300.</td>
</tr>
</tbody>
</table>
For:

Information on MasPar's programming languages and compilers

See:

*MPL User Guide.*
ANSI C version
PN 9302-0101.

*MPL Reference Manual.*
ANSI C version
PN 9302-0001.

*MasPar Fortran User Guide*
PN 9303-0100.

Complete information on administration or ULTRIX and your DECstation

ULTRIX-32 general information,
volume 1, reader’s guide, and
master index.
Digital Equipment Corporation. 1988
Order number AA-ME82A-TE.

ULTRIX-32 software development,
volume 1, guide to VAX C.
Digital Equipment Corporation. 1988
Order number AA-ME83A-TE.

ULTRIX worksystem software
reference pages.
Digital Equipment Corporation. 1988
Order number AA-MA85A-TE.

You may also want to refer to the documentation for your window manager.

Help

To get general information about MasPar, including how to contact MasPar, type
"man maspar" at the shell prompt.

To get on-line help on MasPar commands, compilers, and library routines, type
"man -k maspar" or "apropos maspar" at the shell prompt to see what MasPar man
pages are available.

To get more general help on how to program the MasPar parallel processing system,
study the examples in $MP_PATH/examples.

If you have additional questions, or if you want to report defects or make comments on
MasPar products and documentation, you can contact MasPar via electronic mail,
telephone, or FAX at the following locations:
North America and the Pacific Rim:
Customer Support
MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale
California
94086
U.S.A.

phone: 1-800-526-0916 (8-6 PST)
FAX: 1-408-736-9560
e-mail: support@maspar.com

Europe and the United Kingdom:
Customer Support
MasPar Computer Ltd.
8 Commerce Park
Brunel Road
Theale RG7 4AB
Berkshire
England

phone: +44-734-305444 (9-5 BST)
FAX: +44-734-305505
e-mail: support@mpread.co.uk
Preface
# Table of Contents

## Preface

How To Use This Manual ................................................... i
Typographic Conventions ................................................. ii
Terminology .................................................................. iii
For More Information ...................................................... iii
Help .............................................................................. iv

## Chapter 1. Introducing MPPE 1-1

How MPPE Fits In .......................................................... 1-2
How Do You Use MPPE? ................................................... 1-3
What You Need To Use MPPE ........................................... 1-4

## Chapter 2. Starting An MPPE Session 2-1

Starting MPPE ............................................................... 2-2
Running MPPE on a Local Workstation ................................. 2-5
Starting MPPE Over the Internet ......................................... 2-5
Startup on Fault ................................................................ 2-6
Forcing a fault ................................................................... 2-6
Quitting MPPE ................................................................... 2-7

## Chapter 3. Exploring the MPPE Interface 3-1

MPPE Cursors ................................................................... 3-2
Working In MPPE Windows .............................................. 3-2
Table of Contents

Chapter 4. Opening and Executing Programs  4-1

Selecting a Program to Open  4-2
Moving Around in the Code  4-4
    Indexing into the Code  4-4
    Searching For a String  4-6
Controlling Program Execution  4-7
    Animating Your Program Execution  4-9
    Stopping Program Execution  4-10
    Interrupting and Resetting Program Execution  4-11
Using the Machine Visualizer Window  4-12

Chapter 5. Working With Breakpoints  5-1

Setting a Breakpoint at a Line  5-2
Setting a Breakpoint On a Routine  5-3
Removing Breakpoints  5-3
Showing Breakpoints in the Index Pane  5-3
    Disabling or Enabling Breakpoints  5-5
    Ignoring Breakpoints  5-5
    Setting Conditional Breakpoints  5-6
Using the Breakpoints Window  5-7

Chapter 6. Inspecting, Evaluating, and Visualizing Data  6-1

What is Inspecting?  6-2
    Magnifying Variables  6-2
How to Inspect  6-3
Using Inspector Windows  6-4
    Using the Menus  6-5
    Following Variables  6-6
    Casting Variables  6-7
Evaluating Expressions  6-8
    Legal Expressions  6-8
    Expression Errors  6-10
Appendix B. Invoking MPPE

Setting Environment Variables for MPPE ........................................ B-1
Invoking and Displaying ................................................................. B-2
Displaying MPPE .............................................................................. B-3
Invoking MPPE .................................................................................. B-4

Appendix C. Connecting to AVS

Why Use AVS? ................................................................................ C-1
Configuring the MPPE/AVS Connection ........................................... C-2
Visualizing Data With MPPE and AVS ............................................. C-2
  About the readMPPE Module ......................................................... C-3
About the Networks .......................................................................... C-4

Appendix D. Troubleshooting

Limitations ....................................................................................... D-1
Single-Statement else Blocks ......................................................... D-2
Single-Statement Loop Constructs ................................................ D-2
Control-M in Source Code Files ..................................................... D-2
Include Files .................................................................................... D-3
X Applications ................................................................................ D-3
Reporting MPPE Problems ............................................................ D-4

Index
Chapter 1
Introducing MPPE

The MasPar Programming Environment (MPPE) is a set of integrated tools in an interactive windowing environment that help you run, monitor, debug, and tune the performance of programs that run on the MasPar® system. MPPE’s graphically oriented user interface is designed to streamline your programming and debugging tasks.
Whether you are designing a new data-parallel application or are porting an existing application from a serial system, MPPE can simplify the task of designing and adapting applications for data parallel execution. The easy-to-use features and graphical user interface of MPPE support exceptionally rapid iteration in program development. You can use MPPE to optimize and debug your programs with a minimum of time, effort and recompiations.

MPPE displays basic program information conveniently and in a variety of ways. You can use MPPE to profile a program by statement or by subroutine, or display activity and data values in the PE array.

How MPPE Fits In

You use MPPE with the MasPar parallel processing system, a massively data parallel computer. Massively parallel means that the system has at least 1,000 processors acting in parallel. Data parallel means that the parallel data processors are acting in unison on a single instruction at a time but on different data points; this means that the parallelism is based on the data rather than the control structures.

The MasPar hardware consists of a front end and a data parallel unit (DPU). The front end hardware includes a workstation (usually a DECstation 5000) that runs an implementation of the UNIX operating system and communicates via standard I/O. The data parallel unit is the part of the system that performs all the parallel processing. It consists of an Array Control Unit (ACU), an array of at least 1,024 Processor Elements or PEs, and various PE communications mechanisms. The following figure shows a diagram of this system.

More detailed descriptions of the parts of the system are given in the MasPar System Overview.

The MasPar system is based on a SIMD (Single Instruction, Multiple Data) architecture. In SIMD systems, a single program instruction can be executed on many relatively small
processors simultaneously, each with different local memory. The instruction stream is processed serially, but the data processors all function simultaneously on each parallel instruction. MasPar's implementation extends this concept to ASIMD (Autonomous SIMD), where each processor is somewhat more independent; for example, on the MasPar system each PE can address a different location in memory.

Your user program (the one you want to debug) can only run on a MasPar front end and DPU combination. MPPE however, can run on a different computer that runs UNIX and has windowing capability, such as:

- a DECstation under ULTRIX and UWS.
- a Sun SPARCstation under SunOS with OpenLook.

You can run and display on both of the above systems without accessing the DPU (running what is called a "front-end only" program). You can also display MPPE's X-based graphic output on any X terminal that is running standard MIT X11 version R4 window software.

How Do You Use MPPE?

You use MPPE to understand, debug, and optimize programs that were compiled using a MasPar compiler or linker. MPPE allows you to see what is happening in your program at crucial points by providing several different ways to examine the variables, expressions, and routines in your program. MPPE can only run programs that were compiled with a MasPar compiler.

You typically use MPPE by:

1. Setting breakpoints at crucial points in your program.
2. Controlling the execution of your program and selecting items to examine.
   For example, you might decide to examine the contents of PEs, to see which ones behaved as you expect or where unexpected behavior occurred.
3. Running your program while watching changes in those items.
   For example, you can look at the values of the variables in statements to make sure they are within the ranges you expect.
Since many MPPE tools have their own windows, using MPPE may involve many windows. Chapter 3 reviews which windows go with each function or tool.

(Some MasPar programming tools (such as compilers) are accessed by entering a command at a shell prompt. See the MasPar Commands manual for information on how to use commands that are unique to the MasPar system. See your system documentation for descriptions of other shell commands.)

What You Need To Use MPPE

Before you begin using MPPE:

- You should know how to use your window manager to perform basic operations. (The OSF Motif window manager is the recommended window manager; MPPE’s own window controls conform to Motif standards and work in a Motif-like fashion even under other window managers.)

- Since MPPE’s most frequently used features are those that support debugging data parallel programs, you should be familiar with data parallel programming and at least one of the MasPar programming languages.

We recommend that you use a program that is written in MasPar Fortran or MasPar Programming Language (MPL) when you first work through the steps in this manual, since programs written in other languages (such as ULTRIX C or ULTRIX F77) can’t take advantage of all of the features of MPPE.
Many of the examples and steps shown in the illustrations in this manual are based on the file halftone, which is written in MasPar Fortran and calls an MPL module. The program is an implementation of Donald Knuth’s dot diffusion algorithm, and it converts a gray scale image to a bit-mapped (printable) image. The image is read using parallel I/O operations in MPL.

The MasPar Fortran source is in halftone.f, and the MPL source is in input.m. These source files are provided online in $SMP\_PATH/examples/halftone. Other sample files you may want to use are in the examples area in the MasPar tree ($SMP\_PATH/examples), such as diffuse and life.

- If you use a program that contains MPL code, you need to understand the MPL programming model and the concept of a program that runs on two machines. See the MPL User Guide and the MPL Reference Manual for more information.

- If you use a program that contains MasPar Fortran code, you need to understand the MasPar Fortran programming model. See the MasPar Fortran User Guide and the MasPar Fortran Reference Manual for more information.
Chapter 2

Starting An MPPE Session

This chapter shows you how to:

- Start MPPE on your system
- Set up MPPE to start automatically on a program fault
- Start MPPE over the Internet
- Quit MPPE
Starting MPPE

You can run MPPE on:

- the DECstation front end
- your own DECstation, SUN Sparcstation or X-based display (networked to the DPU)
- or remotely, on another DECstation, SUN Sparcstation, or X-based display using \texttt{rlogin}, \texttt{rsh}, \texttt{ftp}, or a similar command to log into a remote machine and start up MPPE there

The following instructions cover running MPPE on the front end workstation; for information on running MPPE on a local workstation networked to the DPU, see page 2-5; for information on using MPPE over the Internet, see page 2-5. For information on other methods of running MPPE, see Appendix B. For information on having MPPE start automatically when a program has faulted, see page 2-6. For information on having MPPE start when running a program you specified with the MP\_DBTARGET environment variable, see the Appendix B. Depending on the hardware of the remote machine, you might have to change some of the environment variables before MPPE will run correctly; see Appendix A and Appendix B for more information on the different environment variables.

MPPE uses a \textit{client-server model}. The server, started automatically for you, runs on the MasPar front end. The client is the graphical user interface—the part you see. It has windows that can run on other network-connected workstations; this means that interactive performance is maintained without any impact on the front end (the server).

To start MPPE on the DECstation front end:
1. Set your `MP_PATH` environment variable.

To make sure `MP_PATH` is set correctly, type `mppe` at the shell prompt. The message `/usr/maspar/bin/mppe` should appear on screen; if not, set your `MP_PATH` to `/usr/maspar`.

2. Make sure you have `$MP_PATH/bin` in your path.

If you are going to be using halftone to follow along with the examples in the manual, you can also change directories to `$MP_PATH/examples/halftone` and start MPPE directly from there.

3. Type `mppe &` at the system prompt.

If you don’t want to run in the background, just type `mppe`.

You see the MPPE startup screen, and then the Root window appears.

See page 3-10 in Chapter 3 for information on the Root window. See Chapter 4 for information on selecting a program to execute.

Before you start using MPPE, you should check the settings in the Connect window to make sure you are being connected to the correct machine. Click the Connect button in the Root window button bar.
By default, the Host field contains the name of the local machine, so if you’re running on a local workstation and not the DECstation front end, you’ll need to change this field to the correct name. If you try to run MPPE without the name of the MasPar front end in this field, you’ll get an error.

Select Save to have MPPE create a .mppe file, where the Connect and Customize window settings are saved. The next time you start MPPE, the saved settings in the .mppe file will be used.

To customize MPPE for your current (and future) work sessions, click the left mouse button on the Customize button in the Root Window.

When the Customize Window appears, make sure the settings fit your needs.
If you change any settings, select the **Save** button if you plan to use the same settings next time. For information on changing the settings, see Appendix B.

**Running MPPE on a Local Workstation**

You are not limited to using MPPE on an MP-1 front-end machine. You can run MPPE on a DECstation or Sun SPARCstation/Sun-4 if the workstation is connected via TCP-IP to the MP-1 (unless you are debugging a front-end only DECstation-based program). (For information on running MPPE locally and sending the display to a remote machine, see Appendix B. Appendix B also contains information on setting your DISPLAY environment variable if you want to display MPPE on an X terminal.)

Before you do the following set of steps, have your system administrator complete the steps in the *MP-1 System Administration Manual*. Then:

1. Log onto your workstation.

2. **If you're running on a DECstation**, select **Customize** from the Session Manager menu, then select **Security** from the Customize menu and type in the name of the `remoteMachine`. (This is the same as typing `xhost + remoteMachineName`.)

   **If you're running on a SUN Sparcstation**, type `xhost + localmachine`.

3. Type `mppe &` to start MPPE.

4. When the Root Window appears, select **Connect** and make sure the name in the Host field is the name of the DPU front end workstation.

**Starting MPPE Over the Internet**

You can use MPPE over the Internet. To do so, make sure the symbolic name or Internet address of the machine you want to use is in the Host field of the Connect window.
NOTES: Make sure your login name for the remote system is in the Login field. If the remote machine is on the same local network as the machine that is running MPPE, and the user IDs match (and are not the user IDs for root), MPPE requires no password.

Startup on Fault

If your program has been compiled with a MasPar compiler, you can have MPPE start up automatically if the program has a fault that would otherwise cause termination. The fault must be a DPU fault as described in the MP-1 Architecture Specification or a UNIX signal.

When the program faults, MPPE prints out the program stack and displays a list of choices for continuation. If you are running more than one instance of MPPE when your program encounters a fault, you are asked on which MPPE you want the program to appear. If you select one of the listed sessions, then MPPE takes you into the session so you can debug the program—just as if you had originally executed the program from within MPPE.

If you decide not to use one of the sessions—for example, if you don’t want to debug the program at this time—press Return.

If MPPE isn’t running when the program faults, MPPE’s "fault catcher" asks you if you want to start up an MPPE session that will connect to the faulting program.

If the program that faults is running in the background, you don’t see any messages if the program’s stdin/stdout is not connected to a terminal window. When this happens, MPPE attempts to start up the faulted program in the newest MPPE session that is running on the same machine.

In order to start up on fault, MPPE must be registered to a remote machine. If no MPPE is registered, you get the prompt No MPPE registered. Try again. You can either register or type N.

If you iconified MPPE, the new execution window appears when you de-iconify MPPE.

Forcing a fault

You can force a simulated fault with the following command:

```
kill -TRAP pid
```

where pid is the process ID of the running program.

This is useful in checking a job that has been running a long time. Your program appears in an execution window with the Execution bar marker showing where execution stopped. You can continue execution with an Execution Command, described in Chapter 4.
Quitting MPPE

To quit MPPE, move to the Root window and select Quit from the button bar.

All open windows are closed and any running processes are stopped when you quit MPPE.
Chapter 3

Exploring the MPPE Interface

MPPE runs on your workstation and uses your default X window manager. MasPar distributes and recommends OSF Motif™ as the window manager for MPPE. You need to know how to use your window manager to perform basic windowing tasks in MPPE; for more information, see the documentation that came with your window manager.

This chapter shows you:

- What the different mouse cursor shapes mean
- What the various areas of the MPPE windows are used for
- When and why you’ll use the different MPPE windows
- How to use the online Help

The mouse works in MPPE just as it does in your window manager; for a brief review of mouse terminology, see the Preface to this manual.
MPPE Cursors

When you use MPPE, the pointer changes shape to reflect the task you are doing or where the cursor is placed in the window.

<table>
<thead>
<tr>
<th>Pointer Shape is:</th>
<th>Like This:</th>
<th>You:</th>
</tr>
</thead>
<tbody>
<tr>
<td>An arrow</td>
<td></td>
<td>Can select something</td>
</tr>
<tr>
<td>The hour glass</td>
<td>☐</td>
<td>Must wait for the system to process something</td>
</tr>
<tr>
<td>The cross cursor</td>
<td>+</td>
<td>Can &quot;zero in&quot; on or magnify a variable, a profile histogram, or a PE in a Visualizer pane</td>
</tr>
<tr>
<td>A scroll-bar arrow</td>
<td></td>
<td>Can scroll in the indicated direction</td>
</tr>
<tr>
<td>The pane border arrow</td>
<td>←</td>
<td>Can move the border of this pane or window by right-holding the mouse button and dragging to the left or right.</td>
</tr>
<tr>
<td>A trash can, or the letters &quot;GC&quot;</td>
<td>☢</td>
<td>Must wait for MPPE to perform essential internal operations before you can do anything else</td>
</tr>
<tr>
<td>A caret</td>
<td></td>
<td>Can type over the text in this part of the window</td>
</tr>
</tbody>
</table>

Working In MPPE Windows

The windows in MPPE are divided into *panes* and *bars*. A pane is a rectangular region of a window bounded by thick lines. A bar is a rectangular region of a pane, and it is bounded by thin lines. Bars can appear on one or more sides of a pane.
Accessing Menus

Many of the windows in MPPE have pull-down menus associated with them. Pull-down menus are accessed by right-holding on the name of the menu:

Unavailable options on a menu are in gray, while the options you can select are in black.
In addition, many panes have context-sensitive menus associated with them that have the same name as the pane; for example, the Code pane menu is displayed from the Code pane.

To display the context sensitive menu for a pane, middle-hold on that pane. (If no menu is available, the pane flashes.)

**Quick tip:** Pressing Return when your cursor is in the Code pane re-selects the last Execution Control menu item you selected from the Execution pull-down menu or Code pane context-sensitive menu. (See page 4-7 for more information on the Execution Control menu options.)

### Revealing Hidden Text

Many MPPE windows show arrowheads in the text. The arrowheads indicate where a line has been truncated.

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>Location</th>
<th>Enabled</th>
<th>Ignore</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>halftone.fline</td>
<td>13</td>
<td>X</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>halftone.fline</td>
<td>14</td>
<td>X</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>halftone.fline</td>
<td>15</td>
<td>X</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can see the rest of the text by dragging the vertical line next to the arrowhead to the right. If you still can’t see all of the text, resize the window.

### Closing Windows

To close or quit a window, select Close from the Window menu in the upper left corner of any window. You can also press ^d (Control and d). If a process is still running, you see a message from MPPE asking if you really want to quit; select No to cancel or Yes to continue.

^d in the Stdin/Stdout window also works to quit out of the queue for the DPU. However, if you aren’t in the queue for the DPU and you type ^d (or select Close) in the Stdin/Stdout window, it’s the same as telling MPPE you wish to quit the current program. MPPE closes all associated windows, even if you are in the middle of running a program. You do not see any warning messages when you use ^d in the Stdin/Stdout window. (See page 3-12 for more information on the Stdin/Stdout window.)
The MPPE Windows

The following table shows an alphabetical summary of most of the windows in MPPE. It describes how to display the window, and what the window enables you to do.

<table>
<thead>
<tr>
<th>Window:</th>
<th>Getting and Using:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Breakpoints Window</strong></td>
<td>Select View Breakpoints from the Execution window Index pane menu. The window displays all current breakpoints and the code where they are set. Use this window to set and delete breakpoints and conditional breakpoints, and to enable, disable, or set an ignore count for a breakpoint.</td>
</tr>
<tr>
<td>[Image]</td>
<td></td>
</tr>
<tr>
<td><strong>Connect Window</strong></td>
<td>Select the Connect button from the Root Window button bar. Use this window to connect to a different host MasPar machine, or to display the status of the queue waiting for the DPU.</td>
</tr>
<tr>
<td>[Image]</td>
<td></td>
</tr>
<tr>
<td><strong>Evaluation Window</strong></td>
<td>Highlight a variable or expression in the Code pane of the Execution window, then select Evaluate from the Inspection menu. This window displays the variable or expression in a field where you can edit it. Select OK to show the variable or evaluated expression in an Inspector window.</td>
</tr>
<tr>
<td>[Image]</td>
<td></td>
</tr>
</tbody>
</table>
### Window:

#### Global Variables Window

Select **View Global**s from the Execution window Index pane menu. This window displays the name of each global variable in your program, and the code where the global variable is defined.

![Global Variables Window](image)

#### History Window

Select **View History** from the Execution window Index pane menu. This window displays the values of any number of variables and DPU resources over time. See Chapter 8.

![History Window](image)
**Window:**

<table>
<thead>
<tr>
<th>Inspector Window</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Inspector Window" /></td>
</tr>
</tbody>
</table>

**Getting and Using:**

You *evaluate* a variable or expression to see an Inspector window; see the previous table entry for the Evaluation window for more information. An Inspector window displays the name, type, address, and value of a data symbol, or the result of an expression. Inspector windows also display the values of aggregate and plural variables in tabular form. You can also use these windows to display slices (subsets of columns and rows) of MasPar Fortran arrays. You can have multiple Inspector windows open in each MPPE session. (See Chapter 6.)

**Machine Visualizer Window**

![Machine Visualizer Window](image)

Select *Visualize* from the Machine menu of the Execution Window. This window graphically shows you how well your program is using the PE array.

**"Memory" Windows**

![Memory windows](image)

Select *View Memory* from the Execution window Index pane menu, then select one of the following: Registers, Memory, ACU Registers, ACU Special Registers, ACU Memory, PE Registers, and PE Memory. These windows show how your program is using available memory.
### Window:

#### Routine Profiles Window

Select View Routines from the Profiles bar menu in the Execution Window. This window displays the number and percentage of ticks that each subroutine in your program has accumulated. If a subroutine is not listed, it has not accumulated any ticks. See Chapter 7.

### Source Files Window

Select View Files from the Execution window Index pane menu. This window displays a list of all the source files in your program.
Window:

**Statement Profiles Window**

Select View Statements from the Profiles bar menu in the Execution Window. Displays the number and percentage of ticks that each statement in your program has accumulated. If a statement is not listed, it has not accumulated any ticks. See Chapter 7.

---

**Static Routine Profiles Window**

Select View Static Routines from the Profiles bar menu in the Execution window. This window displays the weights accumulated for each routine in the program that has compile-time profiling information (see Chapter 7).
**Window:**

**Static Statement Profiles Windows**

Select View Static Statements from the Profiles bar menu in the Execution window. This window displays the weights accumulated for each line in the program that has compile-time profiling information (see Chapter 7).

**Visualizer Window**

In the Execution window, select Visualize from the Inspection menu, or evaluate an expression or variable and then select Visualize from the Inspect menu of an Inspector window. A Visualizer window graphically displays the value of each instance of a 2-D array variable, a plural variable, or a 2-D slice of an array, relative to some constant that you supply. (See Chapter 6.)

---

**The MPPE Root Window**

When you start MPPE, the Root window appears in the lower right corner of the screen. You use the Root window to invoke your program and to control and customize MPPE. This window also displays messages that come from MPPE. You can only close the Root window by quitting MPPE; however, the window can be iconified.
The **Error/Status** pane displays system status and error messages. It displays information messages when your program execution stops at a breakpoint or after a Go to, for example, and it displays an error message if you try to inspect a variable that is out of scope or to go to a line that is not an executable line.

The **File List** pane shows the executable files in the current directory. Middle-hold to see a menu to change directories or display the current pathname in the Error Status pane.

**NOTE:** If you are running MPPE on a Sun workstation with SUN Automounter, and you change the directory using the menu in the File List pane, you must also change the Directory setting in the Connect window (see page 2-3).

The **Root** window has six buttons.

**Select this button:** **To:**

- **Open** open a Stdin/Stdout window and an Execution window containing the selected program. (If no executable file is highlighted, you are prompted for a filename.) See Chapter 4 for step by step instructions on how to select a program and use the Execution window.

- **Customize** open the Customize window. For information on this window, see Appendix A.

- **Front** move all MPPE windows to the front, relative to all other active windows. (**Note:** This option may not be supported by your window manager.)

- **Connect** display the Connect window. (The status information shown in this window is the same as what you get when you type mp on at the shell.) For information on this window, see page 2-3.

- **Help** access the on-line Help. See page 3-14 for more information.

- **Quit** quit your MPPE session. See page 2-7 for more information.
Quick Tip: If MPPE behaves unexpectedly at any time, check the Error/Status pane in the Root window for messages from MPPE.

The Stdin/Stdout Window

The Stdin/Stdout window is displayed whenever you select the Open button to run a user program under MPPE.

The Stdin/Stdout window is a terminal emulation window. It shows all output generated by your program (stdout), and it’s where you enter any input that your program expects to receive from stdin. However, it is not a “shell” window—you cannot enter UNIX shell commands at this window.

Selecting Close from the Window menu (or typing ^c or ^d) in the Stdin/Stdout window is equivalent to selecting Quit from the Machine menu. When you close the Stdin/Stdout window, all the other windows associated with that particular user program (except for the Root) automatically close, even if you are in the middle of running your program. You do not get any notifiers or warning messages when you do this.

Note: Don’t resize this window—you may lose some of the text.

The Execution Window

You use the Execution window to execute your program and select what program data you want to look at.
The Code pane shows the code for the item highlighted in the Index pane. The Code pane is read-only; you can't edit your code from within MPPE.

Use the Execution window to run your program and examine the results. Chapters 4 through 8 tell you how to execute your program. The following table tells you how to move around in the Execution Window.

<table>
<thead>
<tr>
<th>To:</th>
<th>Do this:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display a list of all currently active routines in the Index pane (this is the default)</td>
<td>Left-click on Stack frames (see page 4-4)</td>
</tr>
<tr>
<td>Display a list of Source files for this program</td>
<td>Left-click on Source files (see page 4-5)</td>
</tr>
<tr>
<td>Display any currently set Breakpoints</td>
<td>Left-click on Breakpoints (see page 4-5)</td>
</tr>
<tr>
<td>Display any Global MPL Variables</td>
<td>Left-click on Global Variables (see page 4-6)</td>
</tr>
<tr>
<td>Select an option from a menu</td>
<td>Middle-hold in the Index pane, the Code pane, or the Machine pane, and highlight the item you want</td>
</tr>
<tr>
<td>Display a different part of the code in the Code pane</td>
<td>Scroll using the scroll bar, or left-click on a item in the index pane</td>
</tr>
</tbody>
</table>
About the Execution Monitor Icons

The icons in the Machine pane tell you whether the front-end machine and the DPU are running or stopped.

Stopped

Running

The Help Window

To get help, select the Help button in the Root window. The MPPE Help window appears.

The Help window displays basic information about MPPE—initially it displays information about the Root window.

It’s easy to find the Help information you want:

You can:

Scroll through the Help text

Display information about a given topic

By:

Dragging the mouse past the top or bottom of a pane, or by using the scroll bars to scroll through either the Help text or the topic list.

Left-clicking on a topic in the Topic List pane.
Search for a string in the Help text

Middle-holding (or middle-double-clicking) in the Help Text pane and typing a word to find. Or, to find the next instance of a word, select it in the Help Text pane, then middle hold. To repeat a search, press Return. To deselect a word, left-click the highlighted word.

When you are finished using the Help window, select Close from the Window menu to close the window.
Exploring the MPPE Interface
In MPPE, once you’ve opened a program in an Execution window, you can execute it to debug and to monitor the behavior and performance of variables and routines. By monitoring what changes and how the program behaves, you can then fine-tune your program to make it run more efficiently. MPPE lets you execute an entire program, or just portions of a program.

This chapter shows you how to:

- Open a program so you can execute, monitor, or debug it
- Move around in the program code
- Control the execution of the program
- Interrupt or reset program execution
- Visualize the PE Array
Selecting a Program to Open

To open a program so you can execute, monitor, or debug it:

1. In the File List pane of the Root window, left-click the name of the program you want.

If you need to change directories, left-click on the name of the directory, or middle-hold in the File List pane, select Go To . . ., and type a new path in the dialog box.

2. Select the Open button.

A message appears in the Error/Status pane informing you that MPPE is starting a session for the file you selected, and the Stdin/Stdout window appears.

(mppeback is the name of the server portion of the MPPE client-server model. mppeback is the portion of the product that runs on the MasPar front end—unless the program is a front-end only non-parallel program.)

The Stdin/Stdout window displays messages about being queued for the DPU; when you have the DPU, the Execution window appears. If another job is already using the DPU, you may notice a few moments delay. (You can check your status in the DPU queue by using the Status button in the Root window.)
The Execution window is the main working window of MPPE. It displays the code for the program you selected. You use this window to set breakpoints, execute the program, and open most of the other windows you use to analyze and debug your program. More information on the Execution window appears in Chapter 3.

The Bar marker shows where execution has stopped. The lines above the bar marker have been completely executed. If you haven’t started executing the program, the Bar marker is at the first line of the program, as shown in the above illustration. (On a fault, the Bar marker proceeds to the line that causes the fault—but execution may be part way into the subsequent statement.)

To quit the Execution window, select Quit from the Machine pull-down menu.

MPPE asks if you want to save the session for the program shown in the execution window. Select Yes or press Return to quit and save the current open windows, breakpoints, profiling information, and so on, related to the current session. DPU resources are released, and the Execution and Stdin/Stdout windows disappear. The next time you open this program, MPPE uses the saved log file to recreate the session up to the point where you quit, so all currently open windows reappear.

Select No to quit and have MPPE discard all the current session info. The next time you open this program, it will appear just as it did when you originally opened it, with no breakpoints set and only the Execution window open.

Select Cancel to not quit and return to the current MPPE session.

NOTE: If you change the executable file after saving the session, you need to be aware that the changes affect what happens when MPPE re-opens the file. For example, if you set a breakpoint and then add lines before the line the
breakpoint is on, the breakpoint will appear at the same line number rather than staying with the same line of code. In the same way, if you delete a variable that you were inspecting in the last MPPE session, you get an error message from MPPE regarding the "missing" variable.

Quick Tip: Sessions are saved to the log file 
.mppe-session-<executablename>-<yourname>. If you want, you can cp that file to another name, so that you can start from the same place more than once. However, in order for MPPE to find the log file, you need to copy the duplicate file back to 
.mppe-session-<executablename>-<yourname>.

Moving Around in the Code

You can move around in your code without executing it. This enables you to review different parts of your code, select a line or routine, set a breakpoint, or inspect a variable.

You can move to different parts of your code by:

- scrolling the Code pane using the scroll bar, or by dragging the mouse so the pointer moves past the bottom or top of the pane.
- indexing into your code by making a selection from the Index pane in the Execution window (see the following section for more information).
- searching for a string (see page 4-6 for more information).

You can move to any part of your code using some combination of these methods.

Indexing into the Code

Indexing is one of the ways MPPE provides to help you move through your code. Indexing divides a program's text into manageable pieces that you can then browse or search through. Indexing uses four different keys to divide the program's text:

- subroutine calls
- breakpoints
- files
- global variables

The Index pane on the Execution window displays a different list or index depending on which of the four buttons you select from the top of the Execution window. When you select (highlight) any of the items in the Index pane, the code for that item is then shown in the code pane.

To index with a subroutine call:
1. Left-click the Stack frames button.

The Index pane displays a list of subroutine calls. The routine at the top of the list is the one that was most recently called.

```
- Stack frames | > Source files | > Breakpoints | > Global variables

DIFFUSEDOTS (halftone.f:45)
DOTDIFFUSION (halftone.f:14)
```

2. Left-click one of the subroutine calls in the Index pane.

For example, you might select DOTDIFFUSION. The Code pane displays that section of the code.

**NOTE:** If the name of the routine includes question marks instead of the filename line number, then no debug information is available for that routine and its source file cannot be viewed. This is sometimes true of low-level library routines, or the question marks may appear because you interrupted execution of your program.

**To index into a source file:**

1. Left-click the Source Files button.

The Index pane displays a list of source files.

```
- Stack frames | > Source files | > Breakpoints | > Global variables

halftone.f
input.m
```

2. Left-click a file name to display that program in the Code pane. For example, if you select halftone.f, the Code pane shows a different selection of code than if you select input.m.

You can see this same information by showing the Source Files window. Select View Files from the Index menu.

**If you’ve set breakpoints in your code, you can index by breakpoints:**

1. Left-click the Breakpoints button.

The Index pane displays a list of breakpoints.
2. Left-click one of the breakpoints shown in the table.

The Code pane displays the section of the code with the breakpoint in it. (For information on using the table shown in the Index pane, see page 5-3.)

You can see this same information by showing the Breakpoints window. Select View Breakpoints from the Index menu.

If your program is stopped in an MPL routine (or in front end C code), you can see the global variables. To index by global variables:

1. Left-click the Global Variables button.

The Index pane displays a list of global variables.

2. Left-click a variable.

The Code pane displays the portion of MPL code containing the definition of that variable.

You can see this same information by showing the Global Variables window. Select View Globals from the Index menu.

Searching For a String

There are three ways to search for a string:

- Left-double-click on the value, word, or name you want, then select Find from the Code menu
- Highlight the entire string (left-hold and drag the mouse), then select Find from the Code menu
- Select Find from the Code menu, and type a string at the prompt

When MPPE finds the string, it is highlighted and the arrow marker is placed at that line in the Breakpoints bar. If the string isn't found, MPPE displays the message Cannot find <string> in the Error/Status pane of the Root window.
To find the next occurrence of a string, press Return. To deselect the current string, left-click on a blank space or select a new string to find.

NOTE: Find is case-sensitive, and also can’t find phrases that are wrapped around the end of a line.

## Controlling Program Execution

Once you have selected a program to debug or monitor, you start the execution of your program by selecting one of the first five options on the Execution menu.

<table>
<thead>
<tr>
<th>Execution</th>
<th>Inspect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continue</td>
<td>Skip</td>
</tr>
<tr>
<td>Step</td>
<td>Go to Line</td>
</tr>
<tr>
<td></td>
<td>Go to Routine</td>
</tr>
<tr>
<td></td>
<td>Break on Routine</td>
</tr>
</tbody>
</table>

MPPE has five commands that control program execution. These functions enable you to control how execution moves from one statement to another, as opposed to using breakpoints (discussed in Chapter 5) which only tell MPPE where to stop.

**Use:**

<table>
<thead>
<tr>
<th>Command</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continue</td>
<td>resume execution of your program from the point where it last stopped, or start it if it hasn’t yet begun to execute. If you haven’t yet set any breakpoints, your entire program is executed.</td>
</tr>
<tr>
<td>Skip</td>
<td>execute up to the next executable source line in the Code pane. If the current source line is a routine call, the entire routine is executed (unless you have set a breakpoint inside the routine) and execution stops at the next executable source line after the routine call. (If you are familiar with dbx or one of its derivatives, Skip is the same as the Next function.</td>
</tr>
<tr>
<td>Step</td>
<td>execute up to the next executable source line; if the next line is a subroutine call, go to its first executable line. If the current source line is not a routine call or is a call to a routine for which no debug information is available, Step is identical to Skip.</td>
</tr>
</tbody>
</table>
Go To Line

continue execution of your program up to the selected line (which is pointed to by the arrow marker in the Breakpoints bar).

Go To Line works as a temporary breakpoint. If execution stops before the line you selected (for example, if you've set a breakpoint or there is an interrupt) and you still want to go to that line, select Go to Line again. If the selected line never gets executed, the program runs until termination, a breakpoint, an interrupt, or a fault or signal is reached.

NOTE: Before you select this option, you must select an executable line in the Code pane; left-click anywhere on the line to select it.

Go To Routine

continue execution of your program up to the routine whose name is highlighted in the Code pane. Unless execution stops earlier for some other reason (such as a breakpoint or an interrupt), execution stops at:

- the routine declaration statement in DEC Fortran or MasPar Fortran.
- the first open brace after the routine declaration in DEC C.
- the first executable source line in the selected routine in MPL.

NOTES:

- Before you select this option, you must select a routine name; left-click anywhere on the name to select it. The routine doesn't have to be in an executable part of the program—it can be anywhere, even inside a comment.

- If your program contains two definitions with the same routine name (one in MPL and a separate one in ULTRIX C, for example), MPPE asks you to choose one of the routine definitions.
• Execution stops in the selected routine (assuming it does not stop earlier) the next time that routine is called, even if the call you have selected in the Code pane is not the call. Thus, even if execution stops at the routine you selected, it may be stopped at a different place in the program than you expected. You should check the call stack (the list of routines in the Stack pane) to see where the routine was actually called from.

Note that you need to select a place to stop before using Go to Line or Go to Routine. You do not need to select a place to stop before using Continue, Skip, or Step, but you may want to set at least one breakpoint before using Continue. To select a line or routine, left-click on a routine name (for example, EnhanceEdges), or left-hold and drag to highlight an entire line.

If you have not yet begun to execute your program when you select one of these options, you will be asked to enter any command line arguments your program is expecting. You can also use the pop-up to control stdin/stdout redirection. (For example, you could type < mydata.file to have MPPE use data from a file named mydata.file. You must list all command-line program arguments first, before redirecting stdin/stdout. See the csh man page for more information on using >, <, >>, or <<.)

Quick Tip: Use Step or Skip in combination with the Animate feature to monitor your variables from line to line without touching your keyboard or mouse. Or, combine these control options with the History window to see a record of how your variables change without keyboard or mouse input. (See the following section for more information on Animate; see Chapter 8 for more information on History.)

Quick Tip: Pressing Return with your cursor in the Code pane re-selects the last Execution Control menu item you selected from the Execution pull-down menu or Code pane context-sensitive menu. (See page 4-7 for more information on the Execution Control menu options.)

Animating Your Program Execution

You use the Animate button to have MPPE repeat a selected control operation. This lets you examine the behavior of a program without continuously re-selecting the same control operation.

To start animation:

1. In the Execution window, left-click the Animate button.

The Animate button is highlighted.
2. Select any execution control operation from the Execution menu (except for Break on Routine).

MPPE repeatedly performs the selected control operation, such as Step or Skip. MPPE pauses briefly after each iteration of the animated operation and updates all windows, including Inspector and History windows (see Chapter 6 and Chapter 8).

To stop animation, left-click the Animate button in the Machine pane at any time Animate is running. You cannot close or move any windows or perform any other MPPE operations while in animation mode.

Animation proceeds until you turn off the animation mode or your program:

- reaches a breakpoint (unless you've selected the Continue control operation)
- faults
- terminates

For example, if you select Animate and then Step, MPPE steps through your program showing its progress, but stops if it reaches a breakpoint. If you select Go To Routine, MPPE pauses each time the routine is called or a breakpoint is reached. However, if you select Continue, your program executes to the end, even if you have breakpoints set, but it pauses at each breakpoint.

Quick Tip: One useful technique is to set breakpoints (see Chapter 5) at spots where you'd like to look at some variables and select Animate and then Continue. When your program hits a breakpoint, MPPE will pause to update the Data Inspector and History windows so you can see what changed in your variables or resources.

Quick Tip: The more windows you have open when you are animating, the longer the pauses in the animation while the windows are being updated. Therefore, before you use animation, close any windows that you are not interested in watching during the animation.

Stopping Program Execution

Execution of your program will stop if:

- It reaches a breakpoint (see Chapter 5) that is enabled and has an ignore count of 0.
• It reaches a conditional breakpoint that is enabled and the condition evaluates to non-zero.
• There is a fault or signal.
• You interrupt it (using Interrupt on the Machine menu).
• You left-click the Animate button when Animate has started (see the previous section).
• You select Go To Line or Go To Routine.
• You select Step or Skip and the program reaches the beginning of a source statement.
• The end of the program is reached.
• You select Quit from the Machine menu.

When the end of the program is reached, all inspectors and visualizers are grayed out, and and the History window shows the message "Out of scope" for any variables. If you execute the program again, once the variables are back in scope, the visualizers, inspector windows, and history window are are reactivated and are no longer gray.

Interrupting and Resetting Program Execution

The Machine menu at the top of the Execution window has a menu of options for quitting, visualizing machine use, and interrupting and resetting your program execution.

<table>
<thead>
<tr>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
</tr>
<tr>
<td>Quit</td>
</tr>
<tr>
<td>[Interrupt]</td>
</tr>
<tr>
<td>Visualize</td>
</tr>
</tbody>
</table>

See page 2-7 for information on quitting MPPE, and the following section for information on visualizing machine use. This section describes interrupting and resetting program execution.

If you select Reset, MPPE goes back to the beginning of the program, but does not begin executing the program. Execution of your program stops and the bar marker moves back to before the first line of the program. Selecting Reset returns you to the state you were in when the Execution window first appears.

If you select Interrupt, program execution is stopped at the line or in the routine that MPPE is currently executing. You can only select this option when then machine icons show that a program is running. When the program stops, MPPE looks and acts the same as when the program is stopped at a breakpoint.
Using the Machine Visualizer Window

Once you have begun execution of an MPL program, one window you might find useful is the Machine Visualizer Window. This window graphically represents the current state of the DPU. It also helps you understand how your program is using the PE array.

To view the Machine Visualizer Window, move to the Machine pane in the Execution window and middle-hold for the menu, then select Visualize.

The default size of the window is proportional to the size of the PE array. Each PE is represented by one pixel, and there is one blank separation pixel between each pair of pixels that represent PEs. You can resize the window if it’s hard to see the entire display.

If you press the left mouse button while the cursor is in the window, a magnifier appears that gives the processor ID of the PE shown in the center of the cross-shaped cursor. The location is the row number followed by the column number of this PE.

The default display in the Machine Visualizer window is the active set of PEs (shown as ebit in the menu below). You can display other PE states by selecting different options from the Machine Visualizer menu:
Select:

>`ebit`  |  To:
   |  Show which PEs are currently in the active set. The pixels that are on represent the PEs that are currently enabled. See the *MPL Reference Manual* for a definition of the active set.

>`mbit`  |  Show which PEs are currently participating in a memory instruction.

>`rflag`  |  Show which PEs have just received a communication. The values that this option displays will be the same for every MPL statement. This option is meaningful if you are running MPAS (MasPar Assembly Language) code or if you use MPL low-level routing library calls. See the *MasPar Architecture Specification* manual for more information on `rflag`.

>`tflag`  |  Show which PEs are trying to transmit a communication. The values that this option displays will be the same for every MPL statement. This option is meaningful if you are running MPAS (MasPar Assembly Language) code or if you use MPL low-level routing library calls.

All other flags are described in the *MasPar Architecture Specification* manual.

The label at the top of the Machine Visualizer window changes to reflect the PE state being displayed.
When your program is executing, and a breakpoint is reached, MPPE stops at the line or routine with the breakpoint. When your program has stopped, you can use the inspection and visualization features to look at the state of your program and make sure it is behaving as you wish. You use breakpoints and the execution control commands described in Chapter 4 together to control the execution of your program.

This chapter describes how to use breakpoints by:

- setting a breakpoint at a line
- setting a breakpoint at a routine
- removing breakpoints
- disabling or enabling a selected breakpoint
- disabling or enabling all breakpoints
- having the MPPE ignore a selected breakpoint a specified number of times
- setting up conditional breakpoints
Setting a Breakpoint at a Line

Set a breakpoint on a line to tell MPPE to stop at that point, so you can examine the code or the changes to variables in the program. (Setting a breakpoint on a routine is explained on page 5-3, setting conditional breakpoints is explained on page 5-6.) To set a breakpoint at a line:

1. In the Code pane of the Execution window, move to the part of the code where you want to set a breakpoint.

   You can use the scroll bars or drag the mouse to page through the code.

2. Left-click in the Breakpoints bar next to the line where you want MPPE to stop.

   If you now select Continue from the Execution menu, the program executes up to the line where you set the breakpoint.

   ![Breakpoint set in Execution window](image.png)

When you left-click in the Breakpoints bar, a breakpoint marker (●) appears at that line. If that line is already selected and has an arrow marker in the Breakpoints bar, MPPE shows a combined breakpoint and arrow marker: ●. If you try to set a breakpoint on a line that is not an executable line, an error message appears in the Error/Status pane of the Root window. If you want to set a breakpoint on a multiple-line statement, you must put it on the last physical line of the statement.

**NOTE:** In DEC C, program execution might not stop where you expect it to if you set a line breakpoint at a single-statement `else` block or after a single-statement loop construct (see Appendix D).
Setting a Breakpoint On a Routine

You set a breakpoint on a routine (instead of on a line) when you want your program to stop at the call to that routine, without going and finding the routine.

To set a breakpoint on a routine:
1. Move to a line that contains the name of the routine.
2. Left-click anywhere in the name of that routine.
   For example, in halftone, you might place a breakpoint on DiffuseDots. You can select any occurrence of the routine name—even inside a comment. The routine name is highlighted, and an arrow marker is displayed on the selected line in the Breakpoints bar.
3. Select Break On Routine from the Execution menu.
   MPPE places a breakpoint marker in the Breakpoints bar next to the first executable source line in that routine (in MPL), at the routine declaration statement (in Fortran), or at the first open brace after the routine declaration (in Ultrix C).

If what you selected is not the name of a routine in your program, you see an error message in the Error/Status pane.

If your program contains two definitions of the same routine name (one in MPL and a separate one in ULTRIX C, for example), MPPE displays a menu so that you can choose one of the routine definitions.

Removing Breakpoints

To remove a breakpoint at a routine or a line:
- Left-click the breakpoint marker,
- Or-
- Select the routine or line in the Breakpoints List, then select Remove from the Index pane menu.

Showing Breakpoints in the Index Pane

To see a list of all the breakpoints set in your program, click the Breakpoints button in the Execution window. The Index pane now shows a table that lists all of the breakpoints in your program, and the Code pane displays the code for the highlighted breakpoint in the table. The Code pane is empty if no Breakpoint is highlighted; when you first show this Index, a breakpoint is only highlighted if program execution is stopped at a line that contains a breakpoint.
If you set or remove a breakpoint in the Code pane, the list of breakpoints in the Index pane is updated to reflect the change, and vice-versa.

To view the code for a breakpoint in the Code pane, left-click anywhere in a row of the table; the Code pane then displays the code for that row.

To see the context-sensitive menu for the Breakpoints List, middle hold in the Index pane:

View Files
View Breakpoints
View Globals
View History

Remove
Set Ignore Count
Set Condition
Enable
Disable

Enable All
Disable All
Disabling or Enabling Breakpoints

When you set a breakpoint, it is enabled by default. However, there may be times when you want to disable a breakpoint but keep the breakpoint as a placeholder. For example, you may have set several breakpoints and want to disable one of them for this execution of a program—instead of deleting it, running the program, and then re-setting it, you can quickly disable and re-enable it using the table in the Index pane.

To disable a specific breakpoint, select that box in the Enable column. Middle-hold for the menu and select Disable.

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>Location</th>
<th>Enabled</th>
<th>Ignore</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>halftone.fline</td>
<td>14</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>halftone.fline</td>
<td>15</td>
<td>X</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>halftone.fline</td>
<td>25</td>
<td>X</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Then, to re-enable it, select the box again and select Enable from the menu.

To disable all the breakpoints set in your program, middle-hold and select Disable All. To re-enable all breakpoints, select Enable All.

Ignoring Breakpoints

You can have MPPE ignore a breakpoint for a specified number of times. This is useful, for example, in enabling you to look at a loop after several iterations without having to step through the loop manually—and without having to disable the breakpoints.

To ignore a breakpoint:

1. Select the breakpoint in the Index pane.

2. Select Ignore Count from the Index menu. A Notifier box prompts you for the number of times you want the breakpoint ignored.

The ignore count number you entered appears in the Ignore Count field of the breakpoint you selected. (You can also left-click in the Ignore Count column and type a number for the breakpoint you want to ignore.)

When you execute the program, the ignore count number decreases by one each time the breakpoint is reached. When the ignore count number reaches zero, MPPE stops at the breakpoint and your ignore count is reset.

When the program terminates, or when the breakpoint stops execution, the ignore count number is restored to the last number you set. To reset the ignore count to 0 and stop ignoring the breakpoint, select the Ignore Count box and type a 0.
Setting Conditional Breakpoints

Expressions can be linked to a breakpoint to make it a conditional breakpoint. When a conditional breakpoint is reached during program execution the expression is evaluated; if the value of the expression is non-zero or not "FALSE." in Fortran logical expressions (and the breakpoint is enabled and has no ignore count), the program is stopped. Otherwise, the breakpoint is ignored. Breakpoint expressions must result in singular scalar values.

To set a conditional breakpoint:

1. In the Execution window with the Stack Frames showing in the Index pane, click in the Breakpoints bar on the line where you want to set a conditional breakpoint.

   The new breakpoint is added to the list in the Index pane.

2. Move to the Index pane, click the Breakpoints button, and select the Condition column for the new breakpoint.

3. You can either:
   - Type an expression in the Condition box and press Return
     -Or-
   - Select Set Condition from the Index pane menu, and type the expression in the pop-up.

4. Return to the Code pane, and execute your program. If the expression evaluates to non-zero (true), then the program execution halts when it reaches that expression. Otherwise, program execution continues as normal.

To remove a conditional breakpoint, select the Condition column and press Del and Return.

NOTE: If you set a conditional breakpoint and have an Ignore Count set, the breakpoint will be ignored for the number of times to which the count is set, and the expression will not be evaluated until the Ignore Count reaches zero.

Quick Tip: Checking the ignore count is much faster for MPPE than evaluation of a conditional breakpoints’ expression. If your run-time speed is a concern, you should use ignore counts instead of conditional breakpoints.
Using the Breakpoints Window

If you want to have the Breakpoints list pane available at all times, but need to show a different index in the Execution window, display a Breakpoints window. Select View Breakpoints from the Index pane menu.

```
program DotDiffusion
  c
  c Implementation of error diffusion halftoning from "Digital Halftones
  c by Dot Diffusion," by Donald Knuth in ACM Transactions on Graphics
  c vol. 6, no. 4, October 1987, pp. 245-273.
  c
  parameter (imageX=64, imageY=64)
  dimension A(imageX,imageY), B(imageX,imageY)
  A = 0.0
  B = 0.0
  call Loadimage(A,imageX,imageY)
  call EnhanceEdges(A,imageX,imageY)
  call DiffuseDots(A,imageX,imageY,B)
  stop
end
subroutine Loadimage(A,X,Y)
  integer :: X,Y
  real, array(X,Y) :: A
```

This window looks just like the Execution window except the Machine pane is missing and the Profile bar is hidden. Use this window to set and remove breakpoints, and to disable or ignore breakpoints.
Chapter 6

Inspecting, Evaluating, and Visualizing Data

This chapter shows you how to:

- Select a variable or expression to inspect; MPPE displays it in an Evaluation window for you to edit, and then displays its value in an Inspector Window
- Visualize an evaluated variable, plural variable, or expression

NOTE: MPPE also provides a link to a standard scientific visualization package called AVS, which you may want to use in addition to MPPE's built-in visualizing abilities. AVS provides general and powerful visualization support. See Appendix C for instructions on using AVS with MPPE.
What is Inspecting?

In MPPE, the concept of inspecting (finding out more about a variable or expression) involves several windows: the Execution window, where you select an item; the Evaluation window, where you accept or change the item being evaluated; and the Inspection window, where you see detailed information about a variable or the results of an evaluated expression. You can also use the Visualizer window to see some types of information in a graphical display.

You: Evaluate expressions Inspect variables Visualize
To: See the result of an expression involving constants and program variables. See the type, address, and value of a particular variable (including Global variables). See a graphic representation of the values of a two-dimensional array variable or an MPL plural variable compared to a constant that you specify. You can visualize any 2D array.

You can also inspect a variable in the Code pane—this is called magnifying (see the following section).

When an Inspector window is created, its scope is remembered. If the window goes out of scope due to program execution, it will be grayed out. When it comes back into scope, its contents will be updated.

Magnifying Variables

Most variables, functions, and subroutines may be magnified. Exceptions include functions and subroutines compiled without debug information (such as library functions).

To magnify variables:

1. In the Code pane of the Execution window, move to the variable you want to magnify.

   For example, for halftone, you might inspect ixproc in the input.m source file.

2. Left-hold on the variable identifier.

   A small box pops up:

   ```
   plural int ixproc
dpu_crt0.S:13
   0...
   ```
If the last line in the magnifier shows an ellipsis (such as \(0 \ldots\)), then the variable is a plural variable. To see all of the values, Inspect the variable.

**How to Inspect**

The steps for inspecting and evaluating are the same. (You can even use this same sequence to Visualize by adding a final step; see page 6-12.) To evaluate an expression, your program must have started executing and have stopped.

To inspect, evaluate, or visualize an item in the code pane:

1. Press the left mouse button and click or drag to highlight the variable or expression you want to inspect.

   You can select a variable identifier anywhere (even in a comment) as long as that variable is in scope. Also, there is no limit to how large of an expression you can select. However, if you select a Fortran expression that is more than one line long, you will need to delete the line-continuation characters before MPPE can evaluate the expression correctly.

2. Select Evaluate from the Inspection menu.

   ![Expression in LOADIMAGE()](image)

   When the Evaluation window appears, you can edit the item you selected by typing over it. (You can also select Evaluate without having an expression or variable selected. When the Evaluation window appears, type in the expression you want to evaluate. The expression you type must follow the rules for legal expressions shown on page 6-8.)

3. Select OK to close the Evaluation window and show the variable or the results of the expression in an Inspector window. (You can also select Cancel to return to the code pane without inspecting and without saving any changes you made in the Evaluation window.)

The following figure shows two Data Inspector windows. One window shows a built-in type variable, and the other shows a MasPar Fortran array variable.
NOTE: You can also display an Inspector window by selecting Inspect, Inspect Plural, or Follow from the Inspect menu on a Data Inspector window, or from the Index pane menu in a Global Variables window.

Using Inspector Windows

An Inspector window has a minimum of two panes: a Name pane at the top and a Value pane below the Name pane. Every time execution stops, the Value pane in each Data Inspector window is updated with current information (if its contents are in scope). If the variable you are inspecting is a MasPar Fortran array, then the Data Inspector window has a third pane: a Slice pane is shown between the Name pane and the Value pane.

The first line of text in the Name pane shows the type of variable you are inspecting, followed by its name. The second line in the Name pane shows the address space where this variable resides, followed by the address of this variable within that address space.

The address space descriptor is one of:

- FEMEM Front end memory
- FEREG Front end register
- ACUMEM ACU memory
- ACUREG ACU register
- PEMEM PE memory
- Pereg PE register

The address space descriptor is updated dynamically if the variable is moved between register space and memory due to compiler optimizations.

Every time execution stops, the Value pane in each Data Inspector window is updated with current information.
You can change the value of a variable in a Value pane. When you edit a value, the new value is used when the program continues executing, just as if that new value had been assigned inside the program.

If the variable is a pointer type, the value displayed is the address held by the pointer (the absolute address of the object the pointer points to).

If the variable is an aggregate type, the value of each element of the aggregate is displayed. If the variable is a plural variable, the value of the variable in each PE is displayed. To focus on just one element of an aggregate or the value of one PE of a plural, select the element, member, or PE in the Value pane, and then select Inspect or Inspect Plural from the Value menu.

Tables in the Value pane have two columns.

<table>
<thead>
<tr>
<th>If:</th>
<th>Then:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The variable in the Name pane is an array</td>
<td>The first column in the Value pane is labeled Index and the table contains one row for each element of the array.</td>
</tr>
<tr>
<td>The variable is a structure</td>
<td>The first column in the table is labeled Member and the table contains one row for each member of the structure.</td>
</tr>
<tr>
<td>The variable is a plural variable</td>
<td>the first column in the table is labeled PE and the table contains one row for each PE in the PE array. (See the following section for more information.)</td>
</tr>
<tr>
<td>The array element, structure member, or PE variable is itself an aggregate or plural type variable</td>
<td>The second column in the Value pane is labeled Value... and only the first value in the aggregate is listed in the table. Otherwise, the second column is labeled Value.</td>
</tr>
<tr>
<td>The variable is a pointer type</td>
<td>The Value pane displays the address held by the pointer (the absolute address of the object the pointer points to).</td>
</tr>
</tbody>
</table>

**Using the Menus**

The Inspect menu at the top of an Inspector window is used to inspect, follow, cast, or visualize the element, member, or PE variable selected in the Value pane, or the variable shown in the Name pane.

<table>
<thead>
<tr>
<th>Select:</th>
<th>To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspect</td>
<td>Display a Data Inspector window focused on the selected variable.</td>
</tr>
<tr>
<td>Inspect Plural</td>
<td>Display a Data Inspector window focused on the selected plural variable.</td>
</tr>
</tbody>
</table>
Follow Display a Data Inspector window focused on the variable that the selected pointer variable points to. For more information, see page 6-6.

Cast... Enter another variable type that you would like the selected variable displayed as. For more information, see page 6-7.

Visualize Display a Data Visualizer window focused on the selected variable. If the selected variable isn’t a two-dimensional array variable or an MPL plural variable, this option is unavailable (grayed out).

Copy Add this variable to the History window (see chapter 8).

Send to AVS Send this variable’s data to AVS for post processing by AVS.

The Format menu prompts you to enter a printf formatting string to specify how you want values formatted in the Value pane. The current format is shown in the notifier. This option is grayed out unless you select a particular value in the Value pane to reformat or you select the column heading “Value” to reformat all the values in this Value pane.

If you enter something in the Format notifier that is not a valid printf string, then whatever you enter is displayed as the value of the selected variables. To redisplay valid values, simply reselect Format and enter a valid printf string.

Following Variables

You can follow MPL and C pointers. To follow a pointer:

1. In the Code pane, select an MPL pointer.
   For example, you might select pixels from the source file input.m.

2. Left-hold to magnify the pointer.
   For the current example, the magnifier shows that pixels is a plural unsigned char *, or a singular pointer to a plural variable.

3. Select Evaluate from the Inspection menu.

4. Click OK.
   An Inspector window appears for pixels.

5. Select Follow from the Inspect menu to look at the variable that the pointer variable points to.
   A new inspector window appears.
The window on the right is the newest window.

The addresses in the two windows show that the pointer itself is in ACU memory, while the plural variable being pointed to is in PE memory. The name of the variable is the same in both Data Inspector windows.

Casting Variables

When you select Cast, MPPE prompts you to enter another variable type that you would like to use to reinterpret the variable (just like "cast" in the C sense).

To display a string value in MPL or ULTRIX C, do the following:

1. Inspect the char* variable
2. Follow the char* variable
3. Cast... the variable pointed to by char* to char[n], where n is the number of characters in the string
4. Select the Value column title in the inspector so that all the values are selected
5. Select Format, and enter %c in the notifier

NOTES: In C, if you pass an array to a function, in the formal parameter the function will be as a pointer; to get back to the array (if you want to look at it as an array), you have to follow the pointer, then cast the result to an array.

Casts involving named types, such as structures, unions, enums, and type defs are not supported.
Evaluating Expressions

MPPE allows you to evaluate C and Fortran style expressions made up of constants, program variables, and operators, once your program has started executing. (Operators involving assignments are not allowed.) Expressions may be entered into an Evaluation window by using the mouse to highlight an expression in the Code pane, or by typing an expression directly into the Evaluation window.

You can edit the expression to change it if you want. For example, you might change

\[ \text{tilesX} = (\text{imageX} + \text{nxproc} - 1)/\text{nxproc} \]

to

\[ \text{tilesX} = (\text{imageX} + \text{nxproc} - 2)/\text{nxproc} \]

C or Fortran scoping rules are used in variable evaluation.

Legal Expressions

Expressions can consist of:
- constants
- variables
- operators
- calls to MPL user functions

The following table gives more explanation about each of these items.

<table>
<thead>
<tr>
<th>Constants</th>
<th>ANSI C Integer formats</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• decimal, hex, and octal formats, l and u suffixes</td>
</tr>
<tr>
<td>ANSI C character formats</td>
<td>• 'x', '\hh', '\0xxx'</td>
</tr>
<tr>
<td>Floating point</td>
<td>• decimal with or without a decimal point</td>
</tr>
<tr>
<td></td>
<td>• scientific notation with 'e' for exponent</td>
</tr>
<tr>
<td>Fortran logical</td>
<td>• .true. and .false. for</td>
</tr>
<tr>
<td>Variables</td>
<td>MPL and C</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>• built-ins</td>
</tr>
<tr>
<td></td>
<td>• short, int, long, char, float, double</td>
</tr>
<tr>
<td></td>
<td>• signed and unsigned for integral types</td>
</tr>
<tr>
<td></td>
<td>• plural built-ins</td>
</tr>
<tr>
<td></td>
<td>• pointers, arrays</td>
</tr>
<tr>
<td></td>
<td>• all singular and plural varieties involving the above supported built-ins</td>
</tr>
<tr>
<td>MasPar Fortran and F77</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• LOGICAL, INTEGER, REAL, DOUBLE PRECISION</td>
</tr>
<tr>
<td></td>
<td>• arrays of the supported types</td>
</tr>
<tr>
<td></td>
<td>• triplet notation for array subscripts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operators</th>
<th>arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• +  -  *  /  %</td>
</tr>
<tr>
<td>relational (Fortran or C style)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• &gt;  &lt;  &gt;=  &lt;=  ==  !=</td>
</tr>
<tr>
<td></td>
<td>• .eq.  .ne.  .lt.  .gt.  .ge.  .le.</td>
</tr>
<tr>
<td>logical (Fortran or C style)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• !</td>
</tr>
<tr>
<td></td>
<td>• .or.  .and.  .not.</td>
</tr>
<tr>
<td>C pointer and array</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• *  []  &amp;</td>
</tr>
<tr>
<td>F77 and F90</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• F77 array subscripts, such as x (3, 4)</td>
</tr>
<tr>
<td></td>
<td>• F90 array sections using triplets, such as x (3:5:2, 7:9:1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>MPL user functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• arguments: plural or single built-in types and pointers</td>
</tr>
<tr>
<td></td>
<td>• return values: plural or single built-in types and pointers</td>
</tr>
</tbody>
</table>
Expression Errors

Errors that occur during expression evaluation are reported in the Error/Status pane of the Root window. Three kinds of expression errors can occur:

- If a variable name is mistyped, MPPE reports that the name is not recognized.
- If an illegal expression causes a parse error, MPPE shows the expression in the Error/Status pane, and inserts an error message into the expression close to the item that caused the error.
- If an evaluation error occurs (for example, if you attempt to evaluate a variable that is not in scope), MPPE displays an error message in the Error/Status pane.

The following table shows what forms of expressions are not supported.

<table>
<thead>
<tr>
<th>Variables</th>
<th>MPL and C</th>
<th>MasPar Fortran and F77</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>• long long types, complex types, structures, unions, and enums</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• COMPLEX, DOUBLE COMPLEX, CHARACTER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constants</th>
<th>MPL</th>
<th>MasPar Fortran and F77</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>• strings</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CHARACTER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 'd' exponent character in floating point</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operators</th>
<th>Operators involving assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• = ++ -- +- *= +=, and so on.</td>
</tr>
<tr>
<td>MPL and C</td>
<td>• plural variables for array subscripts</td>
</tr>
<tr>
<td></td>
<td>• bitwise operators (&lt;&lt;, &gt;&gt;, &amp;, and so on)</td>
</tr>
<tr>
<td></td>
<td>• xnet, router, and so on</td>
</tr>
<tr>
<td></td>
<td>• cast, sizeof</td>
</tr>
<tr>
<td>MasPar Fortran and F77</td>
<td>• .eqv. .neqv.</td>
</tr>
<tr>
<td></td>
<td>• exponentiation (**))</td>
</tr>
</tbody>
</table>
Inspecting Data

You can use MPPE to see the type, address, and value of a variable, to inspect the contents of variables in PEs (inspect plural data), and to slice large Fortran arrays. Showing this information in an Inspector window is especially helpful when the variable is a plural variable, since the magnifier pop-up shows only the first value.

NOTE: In MasPar Fortran, changes made to a variable's value in a subroutine might not be reflected in the calling routine until the subroutine returns. For example, assume subroutine A calls subroutine B and passes argument arg, and subroutine B changes the value of arg. If you execute your program past the point where B changes the value of arg but not to the end of B, and open an inspector on arg from A and another inspector on arg from B, the inspector you opened from B will show the value just assigned in B, but the inspector you opened from A may continue to show the value of arg that was passed to B until B returns.

Inspecting Plural Variables

The Inspect Plural menu option is a specialized form of the Inspect option. Inspect Plural is only available if the selected variable is a plural variable or an aggregate; if the variable you select isn't plural or aggregate, the Inspect Plural option will be grayed out. For a plural or aggregate variable, the Value pane contains a table where the value of each element of the plural variable is displayed.
Inspecting MasPar Fortran Arrays

If the variable you are inspecting is a MasPar Fortran array, the Data Inspector window has a Slice pane between the Name pane and the Value pane. The Slice pane allows you to subselect slices (sets of elements) of the array to display in the Value pane. This can make it much easier to inspect the values of a very large array and to focus on the most important data.

The Slice pane is a text pane that displays the beginning index, the ending index, and the stride for each dimension of the MasPar Fortran array. (These are essentially the same as Fortran 90 triplets.) For example, if you make the declaration dimension A(64,64) the Slice in a Data Inspector window focused on A initially looks like this:

```
1:64:1, 1:64:1
```

- `first` is the index of the first element displayed in that dimension
- `last` is the index of the last element displayed in that dimension
- `stride` is the stride index, which is one greater than the number of array elements skipped between each one that is displayed. A stride of 4 displays the first element displayed in that dimension, then every fourth element.

When you first select `Inspect`, the value of each element of the array is displayed in the Value pane. You can select and edit the index and stride values in the Slice pane to display only a subset of the array element values in the Value pane. The display in the Value pane is modified after you press `Return` in the Slice pane.

**Quick Tip:** If you select `Inspect` after you have edited the values in the Slice pane, the new Data Inspector window is focused on the sliced array defined in the original Data Inspector window. If you select `Visualize` after you have edited the values in the Slice pane, the Visualizer pane in the Data Visualizer window shows values only for the array elements selected in the original Data Inspector window.

Visualizing Data

The Data Visualizer window allows you to graphically display or visualize the value of a two-dimensional array variable or an MPL plural variable relative to some constant that you supply. This enables you to check all at once the status of a variable on every PE. You can also magnify the data in the window to find the contents of individual PEs. This is useful in locating potential trouble spots on individual processors or individual array elements.

To visualize a plural aggregate type variable, Evaluate it, then select an element or number of the aggregate and select `Visualize` from the Inspect menu.
Allowed Variable Types

This section discusses the types of variables that you are allowed to visualize in the different languages that MPPE supports.

MPF and DEC Fortran

In general, in MPF and DEC Fortran, you can only visualize two-dimensional arrays of built-in type variables. If the selected variable does not meet the criteria stated below, then the Visualize option is grayed out.

In MPF, you can visualize two-dimensional arrays of integer, real, double precision, character, logical, or byte type variables.

In DEC Fortran, you can visualize anything you can visualize in MPF. In addition, you can visualize two-dimensional arrays of integer*2 type variables.

For example, given the following three declarations,

\[
\begin{align*}
\text{real} & \ a(32, 16) \\
\text{real} & \ b(32) \\
\text{integer} & \ c \\
\end{align*}
\]

you can visualize \( a \), but you cannot visualize \( b \) or \( c \). Neither \( b \) nor \( c \) is a two-dimensional array.

MPL and ULTRIX C

In general, in ULTRIX C, you can only visualize two-dimensional arrays of built-in type variables. In MPL, you can visualize anything you can visualize in ULTRIX C, and you can also visualize plural built-in type variables. If the selected variable does not meet the criteria stated below, then the Visualize option is grayed out.

In ULTRIX C, you can visualize two-dimensional arrays of pointers or two-dimensional arrays of \text{char}, \text{short}, \text{int}, \text{long}, \text{float}, \text{unsigned?}, \text{or double} type variables.

In MPL, you can visualize anything you can visualize in ULTRIX C. In addition, you can visualize two-dimensional arrays of \text{long long} type variables, and you can visualize plural built-in type variables, where built-in types are all the types listed for ULTRIX C plus the \text{long long} type.

For example, given the following six declarations,

\[
\begin{align*}
\text{int} & \ a[32][16]; \\
\text{long long} & \ b[32][16]; \\
\text{plural int} & \ c; \\
\text{int} & \ d[4]; \\
\text{int} & \ e; \\
\text{plural int} & \ f[2][4]; \\
\end{align*}
\]

you can visualize \( a \), \( b \), and \( c \), but you cannot visualize \( d \), \( e \), or \( f \). Neither \( d \) nor \( e \) is a two-dimensional array, and \( f \) is a plural aggregate type, not a plural built-in type.
To visualize a plural aggregate type variable, first inspect it (select \texttt{Inspect while you} have the parallel aggregate variable selected), and then from the new Data Inspector window visualize an element or member of the aggregate.

To visualize an expression, follow the directions in the previous paragraph.

**Using The Data Visualizer Window**

The Data Visualizer window is displayed when you select \texttt{Visualize from}:
- The Inspection menu in the Execution window
- the Inspect menu of any Data Inspector window
- the Global Variables List menu of the Global Variables window

The window consists of two panes:

![Data Visualizer Window Diagram]

**Specifying a Comparison Expression**

The Data Visualizer Comparison pane displays an expression at the bottom of the window where:
- The left-hand side of the expression is the name of the variable you are visualizing
- The right-hand side of the expression is a constant
- The operator is one of the allowed operators for expressions (see page 6-8)
The expression in the Comparison pane is initialized to:

\[
\text{variableName} \; \equiv \; 0
\]

To visualize a different variable, you must open another Data Visualizer window. If you try to visualize another variable by changing the left-hand side of the expression, you get an error message. If you do this, middle-hold and select Cancel from the Comparison menu.

If you’re visualizing a variable, you can change the operator on the right-hand side of the expression by typing a new operator or value. When you press Return, the column of dots in the visualizer pane moves. If you’re visualizing an expression, the items in the Comparison pane cannot be changed.

**Interpreting the Visualization**

Each PE or each element of a two-dimensional array (depending on the type of variable you are visualizing) is represented by one pixel in the Visualizer pane, and there is one blank separation pixel between each pair of pixels that represent PEs or array elements. You can make the display easier to see by enlarging the Data Visualizer window.

Pixels that are on represent PEs or array elements in which the value of the variable you are visualizing meets the criterion given by the expression in the Comparison pane. In other words, if your comparison frame shows \( B==0 \), then the pixels that are on meet that criteria. Pixels that are off represent PEs or array elements in which the value of the variable you are visualizing does not meet the criterion stated in the Comparison expression. (For most monochrome systems, on will be black and off will be white. However, your system may use a different color map.) The Visualizer pane is updated as soon as you enter a legal expression and then either press the Return key or select Accept from the Comparison menu.

**NOTE:** If you visualize an expression that doesn’t return a boolean value, the dots in the visualizer will be white for zero and black for non-zero (following C conventions for true and false where non-zero is true).

**Using the Visualizer Menu**

The Visualizer menu contains two options:

**Use:**

**To:**

**Format**

Enter a printf formatting string to specify how you want values formatted in the magnifier. You can then use either format (the format specified in the variable declaration or the format you entered after you selected Format) in the Comparison pane.
Magnifier Size  Enter the number of values you want to see in the magnifier (the number of array elements or the number of instances of a plural variable). The number of values displayed is the square of the number you enter for Magnifier Size, and the values are displayed in a square. For example, if you enter “2”, then the magnifier displays the location (x,y) in array notation on the first line, the values at (x,y) and (x+1,y) on the second line, and the values at (x,y+1) and (x+1,y+1) on the third line.

Magnifying Values

You can magnify values in the Visualizer pane the same way you do in the Code pane: left-hold over the value you want to magnify. The pop-up displays the PE coordinates and the value of the variable you are visualizing. You can also left-hold and drag the cursor around in the Visualizer pane to see how these items change for the different pixel locations.

If you are visualizing an array variable, the number for location indicates which element of the array the cursor is over. If you are visualizing an MPL plural variable, the location is the row number followed by the column number of this PE in the PE array (recall that row and column numbering of the PE array start at zero). The origin of the display in the Visualizer pane is the upper left corner.
Chapter 7

Profiling

This chapter discusses the ways you can get information on how efficient your program is. MPPE provides you with two kinds of information on a program: a static or compile-time analysis is provided before the program is run, and a dynamic or execution analysis is presented as the program is executed. This chapter tells you:

- how to display both the compile- and execution-time profiling data for your program
- how to print the profiling data

QUICK TIP: You may want to display a History window as you’re working through your program and examining the program’s execution profile. The History window displays information on how resources and variables change over time, and you may want to compare this information with the profiling information. See Chapter 8 to learn more about the History window.

NOTE: Compile-time profiling information is only available for MasPar Fortran programs. Execution profiling is available for both MPL and Fortran programs.
Viewing Profiling Information

The Statement Profiles bar is a tall, narrow rectangle on the right side of each Code pane. If you have not yet begun to execute your program, the bar shows compile-time profiling information. If you’ve begun execution, the bar shows execution profiling information.

There is a line in the Statement Profiles bar for each physical line of code in the Code pane. The Statement Profiles bar and the Code pane scroll together when you use the scroll bar.

What is Compile-time Profiling?

When you first open your MasPar Fortran program in MPPE, the Execution window provides you with a compile-time profile of your program, shown in the Profile bar on the right side of the screen:
This analysis shows the estimated amounts of resources your program will use, based on an analysis of the code performed at compile time. The size of the histogram bar tells you the approximate amount of resources that is consumed by the routine or statement, based on an internal system of weights. Weights are numbers assigned by the compiler to an operation. Higher numbers are assigned to operations that typically require more time. (For more information on what these operations are, see Chapter 2 of the MasPar Commands manual.) You can use the compile-time information to start doing some performance tuning before running your program.

To find out why your compile-time histogram bars are the size they are, you can magnify the bar (see page 7-4).

What is Execution Profiling?

You must execute a statement or routine before the execution profiling information is available. Execution profiling tells you how much time your MPL or MasPar Fortran program actually spends executing each routine and statement, based on the number of ticks that are credited to each routine and statement.

Each time your program receives a UNIX clock interrupt, two ticks are added for each statement and routine that are currently executing—one tick for the front-end and one for the ACU. Profiling provides you with a statistical way to search for hot spots in programs—you can use the records of the ticks to spot areas that could be optimized.
NOTE: Because ticks are accumulated in a sampling procedure, if a statement is very simple, it might execute without accumulating any ticks. In addition, the same statement may accumulate a different number of ticks in different runs of the program, due to statistical variations.

There are two kinds of execution profiling available in MPPE, hierarchical profiling and flat profiling. Which type of profiling you get depends on what you type at the command line when you compile the program; the differences between the two types of execution profiles are explained in the MasPar Commands Reference manual.

To find out why your execution profile is the size it is, you can magnify the histogram to get more information (see the following section).

**Magnifying the Information**

You can view the compile-time and execution profiling information by magnifying the histograms, or by showing the information in a window. The windows are discussed in the following section.

To magnify a histogram or to see the compile-time or execution profile for a statement:

1. In the Statement Profiles bar, move to the histogram you want to magnify.

2. Place the cursor over the histogram and hold the left mouse button.

   A small box pops up that contains the profile information. The first two lines tell you the number of ticks and percentage of total program ticks that
the statement has accrued; if the statement hasn’t executed yet, the two
two lines referring to ticks are not displayed. Only the compile-time profiling
messages are displayed if the program hasn’t executed yet.

| 94 ticks  
| 16%  
| 4X (4) #391 scalar slosh to the DPU  
| 4X (4) #392 scalar slosh from the DPU  
| 4X (0) #1803 array temp made on DPU  

In the example shown above, the term *slosh* refers to the message passing between the
DPU and the front end. An array temp is an array created by the compiler to hold
intermediate results. For information on what each of these messages mean, and on what
to change in your program in response to the profiling information, see the MasPar
Fortran manuals.

**Using the Profile Windows**

You use the context-sensitive menu in the Profiles bar if you want more detail on the
compile-time or execution profiling information, or if you want to print or delete the
execution profiling information.

**Middle hold in the Profiles bar to select one of the following from the menu.**

**Select:**  
**To:**

**View Routines**  
Display the Routine Profiles window (see page 7-6).

**View Statements**  
Display the Statement Profiles window (see page 7-8).

**View Static Routines**  
Display the Static Routines window (see page 7-6).

**View Static Statements**  
Display the Static Statements window (see page 7-8).

**Clear**  
Clear all execution profiling information. Only the
compile-time information is left in the currently opened
program, and all histogram bars are removed from the
Profile bar.
Print

Before you begin to execute your program, this menu option writes a .ct file of compile-time profiling information.

Once you’ve begun to execute, this menu option writes a postscript file (with a .ps suffix) and an ASCII file (with an .as suffix) with the same specified name.

These files contain execution profiling information on the entire program, not just the visible file. For more information on the printed files, see page 7-9.

Routine Profiles Window and Static Routines Window

To display the Routine Profiles window, select View Routines from the Profiles bar menu. To display the Static Routines Window, select View Static Routines.

The two windows for Routine profiling information have the same layout and basic functionality. Both windows have a Profile List pane at the top and a Code pane with a Profiles bar at the bottom.

The following figure shows the Routine Profiles window:

![Routine Profile Window](image)

The Profile List in the Index pane contains a table with one row for each routine that has profile data (sorted by costliest routine). When you select a line in the Index pane, the Code pane displays the file where that routine is defined, with the definition of that routine near the center of the Code pane.
For the Static Routine profiles window the %, Ticks, and Hierarchical columns in the Index pane are replaced with the Weight column. This column displays a number that shows the amount of relative importance this routine was assigned at compile time, based on the weights.

```
where (oshift(template_dim+2,shift+1) > template)
  scapegoat:Array+scapegoat:Array + 1
end where
do i = 1,8
do j = 1,8
  scapegoat(template(i,j) + 1) = scapegoat:Array(i,j)
end do
done subroutine FindScapegoat

subroutine DistributeError(active, scapegoat, error, x, y, A)
  integer : x, y, scapegoat
  logical, array(x,y) : active
  real, array(x,y) : error, A
where (oshift(active_dim+1,shift+1))
  A = A + oshift(error_dim+1,shift+1) / scapegoat
end where
where (oshift(active_dim+1,shift+1))
  A = A + oshift(error_dim+1,shift+1) / scapegoat
end where
where (oshift(active_dim+2,shift+1))
```

**Statement Profiles Window and Static Statement Profiles Window**

To display the Statement Profiles window, select View Statements from the Profiles bar menu. To display the Static Statement Profiles Window, select View Static Statements.

The two windows for Statement profiling information have the same layout and basic functionality. Both windows have a Profile List pane at the top and a Code pane with a Profiles bar at the bottom.

The following figure shows the Statement Profiles window:
The Profile List in the Index pane contains a table with one row for each statement that has profile data (sorted by costliest routine). When you select a line in the Index pane, the Code pane displays the file where that routine is defined, with the definition of that routine near the center of the Code pane.

The list of statements in the Statement Profiles List pane is shown as a table with one row for each statement that has accumulated at least one tick and with the following columns:

- The line number in the source file
- The routine where this statement appears
- The file
- The % of total program ticks accumulated
- Whether or not the statement was compiled with hierarchical profiling

The information in this pane is updated each time execution stops.

The statements are listed in decreasing order, with the statement that has the most ticks at the top. Statements that have not executed are not listed; neither are statements that have been executed but have not accumulated any ticks. (A statement will not show any ticks if you selected Clear from the Profiles bar menu after that statement executed.)

When you select a statement in the Statement Profiles List pane, the file where that statement appears is displayed in the Code pane, with the selected statement near the center of the Code pane.
For the Static Routine profiles window the %, Ticks, and Hierarchical columns in the Index pane are replaced with the Weight column. This column displays a number that shows the amount of relative importance this routine was assigned at compile time, based on the weights.

- Printing Profiling Information

When you select Print from the Statement Profiles menu, MPPE saves the current profiling information in one or more files. If you have not yet begun to execute the opened program, Print saves the information in a file with a .ct suffix. This file shows compile-time messages. It’s in ASCII, so you can print it or import it into another program.

If you’ve started to execute the program, MPPE saves the profiling information for your current program in two formats: as a Postscript file and as an ASCII file. The two files have the same name as the executable file, but the ASCII file has a .as suffix and the Postscript file has a .ps suffix. The Postscript file can be printed on any printer that prints Postscript files; the ASCII file can be printed, or the data could be imported and used in another program.

See the MasPar Commands manual for information about the format of the output files.
Chapter 8

Resource and Variable History

MPPE enables you to look at how variables and resources change over time. You can examine:

- The value of any number of program variables
- The percentage of active PEs
- The time spent communicating
- The time spent stalled waiting for page swaps (fetching instructions)
- The total computation time

This chapter shows you how to:

- Display a History window
- Examine the history of a variable
- Examine the history of a resource
Displaying a History Window

To display a History window:
1. Move to the Index pane of the Execution window.
2. Middle-hold and select View History from the menu.

An empty History window appears:

![History Window Diagram]

Each tick mark along the left side of the window represents one instance of an execution control operation.

You use a History window by pasting variables and resources onto it. The History window shows one tick for every execution control operation performed during this session. Time in a History window is displayed downwards, with the first operation you executed shown at the top and the most recent operation shown last. This figure shows a window with variables and a resource:
When used with the Animate feature, the History window displays a record of the values of the variables of your choice and resource utilization as your program executes.

MPPE updates the History window whenever your program is stopped. Animate re-starts execution automatically as described in Chapter 4. This means that if you choose Step or Skip, your History window is updated continually.

Pasting Variables

Pasting variables next to each other enables you to compare their behavior easily.

To paste a scalar variable onto a History window:
1. Make sure the variable is in scope (is still defined at the point where the program is executing).
2. Left-click the variable’s identifier in the Execution or Inspector window.
3. Display the Code menu and select Copy. Move to the History window.
4. Display the History menu and select Paste.
5. The variable is added to the window.

If there were already variables or resources in the window, the newest item is added to the right.

Examining Variables

You can display the contents of a variable in two ways. Middle-hold on the variable’s column and select one of the following options from the History menu:

Select: To:
Display Values

Show a value for each tick. If the value of a variable does not change, the block it occupies expands for each tick.

Display Graph

Plots the value of the variable as a line between its minimum and maximum values.

By default, the contents of the variable are displayed by value, but you can easily change that by selecting Display Graph from the context-sensitive menu in the History window.

Formatting a Variable

You format how a variable is displayed by selecting format from the context-sensitive History menu. A Notifier box appears that prompts you to enter a printf formatting string.

You can’t change the format for resources; only variables can be formatted.

Changing Tick Spacing

You can change the distance between the tick marks on the left side of the History window. You might change the tick spacing to show more data in the window, for example, or to change the look of the graph.

You left-hold and drag a tick mark to change the spacing. Drag upwards to decrease the distance between ticks. Drag downwards to increase the distance between ticks.

Pasting a Resource

You paste resources to see how well your program is using PEs, communicating with the DPU, and either waiting or performing computations. This information can be helpful if you’re working at optimizing your program.
To paste a resource:

1. In the History window, middle-hold to display the History menu and select Paste Resource....

2. Select an option from the submenu:

<table>
<thead>
<tr>
<th>Select:</th>
<th>To Display:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active PEs</td>
<td>The percentage of active PEs.</td>
</tr>
<tr>
<td>Total</td>
<td>The total computation time on the DPU.</td>
</tr>
<tr>
<td>Communication</td>
<td>An estimate of the time spent communicating on the DPU. This time is calculated by dividing the number of communication operations over the X-Net or the router by an empirical constant which represents the average length of a communications operation.</td>
</tr>
<tr>
<td>Instruction Fetch</td>
<td>The time spent stalled waiting for IMEM page swaps on the DPU.</td>
</tr>
<tr>
<td>Computation</td>
<td>The time spent performing computations on the DPU.</td>
</tr>
</tbody>
</table>

The time is in microcycles, based on an 80ns clock rate for the MP-1.
Resource and Variable History
Appendix A

Setting User Preferences

This appendix describes some of the settings you may need to be aware of when using MPPE.

If you would like to change the way some MPPE features work or the way the MPPE display looks, you can either select Customize from the Root window button bar to display the Customize window, or you can edit the information in your .mppe file.

The .mppe file uses the same parameter names and values that the Customize and Connect windows use. You can edit the file directly to change those windows’ settings.

.Xrdefauts File

Before you use MPPE, you may want to check your .Xdefaults file. For example, you may want to change the window colors. There are two simple items to check:

1. Make sure you have a .Xdefaults file in your login directory. If you do not, you can use the .Xdefaults file that came with your MasPar system (/usr/maspar/.Xdefaults or $MP_PATH/.Xdefaults).

2. Make sure overlapping objects (such as foreground and background) are specified to be different colors. This should already be set up correctly when you get your .Xdefaults file. If you want to edit your .Xdefaults file, refer to the X man page.
An mppe prefix exists that is similar to the sm prefix. You can use this prefix to specify foreground and background colors for MPPE that are different from your default foreground and background colors. For example:

```plaintext
sm.foreground: black
mppe.foreground: grey
```

If you do end up with the foreground the same color as the background, you will not be able to do any work in MPPE because you cannot see the menus or anything else. To get yourself out of this situation, iconify MPPE Root window the same way you iconify any other window, and then terminate MPPE program by killing the mppe process.

The Customize Window

The Customize window is displayed when you select Customize from the Root window button bar.

![Customize Window](image)

The Customize window allows you to change some MPPE characteristics, such as where the Execution window appears and what font the windows use. You should carefully read the following descriptions of these parameters and their values before you change any of the values.

If you have more than one session of MPPE running, it is possible to open more than one Customize window at one time, but you can only have one Customize window open per session. Note that any changes you make to parameter values in one window will not automatically be reflected in other Customize windows.
Changed values only apply to the current MPPE session unless you select Save from the Customize button bar.

If you don’t want to save the settings you’ve changed for a future session, close the window without saving. MPPE will use the changes for this session only, and will use the settings previously saved the next time you open MPPE.

**Source Directories**

*Value:* a list of directory names separated by spaces  
*Default:* .  
*Description:* If your sources are not in the same directory as your executables, use this parameter to list the directories MPPE should search to find the source files to display. The default value (a single period) refers to the directory where the executable resides.

**Font**

*Value:* [fixed|small|default|large|demo]  
*Default:* default  
*Description:* This sets the font for everything that is displayed in MPPE. fixed is a fixed-width font; small, default, and large are three sizes of the same font; and demo is a very large font.

**Show Stack Frames**

*Value:* [user|all]  
*Default:* user  
*Description:* This specifies which routines in the call stack are displayed in the Stack pane. The user value means display all user-written routines. The all value means display the routines included in user plus MasPar internal routines.

**Float Formats**

*Value:* [%f | %g | %e]  
*Default:* %g  
*Description:* This parameter is a printf formatting string that specifies the format used to display floating point variables in the inspectors and magnifiers. For more information on printf format strings, enter man printf at the command line. Although the value must be expressed as a printf format string, the resulting format applies equivalently to floating point variables in any MasPar-supported language.
**Code Pane Height**

**Stack Pane Height**

Value: real number greater than 0

Default: Code Pane Height: 4  
Stack Pane Height: 1

Description: This parameter specifies the relative proportion of the vertical space that these two panes occupy in the Execution window. The Code Pane Height parameter includes the Breakpoints bar and the Statement Profiles bar, and the Stack Pane Height parameter includes the Machine pane.

**Execution Window Location**

**Stdin/Stdout Window Location**

Value: two integers, separated by an “@” sign

Default: Execution Window Location: 20@20  
Stdin/Stdout Window Location: 20@542

Description: These parameters position the Execution and Stdin/Stdout windows. The first integer is the x ordinate of the upper left corner of the window, and the second integer is the y ordinate of the upper left corner of the window. You can type the window location preferences into your .mppe file or into the Customize table, but a much easier method for determining these values is to simply move the windows where you want them and then select Window Locations and then Save from the Customize window. Note that if you have resized the windows or the panes within the windows, these new sizes are not saved. Only the position of the upper left corner is saved.

**NOTE:** The window location preferences are a little different from the other preferences in the way their values are determined. You can enter the coordinates into your .mppe file or into the Customize table yourself, but a much easier way to do it is to simply move the MPPE windows where you want them and then select Window Locations and then Save from the Customize window.

**Cursor Warping**

Value: [true | false]

Default: true

Description: When a new window is displayed, it is positioned so that the upper left corner of the window is where the arrow cursor is. If the Cursor Warping parameter is set to true, then the arrow cursor is positioned in the middle of the window as soon as the window is displayed. If this parameter is set to false, then the arrow cursor will remain at the upper left corner of the new window until you
move it by moving the mouse. You may want to set this parameter to false for certain applications that are monitoring the cursor and might be expecting the cursor to be doing something else.

The Connect Window

The Connect window is displayed when you select Connect from the Root window button bar.

The Connect window contains settings for the name of the MasPar machine you are running on, the directory name where the executables are stored, and your login name on the MasPar machine. The settings in the Connect window are also saved to your .mppe file, just like the Customize window settings.

Host

Value: the name of the DECstation 5000 that is directly connected to a DPU
Default: There is no default value for this parameter.
Description: By default, MPPE assumes the machine you are logged into when you invoke your user program is one that is directly connected to a MasPar DPU. If the machine you are logged into is not directly connected to a DPU, then you need to enter the name of the MasPar machine that you want to run your user program on as the value of this parameter. See the discussions on displaying and invoking MPPE and invoking your user program starting on page B-2.
MP_PATH

Value: the path to the MasPar software on the MasPar machine (see above)
Default: There is no default value for this parameter.
Description: You do not need to use this parameter unless you are running MPPE and the program being debugged on two different machines and the name of the path to the MasPar software is different on the two machines. The installation instructions for the MasPar software recommend that the path to the MasPar software on each machine be aliased to /usr/maspar. For example, if you are running MPPE on a DECstation and running your user program on a Sun workstation, the MasPar software resides in different places on the two machines. If the path to the MasPar software on the DECstation and the path on the Sun are both aliased to /usr/maspar, then you do not need to use the parameter. If the path names on the two different machines are not both aliased to the same name, then in this example you need to set your local MP_PATH to the name of the path on the DECstation, and you need to set this option to the name of the path on the Sun.

Directory

Value: the path to the executables on the MasPar machine (see above)
Default: There is no default value for this parameter.
Description: You do not need to use this parameter unless you are running the MPPE and your user program on two different machines and the name of the path to your executables is different on the two machines. When the directory where your executables reside is mounted on the MasPar machine, it may have a different full path name than it has on the local machine. You need to select the correct local directory in the File List pane in the Root window, and then set this Directory parameter to the path name on the MasPar machine that represents this same directory.

Login

Value: your user ID on the MasPar machine
Default: There is no default for this parameter.
Description: This parameter tells MPPE the user ID to use when MPPE is to be run on a remote machine. You do not need to use this parameter unless you are running MPPE and your user program on two different machines and your user ID is different on the two machines. If so, enter the user ID for the MasPar machine for this parameter.
The .mppe File

As an alternative to changing the values in the Connect or Customize windows, you may want to create or modify a .mppe file. The .mppe file is created or modified for you in your login directory when you select Save in the Customize window. You can also create or edit your .mppe file directly.

MPPE reads your .mppe file when you invoke MPPE and whenever you save the settings in the Customize window or Connect window. MPPE looks for the .mppe file in your login directory. If you do not have a .mppe file in your login directory, then the default values listed on page A-8 are the values MPPE will use.

The .mppe file uses the same parameter names and values that the windows use.

If you get an error message in the Error/Status pane in the Root window that indicates that something is wrong in your .mppe file (such as an out-of-range value), you can either:

- edit the offending value in the applicable window and then select Save, or
- edit your .mppe file directly (from a shell window outside the MPPE), and then select Use Last Saved in the Customize window.

The general format of the .mppe file is a parameter name, followed by a colon (:), followed by a value. Lines that begin with a pound character (#) are ignored.

Parameters can be given in any order in the .mppe file, but each parameter name must start at the beginning of a new line. If you list a parameter more than once, the last value listed for that parameter is the one that will be used.

Most parameter names consist of more than one word. Separate words within one parameter name must be separated by a single space; do not use tabs or multiple spaces within a parameter name. You can use multiple tabs and spaces (or no space) between the parameter name and the colon and between the colon and the value.

Example .mppe File

This example lists all the parameters. In a real .mppe file, you do not need to list a parameter unless you want to give it some value other than the default value. Your .mppe file will probably only list a few parameters, because in most cases the default values are the values you want to use.
# This is an example .mppe file.

source directories: . /moi/tests/sources /moi/examples/sources

host: mppe

mp_path: /another/maspar
# "remote mppath" is not needed if MasPar software is in /usr/maspar
# on all machines
Directory: /havefun/tests/executables

font: helv12

show stack frames: user

float format: %g

code pane height: 4

stack pane height: 1

execution window location: 20020

stdin/stdout window location: 200542

cursor warping: true
Appendix B

Invoking MPPE

This appendix discusses:

- The environment variables MPPE uses when it’s invoked.
- How to invoke and display MPPE.
- How to run your user program under MPPE.

Setting Environment Variables for MPPE

Although your system administrator will do most of the setup for you, there are four environment variables that may be different for each user, or even each MPPE session. You will need to know how to set these environment variables in order for MPPE to work correctly.

**MP_PATH**

Specifies the path to the top of the MasPar tree on your system. Include $MP_PATH/bin in your path variable. MP_PATH should be set conditionally in your .cshrc file. It should be set to /usr/maspar or to the top of the tree of the MasPar release that you want to use.

**DISPLAY**

By default, the value of this environment variable is :0.0, which means the display is sent to the machine that you are currently logged into. If you remotely log into another machine, make sure your DISPLAY variable is reset so that the display will go to the machine you are looking at and not the one you are remotely logged into. If you run MPPE after a remote login (rlogin) to a workstation other
than the one at which you are sitting, set the DISPLAY variable to
tell X to display the windows on your screen. (This is an X server
issue, not specific to MPPE.)

See the discussion below on invoking and displaying MPPE.

**MP.DbTARGET** Specifies the name of an executable MOFF (MasPar Object File
Format) file that you want to debug. If the name of a
MasPar-compiled program is the value of the MP.DbTARGET
environment variable, then whenever you invoke that program,
MPPE will automatically attempt to start on that program. If you do
not already have MPPE running on the machine where you invoked
your program, then you will be prompted to start it.

**MP_NODEBUG** Does not have a required value; is either set or not set. By default,
MPPE automatically attempts to start on any MasPar-compiled
program that faults. If MP_NODEBUG is set, MPPE will **not**
automatically attempt to start on a MasPar-compiled program that
faults. Instead, the user program that faults will exit with a status
equal to the signal that caused the fault. MPPE can be invoked
explicitly on any program, even if MP_NODEBUG is set.

**Invoking and Displaying**

The following table summarizes the various ways you can run and display MPPE.
Increasing the performance of MPPE and decreasing the load on the MP-1 front end
machine are the primary advantages of invoking MPPE and your user program on
different machines.

<table>
<thead>
<tr>
<th>If MPPE is displayed on this:</th>
<th>And MPPE is invoked on this:</th>
<th>Then you need to set the following:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP-1</td>
<td>MP-1</td>
<td>nothing (use the defaults)</td>
</tr>
<tr>
<td>remote X terminal</td>
<td>MP-1</td>
<td>your DISPLAY variable</td>
</tr>
<tr>
<td>SPARCstation/Sun-4</td>
<td>SPARCstation/Sun-4</td>
<td>the Host parameter</td>
</tr>
<tr>
<td>remote DS</td>
<td>MP-1</td>
<td>your DISPLAY variable</td>
</tr>
<tr>
<td>remote DS</td>
<td>remote DS</td>
<td>the Host parameter</td>
</tr>
<tr>
<td>remote X terminal</td>
<td>remote DS/Sun</td>
<td>both DISPLAY and Host</td>
</tr>
</tbody>
</table>

In general, if the machines named in the first and second columns are different, then you
need to set your DISPLAY variable; and if the machines named in the second column is
not the MasPar MP-1, then you need to set your Host parameter.

Here is what you need to do with the information in the third column:

- **default.** If you have not reset your DISPLAY environment variable, then
  the system automatically sends MPPE display to the machine that you are
logged into when you invoke MPPE. Then if you select Open in the MPPE Root window and you have not set the Host parameter, the system will automatically try to invoke your user program on the same machine where you invoked MPPE.

- **DISPLAY.** If you want to invoke MPPE on one machine and display it on another machine, then you need to be sure your DISPLAY environment variable is set to send MPPE display to the machine you are looking at and not the one you are remotely logged into. Log into localMachine, remotely log into remoteMachine, set your DISPLAY environment variable to localMachine, and invoke MPPE (on remoteMachine).

- **Host.** If you want to invoke MPPE on one machine and invoke your user program on another machine, then you need to specify which MP-1 DECstation front end you want to invoke your user program on. To do this, use the Host parameter in the Connect window or edit your .mppe file (see page A-7). Set the Host keyword to MPmachine (the name of the MasPar front end machine where you want to run your user program), log into otherMachine (where you want to run MPPE), and invoke MPPE. Then if you select Execute or Profile in MPPE Root window, the system will automatically try to invoke your user program on MPmachine.

The term “MP-1” in the table means a MasPar front end machine; which is always a DECstation 5000 that is directly connected to a MasPar DPU. When this term appears in more than one column in one row of the table, it refers to the same machine in each case, even if you have more than one MP-1 at your site. For example, on the first line of the table you are invoking and displaying the MPPE on the same machine where you are invoking your user program.

If your user program does not contain any DPU code (it is a front-end-only program), then you can invoke it on any DECstation where you can invoke MPPE. But if your program contains any parallel code (any code that executes on the DPU), then you must invoke it on an MP-1. The table only shows the cases where your program contains DPU code.

If you are running remotely, data that you send to stdout may not be displayed until your program terminates.

## Displaying MPPE

You can invoke MPPE on a DECstation or on a Sun SPARCstation/Sun-4. You do not have to invoke MPPE on a MP-1 front end machine. If you invoke MPPE on a DECstation or SPARCstation, then you can display MPPE on any X terminal on the network, where “X terminal” is any terminal running standard MIT X11 window software.

This section explains how to send MPPE display to some machine other than the one where you invoked MPPE.
To invoke MPPE on remoteMachine and send MPPE display to localMachine:
1. Log into localMachine.
2. Use one of the following two methods to give remoteMachine permission
to send a display to localMachine (add the name of remoteMachine to the
list of hosts for localMachine):
   a. If localMachine is a DECstation, then on localMachine choose
      Customize from the DECwindows Session Manager menu, and then
      choose Security from the Customize menu. Then type in the name
      of remoteMachine.
   b. If localMachine is not a DECstation, then on localMachine enter
      xhost+remoteMachineName. This change stays in effect for
      the duration of the current session.
3. From localMachine, log into remoteMachine.
4. Set your DISPLAY environment variable to the name of localMachine, as
   follows: setenv DISPLAY localMachineName:0
5. Invoke MPPE (see below).

Invoking MPPE

To invoke MPPE window interface, log into the machine you want to invoke MPPE on
and type mppe at any shell prompt. See the on-line man page for more information on
the mppe command.

You do not have to invoke MPPE on a MP-1 front end machine (a DECstation 5000 that
is directly connected to a DPU). If your front end machine is a DECstation, then you can
invoke MPPE on any other DECstation or on a SPARCstation. See the MasPar MP-1
System Administration Manual to find out how to set the MPPE software onto your
system with NFS or rcp.

If you get a “command not found” error message, check the value of MP_PATH. If
MPPE seems to start up but you never see a display, check the value of DISPLAY. See
the discussion of environment variables above.

If MPPE begins to display but then hangs, check to make sure your system date is not too
far ahead of the true date. You must be logged in as root to do this.

When MPPE is successfully invoked, MPPE Root window is displayed.
Appendix C

Connecting to AVS

MPPE enables you to connect and send data to the Application Visualization System (AVS), an optional product available from DEC or AVS, Inc. for interactive data visualization and analysis. (AVS is also available for SUN workstations.)

This appendix assumes that you are familiar with AVS. For instructions on the installation and use of AVS, read the AVS manuals.

Why Use AVS?

The MasPar MPPE/AVS connection provides:

- enhanced visualization of array data while debugging your program.
- a flexible and powerful means of processing variable data in order to assist the analysis phase of debugging.
- a tool for use in rapid prototyping of scientific and other programs.

The MPPE/AVS connection allows you to select a variable in MPPE and send it to AVS using a menu option, much like the way you inspect variables in MPPE. Once the data has been sent to AVS, it may be processed and visualized using one of the pre-wired MasPar networks or by a network you've constructed. This is a very flexible and powerful means of analyzing and visualizing data stored in program variables.
There are two primary parts to the MPPE/AVS connection:

- an AVS module called readMPPE.
- a set of networks which use readMPPE.

This appendix explains how to configure and use the MPPE/AVS connection, and gives you details about the readMPPE module, the networks MasPar provides, and how to customize the MPPE/AVS connection.

The MPPE/AVS connection can be used for arrays at character, integer, single precision real or double precision real data. The MasPar networks provide visualization techniques for one-, two-, and three-dimensional arrays. The readMPPE module can be used to import arrays of rank up to 7 into the AVS environment.

Configuring the MPPE/AVS Connection

The MPPE/AVS connection has been written for DEC AVS 3.0. For instructions on the installation and use of AVS, consult the manuals that came with AVS.

The connection has been designed so AVS does not need to run on the same machine as MPPE or the program being debugged. If the machine on which AVS is running (the AVS machine) is a DECstation, you can use the executable readMPPE module provided in the directory $MP_PATH/avs/modules. The source code for this module is in $MP_PATH/avs/src, so that it can be ported to other AVS platforms. An example makefile is also in the directory to show how it should be compiled and linked. The networks for the MPPE/AVS connection are stored in $MP_PATH/avs/networks.

Note that the $MP_PATH/avs directory must be mounted on or copied to the AVS machine so that AVS can access the MasPar networks and module. If the structure of the MasPar AVS directory is modified, it may be necessary to modify the network scripts so AVS can still access the readMPPE module.

Visualizing Data With MPPE and AVS

To visualize a 1-, 2-, or 3-dimensional array of character, integer, single precision real, or double precision real data:

1. To start AVS, type:

   ```
   avs -netdir $MP_PATH/avs/networks -modules \
   $MP_PATH/avs/modules -modules /usr/avs/avs_library
   ```

   Substitute the actual path for the MasPar software directory on your system for $MP_PATH. The command must be typed on one line in order to work correctly.
2. In the main AVS menu, select Network Editor.

3. In the Network Editor menu, select Network Tools.

4. Read-in the MasPar net123d network using the Read Network option. (net123d is one of the networks in the directory $MP_PATH/avs/networks.)

Three graphics windows appear. One is for 1-dimensional data, one is for 2-dimensional data, and one is for 3-dimensional data.

5. Left-click the square button on the readMPPE icon to display the readMPPE menu panel.

6. In the Hostname widget for readMPPE, check whether the hostname listed is the name of the DPU machine on which the program to be debugged will run or is running. If the hostname listed is not correct, delete it and type in the correct one.

7. Make sure the view directly button in the readMPPE menu panel is on (lighted up). This permits data received from MPPE to flow directly to the visualization modules.

8. From MPPE, select the variable to be sent to AVS, middle-hold to display the Code or Inspector menu, and select Send to AVS.

The data will now be sent to the AVS machine and should appear in one of the three graphics windows. Refer to the AVS documentation to learn how to manipulate the data for different views.

About the readMPPE Module

The readMPPE module has the following parameter widgets:

- a Hostname widget
- a View Directly widget
- a file browser widget
- Retrieve and Delete widgets
- a Storage Area widget

The Hostname widget specifies the DPU machine on which the program to be debugged is running. Changing this machine name causes the module to attempt to register with the MasPar maspard daemon running on the given host. If the View Directly widget is on (highlighted), data from MPPE flows out one of the output ports immediately on receipt and the other widgets are not used. If the View Directly widget is off, then readMPPE stores incoming data in a file in the directory specified in the Storage Area widget. The file name has the form module.variable.version.var, where:

- module is the module name where MPPE is stopped,
variable is the name of the variable whose data is being transferred, and

version is an internal count of the number of times data for that variable has been sent to AVS. The version number means that you can save multiple versions of the same variable’s data. It is incremented each time the same variable’s data file is sent to AVS.

When View Directly is off,

• pressing the Retrieve button causes the file selected in the file list widget to be sent to an output port

• pressing the Delete button deletes the file

readMPPE has 4 output ports. Three of the ports act like the output of a demultiplexer: 1D data goes out one port, 2D data on another, and 3D data on the third. All types of data are sent out on the fourth port. This port configuration provides both a demultiplexing function which is useful for network construction and a generic N-dimensional port which you can use for higher-dimensional arrays. Because source code for readMPPE is provided, you can change it as desired.

About the Networks

MasPar provides four networks for viewing data: the net123d network can automatically display 1-, 2-, or 3-dimensional data whereas the net1d, net2d, and net3d networks are specialized to 1-, 2-, or 3-dimensional data only. These networks provide an example of how to use the readMPPE module and also provide a starting point from which customized networks can be created.
Appendix D

Troubleshooting

Limitations

This appendix first states some general limitations. It then discusses some issues specific to the MPPE: problems related to defects in ULTRIX C, the effect of having ^M in your source code files, and limitations in debugging include files and X applications.

You can invoke the MPPE on any DECstation, and you can redirect the MPPE display to any X terminal. However, a user program that calls DPU code can only be invoked on a workstation that is directly connected to a DPU. See page B-2 for more information.

You may notice some delay in starting the MPPE on a program that uses the DPU if another job is already using the DPU. You can check the DPU job queue by using the mpq utility (see the MasPar Commands Reference Manual), or by using the Connect window.

Only programs that were compiled with one of the MasPar compilers will run under the MPPE. This is true even if the program is a front-end-only program. See the MasPar Commands Reference Manual for more information on the MasPar compilers.
Single-Statement else Blocks

In ULTRIX C, if you have an else block that consists of a single statement rather than multiple statements, then you need to insert a carriage return after the word “else”. Otherwise, program execution may not stop at the correct place if you set a breakpoint at the line that contains “else”. MPL does not have this limitation.

Single-Statement Loop Constructs

In ULTRIX C, if you have a loop construct that consists of a single statement rather than multiple statements, then you need to enclose the statement in braces. Otherwise, program execution may not stop at the correct place if you set a breakpoint at the statement following the loop statement. Inserting carriage returns into the single loop statement does not fix the problem as it does in the single-statement else problem described above.

In the following example, when you stop at the line that begins with “num”, execution is still inside the loop. If you set a breakpoint at the line that begins with “num” and then select Step, execution continues through the next iteration of the loop and stops at the breakpoint at “num” again.

```c
for (i=0; i<M; i++) /* UNEXPECTED BEHAVIOR */
    if (primes[i]) printf("%d
", 2*i + 1);
num = 2*M;
```

In the following example, when you stop at the line that begins with “num”, the loop has finished executing. If you set a breakpoint at the line that begins with “num” and then select Step, execution continues to the next executable line after the line that begins with “num”.

```c
for (i=0; i<M; i++) { /* CORRECT BEHAVIOR */
    if (primes[i]) printf("%d
", 2*i + 1);
}
num = 2*M;
```

MPL does not have this limitation.

Control-M in Source Code Files

If you have a ^M in a source file of a program that you want to run under the MPPE, the MPPE will interpret this control character as a newline rather than as a single character. This will cause line number information to be off by one line for each ^M in the source file. You can remove these control characters from your source files by executing the following command:
Be sure to remove the ^M characters from your source files before you compile them. Otherwise, the line number information will still be off by one in the MPPE.

### Include Files

You cannot use the MPPE to debug executable code in an include file. You can use the MPPE Data Inspectors to inspect variables that you have declared in include files, but you cannot set a breakpoint inside an include file, and you cannot Skip or Step into an include file. You cannot display the contents of an include file in any MPPE Code pane.

Suppose you have the following include file, named “i.h”:

```c
int i;
i = 9;
printf("%d\n", i);
```

and the following source file that uses this include file:

```c
main()
{
    float f = 4.5;
    #include "i.h"
    printf("done\n");
}
```

You cannot set a breakpoint on either the “i=9” line or the “printf” line in the include file i.h. In addition, if execution is stopped on the “f=4.5” line and you select Step from the Code menu, execution will continue to the “printf(done)” line; it will not stop at the “i=9” line.

### X Applications

If the program you are debugging is an X application, then you need to be careful what you do while execution is stopped. You should avoid all contact with your program’s X window while your program’s execution is stopped. Activities such as moving the cursor across your program’s X window or moving another window over your program’s X window are activities that fill up your X application’s event queue. If the X event queue fills up while execution is stopped, then the X server may decide that your program has died and may close your program’s X window.
If this happens, you can continue to execute and debug the program within the MPPE, though you may have problems when the program tries to send output to the X window that has been closed. To get your program's X window back again, you need to select *Reset* from the Machine menu and begin execution over again.

**Reporting MPPE Problems**

When reporting a problem with MPPE, please include as much information as possible about the problem, such as:

- The output from `mpconfig`. See the *MasPar Commands Reference Manual*.
- The version number of the MPPE. This is displayed when you invoke the MPPE.
- The `/usr/adm/maspard.log` file.
- The `.mppe-<username>` file in the current working directory.
ACU 1-2
Adding variables
  to History window 8-3
Animate 4-10
Animating execution 4-10
apropos iv
Array slicing 6-12
Arrow marker 4-7, 4-8, 5-3
Arrowheads 3-4

Background window A-2
Bars 3-2
Break on Routine 5-3
Breakpoints 4-7, 5-1, D-3
  disabling 5-5
  disabling all 5-5
  enabling 5-5
  enabling all 5-5
  ignoring 5-5
  removing 5-3
  setting conditional 5-6
  setting on a line 5-2
  setting on a routine 5-3
Breakpoints bar 3-13, 4-7, 4-8, 7-5,
  7-6, 7-8
Breakpoints List menu 5-4
Breakpoints List pane 3-13, 5-3
Breakpoints window 3-5

C
  expression evaluation 6-8, 6-10
  scoping rules 6-8

  supported expressions 6-8
  unsupported expressions 6-10
Call stack 4-8, A-3
Calls to MPL 6-9
Cast... 6-6
Clear 7-4, 7-5, 7-8
Clicking mouse buttons iii
Code menu 4-6, 5-3, 6-4
  Continue 4-7
  Go To Line 4-7
  Go To Routine 4-8
  Inspect 6-11
  Skip 4-7
  Step 4-7
Code pane 3-11, 3-13, 4-7, 4-10,
  6-11, 7-2, 7-5, 7-6, 7-8, D-3
Code Pane Height A-4
Comparison menu 6-15
Comparison pane 6-14, 6-15
Compilers D-1
Conditional breakpoints 5-6
Connect window
  Directory A-6
  Host A-5
  Login A-6
  MP_PATH A-6
Constants 6-8
Context-sensitive menus 3-4
Continue 4-7
Copy menu item
Copy variables
  History window 8-3
.csfrc file  B-1
Cursor Warping  A-4
Customize  3-11, A-1, A-2
Customize window  3-11, A-1, A-4,
 A-7, B-3

Data inspection  6-11
Data Inspector panes  6-4
Data Inspector window  3-6, 6-4,
 6-11, 6-14
Data parallel  1-2
See also  SIMD
Data Visualizer window  3-10, 6-6,
 6-12, 6-14
DEC documentation  iv
DEC Fortran  6-13
DECStation  B-3, B-4
DECwindows  2-5, A-2, B-4
Defaults
 Xdefaul ts  A-1
Defects
 MPPE  D-4
Disabling all breakpoints  5-5
Disabling breakpoints  5-5
DISPLAY  B-1, B-2, B-3, B-4
Display Graph
 History window  8-3
Display Values
 History window  8-3
DISPLAY variable  B-1
 setting  B-1
Displaying MPPE  B-2
Documentation, other books  iii
DPU  1-2
DPU View Location  A-4
DPU window  A-4
Enabling all breakpoints  5-5
Enabling breakpoints  5-5
Environment variables
 DISPLAY  B-1
 MP_DBTARGET  B-2
 MP_NODEBUG  B-2
 MP_PATH  B-1
 See also  the individual variable
 names
Error/Status pane  3-11, A-7
Evaluate  3-5
Evaluating Expressions  6-8
 types not supported  6-10
Evaluation window  3-5
Examples  iv
Executable files  A-6
Execute  3-11, 7-2, B-3
Execution
 animating  4-10
 Execution menu  4-10
 Execution window  3-11, 4-7, A-4
 Breakpoints button  4-6
 quitting  4-3
 Execution Window Location
 A-4
Expressions
 types not supported  6-10
F77
 expression evaluation  6-8, 6-10
 supported expressions  6-8
 unsupported expressions  6-10
FE View Location  A-4
FE window  A-4
File List pane  3-11, A-6
find option, Code menu  4-6
Float Format  A-3
Floating point format  A-3
Follow  6-4, 6-6
Font  A-3
Font conventions  ii
 format  6-6, 6-15
Formatting variables
 History window  8-4
Fortran
 scoping rules  6-8
Front  B-3
Front end  1-2, A-5
Front end machine  2-5, B-3, B-4
Front-end-only programs  B-3, D-1
Global variables  3-5, 4-6
Global variables button, Execution
 window button bar  4-6
Global Variables List menu  6-4, 6-14
Global Variables window  3-5, 6-4,
 6-14
Go To Line  4-8
Go To Routine  4-8
Help  iv
Help Text pane, Help window  3-14
Help window  3-14
History window
 Animate  8-3
 Display Graph  8-3
 Display Values  8-3
 displaying  8-2
 displaying variable contents  8-3
Index 3

formatting variables 8-4
pasting resources 8-4
pasting variables 8-3
tick marks 8-2
tick spacing 8-4
Host machine B-2
Host A-5, B-3

Ignore count 5-5
Ignoring breakpoints 5-5
Illegal expressions 6-10
Index pane 3-13
viewing Breakpoints 5-3
Index pane, Execution window 4-4
Indexing into code 4-4
Inspect 6-4, 6-5, 6-11, 6-12
inspect plural 6-4, 6-5
Inspecting 6-11
Inspector panes 6-4
Invoking MPPE B-2

Limitations
control-M D-1, D-2
include files D-1, D-3
single-statement else blocks D-2
single-statement loop constructs D-2
ULTRIX C D-1
X applications D-1, D-3
Line breakpoints 5-2
localMachine 2-5

Machine icons 3-14
Machine menu 3-12
Machine pane 3-13, 3-14, D-4
Machine Visualizer menu
  ebit 4-13
  mbit 4-13
  rflag 4-13
  tflag 4-13
Machine Visualizer window 3-7
Magnifier Size 6-16
Magnifying 6-2
Magnifying data 6-16
Magnifying histograms 7-4
Magnifying variables 6-2
man iv
MasPar Architecture Specification 4-13
MasPar Commands manual 1-4, B-4, D-1, D-4
MasPar documentation iii
MasPar Fortran 3-6

expression evaluation 6-8, 6-10
supported expressions 6-8
unsupported expressions 6-10
See MPF
MasPar machine A-5
MasPar Object File Format
See MOFF
MasPar parallel processor 1-2
MasPar Programming Language
See MPL
Menus 3-3
  Code menu 4-6
displaying 3-2
MIT X11 B-3
MOFF B-2
Mouse terms iii
Moving around in your code 4-4
mpconfig D-4
MPF 1-5, 6-4, 6-11, 6-13
  array slicing 6-12
MPL 1-5, 6-7, 6-13
  expression evaluation 6-8, 6-10
  supported expressions 6-8
  unsupported expressions 6-10
mppe B-4
MPPE B-2
  display B-3
  introduction to 1-1
  invoking B-4
  prerequisites 1-4
  system requirements 1-3
MPPE background window
  See Background window
.mppe file A-1, A-7, B-3
MPPE Reference Manual 5-2
mpq D-1
MP_DBTARGET B-2
MP_NODEBUG B-2
MP_PATH A-1, A-6, B-1, B-4
MP_PATH environment variable 2-3

Name pane 3-13, 6-4
Notation conventions ii

Ogin A-6
OpenLook 2-5, B-3
Operating MPPE 3-1
Other documentation iii

Pane proportions A-4
Panes 3-2
Parameters
  See User preferences
Paste menu item
Pasting resources
  Active PEs 8-4
  communication 8-4
  Computation 8-4
  in History window 8-4
  Instruction Fetch 8-4
  total computation time 8-4
Pasting variables
  History window 8-3
path variable B-1
PE 1-2
PE communications 1-2
Plural variables 6-5, 6-6, 6-11
Pointers 6-5
Preferences
  See User preferences
Pressing Return 3-4, 4-9
Print 7-6
printf A-3
Profiling 7-1, 7-7
  routines 7-6
  statements 7-6
Program Arguments notifier 4-9
Pull-down menus 3-3
Quit 3-11, 3-12
Quitting the Execution Window 4-3
Re-selecting menu items 3-4, 4-9
remoteMachine 2-5
Removing breakpoints 5-3
Repeating execution 4-10
Reset D-4
Root window 3-10, 3-12, A-1, A-2, A-6, A-7, B-3
Routine breakpoints 5-3
Routine Profiles List pane 7-5, 7-6
Routine Profiles window 3-7, 7-5, 7-6
Routines
  profiling 7-6
Save A-3, A-4, A-7
Scroll bars 7-2
Scrolling 4-4
Searching for a string 4-4
Select iii
Setting breakpoints
  on a line 5-2
  on a routine 5-3
Shell A-7
Show Stack Frames A-3
SIMD 1-2
Skip 4-7, D-3
Slice pane 6-4, 6-12
Source Directories A-3
Source Files 3-8, A-3
Source Files List pane 3-13
Source Files window 3-8
SPARCstation 2-5, B-3, B-4
Stack frames A-3
Stack frames button, Execution
  Window button bar 4-5
Stack pane 3-13, 4-8, A-3
Stack Pane Height A-4
Starting up MPPE 2-2
Statement Profiles bar 3-11, 3-13, 7-2, 7-5, 7-6, 7-8
Statement Profiles List pane 7-5, 7-8
Statement Profiles menu 7-4, 7-6, 7-7, 7-8
Statement Profiles window 3-8, 7-5, 7-6, 7-7, 7-8
Statements
  profiling 7-6
Stdin/Stdout window 3-11, 3-12, A-4
Stdin/Stdout Window
  Location A-4
Step 4-7, D-3
String variables 6-7
Sun 2-5
Sun-4
  See SPARCstation
SunTools 2-5, B-3
System diagram 1-2
System stack frames A-3
Tick marks
  in History window 8-2
Tick spacing
  changing 8-4
Ticks 7-8
Topic List pane, Help window 3-14
Typographic conventions ii
ULTRIX C 4-8, 6-7, 6-13
Unsupported expressions 6-10
Use Last Saved A-7
User functions 6-9
User preferences A-1
Using the mouse iii
Value menu 6-4, 6-5, 6-14
  Cast... 6-6
  Visualize 6-6
Follow 6-5
Format 6-6
Inspect Plural 6-5
Inspect 6-5
Value pane 6-4, 6-11, 6-12
Variables
  changing values 6-5
VAX Fortran 6-13
View history 8-2
View Routines 7-5, 7-6
View Statements 7-5, 7-6, 7-7
Visualize 6-6, 6-12, 6-14
Visualizer menu 6-15
Visualizer pane 6-14, 6-15

Window Locations A-4
Window locations A-4
Windows 3-2
  Execution window 4-4
  Execution window, quitting 4-3
  Help window 3-14

X server B-2
X terminal B-3
.Xdefaults file A-1
MasPar Commands Reference Manual

Document Part Number 9300-0300, Revision A8, May 1992

This manual describes the MasPar shell commands released in May 1992.

MasPar Computer Corporation makes no representation or warranties regarding the contents of this document and reserves the right to revise this document or make changes in the specifications of the product described herein at any time without obligation to notify any person of such revision or change.

Copyright © 1990, 1991, 1992 MasPar Computer Corporation, Sunnyvale, California

All Rights Reserved
Printed in the United States of America

This manual, its format, and the hardware, software, and microcode described in it are copyrighted by MasPar Computer Corporation and may not be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the express written permission of MasPar Computer Corporation.

Restricted Rights Legend:
The software and microcode documented in this manual has been developed at private expense and is not in the public domain. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (c)(1)(i) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

MasPar and the MasPar logo are registered trademarks of MasPar Computer Corporation. MasPar Programming Environment, MasPar Fortran, MasPar Programming Language, MasPar Data Display Library, MasPar Image Processing Library, MasPar Mathematics Library, and MasPar Parallel Disk Array are trademarks of MasPar Computer Corporation.
DEC, DEC AVS, DECnet, DECwindows, DECstation, ULTRIX, VAX, and VAXstation are trademarks of Digital Equipment Corporation.
UNIX is a registered trademark of the American Telephone and Telegraph Company.
X Window System is a trademark of the Massachusetts Institute of Technology.
Motif is a trademark of the Open Software Foundation, Inc.
NFS is a trademark of Sun Microsystems.
AVS is a trademark of Advanced Visual Systems, Inc.
PostScript is a registered trademark of Adobe Systems Incorporated.
Sun, SunOS, Sun-4, SPARC, and SPARCstation are trademarks of Sun Microsystems, Inc.

MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale, California 94086
1-800-526-0916
1-408-736-9560 (FAX)
MasPar® commands invoke compilers and other tools that are specific to the MasPar system. This manual provides copies of the man pages for the most often used commands for the MasPar system. It also provides some introductory explanation of some of the commands.

How to Use This Manual

You should read Chapter 1 before you use any of the MasPar commands, especially the compilers. It contains some information that is not in the individual man pages.

Chapter 2 contains details about hierarchical and compile-time profiling, and tells you how to use the command line options to get profiling and print out reports.

The rest of the manual contains copies of man pages, including those for the MasPar compilers and programming environment, commands to check the status of the DPU job queue, commands to manage MPL (MasPar Programming Language) libraries, and other MasPar commands.

Prerequisites

You must know how to use the front end machine (the DECstation 5000) and how to use ULTRIX.
If you are new to the MasPar system, you should read the *MasPar System Overview* manual first to get an idea of how the system works. Since these MasPar commands help you compile and debug data parallel programs and manage libraries of data parallel routines, you should know something about data parallel programming in general and about MPF (MasPar Fortran) and MPL (MasPar Programming Language) in particular. See the *MasPar Fortran User Guide* and the *MasPar Fortran Reference Manual* for more information on MPF. See the *MPL User Guide* and the *MPL Reference Manual* for more information on MPL.

### For More Information

You may want to refer to the following manuals for more information.

**For:**

- Background information on the MasPar MP-1 system and its architecture
  
  See:
  
  *MasPar System Overview.*
  
  PN 9300-0100.

- MasPar Architecture Specification.
  
  PN 9300-5001.

- Information on MasPar system commands
  
  See:
  
  *MasPar Commands.*
  
  PN 9300-0300.

- Information on how to use MPPE
  
  See:
  
  *MPPE User Guide.*
  
  MPPE user guide.
  
  PN 9305-0000.

- A quick overview of the most used features of MPPE.
  
  See:
  
  *MPPE Quick Reference.*
  
  PN 9305-0300.

- Information on MasPar's optional libraries
  
  See:
  
  *MPDDL Reference Manual.*
  
  PN 9302-0200.

  
  PN 9302-0300.

  
  PN 9302-0400.
For:
Information on MasPar's programming languages and compilers

See:
MPL User Guide.
(ANSI C version)
PN 9302-0101.

(ANSI C version)
PN 9302-0001.

MasPar Fortran User Guide
PN 9303-0100.

Complete information on administration or ULTRIX and your DECstation

ULTRIX-32 general information, volume 1, reader's guide, and master index.
Digital Equipment Corporation. 1988
Order number AA-ME82A-TE.

ULTRIX-32 software development, volume 1, guide to VAX C.
Digital Equipment Corporation. 1988
Order number AA-ME83A-TE.

ULTRIX worksystem software reference pages.
Digital Equipment Corporation. 1988
Order number AA-MA85A-TE.

ULTRIX worksystem software advanced installation guide.
Digital Equipment Corporation. 1988
Order number AA-KU43B-TE.

ULTRIX worksystem software DECwindows user's guide.
Digital Equipment Corporation. 1988
Order number AA-MA87A-TE.

You may also want to refer to the documentation for your window manager.
Help

To get general information about MasPar, including how to contact MasPar, type "man maspar" at the shell prompt.

To get on-line help on MasPar commands, compilers, and library routines, type "man -k maspar" or "apropos maspar" at the shell prompt to see what MasPar man pages are available.

To get more general help on how to program the MasPar parallel processing system, study the examples in $MP_PATH/examples.

If you have additional questions, or if you want to report defects or make comments on MasPar products and documentation, you can contact MasPar via electronic mail, telephone, or FAX at the following locations:

North America and the Pacific Rim:
Customer Support
MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale
California
94086
U.S.A.

phone: 1-800-526-0916 (8-6 PST)
FAX: 1-408-736-9560
e-mail: support@maspar.com

Europe and the United Kingdom:
Customer Support
MasPar Computer Ltd.
8 Commerce Park
Brunel Road
Theale RG7 4AB
Berkshire
England

phone: +44-734-305444 (9-5 BST)
FAX: +44-734-305505
e-mail: support@mpread.co.uk

Typographic Conventions

The following conventions are used throughout this book:

This is used: To show:

*text in italics* program variables, new terms

text in typewriter font user input, items the user selects, examples, path

names, command names

Initial Caps on text names of tools, windows, panes, and menus

"text in quotation marks" literals
# Table of Contents

## Preface
- How to Use This Manual ........................................... i
- Prerequisites .......................................................... i
- For More Information ............................................... ii
- Help ................................................................. iv
- Typographic Conventions .......................................... iv

## Chapter 1. Overview ................................................. 1-1

- Environment Setup .................................................. 1-1
  - Setting MP_PATH .................................................. 1-1
  - Accessing the DPU ............................................... 1-2

- Compiling Your MasPar Programs ................................ 1-3
  - MasPar Compiler Drivers ...................................... 1-3
  - Linking With Code That Uses the Global Pointer .......... 1-5
  - Memory Requirements .......................................... 1-5
  - Stack Overflow Error .......................................... 1-5
  - Program Requirements and Machine Size .................. 1-6

- Invoking MasPar Compilers (Makefiles) ....................... 1-6
- ULTRIX make With .c.o Rule Omitted ......................... 1-7

- Libraries .......................................................... 1-7
  - Math and C Libraries ........................................ 1-7
  - User-Defined Libraries ....................................... 1-8
  - Run-Time Libraries .......................................... 1-8
  - SIGEMT ......................................................... 1-9
Chapter 2. Analyzing Profile Information 2-1

What is Compile-Time Profiling? 2-2
What is Execution Profiling? 2-2
   Selecting Execution Profile Type 2-2
   Why Use Hierarchical Profiling? 2-3
   Varying The Level of Hierarchical Profiling 2-4
   About Profile Quality 2-4

Getting Profiling Information 2-5
Printing Profiling Information 2-5

About the .CT file 2-6
   Summary Section 2-9
   Statement Detail Section 2-9
   About the Postscript File 2-10
   About the ASCII file 2-13
   About The Columns 2-14
   About The Rows 2-16

Chapter 3. man Pages 3-1
Chapter 1

Overview

This chapter discusses:

- the environment settings you need to know about before you can run any of the MasPar® tools
  
- the MasPar compilers

Environment Setup

All commands and tools in the MasPar system use the MP_PATH environment variable. MP_PATH specifies the path to the top of the MasPar tree on your system. This section explains how the MP_PATH variable works on the MasPar system.

NOTE: Two other environment variables, MP_DBTARGET and MP_NODEBUG are used by MPPE; see the MPPE User Guide for information on setting those environment variables.

Setting MP_PATH

The MP_PATH environment variable points to the top of the MasPar tree. MP_PATH is set to provide a convenient way to access files in the MasPar tree (such as examples). If MP_PATH is not set the MasPar software uses the default path of /usr/maspar; this should be the correct path for most systems.
If you need to set your MP_PATH, you should set it in your .cshrc file.

If you wish, you can set the MP_PATH conditionally:

```bash
if(! $?MP_PATH) then
  setenv MP_PATH /usr/maspar
  set path = ($MP_PATH/bin $path)
endif
```

You will have problems if you set MP_PATH unconditionally in your .cshrc file. This is because your .cshrc file is executed every time you invoke a new cshell, and you may not always be aware of when you are invoking a new cshell. For example, you invoke a cshell when you invoke a user program under the MPPE because the Stdin/Stdout window is a cshell. If you set MP_PATH at the command prompt and then invoke a user program under the MPPE, any value of MP_PATH that is set unconditionally in your .cshrc file will override the value you set at the command prompt.

There may be reasons that you should not set it unconditionally. See the information in the following section.

If you have MP_PATH set, you should set your path variable so that it includes $MP_PATH/bin. If you do not have MP_PATH set, you should set your path variable so that it includes /usr/maspar/bin.

Reasons that you or your system administrator may want to set or reset MP_PATH include:

- You want to use $MP_PATH as a convenient way to access files in the MasPar directory tree such as demos and examples. In this case, you should set a default MP_PATH conditionally in your .cshrc file.

- The workstation you are logged into is not the one that is directly connected to the MasPar machine. Consult your system administrator.

- The MasPar software was not installed where the Installation Manual suggests it should be installed (under /usr/maspar). Consult your system administrator.

- You have more than one version of the MasPar software installed, and you want to point to different versions at different times. In this case, you should set a default MP_PATH conditionally in your .cshrc file and then reset it as necessary at the command prompt. You should also consult your system administrator in case the two versions are not completely compatible and you have to take additional steps to switch between them.

**Accessing the DPU**

You can invoke the MPPE on a DECstation or a SPARCstation. You can display the MPPE on most X terminals on the network, where “X terminal” is any terminal running standard MIT X11 window software. See Appendix B of the MPPE User Guide for details.
A user program that calls code that runs on the DPU can only be invoked on the workstation that is directly connected to the DPU. Many MasPar tools, such as mpq and mpstat, can only be invoked on the workstation that is directly connected to the DPU. If you are using some other machine, then you must rlogin to the workstation that is directly connected to the DPU when you want to run tools and user programs that access the DPU.

MasPar program composition tools such as the compilers and library management tools can be invoked from anywhere on the network, as long as your MP_PATH is set correctly to find them (see above).

You may notice some delay in executing a program that uses the DPU or in starting the MPPE on such a program if another job is already using the DPU or if your job gets swapped out. You can check the DPU job queue by using the mpq command. You can set job swapping by using the dpumanager and mptimelimit commands.

## Compiling Your MasPar Programs

This section presents general introductory information about the MasPar compilers. For detailed information on each command, refer to the man page sections that follow this chapter. Note that there are two families of MPL compiler drivers: the mpl* drivers invoke the ANSI C-based MPL compilers, and the ompl* drivers invoke the K&R C-based MPL compilers.

### MasPar Compiler Drivers

The MasPar system includes the following compilers: mpfortran, mpl_f77, mpl_cc, mpl, ompl_f77, ompl_cc, and ompl. For information on MPF (MasPar Fortran), see the *MasPar Fortran User Guide* and the *MasPar Fortran Reference Manual*. For information on ANSI C-based MPL (MasPar Programming Language), see the *MPL User Guide* and the *MPL Reference Manual*.

- **mpfortran**: Use this driver if your main routine is written in MPF (MasPar Fortran). This driver compiles MPF and MasPar assembler.
- **mpl_f77**: Use this driver if your main routine is written in DEC Fortran. This driver compiles DEC Fortran, ANSI C-based MPL, DEC assembler, and MasPar assembler. Remember that all files involved in the same compile and link must have unique filenames, even if they have different file extensions. The MasPar mpl_f77 compiler calls the DEC f77 compiler. Note that f77 does not automatically come with your DECstation. You must purchase the f77 compiler and documentation separately from DEC.
- **ompl_f77**: This driver is the same as the mpl_f77 driver except that it invokes the K&R C-based MPL compiler instead of the ANSI C-based MPL compiler.
mpl_cc

Use this driver if your main routine is written in ULTRIX C. This driver compiles ULTRIX C, ANSI C-based MPL, DEC assembler, and MasPar assembler. Remember that all files involved in the same compile and link must have unique filenames, even if they have different file extensions. The mpl_cc compiler calls the ULTRIX cc compiler.

ompl_cc

This driver is the same as the mpl_cc driver except that it invokes the K&R C-based MPL compiler instead of the ANSI C-based MPL compiler.

mpl

Use this driver if your main routine is written in ANSI C-based MPL. This driver compiles ANSI C-based MPL and MasPar assembler.

ompl

This driver is the same as the mpl driver except that it invokes the K&R C-based MPL compiler instead of the ANSI C-based MPL compiler.

NOTE

Source files whose names differ only in their extensions (such as src.f, src.c, and src.m) each yield an object file of that same name (src.o in this example) when compiled. You must give unique filenames to all files involved in the same compile and link so that the resulting object files do not overwrite each other.

To write code that can be compiled with either the mpl* or ompl* drivers, you can #ifdef uses of the new MPL features (such as function prototypes), checking for the predefined macro __STDMPL__. The mpl* drivers set __STDMPL__; the ompl* drivers do not. In the following example, the function prototype will be used when you compile with the mpl* drivers, and it will not be used when you compile with the ompl* drivers:

```c
#include <stdio.h>

int main(void)
{
  printf("Hello, World!");
  return 0;
}
```

```c
#define __STDMPL__
extern double funyet(float a, double w; plural int having)
#endif else
extern double funyet()
#endif
```
Linking With Code That Uses the Global Pointer

The global pointer is a feature of the DECstation software that permits addressing external small data values with two-byte offsets from the global pointer, instead of with four-byte absolute addresses, thereby saving program text space. Code that uses the global pointer in this way must be compiled with -G 8.

All MasPar drivers compile with -G 0 because a two-byte offset is too small to address the DPU. Any code that shares data with MPL code must be compiled with -G 0. Any code that shares data with code that was compiled with -G 8 must be compiled with -G 8 because it may not be able to link with code that was compiled with -G 0.

You may get link error messages such as G P affect overflow if you try to link codes that have been compiled with different -G options.

If you have a program that calls MPL code and calls code (such as a library) that was compiled with -G 8, and if that program references data from both the -G 0 code and the -G 8 code, then you need to separate out the part of the -G 8 code that interacts with the -G 0 code and compile that portion separately using one of the MasPar drivers or using -G 0 explicitly, or you need to separate out the part of the -G 0 code that interacts with the -G 8 code and compile that portion separately using cc or f77. Otherwise, the link may fail with a global pointer offset overflow diagnostic.

If you have this situation, segregate the application code as follows:

- Code that communicates with MPL through routines such as callRequest(), blockIn(), or blockOut() must be compiled with -G 0. This is the default for mpl_cc and mpl_f77.

- Code that contains external data definitions that specify non-zero initializations for variables referenced in code that was compiled with -G 8 must be compiled with -G 8. This is the default for cc and f77. It may be necessary to move some data initializations to get them into modules that do not communicate with MPL.

- The remainder of the code can be compiled either way.

Memory Requirements

This section describes some of the errors you may encounter related to your program’s memory requirements. The MasPar system contains commands that help you determine how much memory your program will need and allow you to request that a particular amount of memory be allocated to your program.

Stack Overflow Error

If your program gets a stack overflow error, you should use mplimit to increase the memory allocation for the program.
The `dpu.manager` provides a large enough partition (up to the machine size) for a job's static memory allocation (static data and bss). However, the job could still get a stack overflow. You may need to use `mplimit` to make sure the job gets enough PE stack and heap space, since the `dpu.manager` cannot predict that usage. The memory allocation cannot be dynamically increased, so it is important that a job request an adequate quantity.

The `mplimit` command writes the DPU usage requirements you specify to the object file header for the named executable. These values tell the `dpu.manager` what the anticipated resource requirements are for the program. The `mpsize` command may be useful in determining what the memory requirements are for a program. A program needs at least as much PE memory as the sum of its PE data and PE bss sections. If it allocates PE memory at runtime, it may need even more. You can use the `mpconfig` command to determine the total amount of CMem and PMem on your machine.

Note that the PMem limit in `mplimit` includes a small amount of system overhead. Therefore, the amount of user allocatable PMem is actually slightly less than the amount you specify to `mplimit`. In this release, the amount of system overhead is 384 bytes. The amount of PMem requested by `dpu.manager` for a job is the smallest multiple of the partition size that accommodates the amount you specified using `mplimit` or the amount of static memory required, whichever is larger. The `mplimit` command will accept a value that is larger than the total amount of PMem available; in that case, the `dpu.manager` will allocate the entire machine to that job.

**Program Requirements and Machine Size**

If your program displays messages about program requirements not matching the machine size, it is likely that the swapping and memory requirements are out of conformance. The `mplimit` and `mpsize` utilities allow you to reset the swapping and memory requirements for your program. Refer to the man pages for more information about these utilities.

**Invoking MasPar Compilers (Makefiles)**

The following examples use `mpl_f77`, but the same examples also apply to the other compilers.

You can invoke the compilers from the shell prompt as follows:

```sh
mpl_f77 DECfortran_module.f MPL_module.m
```

Remember that all files involved in the same compile and link must have unique filenames, even if they have different file extensions.

Alternatively, you may want to put commands such as the following into a makefile:

```sh
mpl_f77 -c DECfortran_module.f
mpl_f77 -c MPL_module.m
mpl_f77 DECfortran_module.o MPL_module.o
```
You can set up the dependencies in the makefile so that only the modules that have changed since the last time you compiled this program will be compiled this time. This can save you a lot of compile time in a large, modular program. For more information on make, enter "man make" at the command line, or see Volume 1 of the ULTRIX Software Development documentation or Part 3 of Volume 2 of the ULTRIX Supplementary Documents (complete references are in the Preface). Sample makefiles are also available in $MP_PATH/examples.

The make utility does not provide default compilation rules for .m (MPL) and .S (MasPar assembler) files. To process these files correctly, you should add lines such as the following to your MasPar makefiles:

```
.m.o:
   mpl -c $<

.S.o:
   mpl -c $<
```

You should also set "CC" equal to "mpl_cc" and set "FC" equal to "mpl_f77" in your MasPar makefiles. If you do this, all .c files processed by this makefile will be compiled with mpl_cc instead of cc and all .f files processed by this makefile will be compiled with mpl_f77 instead of f77. This should not be a problem since mpl_cc calls cc and mpl_f77 calls f77.

The primary differences between mpl_cc and cc and between mpl_f77 and f77 are different defaults, such as debug information, the addition of MasPar libraries and include file directories, a different linker that produces a MOFF (MasPar Object File Format) file, and the fact that the MasPar compilers process the .m and .S files correctly.

**ULTRIX make With .c.o Rule Omitted**

If you use a makefile with no .c.o rule, with CC=mpl_cc, you get -O on the mpl_cc compile line. The ULTRIX make puts the -O on the compile line if the CFLAGS variable is undefined. This means that you will be unable to use MPPE on the resulting executable.

**Libraries**

This section contains information on what libraries are linked by default and how to link a user-defined library, and it contains a warning about using names for global variables that conflict with the names of section 2 or section 3 routine names.

**Math and C Libraries**

The math (-lm) and C (-lc) libraries are automatically linked in. It is not necessary to specify them on the command line. If you do specify these libraries on the command line, the compile will run a little slower because these libraries will be scanned twice.
User-Defined Libraries

To link user-defined libraries, use the `mpar` and `mpranlib` commands.

Run-Time Libraries

The run-time libraries linked with all MPL programs make use of a number of system library functions. These include:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>access(2)</td>
<td>abort(3)</td>
</tr>
<tr>
<td>execve(2)</td>
<td>alloca(3)</td>
</tr>
<tr>
<td>ioctl(2)</td>
<td>exit(3)</td>
</tr>
<tr>
<td>lseek(2)</td>
<td>fprintf(3)</td>
</tr>
<tr>
<td>read(2)</td>
<td>fputs(3)</td>
</tr>
<tr>
<td>sigvec(2)</td>
<td>getenv(3)</td>
</tr>
<tr>
<td></td>
<td>index(3)</td>
</tr>
<tr>
<td></td>
<td>malloc(3)</td>
</tr>
<tr>
<td></td>
<td>perror(3)</td>
</tr>
<tr>
<td></td>
<td>putenv(3)</td>
</tr>
<tr>
<td></td>
<td>signal(3)</td>
</tr>
<tr>
<td></td>
<td>sprintf(3)</td>
</tr>
<tr>
<td></td>
<td>sscanf(3)</td>
</tr>
<tr>
<td></td>
<td>strcat(3)</td>
</tr>
<tr>
<td></td>
<td>strncmp(3)</td>
</tr>
<tr>
<td></td>
<td>strlen(3)</td>
</tr>
</tbody>
</table>

Note that the above list is not necessarily complete. Some of the section 3 routines make additional calls that may conflict. As a general rule, you should avoid using names that conflict with section 2 or section 3 library routine names.

The system calls `fork(2)`, `vfork(2)`, and `execve(2)` should be used only with caution in programs that will run on the MasPar DPU.

- You should not use `fork` or `vfork` in a DPU job unless they intend to immediately `exec` a non-DPU program. Failure to follow this rule may result in anomalous system behavior.
- The DPU code from only one `exec` file can be used in a DPU job. The behavior of a process that `execs` a second MasPar DPU program is undetermined.

SIGEMT

The signal SIGEMT (signal number 7) is used by the MasPar run-time environment to report DPU halt and fault conditions. You should not use SIGEMT for other purposes.
Overview
Chapter 2

Analyzing Profile Information

This chapter explains:
- what profiling is
- when and why to use the different types of profiling
- how to get profiling information for your program on-screen
- how to print out reports of the profiling data

You can get information on how efficient your data parallel program is by using command line options, or when debugging your program within MPPE. Profiling helps you to understand where bottlenecks occur in your program. If you know where your program contains a lot of overhead (for example, places where there is a lot of data being passed between the DPU and the front end), then you can insert directives or make other changes to minimize the overhead.

There are two types of profiling available on the MasPar system: a static or compile-time analysis is provided before the program is run, and a dynamic or execution analysis is presented as the program is executed.

NOTE: Compile-time profiling information is only available for MasPar Fortran programs. Execution profiling is available for both MPL and Fortran programs.
What is Compile-Time Profiling?

Compile-time analysis shows the *estimated* amounts of resources your Fortran program will use, based on an analysis of the code performed at compile time. The analysis of the program is based on an internal system of *weights*. The weights are arbitrary numbers assigned to various time-intensive operations. Higher numbers are assigned to operations that typically require more time. The compile-time profile weights for typical programs come from data communications operations, such as array sloshing, scalar sloshing, router operations, and xnet operations.

The compile-time profile for a routine or statement does not take into account the variation in execution time for an operation depending on array size. For instance, all array sloshes are given the same weight. The compile-time profile also does not take looping into account. An array slosh gets the same weight regardless of how many times a routine gets called from a given call site.

What is Execution Profiling?

Execution profiling tells you how much time your MPL or MasPar Fortran program actually spends executing each routine and statement, based on the number of *ticks* each routine and statement accrues. You must execute a statement or routine before the execution profiling information is available.

Each time your program receives a UNIX clock interrupt, two ticks are added for each statement and routine that are currently executing—one tick for the front-end and one for the ACU. (Each tick is approximately 2 milliseconds of clock time.) Execution profiling provides you with a statistical way to search for *hot spots* in programs—you can use the records of the ticks to spot areas that could be optimized.

**NOTE:** Because ticks are accumulated in a sampling procedure, if a statement is very simple, it might execute without accumulating any ticks. In addition, the same statement may accumulate a different number of ticks in different runs of the program, due to statistical variations.

Selecting Execution Profile Type

There are two types of execution profiling: flat and hierarchical.

The type of profile you get for a routine depends on how it has been compiled. You get a hierarchical profile if the routine has been compiled for hierarchical profiling (`-hprofile` option), and a flat profile otherwise (`-nohprofile` option).

The default for MasPar Fortran and MPL is to be compiled for hierarchical profiling. You can specify `-nohprofile` to have MasPar Fortran programs compiled with flat profiling.
For a hierarchical profile, the two ticks are credited not only to the executing statement and routine, but also to the lowest level hierarchically profiled routine that called the executing flat profiled routine.

For a flat profile, the two ticks are credited only to the routine and statement executing at the time of the clock interrupt. Routines compiled with flat profiling do not include the time spent in calls to other routines.

For example, consider four routines that call each other as follows:

A()  
calls B()  
calls C()  
calls D()

These routines have flat profile ticks flatA, flatB, flatC, and flatD.

If the A() and B() routines are compiled with hierarchical profiling, then the profile ticks are counted as follows:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Profiling Ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A()</td>
<td>hierA = flatA</td>
</tr>
<tr>
<td>B()</td>
<td>hierB = flatB + flatC + flatD</td>
</tr>
<tr>
<td>C()</td>
<td>flatC</td>
</tr>
<tr>
<td>D()</td>
<td>flatD</td>
</tr>
</tbody>
</table>

Only routine B() has its profile affected by hierarchical profiling.

The –nohprofile and –hprofile options only affect MPL and MasPar Fortran code—there is no provision for hierarchical profiling of front-end C and Fortran 77 routines. MasPar system libraries are compiled for flat profiles.

**Why Use Hierarchical Profiling?**

Hierarchical profiling is useful for finding where your program is incurring overhead for both library routines and your own routines. A MasPar Fortran program can spend significant time in array sloshing (that is, copying arrays between the ACU and front-end).

With a flat profile, the execution profile does not show where in the program the sloshing takes place. With hierarchical profiling, the sloshing at the entry to a routine shows up at the FUNCTION or SUBROUTINE statement of the routine, and the sloshing at the end of a routine shows up at its END statement.

With a flat profile, a MasPar Fortran programmer has no indication of whether the profile for a statement containing an intrinsic includes the time for the intrinsic or not. If the intrinsic is coded in-line, the profile includes the intrinsic. However, if the intrinsic is encoded with a library call, the profile for the statement does not include the time for the
intrinsic. Normally, the programmer does not know which routines are encoded which way. With hierarchical profiling, the time for the library routine is included on the line where the intrinsic appears. Therefore, with hierarchical profiling, the MasPar Fortran intrinsic time always shows up on the statement.

A Fortran programmer can also make explicit calls to libraries, such as the MPML and MPDDL libraries. Hierarchical profiling shows the time spent in those calls. Users can choose to have their own code treated like system libraries in this respect by simply omitting hierarchical profiling when compiling the code.

The situation is similar for MPL. MPL does not have intrinsics as such, but it does have hidden routines (for example, a routine to do scalar divide), and it has explicit library calls, such as tan().

**Varying The Level of Hierarchical Profiling**

You can use hierarchical profiling at various levels in your program. You can start by compiling all of your code with hierarchical profiling. Then as you fine-tune your lowest-level routines, you can turn off hierarchical profiling on them. This results in the time for these low-level routines being added to the profiles for the remainder of your routines. You can continue turning off the profiles at various levels until you are at your main routine. This approach allows you to make maximum use of hierarchical profiling throughout the tuning phase of your development.

**NOTE:** You should not turn off the hierarchical profiling on middle level routines before turning it off for the low level routines. Doing this causes a portion of the time from the middle level routines to be added to the high level routines, making their profiles very difficult to interpret.

**About Profile Quality**

Usually, as your program executes longer and gets more ticks, the profile becomes a more accurate reflection of where time is being spent in your program. However, there can be non-statistical errors in the profile due to the fact that ACU sampling is done at front-end clock interrupts. The sampling at front-end clock interrupts can cause errors in two ways:

- It can exaggerate the effect of any accidental synchronization between the front-end and the ACU. Such synchronization can be caused by the ACU waiting for the front-end because the ACU needs front-end servicing during front-end clock interrupt processing.

- The front-end sampling of the ACU can overemphasize the state where the ACU is stalled waiting for the front-end during clock processing.

If you suspect such errors in the profile of your program, you can check the quality of the profile using information in the ASCII printout of the profile information (see page 2-13 for more information on ASCII printouts of profile data).
One clue that your program may contain such errors is a variation in the profile from run to run that is bigger than what you would expect from statistical variation. Another clue is a statement that has an extremely low profile for no apparent reason.

The programs that are most susceptible to these errors are MPL programs and any MasPar Fortran programs that have arrays less than or equal to machine size. Errors are less likely in MasPar Fortran with arrays that have a high virtualization ratio.

The profile errors tend to be less pronounced in routine profiles than in statement profiles. Also, they are less likely in MasPar Fortran routine profiles than in MPL routine profiles.

Getting Profiling Information

If you would like to see the compile-time messages displayed on screen as you are compiling, you can use these mpfortran options:

- `-report`
- `-threshold=<n>`
- `-blocked`
- `-storage`

If you also specify `-V` with any of the above options, you get a `.lis` file that has the compile-time information collated into the source listing. These options are explained in detail in the mpfortran man page, later in this manual.

To get the compile-time information from a `.o` or `.out` file (which you can display in batch mode), you use `mprof -noexecute <file>`. These messages are saved in a `.ct` file, the format of which is described in the following sections. You can also use the `-threshold=<n>`, `-blocked`, and `-storage` options on mprof when creating your `.ct` file. (`mprof` is explained in detail in the man page later in this manual.) In addition, you can print the `.ct` file from within MPPE (see the `MPPE User Guide`).

Both compile-time and execution profiling information can be viewed in MPPE. See the `MPPE User Guide` for more information.

Printing Profiling Information

You can print profiling information from within MPPE, by using the `mprof` command line option, or by asking the mpfortran compiler to display compile-time information.

You can print your profiling information for your current program in three formats: the compile-time profile is saved only in ASCII format, but the execution profiling
information is saved as both a Postscript file and as an ASCII file. The three files have the same name as the executable file, but the ASCII execution profile has a .as suffix, the Postscript execution profile has a .ps suffix, and the compile-time information is saved in a file with a .ct suffix. The following sections explain the differences between the formats. Although both execution formats show the number of ticks and the percentage of total ticks for the program, the Postscript file shows source code that isn’t available in the ASCII file, and the ASCII file provides details about the profiling that aren’t available in the Postscript output.

The Postscript file can be printed on any printer that prints Postscript files; the two ASCII files can be printed, or the data could be imported and used in another program.

For instructions on how to get printed profiling information in MPPE, see the MPPE Users Guide.

About the .CT file

The .ct file contains the compile-time analysis messages for a relocatable (.o) or executable (.out) MOFF file.

The file is divided into two sections, a summary section, that shows counts of messages by message category for each routine, and a statement detail section, that lists the individual messages for each source line.
<table>
<thead>
<tr>
<th>WEIGHT</th>
<th>SUBTOTAL</th>
<th>FREQUENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTRIBUTERROR (halftone.f)</td>
<td>504</td>
<td>total</td>
</tr>
<tr>
<td>DISTRIBUTERROR (halftone.f)</td>
<td>300</td>
<td>3 ASL: array sloshing to/from DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR (halftone.f)</td>
<td>188</td>
<td>188 SSL: scalar sloshing to/from DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR (halftone.f)</td>
<td>16</td>
<td>8 XOP: xnet operations</td>
</tr>
<tr>
<td>FINDSCAPEGOATS (halftone.f)</td>
<td>220</td>
<td>total</td>
</tr>
<tr>
<td>FINDSCAPEGOATS (halftone.f)</td>
<td>200</td>
<td>2 ASL: array sloshing to/from DPU</td>
</tr>
<tr>
<td>FINDSCAPEGOATS (halftone.f)</td>
<td>10</td>
<td>10 SSL: scalar sloshing to/from DPU</td>
</tr>
<tr>
<td>FINDSCAPEGOATS (halftone.f)</td>
<td>8</td>
<td>4 XOP: xnet operations</td>
</tr>
<tr>
<td>FINDSCAPEGOATS (halftone.f)</td>
<td>2</td>
<td>2 AEA: array element access on DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE (halftone.f)</td>
<td>156</td>
<td>total</td>
</tr>
<tr>
<td>PIXELAVERAGE (halftone.f)</td>
<td>132</td>
<td>132 SSL: scalar sloshing to/from DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE (halftone.f)</td>
<td>24</td>
<td>12 XOP: xnet operations</td>
</tr>
<tr>
<td>DIFFUSEDOTS (halftone.f)</td>
<td>24</td>
<td>total</td>
</tr>
<tr>
<td>DIFFUSEDOTS (halftone.f)</td>
<td>24</td>
<td>scalar sloshing to/from DPU</td>
</tr>
<tr>
<td>MAKEMASK (halftone.f)</td>
<td>16</td>
<td>total</td>
</tr>
<tr>
<td>MAKEMASK (halftone.f)</td>
<td>16</td>
<td>scalar sloshing to/from DPU</td>
</tr>
<tr>
<td>ENHANCEDGES (halftone.f)</td>
<td>9</td>
<td>total</td>
</tr>
<tr>
<td>ENHANCEDGES (halftone.f)</td>
<td>9</td>
<td>scalar sloshing to/from DPU</td>
</tr>
<tr>
<td>LOADIMAGE (halftone.f)</td>
<td>8</td>
<td>total</td>
</tr>
<tr>
<td>LOADIMAGE (halftone.f)</td>
<td>8</td>
<td>scalar sloshing to/from DPU</td>
</tr>
<tr>
<td>DOTDIFFUSION (halftone.f)</td>
<td>2</td>
<td>total</td>
</tr>
<tr>
<td>DOTDIFFUSION (halftone.f)</td>
<td>2</td>
<td>scalar sloshing to/from DPU</td>
</tr>
</tbody>
</table>

**STATEMENT DETAIL:**

<p>| DISTRIBUTERROR (halftone.f:127) | 100 | 1 ASL: #21 |
| DISTRIBUTERROR (halftone.f:127) | 100 | array slosh from DPU arg 1 in DIFFUSEDOTS (halftone.f:49) |
| DISTRIBUTERROR (halftone.f:127) | 100 | to FE arg ACTIVE in DISTRIBUTERROR() |
| DISTRIBUTERROR (halftone.f:127) | 100 | array slosh from DPU arg 6 in DIFFUSEDOTS (halftone.f:49) |
| DISTRIBUTERROR (halftone.f:127) | 100 | to FE arg A in DISTRIBUTERROR() |
| DISTRIBUTERROR (halftone.f:127) | 100 | array slosh from DPU arg 3 in DIFFUSEDOTS (halftone.f:49) |
| DISTRIBUTERROR (halftone.f:127) | 100 | to FE arg ERROR in DISTRIBUTERROR() |
| DISTRIBUTERROR (halftone.f:127) | 6 | 6 SSL: #392 |
| DISTRIBUTERROR (halftone.f:127) | 2 | scalar slosh from the DPU |
| DISTRIBUTERROR (halftone.f:131) | 13 | 13 SSL: #391 |
| DISTRIBUTERROR (halftone.f:131) | 12 | scalar slosh to the DPU |
| DISTRIBUTERROR (halftone.f:131) | 12 | scalar slosh from the DPU |
| DISTRIBUTERROR (halftone.f:131) | 12 | scalar slosh to the DPU |
| DISTRIBUTERROR (halftone.f:131) | 12 | scalar slosh from the DPU |
| DISTRIBUTERROR (halftone.f:132) | 11 | 11 SSL: #391 |
| DISTRIBUTERROR (halftone.f:132) | 2 | scalar slosh to the DPU |
| DISTRIBUTERROR (halftone.f:132) | 2 | xnet variable distance |
| DISTRIBUTERROR (halftone.f:134) | 13 | 13 SSL: #391 |
| DISTRIBUTERROR (halftone.f:134) | 9 | 9 SSL: #392 |
| DISTRIBUTERROR (halftone.f:134) | 9 | scalar slosh to the DPU |
| DISTRIBUTERROR (halftone.f:134) | 9 | xnet variable distance |
| DISTRIBUTERROR (halftone.f:135) | 11 | 11 SSL: #391 |
| DISTRIBUTERROR (halftone.f:135) | 9 | scalar slosh to the DPU |
| DISTRIBUTERROR (halftone.f:135) | 9 | scalar slosh from the DPU |</p>
<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>LINE</th>
<th>OPERANDS</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:135)</td>
<td>2 XOP:</td>
<td>#302</td>
<td>xnet variable distance</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:137)</td>
<td>13 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:137)</td>
<td>12 SSL:</td>
<td>#392</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:137)</td>
<td>2 XOP:</td>
<td>#302</td>
<td>xnet variable distance</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:138)</td>
<td>12 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:138)</td>
<td>11 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:138)</td>
<td>1 XOP:</td>
<td>#302</td>
<td>xnet variable distance</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:140)</td>
<td>13 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:140)</td>
<td>9 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:140)</td>
<td>2 XOP:</td>
<td>#302</td>
<td>xnet variable distance</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:141)</td>
<td>11 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:141)</td>
<td>9 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>DISTRIBUTERROR</td>
<td>(halftone.f:141)</td>
<td>2 XOP:</td>
<td>#302</td>
<td>xnet variable distance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>LINE</th>
<th>OPERANDS</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINDSCAPEGOATS</td>
<td>(halftone.f:102)</td>
<td>100 ASL: #21</td>
<td></td>
<td>array slosh from DPU arg 1 in MAKEMASK(halftone.f:98) to FE arg TEMPLATE in FINDSCAPEGOATS()</td>
</tr>
<tr>
<td>FINDSCAPEGOATS</td>
<td>(halftone.f:102)</td>
<td>100 ASL: #21</td>
<td></td>
<td>array slosh from arg 4 in DIFFUSEDOTS(halftone.f:42) thru arg SCAPEGOATS in MAKEMASK(halftone.f:98) to FE arg SCAPEGOATS in FINDSCAPEGOATS()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>LINE</th>
<th>OPERANDS</th>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:55)</td>
<td>4 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:59)</td>
<td>1 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:60)</td>
<td>6 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:60)</td>
<td>1 SSL:</td>
<td>#391</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:61)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:61)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:62)</td>
<td>8 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:62)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:63)</td>
<td>40 SSL:</td>
<td>#391</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:63)</td>
<td>39 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:63)</td>
<td>8 XOP:</td>
<td>#302</td>
<td>xnet variable distance</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:67)</td>
<td>4 XOP:</td>
<td>#301</td>
<td>xnet constant distance</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:67)</td>
<td>4 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:67)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:68)</td>
<td>4 XOP:</td>
<td>#301</td>
<td>xnet constant distance</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:68)</td>
<td>4 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:68)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:69)</td>
<td>4 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:69)</td>
<td>4 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:69)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:69)</td>
<td>1 XOP:</td>
<td>#301</td>
<td>xnet constant distance</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:70)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:70)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>PIXELAVERAGE</td>
<td>(halftone.f:70)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>DIFFUSEDOTS</td>
<td>(halftone.f:36)</td>
<td>4 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>DIFFUSEDOTS</td>
<td>(halftone.f:36)</td>
<td>4 SSL:</td>
<td>#392</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>DIFFUSEDOTS</td>
<td>(halftone.f:42)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
<tr>
<td>DIFFUSEDOTS</td>
<td>(halftone.f:42)</td>
<td>1 SSL:</td>
<td>#391</td>
<td>scalar slosh to the DPU</td>
</tr>
<tr>
<td>DIFFUSEDOTS</td>
<td>(halftone.f:44)</td>
<td>1 SSL:</td>
<td>#392</td>
<td>scalar slosh from the DPU</td>
</tr>
</tbody>
</table>
Summary Section

To help you locate routines that may consume the most time, the summary section is ordered with the routines with the highest weight coming first. The weight for a routine is the sum of the weights of all the messages for that routine. The weight assigned to a message reflects the amount of time consumed by the effect described in the message. The weight computation does not take account of the looping in the program. Therefore, it may be necessary to ignore a high weight given to an initialization routine that contains time consuming operations which are executed only a few times.

For each routine in the summary, there is a list of categories, ordered highest weight category first. The weight for each category is the sum of the weights of the messages in that category, and the frequency for each category is the count of messages occurring in that category. In addition to the category name, each category has a three-character mnemonic to identify the category. These mnemonics are provided to assist in correlating the category totals with the individual messages in the statement detail section.

Statement Detail Section

For ease in correlation with the summary section, the statement detail section is organized by routine with the routines in the same order as in the summary section. The messages are ordered and identified by statement number within each routine.

Each message has weight, frequency, and category mnemonic values similar to those in the summary section. In addition each message has a message description and a message number. The message number is used for locating supplementary information about the
message in the mpfortran manuals. This supplementary information is generally hints for what to due to reduce the effect described in the message.

**About the Postscript File**

The .ps file is very similar to what you might see if you could print the Code pane and Profiles bar of the MPPE Execution Window. The .ps file shows the entire code with the related histograms and the number of ticks and percentage of total ticks:
halftone.f (Hierarchical Profile)

program DotDiffusion
  c Implementation of error diffusion halftoning from "Digital Halftones"
  c by Dot Diffusion," by Donald Knuth in ACM Transactions on Graphics,
  c vol. 6, no. 4, October 1987, pp. 245-256.
  parameter (imageX=84,imageY=84)
  dimension A(imageX,imageY), B(imageX,imageY)
  A = 0.0
  B = 0.0
  call LoadImage(A,imageX,imageY)
  call EnhanceEdges(A,imageX,imageY)
  call DiffuseDots(A,imageX,imageY,B)
  stop
end

subroutine LoadImage(A,x,y)
  integer :: x,y
  real, array(x,y) :: A
  cmpl mpl READ_FILE
  7 0 8
call READ_FILE(A)
  4 0 8
  A = transpose(A)
  return
end subroutine LoadImage

subroutine EnhanceEdges(A,x,y)
  integer :: x,y
  real, array(x,y) :: A,Abar
  call PixelAverage(A,x,y, Abar)
  2 0 8
  A = (A - 0.5 * Abar) / (1.0 - 0.5)
  return
end subroutine EnhanceEdges

subroutine DiffuseDots(A,x,y,B)
  integer :: x,y
  real, array(x,y) :: A,B, error
  integer, array(x,y) :: mask
  logical, array(x,y) :: active
  integer, array(84) :: escapegoats
  call MakeMask(mask,x,y, escapegoats)
  do i=1,84
  10 1 8
    active = mask == i
    16 2 8
    where (active .and. A < 0.5) B = 0.0
    20 2 8
    where (active .and. A >= 0.5) B = 1.0
  8 0 8
    error = A - B
    if (escapegoats(i) > 0) then
      10 1 8
        call DistributeError(active,escapegoats(i), error,x,y,A)
      end if
    end do
  return
end subroutine DiffuseDots

subroutine PixelAverage(A,x,y,Abar)
  integer :: x,y
  real, array(x,y) :: A,Abar
  integer, array(x,y) :: neighbors
  neighbors = 0
  neighbors(1:x-1,:) = 6  ! right and left edges
  neighbors(:,1:y-1) = 6  ! top and bottom edges
  neighbors(1:x-1,1:y-1) = 4  ! corners
  Abar = A + eshift(A,dim=1,shift=1)
  & + eshift(A,dim=2,shift=1)
  & + eshift(A,dim=2,shift=-1)
The summary pages show the hierarchically-profiled and flat-profiled information in separate listings. (The MPPE windows have hierarchical and flat profiles combined.)
### About the ASCII file

The ASCII profile output contains nine columns of detailed information:

<table>
<thead>
<tr>
<th>FE</th>
<th>ACU</th>
<th>max</th>
<th>only ACU</th>
<th>FE only</th>
<th>hprof total prcnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>194</td>
<td>10</td>
<td>12</td>
<td>442</td>
<td>178</td>
<td>285</td>
</tr>
<tr>
<td>p 29.5</td>
<td>1.5</td>
<td>1.8</td>
<td>67.2</td>
<td>27.1</td>
<td>43.3</td>
</tr>
</tbody>
</table>

#### routines with HIERARCHICAL profiling

| p 180 | 7 | 10 | 430 | 174 | 286 | 627 | 95.3 | HIERARCHICAL totals |
| p 27.4 | 1.1 | 1.5 | 65.3 | 26.4 | 43.5 | 95.3 | 95.3 | percents |

**Hierarchical Information for halftone**

<table>
<thead>
<tr>
<th>FE</th>
<th>ACU</th>
<th>max</th>
<th>only ACU</th>
<th>FE only</th>
<th>hprof total prcnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>126</td>
<td>6</td>
<td>6</td>
<td>128</td>
<td>150</td>
<td>132</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>162</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>62</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>34</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>18</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

| L 0 | 0 | 0 | 2 | 2 | 0 | 2 | 0.3 | 127 DISTRIBUTERROR() |
| L 18 | 0 | 0 | 10 | 16 | 18 | 28 | 4.3 | 131 DISTRIBUTERROR() |
| L 22 | 1 | 1 | 22 | 32 | 23 | 46 | 7.0 | 132 DISTRIBUTERROR() |
| L 8 | 1 | 1 | 8 | 12 | 9 | 18 | 2.7 | 134 DISTRIBUTERROR() |
| L 18 | 0 | 0 | 12 | 16 | 18 | 30 | 4.6 | 135 DISTRIBUTERROR() |
| L 18 | 0 | 0 | 20 | 24 | 18 | 38 | 5.8 | 137 DISTRIBUTERROR() |
| L 18 | 2 | 2 | 18 | 16 | 20 | 40 | 6.1 | 138 DISTRIBUTERROR() |
| L 12 | 2 | 2 | 16 | 14 | 14 | 32 | 4.9 | 140 DISTRIBUTERROR() |
| L 10 | 0 | 0 | 20 | 18 | 10 | 30 | 4.6 | 141 DISTRIBUTERROR() |
| L 2 | 0 | 0 | 0 | 0 | 2 | 2 | 0.3 | 143 DISTRIBUTERROR() |

| L 4 | 0 | 0 | 8 | 2 | 12 | 12 | 1.8 | 102 FINDSCAPEGOATS() |
| L 2 | 0 | 0 | 0 | 0 | 2 | 2 | 0.3 | 122 FINDSCAPEGOATS() |
| L 8 | 1 | 1 | 154 | 0 | 16 | 164 | 24.9 | 125 FINDSCAPEGOATS() |

| L 12 | 0 | 0 | 44 | 0 | 56 | 56 | 8.5 | 23 LOADIMAGE() |
| L 0 | 0 | 0 | 6 | 0 | 6 | 6 | 0.9 | 24 LOADIMAGE() |

| L 4 | 0 | 0 | 2 | 2 | 4 | 6 | 0.9 | 44 DIFFUSEDOTS() |
| L 6 | 0 | 0 | 16 | 6 | 6 | 22 | 3.3 | 45 DIFFUSEDOTS() |
| L 6 | 0 | 0 | 10 | 6 | 6 | 16 | 2.4 | 46 DIFFUSEDOTS() |
| L 2 | 0 | 0 | 6 | 6 | 2 | 8 | 1.2 | 47 DIFFUSEDOTS() |

| L 0 | 0 | 0 | 6 | 0 | 6 | 6 | 0.9 | 35 READ_FILE() |
| L 0 | 0 | 0 | 2 | 0 | 2 | 2 | 0.3 | 36 READ_FILE() |
| L 0 | 0 | 0 | 2 | 0 | 2 | 2 | 0.3 | 45 READ_FILE() |
| L 0 | 0 | 0 | 22 | 0 | 22 | 22 | 3.3 | 46 READ_FILE() |

| L 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0.2 | 55 PIXELAVERAGE() |
| L 0 | 0 | 1 | 2 | 0 | 3 | 0.5 | 62 PIXELAVERAGE() |
| L 0 | 0 | 1 | 2 | 0 | 2 | 4 | 0.6 | 63 PIXELAVERAGE() |
| L 0 | 0 | 0 | 2 | 0 | 0 | 2 | 0.3 | 68 PIXELAVERAGE() |
| L 0 | 0 | 0 | 12 | 0 | 0 | 12 | 1.8 | 73 PIXELAVERAGE() |

| L 6 | 0 | 0 | 4 | 2 | 6 | 10 | 1.5 | 95 MAKEMASK() |
| L 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0.2 | 31 ENHANCEDGES() |
| L 0 | 0 | 0 | 2 | 0 | 0 | 2 | 0.3 | 34 ENHANCEDGES() |
| L 2 | 0 | 0 | 0 | 0 | 2 | 2 | 0.3 | 15 DOTDIFFUSION() |
The .as file is in two sections—first is information for hierarchical routines, and then information for flat routines.

<table>
<thead>
<tr>
<th>routines with FLAT profiling</th>
</tr>
</thead>
<tbody>
<tr>
<td>F 36 0 0 44</td>
</tr>
<tr>
<td>F 0 0 38</td>
</tr>
<tr>
<td>F 0 0 34</td>
</tr>
<tr>
<td>F 0 0 32</td>
</tr>
<tr>
<td>F 0 0 26</td>
</tr>
<tr>
<td>F 10 0 0</td>
</tr>
<tr>
<td>F 4 2 0 0</td>
</tr>
<tr>
<td>F 0 0 6</td>
</tr>
<tr>
<td>F 0 0 4</td>
</tr>
<tr>
<td>F 0 0 4</td>
</tr>
<tr>
<td>F 0 0 4</td>
</tr>
<tr>
<td>F 0 0 4</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 2 0 0</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 2 0 0</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 2 0 0</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 0 0 2</td>
</tr>
<tr>
<td>F 0 0 0</td>
</tr>
<tr>
<td>F 0 0 0</td>
</tr>
</tbody>
</table>

Flat Profiling Information for halftone

Within these sections, the information is sorted into rows of similar information. The information on total counts and percents shows first, then information on symbol rows (ordered hottest first), and finally information on statement rows.

About The Columns

The first seven columns of information in the .as file display information not included in the .ps file.
The first column shows the letters F, L, P, p, U, and K. This column doesn't contain any profiling information; the letters are provided to make it easier for you to sort the ASCII file using other programs, such as awk scripts.

F  the information relates to Functions
L  The information relates to Lines
P  the information in these lines are Totals
p  the information in these lines are Percents
U  the information relates to unmatched ticks
K  this information relates to the Kernel

The next four columns show the profile counts broken down into FE only, ACU only, FE with ACU, and ACU with FE. If you add the numbers in any row for those 4 columns, your result should be the same as the number in that row for the Total column.

The next column indicates the number of counts that may be assigned to incorrect functions or statements due to interaction between the profile processing and the FE system clock processing. These "max clock error" counts indicate events where the ACU advanced to a stall state while the FE was held up by clock processing, as well as events counted after the ACU has become synchronized with the system clock. This number is a measure of possible error in the profile. Profile counts smaller than this value may be in error.

The hprof column has different meanings for hierarchical and flat profiles. In the hierarchical profiles, the hprof count is the number of Total counts that came from hierarchical profiling. In the flat profiles, the hprof count is the number of Total counts that went to hierarchical profiling. Ideally, the sum of the hprof counts in the two sections should be equal.

The Total column shows total ticks, and the Percents column shows a breakdown of the percent of ticks.

The final column tells you which category, function, or line the numbers in that row are attributed to.
About The Rows

The different symbol rows in the ASCII output are labeled in the final column.

| F | 126 | 6 | 6 | 128 | 150 | 132 | 266 | 40.4 | dbg DISTRIBUTERROR() |
| F | 14  | 1 | 1 | 162 | 2   | 30  | 178 | 27.1 | dbg FINDSCAPEGOATS() |
| F | 12  | 0 | 0 | 50  | 0   | 62  | 62  | 9.4  | dbg LOADIMAGE()       |
| F | 18  | 0 | 0 | 34  | 20  | 18  | 52  | 7.9  | dbg DIFFUSEDOTS()     |
| F | 0   | 0 | 0 | 32  | 0   | 32  | 32  | 4.9  | dbg READ_FILE()       |
| F | 2   | 2 | 18 | 0   | 3   | 22  | 3.3 | 1.5  | dbg PIXELAVERAGE()    |
| F | 6   | 0 | 0 | 4   | 2   | 6   | 10  | 1.5  | dbg MAKEMASK()        |
| F | 0   | 0 | 1 | 2   | 0   | 1   | 3   | 0.5  | dbg ENHANCEEDGES()    |
| F | 2   | 0 | 0 | 0   | 0   | 2   | 2   | 0.3  | dbg DOTDIFFUSION()    |

The symbol rows are labeled dbg, FEP, and DPU, where:

- **dbg rows** show summaries for functions with debug information. These rows may contain both FE and ACU counts.
- **FEP and ACU rows** show summaries for functions without debug information.

For halftone, the dbg information appears in the hierarchical routine section, and the FEP and ACU symbols appear in the flat routine section of the file.

Following the symbol rows are the statement rows for the symbols with debug information. The last few rows show FE and ACU kernel information and one row for unmatched symbols. (The unmatched symbols row should display zeros in all columns.)
Chapter 3

man Pages

The following man pages show some of the most often used MasPar commands. Most of the commands must be invoked from the shell command line.

<table>
<thead>
<tr>
<th>Man page name:</th>
<th>Discusses how to use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpar(1)</td>
<td>the MasPar archive and library maintainer</td>
</tr>
<tr>
<td>mpconfig(1)</td>
<td>the MasPar DPU configuration information</td>
</tr>
<tr>
<td>mpctl(1)</td>
<td>the MasPar job manager and job control interface</td>
</tr>
<tr>
<td>mpdc(1)</td>
<td>the MasPar desktop calculator</td>
</tr>
<tr>
<td>mpdecl(1)</td>
<td>compose MPL and C type declarations</td>
</tr>
<tr>
<td>mpfile(1)</td>
<td>MasPar file classifier</td>
</tr>
<tr>
<td>mpfortran(1)</td>
<td>MasPar Fortran compiler</td>
</tr>
<tr>
<td>mpi(1)</td>
<td>MasPar DPU configuration, network style</td>
</tr>
<tr>
<td>mpl(1)</td>
<td>the MasPar ANSI C-based MPL compiler</td>
</tr>
<tr>
<td>mpl_cc(1)</td>
<td>the MasPar MIPS C and ANSI C-based MPL compiler</td>
</tr>
<tr>
<td>mpl_f77(1)</td>
<td>the MasPar DEC Fortran and ANSI C-based MPL compiler</td>
</tr>
<tr>
<td>mpld(1)</td>
<td>the MasPar linker</td>
</tr>
<tr>
<td>mplimit(1)</td>
<td>print or set MasPar program’s DPU usage limits</td>
</tr>
<tr>
<td>mplorder(1)</td>
<td>determine relation for a MasPar object library</td>
</tr>
<tr>
<td>mpnm(1)</td>
<td>print MasPar program’s name list</td>
</tr>
<tr>
<td>Man page name:</td>
<td>Discusses how to use:</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>mppe(1)</td>
<td>the MasPar Programming Environment</td>
</tr>
<tr>
<td>mpprof(1)</td>
<td>the MasPar report and profile generation front end</td>
</tr>
<tr>
<td>mpq(1)</td>
<td>the MasPar job queue examination program</td>
</tr>
<tr>
<td>mpranlib(1)</td>
<td>the MasPar archive table of contents builder</td>
</tr>
<tr>
<td>mpsize(1)</td>
<td>print MasPar program’s sizes</td>
</tr>
<tr>
<td>mpstat(1)</td>
<td>print MasPar job accounting statistics</td>
</tr>
<tr>
<td>mpstrip(1)</td>
<td>the MasPar object file stripper</td>
</tr>
<tr>
<td>mpswopt(1)</td>
<td>print or set MasPar program’s PE-array size requirements</td>
</tr>
<tr>
<td>mpvast(1)</td>
<td>the MasPar VAST-2 driver</td>
</tr>
<tr>
<td>pcmp(1)</td>
<td>the MasPar parallel compare two files</td>
</tr>
<tr>
<td>vmestat(1)</td>
<td>print status of MasPar VMEFS files</td>
</tr>
<tr>
<td>dpumanager(8)</td>
<td>the MasPar DPU job manager daemon</td>
</tr>
<tr>
<td>mpd(8)</td>
<td>the MasPar system information daemon</td>
</tr>
<tr>
<td>mpshutdown(8)</td>
<td>terminate the MasPar DPU job manager</td>
</tr>
<tr>
<td>mptimelim(8)</td>
<td>set the MasPar DPU maximum time length</td>
</tr>
<tr>
<td>mpzap(8)</td>
<td>the MasPar Binary Zapper</td>
</tr>
</tbody>
</table>
NAME
mpar — MasPar archive and library maintainer

SYNTAX
mpar -key [ posname ] arfile name ...

DESCRIPTION
The MasPar archive command, mpar, maintains groups of MOFF (MasPar Object File Format) and extended COFF (MIPS ULTRIX) object files combined into a single archive file. The mpar command is used to create and update library files as they are used by the MasPar linker, mpld(1). You should use mpar for any library that contains MOFF files.

In the current implementation, mpar is the same as the ULTRIX archive command, ar(1).

SEE ALSO
ar(1), mpld(1), mplorder(1), mpranlib(1), mpar(5)
NAME
mpconfig — MasPar DPU configuration information

SYNTAX
mpconfig

DESCRIPTION
The mpconfig command prints out information about the DPU. This command can only be run from a machine attached to a MasPar DPU. The information generated is similar to the following:

MasPar DPU Model MP-1104 (64 rows, 64 columns)
ACU IMEM size: 1 MByte
ACU CMEM size: 128 KBytes
PE memory size: 16 KBytes
PVME in I/O slot #2 (8 MBytes)
MPDA unit #0: 8 data drives of type DK516-15
NAME
mpctl - MasPar job manager and job control interface

SYNTAX
mpctl [ option ] pid ...
mpctl [ option ] all
mpctl [ -sysslice ]

Description
The mpctl command is used to modify the dpumanager's default setting as well as the state of a job that is known to the dpumanager. It takes a list of switches and a list of process id’s (or the keyword "all" for all jobs). The user "root" can change any job, otherwise a user can affect only his own job. Options are listed below.

Options
-h Turn on jobs' "held" flag. A currently running job will be stopped and swapped out. Jobs will be not eligible for swapin as long as it stays in "held" state.
-nh Turn off the "held" flag.
-b Turn on jobs' "background" flag, reducing the priority to background priority level. Background jobs will be immediately swapped out if any regular job require their dpu memory.
-nb Turn off the "background" flag.
-s Turn on jobs' "swappable" flag. A "swappable" job will be swapped out if its dpu slice limit is exceeded and some other jobs require its dpu memory. This is default.
-ns Turn off the "swappable" flag. A not "swappable" job can't be swapped out into the disk.
-prio Manually change the priority of the job, therefore the job's position in the queue is changed. Must be in superuser mode.
-limit Set a maximum time limit on the job, in DPU seconds. After the time limit the job will be killed with a SIGXCPU signal. Periods where the job is waiting in queue or swapped out do not count. This cannot be greater than the system maximum time specified on the dpumanager command line if the user is not a super user.
-sysslice Change the default system time limit value. No command argument is required for this option. Must be in superuser mode. This command will not affect those jobs already started.
-swapout Manually swap the job out, do not care whether the job is swappable. The job will be swapped out immediately if it is not holding IORAM. Must be in superuser mode.
-swapin Manually swap the job in, do not care whether the job is background or finished its
mpctl (1)

swap slice time. The job will be swapped in immediately if there is enough PMEM space, or else it will wait for PMEM space at the top of the waiting queue. Must be in superuser mode.

-swapon
Disable the dpu swap facility. Absolutely no swap (in/out) will be taken place. Swapped-out jobs will stay in the disk until the facility is enabled again. Must be in superuser mode.

-swapon
Enable the dpu swap facility again. Must be in superuser mode.

Restriction
By default, when the swapping facility is enabled, a job is not "held", not "background", and "swappable". However, diagnostic jobs' flag can't be changed.

See also
dpumanager (8), swap (5), mpq (1)
NAME
mpdc — MasPar desktop calculator

SYNTAX
mpdc [ -Zq ]

DESCRIPTION
The mpdc command is an arbitrary precision arithmetic package. It differs from dc(1) in that it uses the MasPar DPU for its calculations and a number of features (strings, macros, and alternate bases) are not implemented.

The overall structure of mpdc is a stacking (reverse Polish) calculator. The following constructions are recognized:

number The value of the number is pushed on the stack. A number is an unbroken string of the digits 0-9. It may be preceded by an underscore _ to input a negative number. Numbers may contain decimal points.

+ — / * % ^
The top two values on the stack are added (+), subtracted (—), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

sx The top of the stack is popped and stored into a register named x, where x may be any character. If the s is capitalized, x is treated as a stack and the value is pushed on it.

lx The value in register x is pushed on the stack. The register x is not altered. All registers start with zero value. If the l is capitalized, register x is treated as a stack and its top value is popped onto the main stack.

d The top value on the stack is duplicated.

p The top value on the stack is printed. The top value remains unchanged.

f All values on the stack are printed.

q Exits the program.

X Replaces the number on the top of the stack with its scale factor.

v Replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

! Interprets the rest of the line as a UNIX command.

c All values on the stack are popped.

i The top value on the stack is popped and used as the number radix for further input.

k The top of the stack is popped, and that value is used as a non-negative scale factor: the appropriate number of places are printed on output, and maintained during multiplication, division, and exponentiation. Unlike dc(1), this program does not limit the scale to 99.

z The stack level is pushed onto the stack.
mpdc(1)

Z Replaces the number on the top of the stack with its length.

RESTRICTIONS
The size of values that can be represented is determined by the number of processors in the DPU: A maximum of 8 digits per PE can be stored, minus a few digits for overhead. A 2K DPU can handle about 16,000 digits. Warning: if an intermediate result has too many digits to be represented, on either side of the decimal point, the program may silently generate an incorrect result.

DIAGNOSTICS
'stack empty' for not enough elements on the stack to do what was asked.

SEE ALSO
dc(1)
NAME
mpdecl — Compose MPL and C type declarations

SYNTAX
mpdecl [-a | + | -p | -r] [-ci]
[[files ...] | explain ... | declare ... | cast ... | set ... | help ... | ? ...]

DESCRIPTION
mpdecl is a program for encoding and decoding MPL type-declarations. Since MPL is a
superset of C, mpdecl can be used for C type-declarations as well. The default language (or
with the -p option) is based on the pre-ANSI definition defined by Kernighan & Ritchie’s
The C Programming Language book; optionally, the language may be based on the (draft
proposed) X3J11 ANSI Standard (the -a option is used), or the language defined by the
Ritchie PDP-11 C compiler (the -r option is used).

mpdecl reads the named files for statements in the language described below. A
transformation is made from that language to MPL or pseudo-English. The results of this
transformation are written on standard output. If no files are named, or a filename of "-"
is encountered, standard input will be read. If standard input is coming from a terminal, (or the
-i option is used), a prompt will be written to the terminal before each line.

You can use mpdecl as you create an MPL program with an editor like vi(1) or emacs(1).
You simply type in the pseudo-English version of the declaration and apply mpdecl as a filter
to the line. (In vi(1), type ‘‘!!mpdecl<cr>’’.)

If the create program option —c is used, the output will include semi-colons after variable
declarations and curly brace pairs after function declarations.

COMMAND LANGUAGE
There are six statements in the language. The declare statement composes an MPL type-
declaration from a verbose description. The cast statement composes an MPL type-cast as
might appear in an expression. The explain statement decodes an MPL type-declaration or
cast, producing a verbose description. The help (or ?) statement provides a help message.
The quit (or exit) statement (or the end of file) exits the program. The set statement allows
the command line options to be set interactively. Each statement is separated by a semi-
colon or a newline.

The following grammar describes the language. In the grammar, words in "<>" are non-
terminals, bare lower-case words are terminals that stand for themselves. Bare upper-case
words are other lexical tokens: NOTHING means the empty string; NAME means a C
identifier; NUMBER means a string of decimal digits; and NL means the new-line or semi-
colon characters.

Some synonyms are permitted during a declaration: character → char, constant → const,
enumeration → enum, func → function, integer → int, ptr → pointer, ref → reference, ret →
returning, structure → struct, and vector → array.

<program> ::= NOTHING
| <program> <stmt> NL
<stmt> ::= NOTHING
| declare NAME as <decl>
| declare <decl>
mpdecl(1)

| cast NAME into <decl>
| cast <decl>
| explain <optstorage> <ptrmodlist> <type> <cdecl>
| explain <storage> <ptrmodlist> <cdecl>
| explain ( <ptrmodlist> <type> <cast> ) optional-NAME
| set <options>
| help | ?
| quit
| exit

<cdecl> ::= array of <decl>
| array NUMBER of <decl>
| function returning <cdecl>
| function ( <cdecl-list> ) returning <cdecl>
| <ptrmodlist> pointer to <cdecl>
| <ptrmodlist> pointer to member of class NAME <cdecl>
| <ptrmodlist> reference to <cdecl>
| <ptrmodlist> <type>

<cdecl> ::= <cdecl1>
| * <ptrmodlist> <cdecl>
| NAME :: * <cdecl>
| & <ptrmodlist> <cdecl>

<cdecl1> ::= <cdecl1> ( )
| <cdecl1> ( <castlist> )
| <cdecl1> [ ]
| <cdecl1> [ NUMBER ]
| ( <cdecl> )
| NAME

<cast> ::= NOTHING
| ( )
| ( <cast> ) ( )
| ( <cast> ) ( <castlist> )
| ( <cast> )
| NAME :: * <cast>
| * <cast>
| & <cast>
| <cast> [ ]
| <cast> [ NUMBER ]

<typename> ::= <typename> | <modlist>
| <modlist> <typename>
| struct NAME | union NAME | enum NAME | class NAME

<castlist> ::= <castlist> , <castlist>
| <ptrmodlist> <type> <cast>
| <name>

<cdecllist> ::= <cdecllist> , <cdecllist>
| NOTHING
| <name>
mpdecl(1)

| <adecl>  
| <name> as <adecl>
<typename> ::= int | char | double | float | void
<modlist> ::= <modifier> | <modlist> <modifier>
<modifier> ::= short | long | unsigned | signed | <ptrmod>
<ptrmodlist> ::= <ptrmod> <ptrmodlist> | NOTHING
<ptrmod> ::= const | volatile | noalias
<storage> ::= auto | extern | register | auto
<optstorage> ::= NOTHING | <storage>
<options> ::= NOTHING | <options>
| create | nocodecreate | interactive | nointeractive
| ritchie | preansi | ansi | cplusplus
| debug | nodebug | yydebug | noyydebug

**EXAMPLES**

To declare an array of pointers to functions like malloc(3), do

```
declare fptab as array of pointer to function returning pointer to char
```

The result of this command is

```
char *(*fptab[])
```

When you see this declaration in someone else's code, you can make sense out of it by doing

```
explain char *(*fptab[])
```

The proper declaration for signal(2), ignoring function prototypes, is easily described in

*mpdecl*’s language:

```
declare signal as function returning pointer to function returning void
```

which produces

```
void (*signal())
```

The function declaration that results has two sets of empty parentheses. The author of such a function might wonder where to put the parameters:

```
declare signal as function (arg1,arg2) returning pointer to function returning void
```

provides the following solution (when run with the —c option):

```
void (*signal(arg1,arg2))() { }
```

If you wanted to add in the function prototypes, the function prototype for a function such as

```
_exit(2)
```

would be declared with:

```
declare _exit as function (retval as int) returning void
```

```
giving

```
void _exit(int retval) { }
```

As a more complex example using function prototypes, signal(2) could be fully defined as:

3-13
Declare signal as function(x as int, y as pointer to function(int) returning void)
returning pointer to function(int) returning void

giving (with -c)
void (*signal(int x, void (*y)(int)))(int) { }

`mpdecl` can help figure out the where to put the "const" and "volatile" modifiers in
declarations, thus

declare foo as pointer to const int

gives
const int *foo

while

declare foo as const pointer to int

gives
int * const foo

**DIAGNOSTICS**
The declare, cast and explain statements try to point out constructions that are not supported
in MPL. In some cases, a guess is made as to what was really intended. In these cases, the
MPL result is a toy declaration whose semantics will work only in Algol-68. The list of
unsupported MPL constructs is dependent on which version of the C language is being used
(see the ANSI, pre-ANSI, and Ritchie options).

SEE ALSO
(draft proposed) ANSI National Standard X3J11

Section 8.4 of the C Reference Manual within *The C Programming Language* by B.
Kernighan & D. Ritchie.

**CAVEATS**
The pseudo-English syntax is excessively verbose.

There is a wealth of semantic checking that isn’t being done.

`mpdecl`’s scope is intentionally small. It doesn’t help you figure out initializations. It expects
storage classes to be at the beginning of a declaration, followed by the the const, volatile and
noalias modifiers, followed by the type of the variable. `mpdecl` doesn’t know anything about
variable length argument lists. (This includes the “/, . . .” syntax.)

`mpdecl` thinks all the declarations you utter are going to be used as external definitions.
Some declaration contexts in C allow more flexibility than this. An example of this is:

declare argv as array of array of char

where `mpdecl` responds with

Warning: Unsupported in C -- 'Inner array of unspecified size'
(maybe you mean "array of pointer")
char argv[][]
Tentative support for the *noalias* keyword has been put in because it is in the current ANSI specifications.
NAME
mpfile — MasPar file classifier

SYNTAX
mpfile [-c] [-f ffile] [-m mfile] filename ...

DESCRIPTION
The mpfile command performs a series of tests on each filename argument in an attempt to classify it. If an argument appears to be ASCII, the mpfile command examines the first 1024 bytes and tries to guess its language.

For character special files, part of this classification is information about which devices the system shows as active. In particular, device-specific information such as controller type and unit, device type and unit, and status (offline, write locked, density, errors) is returned. The general categories currently implemented are disk, tape, and terminal devices. The supported terminal devices include Local Area Terminals (LAT) but not Local Area Network (LAN) pseudo-terminals.

The mpfile command uses the file /usr/lib/file/magic to identify files that have some sort of magic number. A magic number is any numeric or string constant that identifies the file containing the constant. Commentary at the beginning of /usr/lib/file/magic explains its format.

In addition to the file types recognized by file(1), mpfile recognizes MOFF (MasPar Object File Format) and MPL (MasPar Programming Language) file types.

OPTIONS
-c          Checks the magic file for format errors by printing the internal representation of the magic file. No file typing is done under —c.
-f          Interprets the following argument to be a file containing the names of the files to be examined.
-m          Instructs mpfile to use an alternate magic file.

RESTRICTIONS
It often does a poor job of distinguishing C programs, shell scripts, English text, and ascii text.

It does not recognize many programming languages, including Modula, Pascal, and Lisp.

SEE ALSO
file(1)
NAME
mpfortran — MasPar Fortran compiler

SYNTAX
mpfortran [ options ... ] files ... [ options ... ]

DESCRIPTION
The mpfortran(1) command invokes the MasPar Fortran compiler for MP-1 systems. It can also be used to invoke the MasPar assembler and linker.

ARGUMENTS
The mpfortran command accepts the following types of arguments:

- Arguments whose names end with ".f", ".for", or ".FOR". These are taken to be MasPar Fortran source programs. The source code is compiled, and the resulting object code is put in a file whose name is the same as the name of the source file except ".o" is substituted for ".f", ".for", or ".FOR".

- Arguments whose names end with ".F". These are also taken to be MasPar Fortran source programs; these are first processed by the C preprocessor (cpp) before being compiled by mpfortran. (NOT YET IMPLEMENTED)

- Arguments whose names end with ".S". These are taken to be MasPar assembly source programs and are passed to the MasPar assembler, producing a .o file.

- Arguments whose names end with ".o" or ".a". These are taken to be object files and are passed to the link step.

Arguments other than those ending with ".f", ".F", ".for", ".FOR", ".S", ".o", or ".a" are treated as either: (1) Compiler or linker option arguments, or (2) object programs that were produced during an earlier compilation or were extracted from the libraries.

The compiler does not create a .o file when there is only one source file and the -c option is not specified. When there are multiple source files, the compiler always creates a .o file for each source file.

All assembly and object files produced by the mpfortran command, by default, contain debug information and compile-time analysis messages for subsequent use by MPPE and mmpprof(1). This information can be suppressed by using the -nodebug option.

OPTIONS
Options are specified on the command line as a single dash followed by an option string.

The following options are accepted by the mpfortran command:

-[-no]blocked
  Force the -report option, printing only compile-time analysis messages that indicate multilayer array mappings that might affect blocked algorithms. Ignore the -threshold option, and print messages of all levels. The default is -noblocked.

-c
  Suppress linking and produce .o files for each source file.

-continuations=n
  Where n is an integer from 0 to 99 specifying the number of continuation lines allowed in a source program statement. The default is 19 continuation lines.
mpfortran(1)

-[no]cross_reference
If the -V option is specified, include the numbers of the lines in
which symbols are defined and referenced in the storage map
section of the listing file. The default is -nocross_reference.

-nodebug
Do not generate debug information. This option is useful when you
have "out of memory" problems. However, you cannot use MPPE or
mpprof(1) on code compiled with this option.

-[no]d_lines
Compile lines with a "d" or "D" in column 1; do not treat them as
comment lines. The default is -nod_lines.

-double_precision
Use double precision as the default precision for REAL and
COMPLEX variables. Declarations of REAL and COMPLEX types
are compiled as DOUBLE PRECISION and DOUBLE COMPLEX.
(REAL*4 and COMPLEX*8 declarations are not affected by this
option.)

-[no]extend_source
Extend the range of statement text from columns 1-72 to columns
1-132. The default is -noextend_source.

-[no]fecommon
Make all COMMON data resident on the front end for common
blocks that contain both scalars and arrays (array operations on
these arrays are not allowed). This option can be useful during
initial porting of Fortran 77 programs to MasPar Fortran, to avoid
having to modify all common blocks that mix scalars and arrays at
once. The default is -fecommon; -nofecommon puts common array
data on the DPU. (See also the ONDPDU directive described in the
MPF Reference Manual.)

-[no]hprofile
Generate extra code to write hierarchical profile information for
analysis under MPPE or with mpprof(1). The default is -hprofile.
Use -nohprofile to suppress generation of this extra code. See the
MPPE manual set for details on hierarchical profiling.

-include=path
If an include file is not found in the current working directory,
search path for source include files, in the order specified.

-Ldir
Adds dir to the list of directories that are searched for libraries.
Directories specified with -L are searched before the standard
directories.

-Lstring
Searches for a library named string, which is a file named
libstring.a, where string is a string. The MasPar linker searches for
libraries first in any directories specified with -L options, then in the
standard directories. The first default library searched is
$MP_PATH/lib/maspar, if MP_PATH is set, or
/usr/maspar/lib/maspar, if MP_PATH is not set. The remaining
default libraries searched are /lib, /usr/lib, and /usr/local/lib. A
library is searched when its name is encountered, so the placement
of a -l in the compiler or linker command line is significant. The
math (-lm) and C (-lc) libraries are automatically linked in; it is not
necessary to specify them on the command line.
-o output

Name the final output file _output_ instead of a.out.

-O

Compile with optimization (the default). Some debugger features are not supported at this level of optimization. Note that in the current version of the compiler, -O yields the same result as -Omin.

-Omin

Compile with minimum optimization. This option allows the use of all debugger features. Note that in the current version of the compiler, -Omin yields the same result as -O.

-Omax

Compile with maximum optimization. This option provides the most optimization available, but none of the debugger features are supported at this level of optimization.

-pesize=integer

Specify the size of the DPU PE array for which code will be compiled; default is 1K PEs. Valid values for _integer_ are:

* 1, 2, 4, 8, or 16, meaning 1K, 2K, 4K, 8K, or 16K PEs.

-pevariable

Compile code for any size machine. Arrays in COMMON and arrays with the SAVE attribute cannot be used with this option. By default, the -pesize option is in effect.

-pmemsize=integer

Specify the size of the PE memory for error checking. By default, the amount of memory assumed in each PE is 64 KBytes. Any value of _integer_ greater than 16384 is treated as 64 KBytes; any value up to 16384 is treated as 16 KBytes. The compiler will verify that the known amount of PE storage used by any routine does not exceed the size specified; it will generate an error diagnostic if it does.

-[no]report

Print a report of the compile-time analysis messages of level 0 for each subprogram on standard output. See the -threshold option for more information on levels. If the the -V option is specified, include the messages in the listing file also. The default is -noreport.

-S

Compile programs and write output to .S files.

-save_all_variables

Retain values of all local arrays and variables after procedure return (as if specified in the SAVE statement). By default, local arrays and variables are not saved. This option is useful for debugging. Note that using this option has high costs at runtime and in DPU memory usage.

-save_fe_variable

The same effect as -save_all_variables, except it does not apply to values on the DPU (it retains front-end values only). As with -save_all_variables, this option is useful for debugging, but has high runtime and memory usage costs.

-[no]storage

Force the -report option, printing only compile-time analysis messages that indicate storage usage. Ignore the -threshold setting and print messages of all levels. The default is -nostorage.

-strip=schedule

Perform load/store scheduling optimizations. To have an effect, this
mpfortran(1)

The mpfortran compiler option must be used in conjunction with the -Omax option, which must be separately specified. The compiler does overlapped arithmetic and memory operations in strip loops (compiler-generated loops when the arrays being operated on are larger than the machine size).

-threshold=n

Force the -report option, printing only compile-time analysis messages with level less than or equal to n. The default is n=0. As a general rule, as the level number increases, the messages become less critical and might be more voluminous. Currently, the highest level used is 2.

-V

For each .f source file specified, create a .lis file containing a source listing. By default, no listing is generated.

-[no]v104

Use intrinsic arguments and syntax from the V1.0 MPF compiler, which was based on V104 of the draft Fortran 90 standard. All subsequent versions of the compiler use -nov104 by default; that is, the latest syntax and arguments of intrinsics that comply with the final ISO Fortran standard are used.

-Zq

Suppress the MasPar copyright and version number notice.

RESTRICTIONS

If the -Omax option is used, debug information is omitted; therefore the MasPar Programming Environment (MPPE) cannot be used.

Be aware that the MPF compiler does not support the DEC f77 compiler command-line options.

FILES

$MP_PATH/lib/mpl/dpu_crt0.o initialization code
/lib/cpp C preprocessor
/bin/cc C compiler
$MP_PATH/lib/mpf/mpf_io/libmpf_io.a I/O library
/usr/lib/lbm.a math library
/lib/libc.a C library

SEE ALSO

cc(1), cpp(1), mppe(1), mpprof(1), mpl(1), as(1)
MasPar Fortran User Guide
MasPar Fortran Reference Manual

3-22
NAME
mpi — MasPar DPU configuration information, network style, Version 2.2

SYNTAX
mpi [-Zq] [hostname]

DESCRIPTION
The mpi command prints out information about all DPU’s on any attached local area
network supporting SO_BROADCAST sockets and the broadcast address
INADDR_BROADCAST. The broadcast message is received by a daemon program,
maspard(8), running on all machines which have a DPU attached. Upon receipt of the
broadcast message, each daemon sends a message describing the DPU attached to the client
program, mpi. Mpi can be run on any machine in the network; even those without a DPU.
The information generated is similar to the following:

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>TYPE</th>
<th>PE</th>
<th>PMEM</th>
<th>CMEM</th>
<th>MODEL</th>
<th>UCODE</th>
<th>QUEUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>svsales</td>
<td>4.2RISC</td>
<td>64x64</td>
<td>65536</td>
<td>114688</td>
<td>1104</td>
<td>2.4.11</td>
<td>0</td>
</tr>
<tr>
<td>krusty</td>
<td>4.2RISC</td>
<td>32x32</td>
<td>16384</td>
<td>114688</td>
<td>1101</td>
<td>2.7.8</td>
<td>0</td>
</tr>
<tr>
<td>pigpen</td>
<td>4.2RISC</td>
<td>64x64</td>
<td>16384</td>
<td>114688</td>
<td>1204</td>
<td>2.6.228</td>
<td>0</td>
</tr>
<tr>
<td>lucy</td>
<td>4.2RISC</td>
<td>64x128</td>
<td>16384</td>
<td>114688</td>
<td>1208</td>
<td>2.6.179</td>
<td>1</td>
</tr>
<tr>
<td>linus</td>
<td>4.2RISC</td>
<td>32x32</td>
<td>16384</td>
<td>114688</td>
<td>1101</td>
<td>2.7.8</td>
<td>0</td>
</tr>
<tr>
<td>alpha2</td>
<td>4.2RISC</td>
<td>32x64</td>
<td>16384</td>
<td>114688</td>
<td>1202</td>
<td>2.6.238</td>
<td>1</td>
</tr>
</tbody>
</table>

DPU systems running MasPar software versions prior to 2.0 will supply the "hardware
option" (hwopt) value, while systems running 2.0 and later will supply the size of usable PE
memory (pmem). Also, the number of jobs in the queue is shown.

FILES
/usr/tmp/dpucfg

SEE ALSO
mpq(1)
NAME
mpl — MasPar ANSI C-based MPL compiler

SYNTAX
mpl [ options ... ] files ...

DESCRIPTION
The mpl(1) command invokes the MasPar MPL compiler. It can also be used to invoke utilities such as the MasPar assembler and linker. It produces MOFF (MasPar Object File Format) executable files, a.out relocatable files, or symbolic assembly language files.

ARGUMENTS
The mpl command accepts the following types of arguments:

• Arguments whose names end with .m. These arguments are treated as MPL source programs. The source code is compiled and the resulting object code is put in a file whose name is the same as the name of the source file except .o is substituted for .m. When you compile and load a single program in one step, mpl deletes the intermediate object code file.

• Arguments whose names end with .S. These arguments are treated as MasPar assembly source programs and are passed to the MasPar assembler, which produces a .o file.

Arguments other than those ending with .m or .S are treated as either: (1) Compiler or linker option arguments. (2) Object programs that were produced during an earlier compilation or were extracted from the libraries.

OPTIONS
The following options are accepted by the mpl command:

-b
Does not pass the -lc library to the linker.

-c
Suppresses the loading phase of the compilation and forces an object file to be produced even if only one program is compiled. Does nothing with input object files.

-Dname=def
-Dname
Defines name to the preprocessor. This functions as if an additional #define preprocessor directive were embedded in the source code. If no definition is given, the name is defined as "1". See also the -U option.

-E
Runs only the macro preprocessor on each .m file and sends the result to the standard output.

-Em
Tells the preprocessor to output a rule suitable for make describing the dependencies of each .m source file. For each .m source file, the preprocessor outputs one make rule whose target is the object file name for that source file and whose dependencies are all the files included in it with "#include". This rule may be a single line or may be continued with "\n"-newline if it is long. This option causes the preprocessor only to be run.

-g
Generate additional symbol table information for the MPPE (MasPar
mpl(1)

Programming Environment). This is the default. Also pass the -lg flag to
the link step.

-[no]hprofile

Generate extra code to write hierarchical profile information for analysis
under MPPE or with mprof(1). See the MPPE manual set for details on
hierarchical profiling. The default is -hprofile.

-I-

For directories specified with -Idir options before the -I- option, search
only for the case of #include file; do not search those directories for the
case of #include <file>. For directories specified with -Idir options after
the -I- option, search for all #include directives. In addition, inhibit the
use of the current directory (where the current input file came from) as the
first search directory for #include file. There is no way to override this
effect of -I-. You can use "-I." to specify searching the directory that was
current when the compiler was invoked. That is not exactly the same as
what the preprocessor does by default, but it is often satisfactory. The -I-
option does not inhibit the use of the standard system directories for
header files.

-Idir

Looks in the following directories for #include files whose names do not
begin with a directory specification (/): (1) the directory of the file
argument on the command line, (2) directories (dir) named in -I options,
and finally (3) directories in a standard list.

-Ldir

Adds dir to the list of directories that are searched for libraries.
Directories specified with -L are searched before the standard directories.

-Lstring

Searches for a library named string, which is a file named libstring.a,
where string is a string. The MasPar linker searches for libraries first in
any directories specified with -L options, then in the standard directories.
The first default library searched is $MP_PATH/lib/maspar, if MP_PATH
is set, or /usr/maspar/lib/maspar, if MP_PATH is not set. The remaining
default libraries searched are /lib, /usr/lib, and /usr/local/lib. A library is
searched when its name is encountered, so the placement of a -l in the
compiler or linker command line is significant. The math (-lm) and C (-lc)
libraries are automatically linked in; it is not necessary to specify them on
the command line.

-O

This option is provided for compatibility, but it does not do anything (it is
ignored). The mpl compiler does perform some optimizations, but these
are not user-selectable.

-o output

Names the final output file output instead of a.out. If this option is used,
the file a.out is left undisturbed. If the named file has a .o, .a, .c, .m, .f, .s,
or .S file extension, the following error message is displayed: "-o would
overwrite".

-S

Compiles programs and writes output to .S files. Do nothing with input
assembler source files or object files.

-Uname

Removes any initial definition of name to the preprocessor, as if by
#define. See also the -D option.
-W[no-]implicit Warn whenever a function is implicitly declared (whenever a function is referenced before it is declared). The default is -Wimplicit.

-Wreturn-type Warn whenever a function is defined with a return type that defaults to int. Also warn about any return statement with no return value in a function whose return type is not void.

-Wunused Warn whenever a local variable is unused aside from its declaration, whenever a function is declared static but never defined, and whenever a statement computes a result that is explicitly not used.

-Wswitch Warn whenever a switch statement has an index of enumeration type and lacks a case for one or more of the named codes of that enumeration and lacks a default label. Also warn whenever a switch statement has an index of enumeration type and has case labels outside the enumeration range.

-Wcomment Warn whenever a comment-start sequence (/*) appears inside a comment.

-Wtrigraphs Warn if any trigraphs are encountered.

-Wall All of the above -W options combined. These are all the options that pertain to usages that are often considered to be poor programming practice. These usages should be easy to avoid, even in conjunction with macros. The remaining -W options described below are not included in -Wall because certain kinds of useful macros are very difficult to write without causing those warnings.

-Wshadow Warn whenever a local variable shadows another local variable. For example, in the following code fragment:

```c
f0()
{
   int i;
   /* i = .... */
   {
      int i;
      /* i = ... */
   }
}
```

the second i shadows the first, in that you cannot refer to the i in the outer block when in the inner block, even though it would be in scope if the second i were not declared.

-Wpointer-arithmetic Warn about anything that depends on the size of a function type or of void. MPL assigns these types a size of 1 for convenience in calculations with void * pointers and pointers to functions.

-Wcast-qual Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a const char * is cast to an ordinary char *.

-Wwrite-strings Warn whenever the address of a string constant is copied into a non-constant char * pointer. For example, in the following code fragment:

```c
f0()
{
   char *nonConstant;
   const char *constant;

   /* Unwise */
   char *str = const_cast<char*>(nonConstant);
```
mpl(1)

nonConstant = "abc";
Constant = "xyz";
nonConstant = Constant;
}

warning messages are given for lines 3 and 5 if you set this flag.

-w Suppresses all warning messages.
-Zn Do not turn on -g.
-Zq Suppress the MasPar copyright and version number notice.
-Zv Print the commands executed by the preprocessor, compiler, assembler, and linker. Also print resource usage in the C-shell time(1) format.

DEFAULT SYMBOLS
The mpl compiler recognizes the following predefined symbols. These symbols are assigned the value of ‘‘1’’. These symbols are useful in #ifdef statements to isolate code for a particular case. Thus, these symbols can be useful for ensuring portable code.

_MPI any MasPar system
_MPI1 any of the MasPar MP-1 Family
__STDMPL__ ANSI C features and MPL extensions

The mpl compiler recognizes the following predefined macros:

__DATE__ Evaluates to a string that specifies the date the program was compiled.
__FILE__ Evaluates to a string that specifies the current source file.
__LINE__ Evaluates to an integer that specifies the line that contains the macro reference.
__TIME__ Evaluates to a string that specifies the time the program was compiled.

RESTRICTIONS
None.

FILES
$MP_PATH/bin/mpl command program
$MP_PATH/lib/mpl/mpl libraries
$MP_PATH/include search path for #include files
$MP_PATH/include/mpl search path for #include files
$MP_PATH/log/mplcontrol disables mpl logging
/usr/tmp/mplXXXXXX.X temporary files

SEE ALSO
adb(1), as(1), cpp(1), dbx(1), ld(1), mpl_cc(1), mpl_f77(1), mplld(1), mpprof(1), prof(1), monitor(3)
MasPar Programming Language (MPL) User Guide
MasPar Programming Language (MPL) Reference Manual

3-28
NAME
mpl_cc — MasPar MIPS C and ANSI C-based MPL compiler

SYNTAX
mpl_cc [ options ... ] arguments ...

DESCRIPTION
The mpl_cc(1) command invokes the MIPS ucode C compiler, the MasPar MPL compiler, and the MasPar linker. It produces MOFF (MasPar Object File Format) executable files, MIPS extended COFF relocatable files, or symbolic assembly language files.

ARGUMENTS
The mpl_cc command accepts the following types of arguments:

- Arguments whose names end with .c. These arguments are treated as MIPS C source programs. The source code is compiled using cc(1), and the resulting object code is put in a file whose name is the same as the name of the source file except .o is substituted for .c. When you compile and load a single program in one step, mpl_cc deletes the intermediate object code file.

- Arguments whose names end with .m. These arguments are treated as MPL source programs. The source code is compiled and the resulting object code is put in a file whose name is the same as the name of the source file except .o is substituted for .m. When you compile and load a single program in one step, mpl_cc deletes the intermediate object code file.

- Arguments whose names end with .s. These arguments are treated as MIPS assembly source programs and are passed to the MIPS assembler, which produces a .o file.

- Arguments whose names end with .S. These arguments are treated as MasPar assembly source programs and are passed to the MasPar assembler, which produces a .o file.

- Arguments whose names end with .i. These arguments are interpreted as C source after being processed by the C preprocessor. They are compiled without being processed by the C preprocessor. Arguments other than those ending with .c, .m, .s, .S, or .i are treated as either: (1) Compiler or linker option arguments. (2) Object programs that were produced during an earlier compilation or were extracted from the libraries.

OPTIONS
The following options are accepted by the mpl_cc command:

-C For .c files, stops the macro preprocessor from omitting comments.

-c Suppress the loading phase of the compilation and force an object file to be produced even if only one program is compiled. Do nothing with input object files.

-[no]cpp Run the C macro preprocessor cpp(1) on .c and .s source files before compiling. The -cpp option is the default. The -nocpp option avoids running cpp on these files.

-Dname=def
mpl.cc(1)

-Dname  Defines name to the preprocessor. This functions as if an additional
#define preprocessor directive were embedded in the source code. If no
definition is given, the name is defined as "1". See also the -U option.

-E     Run only the C macro preprocessor on the source files (regardless of the
suffix), and send the result to the standard output. This sets the -cpp
option.

-Em    Tell the preprocessor to output a rule suitable for make describing the
dependencies of each .m or .c source file. For each .m or .c source file, the
preprocessor outputs one make rule whose target is the object file name
for that source file and whose dependencies are all the files included in it
with "#include". This rule may be a single line or may be continued with
"\n"-newline if it is long. This option causes the preprocessor only to be
run.

-g0    Do not produce symbol table information for symbolic debugging. Do not
use this option if you are going to execute this program under the MPPE.

-g1    Produce additional symbol table information. Provides accurate, but
limited symbolic debugging of partially optimized code. Do not use this
option if you are going to execute this program under the MPPE.

-g or -g2  Produce additional symbol table information for full symbolic debugging,
but do not perform optimizations that limit full symbolic debugging. This
is the default.

-g3    Produce additional symbol table information for full symbolic debugging
for fully optimized code. Do not use this option if you are going to
execute this program under the MPPE.

-[-no]hprofile  Generate extra code to write hierarchical profile information for analysis
under MPPE or with mpprof(1). Generates hierarchical profile
information for MPL code only. See the MPPE manual set for details on
hierarchical profiling. The default is -hprofile.

-I     Prevents cpp from searching for #include files in the standard directory
/usr/include.

-I-    For directories specified with -Idir options before the -I- option, search
only for the case of #include file; do not search those directories for the
case of #include <file>. For directories specified with -Idir options after
the -I- option, search for all #include directives. In addition, inhibit the
use of the current directory (where the current input file came from) as the
first search directory for #include file. There is no way to override this
effect of -I-. You can use "-I." to specify searching the directory that was
current when the compiler was invoked. That is not exactly the same as
what the preprocessor does by default, but it is often satisfactory. The -I-
option does not inhibit the use of the standard system directories for
header files.

-Idir  Looks in the following directories for #include files whose names do not
begin with a directory specification (/): (1) the directory of the file argument on the command line, (2) directories (dir) named in -I options, and finally (3) directories in a standard list.

-Ldir
 Adds dir to the list of directories that are searched for libraries. Directories specified with -L are searched before the standard directories.

-Istring
 Searches for a library named string, which is a file named libstring.a, where string is a string. The MasPar linker searches for libraries first in any directories specified with -L options, then in the standard directories. The first default library searched is SMP_PATH/lib/maspar, if MP_PATH is set, or /usr/maspar/lib/maspar, if MP_PATH is not set. The remaining default libraries searched are /lib, /usr/lib, and /usr/local/lib. A library is searched when its name is encountered, so the placement of a -l in the compiler or linker command line is significant. The math (-lm) and C (-lc) libraries are automatically linked in; it is not necessary to specify them on the command line.

-O
 This option is provided for compatibility, but it does not do anything (it is ignored). The mpl compiler does perform some optimizations, but these are not user-selectable.

-00
 Turn off all optimizations. This is the default.

-01
 Turn on all optimizations that complete fast. Do not use this option if you are going to execute this program under the MPPE.

-o output
 Name the final output file output. If this option is used, the file a.out is unaffected.

-P
 Run only the C macro preprocessor and put the result for each source file using suffix convention (for example, .c and .s) in a corresponding .i file. The .i file does not have number lines (#) in it. This sets the -cpp option.

-S
 Compiles programs and writes output to .s files for MIPS C and to .S files for MPL. Does nothing with input assembler source files or object files.

-signed
 Causes all char declarations to be signed char declarations in the C code in your program. This is the default.

-std
 Produce warnings for things that are not standard in the language.

-Unname
 Removes any initial definition of name to the preprocessor, as if by #undefine. See also the -D option.

-unsigned
 Causes all char declarations to be unsigned char declarations in the C code in your program.

-V
 Print the version of the driver and the versions of all passes of the MIPS C compiler. This is done with the what(1) command.

-varargs
 Prints warnings for lines in the C code in your program that may require the varargs.h macros.

-volatile
 Causes all variables to be treated as volatile in the C code in your
program.

-W[no-]implicit Warn whenever a function is implicitly declared (whenever a function is referenced before it is declared). The default is -Wimplicit.

-Wreturn-type Warn whenever a function is defined with a return type that defaults to int. Also warn about any return statement with no return value in a function whose return type is not void.

-Wunused Warn whenever a local variable is unused aside from its declaration, whenever a function is declared static but never defined, and whenever a statement computes a result that is explicitly not used.

-Wswitch Warn whenever a switch statement has an index of enumeration type and lacks a case for one or more of the named codes of that enumeration and lacks a default label. Also warn whenever a switch statement has an index of enumeration type and has case labels outside the enumeration range.

-Wcomment Warn whenever a comment-start sequence (/*) appears inside a comment.

-Wtrigraphs Warn if any trigraphs are encountered.

-Wall All of the above -W options combined. These are all the options that pertain to usages that are often considered to be poor programming practice. These usages should be easy to avoid, even in conjunction with macros. The remaining -W options described below are not included in -Wall because certain kinds of useful macros are very difficult to write without causing those warnings.

-Wshadow Warn whenever a local variable shadows another local variable. For example, in the following code fragment:

```c
f() { int i;
    /* i = .... */
    { int i;
      /* i = ... */
    }
}
```

the second i shadows the first, in that you cannot refer to the i in the outer block when in the inner block, even though it would be in scope if the second i were not declared.

-Wpointer-arith Warn about anything that depends on the size of a function type or of void. MPL assigns these types a size of 1 for convenience in calculations with void * pointers and pointers to functions.

-Wcast-qual Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a const char * is cast to an ordinary char *.

-Wwrite-strings Warn whenever the address of a string constant is copied into a non-constant char * pointer. For example, in the following code fragment:

```c
f() { char *nonConstant;
```
const char *Constant;
nonConstant = "abc";
Constant = "xyz";
nonConstant = Constant;
}
warning messages are given for lines 3 and 5 if you set this flag.

-w  Suppress all warning messages.
-Zn  Use the cc debug default.
-Zq  Suppress the MasPar copyright and version number notice.
-Zv  Print the commands executed by the preprocessor, compiler, assembler, and linker. Also print resource usage in the C-shell time(1) format.

If the environment variable COMP_HOST_ROOT is set, the value is used as the root directory for all pass names rather than the default slash (/). If the environment variable COMP_TARGET_ROOT is set, the value is used as the root directory for all include and library names rather than the default slash (/). This affects the standard directory for #include files, /usr/include, and the standard library, /usr/lib/libc.a. If this is set then the only directory that is searched for libraries, using the -lx option, is COMP_TARGET_ROOT/usr/lib.

If the environment variable TMPDIR is set, the value is used as the directory to place any temporary files rather than the default /tmp/ for the cc compiler.

Other arguments are assumed to be either loader options or C-compatible object files, typically produced by an earlier cc run, or perhaps libraries of C-compatible routines. These files, together with the results of any compilations specified, are loaded in the order given, producing an executable program with the default name a.out.

DEFAULT SYMBOLS

The mpl_cc command always defines the C preprocessor macro LANGUAGE_C when a .c file is being compiled. The mpl_cc command defines the C preprocessor macro
LANGUAGE.Assembly when a .s file is compiled.

The ULTRIX C compiler provides the following default symbols for your use. These symbols are useful in #ifdef statements to isolate code for one of the particular cases. Thus, these symbols can be useful for ensuring portable code.

unix any UNIX system
bsd4_2 Berkeley UNIX Version 4.2
ultrix ULTRIX only
mips any MIPS architecture
MIPSEL little endian variant of MIPS architecture
host_mips native compilation environment (as opposed to cross-compiler)

The mpl_cc compiler recognizes the following predefined symbols when a .m file is compiled. The symbols are all assigned the value of "1". These symbols are useful in #ifdef statements to isolate code for one of the particular cases. Thus, these symbols can be useful for ensuring portable code.
mpl_cc(1)

_MPL_                any MasPar system
_MPL1_               any of the MasPar MP-1 Family
__STDMPL__           ANSI C features and MPL extensions

The mpl_cc compiler recognizes the following predefined macros when a .m file is compiled:

__DATE__               Evaluates to a string that specifies the date the program was compiled.
__FILE__               Evaluates to a string that specifies the current source file.
__LINE__               Evaluates to an integer that specifies the line that contains the macro reference.
__TIME__               Evaluates to a string that specifies the time the program was compiled.

RESTRICIONS
The options -g0, -g1, -g3, and -O1 do not work with the MPPE (MasPar Programming Environment). Do not use these options if you are going to execute this program under the MPPE.

Linking with Code that Uses the Global Pointer
The global pointer is a feature of the DECstation software that permits addressing external small data values with two-byte offsets from the global pointer, instead of with four-byte absolute addresses, thereby saving program text space. Code that uses the global pointer in this way must be compiled with -G 0.

All MasPar drivers compile with -G 0 because a two-byte offset is too small to address the DPU. Any code that shares data with MPL code must be compiled with -G 0. Any code that shares data with code that was compiled with -G 8 must be compiled with -G 8 because it may not be able to link with code that was compiled with -G 0. You may get a link error message such as "GP offset overflow" if you try to link codes that have been compiled with different -G options.

If you have a program that calls MPL code and calls code (such as a library) that was compiled with -G 8, and if that program references data from both the -G 0 code and the -G 8 code, then you need to separate out the part of the -G 8 code that interacts with the -G 0 code and compile that portion separately using one of the MasPar drivers or using -G 0 explicitly, or you need to separate out the part of the -G 0 code that interacts with the -G 8 code and compile that portion separately using cc or f77. Otherwise, the link may fail with a global pointer offset overflow diagnostic.

If you have this situation, segregate the application code as follows:

- Code that communicates with MPL through routines such as callRequest(3), blockIn(3), or blockOut(3) must be compiled with -G 0. This is the default for mpl_cc and mpl_f77.
- Code that contains external data definitions that specify non-zero initializations for variables referenced in code that was compiled with -G 8 must be compiled with -G 8. This is the default for cc and f77. It may be necessary to move some data initializations to get them into modules that do not communicate with MPL.
The remainder of the code can be compiled either way.

**FILES**

file.c
file.m
file.o
a.out
/tmp/ctm?
/usr/lib/cpp
/usr/lib/ccom
/usr/lib/ujoin
/usr/bin/uld
/usr/lib/usplit
/usr/lib/merge
/usr/lib/uopt
/usr/lib/ugen
/usr/lib/as0
/usr/lib/as1
/usr/lib/crt0.o
/usr/lib/mcrt0.o
/usr/lib/libc.a
/usr/lib/libprof1.a
/usr/include
/usr/bin/ld
/usr/lib/fioc
/usr/lib/cord
/usr/bin/btou
/usr/bin/utob
mon.out

$MP_PATH/log/mpllogcontrol disables mpl_cc logging

**SEE ALSO**
cpp(1), dbx(1), ld(1), mpl(1), mpl_f77(1), mplld(1), mpprof(1), pixie(1), prof(1), what(1),
monitor(3)

*MasPar Programming Language (MPL) User Guide*

*MasPar Programming Language (MPL) Reference Manual*
NAME

mpl_f77 — MasPar DEC Fortran and ANSI C-based MPL compiler

SYNTAX

mpl_f77 [ options ... ] files ...

DESCRIPTION

The mpl_f77(1) command invokes the DEC Fortran (f77(1)) compiler, the MasPar MPL compiler, and the MasPar linker. It produces MOFF (MasPar Object File Format) executable files, MIPS extended COFF relocatable files, or symbolic assembly language files. The mpl_f77 command will not work unless f77 is installed on your system.

ARGUMENTS

The mpl_f77 command accepts the following types of arguments:

- Arguments whose names end with .f, .for, or .FOR. These arguments are treated as DEC Fortran source programs. The source code is compiled using f77 and the resulting object code is put in a file whose name is the same as the name of the source file except .o is substituted for .f, .for, or .FOR. When you compile and load a single program in one step, mpl_f77 deletes the intermediate object code file.

- Arguments whose names end with .m. These arguments are treated as MPL source programs. The source code is compiled and the resulting object code is put in a file whose name is the same as the name of the source file except .o is substituted for .m. When you compile and load a single program in one step, mpl_f77 deletes the intermediate object code file.

- Arguments whose names end with .s. These arguments are treated as DEC assembly source programs and are passed to the DEC assembler, which produces a .o file.

- Arguments whose names end with .S. These arguments are treated as MasPar assembly source programs and are passed to the MasPar assembler, which produces a .o file. MasPar assembly language modules must not contain any front end code.

- Arguments whose names end with .o. These arguments are treated as DEC Fortran object files and are passed to the link step.

- Arguments whose names end with .a. These arguments are treated as libraries and are passed to the link step.

- Arguments whose names end with .F. These arguments are also treated as DEC Fortran source programs. These files are first processed by the C preprocessor, then compiled by mpl_f77.

- Arguments whose names end with .i. These arguments are treated as DEC Fortran source programs that have been preprocessed by cpp(1). They are compiled without further preprocessing.

- Arguments whose names end with .e. These arguments are treated as EFL source programs. These files are first processed by the EFL preprocessor, then compiled by mpl_f77.

- Arguments whose names end with .r. These arguments are treated as Ratfor source programs. These files are first processed by the Ratfor preprocessor, then compiled by
mpl_f77(1)

mpl_f77.

When a source program requires preprocessing by cpp, efl(1), ratfor(1), or m4(1), the name of the output file generated by the preprocessor consists of the basename of the source input file with the appropriate preprocessor suffix. For example:

```
cpp myfile.F  => myfile.i
efl myfile.e  => myfile.f
ratfor myfile.r => myfile.f
m4 myfile.r   => myfile.p
```

Note that if you have myfile.F and myfile.f as distinct files, cpp overwrites myfile.f to produce an output file with the same name.

The mpl_f77 command always defines the cpp macros mips and host_mips to cpp. If the -cpp option is specified, mpl_f77 defines the cpp macro LANGUAGE_FORTRAN when an .f, .for, .F, .FOR, .r, or .e file is compiled.

If the environment variable DECFFORT is set, the value is used as the name of the f77 compiler to invoke. If the environment variable TMPDIR is set, the value is used as the directory for temporary files for f77.

OPTIONS

You can override some options specified on the command line by using the OPTIONS statement in your DEC Fortran source program. An OPTIONS statement affects only the program unit where the statement occurs. For more information, see the DEC Fortran Language Reference Manual.

The mpl_f77 command takes the following options:

```
-1    Executes at least one iteration of DO loops. (DEC Fortran DO loops are not executed if the upper limit is smaller than the lower limit.) This option has the same effect as -onetripl.

-66   Allows extensions that enhance FORTRAN-66 compatibility (same as the -nof77 option).

-automatic Places local variables on the run-time stack. This option sets the -assume recursive option. The default is -static.

-C    For .f, .F, .for, and .FOR files, generates code to perform run-time checks on subscript and substring expressions (same as the -check_bounds option). An error is reported if the expression is outside the dimension of the array or the length of the string. The default suppresses range checking.

-c    Suppresses the loading phase of the compilation and forces an object file to be produced even if only one program is compiled. Does nothing with input object files.

-check_bounds Generates code to perform run-time checks on subscript and substring expressions. This option has the same effect as -C.

-col72 Treats the statement field of each source line as ending in column 72 (same as the -noextend_source option).
```
-[no]cpp

Runs `cpp` on all `.F` and `.s` source files before compiling. This includes FORTRAN source files created by `ratfor` or `eef`. The `-cpp` option is the default. The `-nocpp` option avoids running `cpp` on these files.

-Dname=def
-Dname

Defines `name` to the preprocessor. This functions as if an additional `#define` preprocessor directive were embedded in the source code. If no definition is given, the name is defined as "1". See also the `-U` option.

-d_lines

Compiles lines having a D in column 1 of the source program. The default is to treat such lines as comments.

-E

Uses any remaining characters in the argument as `eef` options when processing a `.e` file. The temporary file, used as the output of `eef`, consists of the last component of the source file with an `.f` substituted for the `.e`. This temporary file is removed unless you specify the `-K` option.

-Em

Tells the preprocessor to output a rule suitable for `make` describing the dependencies of each `.m` source file. For each `.m` source file, the preprocessor outputs one `make` rule whose target is the object file name for that source file and whose dependencies are all the files included in it with "#include". This rule may be a single line or may be continued with "\n"-newline if it is long. This option causes the preprocessor only to be run.

-[no]extend_source

Treats the statement field of each source line as ending in column 132, instead of column 72. The default is `-noextend_source`: Treats the statement field of each source line as ending in column 72 of the source program.

-F

Applies the `eef` and `ratfor` preprocessors to relevant files and puts the result in files whose names have their suffix changed to `.f`. (No `.o` files are created.)

-fpe0 or -fpe

Terminates a program during run time if a floating-point operation results in overflow, a division by zero, or invalid data; before termination, the compiler issues a message, and creates a core dump file. In the case of floating-point underflow, the program does not terminate, but continues with the underflow value set to zero. (The compiler issues a warning message if `-check underflow is set`). This is the default.

-fpe1

Continues program execution if a floating-point operation results in overflow, a division by zero, invalid data, or floating-point underflow (the underflow value is set to zero, and the compiler issues a warning message if `-check underflow is set`).

-fpe2

Continues program execution if a floating-point operation results in overflow, a division by zero, invalid data, or floating-point underflow, and prints a warning message in each instance. In the case of floating-point underflow, the underflow value is set to zero. Upon program completion, DEC Fortran provides a count of how many times each exception
-fpe3  Continues program execution if a floating-point operation results in overflow, a division by zero, invalid data, or floating-point underflow (the compiler issues a warning message if -check underflow is set).

- fpe4  Continues program execution if a floating-point operation results in overflow, a division by zero, invalid data, or floating-point underflow, and prints a warning message in each instance. Upon program completion, DEC Fortran provides a count of how many times each exception occurred.

-g0  Prevents symbolic debugging information from appearing in the object file.

-g1  Produces traceback information (showing pc to line correlation) in the object file, substantially increasing its size.

-g2 or -g  Produces traceback and symbolic debugging information in the object file. This is the default. This option sets the -O0 option, unless you specify an explicit -O option. In this case, the specified -O option is effective, and you see an informative message.

-g3  Produces traceback and symbolic debugging information in the object file, and performs whatever optimizations you specify. This option can produce additional debugging information describing the effects of optimizations, but debugging inaccuracies can occur as a result of the optimizations that have been performed.

-[no]hprofile  Generate extra code to write hierarchical profile information for analysis under MPPE or with mpprof(1). Generates hierarchical profile information for MPL code only. See the MPPE manual set for details on hierarchical profiling. The default is -hprofile.

-I  Prevents cpp from searching for #include files in the standard directory /usr/include.

-I-  For directories specified with -Idir options before the -I- option, search only for the case of #include file; do not search those directories for the case of #include <file>. For directories specified with -Idir options after the -I- option, search for all #include directives. In addition, inhibit the use of the current directory (where the current input file came from) as the first search directory for #include file. There is no way to override this effect of -I-. You can use "-I." to specify searching the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory. The -I- option does not inhibit the use of the standard system directories for header files.

-Idir  Looks in the following directories for #include files whose names do not begin with a directory specification (/): (1) the directory of the file argument on the command line, (2) directories (dir) named in -I options,
and finally (3) directories in a standard list.

-i2
Makes default integer and logical variables 2 bytes long (same as the -noi4 option). The default is -i4.

-[no]i4
Makes default integer and logical variables 4 bytes long. The -i4 option is the default. The -noi4 option is the same as the -i2 option.

-Ldir
Directs the linker to search for libraries in dir before searching the standard directories.

-lstring
Searches for a library named string, which is a file named libstring.a, where string is a string. The MasPar linker (mplCd(1)) searches for libraries first in any directories specified with -L options, then in the standard directories. The first default library searched is $MP_PATH/lib/maspar, if MP_PATH is set, or /usr/maspar/lib/maspar, if MP_PATH is not set. The remaining default libraries searched are /lib, /usr/lib, and /usr/local/lib. A library is searched when its name is encountered, so the placement of a -l in the compiler or linker command line is significant. The math (-lm) and C (-lc) libraries are automatically linked in; it is not necessary to specify them on the command line.

-m
Applies the m4 preprocessor to each eff or ratfor source file before transforming it with the ratfor or eff preprocessors. The temporary file used as the output of m4 consists of the last component of the source file with a .p substituted for the .e or .r. This temporary file is removed unless you specify the -K option.

-nof77
Allows extensions that enhance FORTRAN-66 compatibility.

-O
This option is provided for compatibility, but it does not do anything (it is ignored). The mpl compiler does perform some optimizations, but these are not user-selectable.

-O0
Disables all optimizations. Does not check for unassigned variables. This is the default.

-O1
Enables local optimizations and recognition of common subexpressions. The call graph determines the order of compilation of procedures.

-o output
Names the final output file output. The a.out file is unaffected.

-onetrip
Executes at least one iteration of DO loops. (DEC Fortran DO loops are not executed if the upper limit is smaller than the lower limit.) This option has the same effect as -1.

-P
Runs only cpp and puts the result for each source file, by suffix convention (for example, .f, .r, .e, and .s), in a corresponding .i file, after being processed by appropriate preprocessors. The .i file does not have line numbers (#) in it. This option sets the -cpp option.

-R
Uses any remaining characters in the argument as ratfor options whenever processing a .r file. The temporary file, used as the output of ratfor, consists of the last component of the source file with an .f substituted for
the .r. This temporary file is removed unless you specify the -K option.

-r8
Defines REAL declarations, floating-point constants, functions, and
intrinsic as DOUBLE PRECISION (REAL*8), and defines COMPLEX
declarations as DOUBLE COMPLEX (COMPLEX*16).

-static
Causes all local variables to be statically allocated. This is the default.

-std
Produces warnings for things that are not standard in the language (same
as the -stand semantics and -stand syntax options).

-U
Causes DEC Fortran to distinguish between uppercase and lowercase
letters. (Otherwise, "a" is equivalent to "A".)

-Unnamed
Removes any initial definition of name to the preprocessor, as if by
#include name. See also the -D option.

-u
Makes the default type of a variable undefined (IMPLICIT NONE) rather
than using the default FORTRAN rules.

-V
Creates a listing file of the source file with various compile-time
information appended. The name of the listing file is the basename of the
source file, with an .l substituted for the.f,.for, or .FOR.

How you compile source files determines how the listing file is created. If
you compile several source files together, one listing file is created (named
with the basename of the first input file and the .l suffix). If you compile
source files one at a time, a separate listing file is created for each input
file (named with the basename of the input file and the .l suffix).

NOTE
In FORTRAN for RISC Version 2.0, -V was defined to print
the versions of compiler passes. POSIX defines -V as a
FORTRAN command line option to create a listing file. DEC
Fortran follows the POSIX usage for -V.

DEC Fortran follows the POSIX usage for the listing file
suffix .l, even though the suffix conflicts with input files to
lex(1).

-vms
Causes the run-time system to behave like VAX FORTRAN in the
following ways:
• Reinforces certain defaults
  Reinforces the following DEC Fortran defaults: -fpe0, -static, no -U
  (therefore, "a" is equivalent to "A"), and -assume norecursive. For
  example, if you specify -vms -fpe2, you get -fpe0.
  • Alignment
    Aligns all COMMON entries and RECORD fields on the next
available byte boundary, instead of natural boundaries (same as the
-align nocommons and -align norecords options).

- INCLUDE qualifiers
Recognizes /LIST and /NLIST at the end of the file pathname in
an INCLUDE statement at compile time.
If the file name in the INCLUDE statement does not specify the
complete path, the path used is the current directory.

- Quotation mark character ("")
Recognizes a quotation mark as starting an octal constant (such as
"177") instead of a character literal ("...").

- Control character syntax
Does not recognize the \n control character syntax in character
literals (same as the -assume backslash option).

- Deleted records in relative files
When a record in a relative file is deleted, the first byte of that
record is set to a known character (currently '@'). Attempts to read
that record later result in ATTACNON errors. The rest of the
record (the whole record if -vms is not set) is set to nulls for
unformatted files and spaces for formatted files.

- ENDFILE records
When an ENDFILE is performed on a sequential unit, an actual one
byte record containing a Ctrl/D is written to the file. If -vms is not
set, an internal ENDFILE flag is set and the file is truncated.

The -vms option does not affect ENDFILE on relative files; the file
is truncated.

- Reading deleted records and ENDFILE records
The run-time direct access READ routine checks the first byte of the
retrieved record. If this byte is '@' or NULL ("\0"), then
ATTACNON is returned.
The run-time sequential access READ routine checks to see if the
record it just read is one byte long and contains a Ctrl/D. If this is
true, it returns EOF.

- OPEN effects
Carriage control defaults to FORTRAN if the file is formatted, and
the unit is connected to a terminal (checked by means of isatty(3)).
Otherwise, carriage control defaults to LIST.
The -vms option affects the record length for direct access and
relative organization files. The buffer size is increased by one (to
accomodate the deleted record character).
- Implied logical unit numbers
  Recognizes certain environment variables at run time for ACCEPT,
  PRINT, and TYPE statements, and for READ and WRITE
  statements that do not specify a unit number, such as: READ
  (*.1000). For more information, see the DEC Fortran for ULTRIX
- BLANK= defaults
  The defaults for the OPEN keyword BLANK are as follows: for an
  explicit OPEN - 'NULL'; for an implicit OPEN - 'ZERO'; for
  internal files - 'ZERO'.
- List-directed internal WRITEs
  A list-directed internal WRITE blanks the first character of each
  record, and decrements the field width for the first element written
  to that record. For more information, see the DEC Fortran for

-W[no-implicit] Warn whenever a function is implicitly declared (whenever a function is
  referenced before it is declared). The default is -Wimplicit.
-Wreturn-type Warn whenever a function is defined with a return type that defaults to int.
  Also warn about any return statement with no return value in a function
  whose return type is not void.
-Wunused Warn whenever a local variable is unused aside from its declaration,
  whenever a function is declared static but never defined, and whenever a
  statement computes a result that is explicitly not used.
-Wswitch Warn whenever a switch statement has an index of enumeration type and
  lacks a case for one or more of the named codes of that enumeration and
  lacks a default label. Also warn whenever a switch statement has an
  index of enumeration type and has case labels outside the enumeration
  range.
-Wcomment Warn whenever a comment-start sequence (/*) appears inside a comment.
-Wtrigraphs Warn if any trigraphs are encountered.
-Wall All of the above -W options combined. These are all the options that
  pertain to usages that are often considered to be poor programming
  practice. These usages should be easy to avoid, even in conjunction with
  macros. The remaining -W options described below are not included in
  -Wall because certain kinds of useful macros are very difficult to write
  without causing those warnings.
-Wshadow Warn whenever a local variable shadows another local variable. For
  example, in the following code fragment:
  f(){ int i;
       /* i = .... */
       { int i;


/* i = ... */
}
}

the second i shadows the first, in that you cannot refer to the i in the outer
block when in the inner block, even though it would be in scope if the
second i were not declared.

-Wpointer-arith Warn about anything that depends on the size of a function type or of
void. MPL assigns these types a size of 1 for convenience in calculations
with void * pointers and pointers to functions.

-Wcast-qual Warn whenever a pointer is cast so as to remove a type qualifier from the
target type. For example, warn if a const char * is cast to an ordinary
char *.

-Wwrite-strings Warn whenever the address of a string constant is copied into a non-
constant char * pointer. For example, in the following code fragment:
f() { char *nonConstant;
    const char *Constant;
    nonConstant = "abc";
    Constant = "xyz";
    nonConstant = Constant;
}

warning messages are given for lines 3 and 5 if you set this flag.

-w Suppresses all warning messages.

-w1 Suppresses warnings about unused variables, but permits other warnings
(unless you also specify the -w option).

The following are mpl_f77 keyword options. Specify these options in the form: -name
keyword. You must spell out the -name completely, but you can abbreviate the keyword to
its shortest unique prefix (4 characters is the recommended length). For example, you can
specify -assume noaccuracy_sensitive as -assume noac, and -check underflow as
-check unde.

-align commons Aligns all COMMON block entities on natural boundaries, up to 4 bytes.
This option is invalid if you specify -vms. The default is
-align nocommons.

-align norecords Aligns all RECORD fields on the next available byte boundary, instead of
natural boundaries. The default is -align records.

-assume noaccuracy_sensitive Reorders floating-point operations, based on algebraic identities (inverses,
associativity, and distribution), to improve performance. The default is
-assume accuracy_sensitive.

-assume backslash Treats the backslash (\) character literally in character literals. The default
is -assume nobackslash, which treats the backslash character as C-style
control (escape) character syntax.
-assum dummy aliases
Assumes that dummy (formal) arguments to procedures share memory
locations with other dummy arguments or with COMMON variables that
are assigned. These program semantics slow performance and do not
strictly obey the FORTRAN-77 Standard. The default is
-assume nodummy_aliases.

-assume recursive
Compiles all FUNCTION and SUBROUTINE procedures for possible
recursive execution. This sets the -automatic option. The default is
-assume norecursive.

-check bounds
Generates code to perform run-time checks on subscript and substring
expressions. An error is reported if the expression is outside the dimension
of the array or the length of the string (same as the -C and -check_bounds
options). The default is -check nobounds.

-check overflow
Generates code to trap on integer overflow. Floating-point overflow
trapping is always enabled. The default is -check nooverflow.

-check underflow
Produces a message at run time to warn that floating-point underflow has
occurred. Floating-point underflow replaces the result with 0, unless you
specify the -fpe3 or -fpe4 option. The default is -check nounderflow.

-show code
Includes in the listing (if one is generated with -V), a machine language
representation of the compiled code. This machine language cannot be
assembled. The default is -show nocode.

-show include
Includes in the listing (if one is generated with -V), any text file specified
with INCLUDE in the source program. The default is -show noinclude.

-show xref
Includes in the listing (if one is generated with -V), a cross-reference of all
symbols used in the source program, along with line numbers of
definitions and uses. The default is -show noxref.

-stand semantics
Issues informational messages for statements that conform to the ANSI C
Standard but become nonstandard because of how they are used. This
option sets the -stand syntax option. The default is -stand nosemantics.

-stand source_form
Issues informational messages for statements that use tab formatting or
contain lowercase characters outside of character literals and comments.
The default is -stand nosource_form.

-stand syntax
Issues informational messages for syntax extensions to the ANSI C
Standard. The default is -stand nosyntax.

-warn noalignments
Suppresses warning messages for data that is not naturally aligned. The
default is -warn alignments.

-warn declarations
Issues a warning on any undeclared symbols (same as the -u option). The
default is -warn nodeclarations.

-warn nogeneral Suppresses all warning level messages (same as the -w option). The
default is -warn general.

-warn nounused Suppresses warning messages about unused variables (same as the -w1
option). The default is -warn unused.

-Zn Use the f77(1) debug default.
-Zq Suppress the MasPar copyright and version number notice.
-Zv Print the commands executed by the preprocessor, compiler, assembler,
and linker. Also print resource usage in the C-shell time(1) format.

The mpl_f77 driver displays an informative message saying that the following options are
unsupported:

-align8
-align16
-align32 Allows misalignment in COMMON.
-col120 Truncates source after column 120.
-q Does not display file and program names.
-w66 Suppresses F66 compatibility warnings.

The mpl_f77 driver stops compilation if the following option is specified:

-EB Produces big endian byte order.

The compiler assumes other arguments are either loader options or FORTRAN-77-
compatible object files, typically produced by an earlier mpl_f77 run, or by libraries of
FORTRAN-77-compatible routines. These files, together with the results of any
compilations specified, are loaded in the order given, producing an executable program with
the default name a.out.

DEFAULT SYMBOLS

The mpl_f77 compiler recognizes the following predefined symbols when a .m file is being
compiled. These symbols are assigned the value of "1". These symbols are useful in #ifdef
statements to isolate code for a particular case. Thus, these symbols can be useful for
ensuring portable code.

_MPL any MasPar system
_MPL1 any of the MasPar MP-1 Family
__STDMP__ ANSI C features and MPL extensions

The mpl_f77 compiler recognizes the following predefined macros when a .m file is being
compiled:

__DATE__ Evaluates to a string that specifies the date the program was compiled.
__FILE__ Evaluates to a string that specifies the current source file.
__LINE__ Evaluates to an integer that specifies the line that contains the macro

3-47
mpl_f77(1)

__TIME__

Evaluates to a string that specifies the time the program was compiled.

RESTRICTIONS

The standard library, /usr/lib/libc.a, is loaded with the -lc loader option and not a full path name. The wrong library can be loaded if there are files with the name libc.a in the directories specified with the -L loader option, or in the default directories searched by the loader.

The options -g0, -g1, -g3, and -O1 do not work with the MPPE (MasPar Programming Environment). Do not use these options if you are going to execute this program under the MPPE.

Optimized f77 code makes the MPPE output somewhat unpredictable and perhaps incorrect. Also, some bugs are much harder to find if the code is optimized than if it is not optimized. You may not want to optimize f77 code until you are finished debugging. See the DEC FORTRAN for ULTRIX RISC Systems User Manual for more information on debugging optimized DEC Fortran code.

If you are linking any .o files that were compiled with any -G option other than -G 0, see the "Restrictions" section of the mpl_cc man page. The mpl_f77 and mpl_cc drivers automatically compile with -G 0.

DIAGNOSTICS

The mpl_f77 command produces diagnostic messages that are intended to be self-explanatory. The loader can also produce occasional messages.

FILES

a.out loaded output
file.f input file
file.m input file
file.o object file
mon.out file produced for analysis by prof(1)
/tmp/* temporary
/usr/bin/efl Extended FORTRAN Language preprocessor
/usr/bin/ld RISC ULTRIX loader
/usr/bin/ratfor Rational FORTRAN dialect preprocessor
/usr/include standard directory for ‘#include’ files
/usr/lib/cmplrs/fort/decfort DEC Fortran compiler
/usr/lib/cord procedure-rearranger
/usr/lib/cpp C macro preprocessor
/usr/lib/crt0.o run-time startup
/usr/lib/f77 FORTRAN compilation driver
/usr/lib/for_msg.cat FORTRAN run-time message catalog
/usr/lib/ftoc interface between prof and cord.
/usr/lib/libc.a standard library, see intro(3)
/usr/lib/libfor.a DEC Fortran I/O support
/usr/lib/libi internationalization library
/usr/lib/libm.a math library
<table>
<thead>
<tr>
<th>File Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/usr/lib/libbots.a</td>
<td>DEC Fortran run-time string support</td>
</tr>
<tr>
<td>/usr/lib/libprof1.a</td>
<td>level 1 profiling library</td>
</tr>
<tr>
<td>/usr/lib/libUfor.a</td>
<td>DEC Fortran UNIX interface library</td>
</tr>
<tr>
<td>/usr/lib/libutil.a</td>
<td>DEC Fortran miscellaneous run-time support</td>
</tr>
<tr>
<td>/usr/lib/mcrt0.o</td>
<td>startup for profiling</td>
</tr>
<tr>
<td>$MP_PATH/log/mplogcontrol</td>
<td>disables mpl_f77 logging</td>
</tr>
</tbody>
</table>

**SEE ALSO**
- as(1), cc(1), cord(1), cpp(1), dbx(1), efl(1), fpr(1), ftoc(1), ld(1), lex(1), m4(1), mpl(1), mpl\_cc(1), mplld(1), mpprof(1), pixie(1), prof(1), ratfor(1), what(1), monstartup(3)
- *DEC Fortran for ULTRIX RISC Systems User Manual*
- *DEC Fortran Language Reference Manual*
- *DEC Fortran Release Notes* in /usr/lib/cmplrs/f77/relnotes
- *MasPar Programming Language (MPL) User Guide*
- *MasPar Programming Language (MPL) Reference Manual*
NAME
mpld — MasPar linker

SYNTAX
mpld [ options ... ] files ...

DESCRIPTION
The mpld command combines several object programs into one, resolves external references, and searches libraries. In the simplest case, several object files are given, and mpld combines them, producing a MOFF (MasPar Object File Format) object module that can either be executed or can become the input for a further mpld run. (In the latter case, you must use the -r option.) The output of mpld is left on a.out by default. This file is only made executable if no errors occurred during the load.

Argument routines can be in either MOFF or extended COFF format. MIPS ULTRIX files are in extended COFF format. The linker translates the extended COFF files to MOFF format.

The front end entry point of the output is the beginning of the first front end routine, unless you use the -e option. The ACU (Array Control Unit) entry point of the output is the beginning of the first ACU routine, unless you use the -a option.

If an argument is a library, it is searched only once at the point where it is encountered in the argument list. Libraries can contain a mixture of extended COFF and MOFF format object files. Only those routines that define an unresolved external reference are loaded.

Libraries of object files can be created using mplorder(1) to create an ordering of the files that places callers preceding callees, mpar(1) to create the library file, and mpranlib(1) to add a dictionary of symbol references.

Use of mpranlib is optional (see the note below regarding the order of routines in the library), but its use is recommended for speed in linking. When mpranlib is run on a library, it adds a dictionary of symbol references, named ___SYMDEF, as the first file in the library. The dictionary is searched repeatedly to satisfy as many references as possible. Therefore, although ordering of such libraries is not required for correctness, it can speed up linking by reducing the time to read the files from the library.

If a library has not been processed by mpranlib, and if a routine from a library references another routine in the library, the referenced routine must appear after the referencing routine in the library. The order of programs within such libraries is important for successful linking.

The mpld command can link together object files from different languages. The argument routines are linked in the order specified on the command line. This currently works for relocatable object files from MPL (MasPar Programming Language), ULTRIX C, DEC Fortran, MPAS (MasPar Assembly language), and MIPS assembly.

The mpld command provides symbol cross-linking between MPL and the other high level languages using the visible keyword in MPL. Calls between MPL and the other languages can be accomplished using the library routine callRequest(). See documentation of that routine for more details.
It is recommended, when compiling F77 and C routines for linking with MPL, that you use
the `mpl_f77(1)` and `mpl_cc(1)` commands, rather than using the `f77(1)` and `cc(1)` commands
directly. The `mpl_f77` and `mpl_cc` commands provide the correct options to `f77` and `cc` for
the MasPar system. If you invoke `f77` and `cc` directly, you must specify the `-G 0` option to
turn off use of global pointers to ensure correct linking with MPL programs. Also, if you
reference MasPar include files of the form `dir/file`, you must supply the path `-ISMP_PATH`.
Furthermore, if you wish correct debug information, you must specify `-O0` for `f77`, but not for
`cc`.

Eight C and MPL symbols are reserved and set to the first location above the text, initialized
data, and all data for the front end, ACU, and PE address spaces. It is an error to define these
symbols. The symbols are listed in the following tables.

<table>
<thead>
<tr>
<th>C</th>
<th>assembly</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>etext</td>
<td>etext</td>
<td>passed end of front end text</td>
</tr>
<tr>
<td>edata</td>
<td>edata</td>
<td>passed end of front end init data</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td>passed end of front end data</td>
</tr>
<tr>
<td>__etext_acu</td>
<td>__etext_acu</td>
<td>passed end of ACU text</td>
</tr>
<tr>
<td>__edata_acu</td>
<td>__edata_acu</td>
<td>passed end of ACU init data</td>
</tr>
<tr>
<td>__end_acu</td>
<td>__end_acu</td>
<td>passed end of PE data</td>
</tr>
<tr>
<td>__edata_pe</td>
<td>__edata_pe</td>
<td>passed end of PE init data</td>
</tr>
<tr>
<td>__end_pe</td>
<td>__end_pe</td>
<td>passed end of PE data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPL</th>
<th>assembly</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>etext</td>
<td><code>$etext</code></td>
<td>passed end of front end text</td>
</tr>
<tr>
<td>edata</td>
<td><code>$edata</code></td>
<td>passed end of front end init data</td>
</tr>
<tr>
<td>end</td>
<td><code>$end</code></td>
<td>passed end of front end data</td>
</tr>
<tr>
<td>__etext_acu</td>
<td><code>$etext_acu</code></td>
<td>passed end of ACU text</td>
</tr>
<tr>
<td>__edata_acu</td>
<td><code>$edata_acu</code></td>
<td>passed end of ACU init data</td>
</tr>
<tr>
<td>__end_acu</td>
<td><code>$end_acu</code></td>
<td>passed end of PE data</td>
</tr>
<tr>
<td>__edata_pe</td>
<td><code>$edata_pe</code></td>
<td>passed end of PE init data</td>
</tr>
<tr>
<td>__end_pe</td>
<td><code>$end_pe</code></td>
<td>passed end of PE data</td>
</tr>
</tbody>
</table>

For convenience in programs that use these symbols, an include file, `specialSym.h`, is
available that makes the five back end symbols available in C and MPL as `etext_acu`,
`edata_acu`, `end_acu`, `edata_pe`, and `end_pe`.

**OPTIONS**

The `mpld` command has several options. Except for the `-I` option, they should appear before
the file names.

- `-a` Takes the next argument as the name of the ACU entry point of the loaded
  program. Location 0x1000 is the default.

- `-D` Takes the next argument as a hexadecimal number and pads the front end data
segment with zero bytes to the indicated length.

-d Forces definition of common storage even if the -r flag is present.

-e Takes the next argument as the name of the front end entry point of the loaded program. Location 0 is the default.

-H Takes the next argument as a decimal integer, adds it to the end of front end text, and causes the front end data section to start at a higher address.

-Ldir Adds dir to the list of directories that are searched for libraries. Directories specified with -L are searched before the standard directories.

-lx Abbreviates the library name libx.a, where x is a string. The mplex command searches for libraries first in any directories specified with -L options, then in the standard directories. The first default library searched is $MP_PATH/lib/maspar, if MP_PATH is set, or /usr/maspar/lib/maspar, if MP_PATH is not set. The remaining default libraries searched are /lib, /usr/lib, and /usr/local/lib. A library is searched when its name is encountered, so the placement of a -l option is significant.

-M Produces a primitive load map, listing the names of the files that are loaded.

-N Place the data section immediately after the text section and do not make text portion read only or sharable. (Use magic number 0417.)

-n Arrange (by giving the output file a 0420 magic number) that the front end text portion is read-only and shared among all users executing the file when the output file is executed. This involves moving the front end data areas up to the first possible pagesize byte boundary following the end of the front end text.

-o output Names the final output file output instead of a.out.

-r This flag prevents final definitions from being given to common symbols, suppresses the undefined symbol diagnostics, and sets the magic number to RELOC_MAGIC(0432). (For the ULTRIX Id(1) command, this flag preserves relocation information, but for mplex the relocation information is always preserved.)

-T Takes the next argument as a hexadecimal number which sets the front end text segment origin. The default origin is 0.

-t Prints the name of each file as it is processed.

-u Takes the next argument as a symbol and enters it as undefined in the symbol table. This is useful for loading from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-X This option is accepted, but has no effect. (For the ULTRIX Id command, this flag preserves local symbol information, but for the mplex command the local symbol information is always preserved.)

-ysym Indicates each file in which sym appears, its type, and whether the file defines or references it. Many such options may be given to trace many symbols. For
mpl(1)

global variables, it is necessary to begin sym with an underscore-dollar-dollar (___$) for MPL, to leave sym unchanged for C, and use lowercase letters with a trailing underscore (_) for sym for DEC Fortran.

-z Arranges for the process to be loaded on demand from the resulting executable file (423 format) rather than preloaded. This is the default.

RESTRICTIONS
There is no way to force data to be page aligned. The mpl command pads the images that are to be demand loaded from the file system to the next page boundary.

The mpl command cannot process a file built from multiple .o files using ld -r. To combine .o files, use mpl -r.

FILES

$MP_PATH/lib/maspar/lib*.a  libraries
/lib/lib*.a  libraries
/usr/lib/lib*.a  libraries
/usr/local/lib/lib*.a  libraries
/usr/tmp/pldmtm*  temporary
a.out  output file

SEE ALSO

ar(1), as(1), cc(1), f77(1), mpar(1), mpas(1), mpl(1), mplorder(1), mpl_cc(1), mpl_f77(1), mpranlib(1)
NAME
mplimit — print or set MasPar program's DPU usage limits

SYNTAX
mplimit [-x] executable [ resource [ limit ] ]

DESCRIPTION
The mplimit command is used to print or set the DPU usage limits stored in a MasPar Object File Format (MOFF) executable file. These limits tell the dpumanager(8) what the anticipated resource requirements are for the program. If these limits are exceeded, the program may be aborted, usually with a stack overflow error. The two resources are ACU data memory (CMem) and PE memory (PMem). Currently, only the PMem resource is significant.

The mpsize(1) command may be useful in determining what the limits should be for a program. A program needs at least as much PE memory as the sum of its PE data and PE bss sections. If it allocates PE memory at runtime, it may need even more. You can use the mpconfig(1) command to determine the total amount of CMem and PMem on your machine.

The dpumanager provides a large enough partition (up to the machine size) for a job's static memory allocation (static data and bss), regardless of whether you use mplimit. However, the job could still get a stack overflow. The memory allocation cannot be dynamically increased, so it is important that a job request an adequate quantity. You may need to use mplimit to make sure the job gets enough PE stack and heap space, since the dpumanager cannot predict that usage.

The arguments to mplimit are described below. If a limit argument appears, the specified resource limit is set to that value. If no limit argument appears, the current setting of the specified resource limit is printed. If no resource is specified, both current resource limit values are printed.

executable The name of the MOFF executable file.
resource Either "pmem" or "cmem".
limit The word "default", a decimal number of bytes, or a hexadecimal number of bytes. Hexadecimal numbers must be prefixed with either "0x" or "OX". If a number has a "k" or a "K" suffix, it is multiplied by 1024. An executable file just created by the MasPar linker, mplt(1), has its limits set to "default". A limit of "default" will be interpreted when the program starts running to be the system default for that resource. The system default for PMem is set using arguments to dpumanager.

Note that the PMem limit includes a small amount of system overhead. Therefore, the amount of user allocatable PMem is actually slightly less than the amount specified by limit.

The amount of PMem requested for a job (written to the object file header) by mplimit is the smallest multiple of the partition size that accommodates the amount specified by limit or the amount of static memory required, whichever is larger. The mplimit command will accept a value for limit that is larger than the total amount of PMem available; in that case, the dpumanager will allocate the entire machine to that job.
mplimit (1)

OPTIONS
-Zq Suppress printing of the MasPar copyright notice.
-x Print numbers in hex rather than the default decimal.

DIAGNOSTICS
bad format The file is not a MOFF executable.
invalid number The number specified as a limit is invalid.

SEE ALSO
mpconfig(1), mple(1), mpsize(1), dpumanager(8)
NAME
mplorder — determine relation for a MasPar object library, Version 2.0

SYNTAX
mplorder file ...

DESCRIPTION
The input is one or more object or MasPar library archive files. For further information, see mpar(1). The standard output is a list of pairs of object file names, where the first file of each pair refers to external identifiers defined in the second. The output may be processed by tsort(1) to find an ordering of a library suitable for one-pass access by mpld(1).

The following example builds a new library from existing .o files.

    mpar -cr library `mplorder *.o | tsort`

The use of mplorder(1) is not absolutely necessary when you use mpranlib(1), which converts an ordered archive into a randomly accessed library. However, its use may speed up library processing by mpld(1).

RESTRICTIONS
The names of object files, in and out of libraries, must end with ".o"; otherwise, the results do not make sense.

FILES
*symref, *symdef

SEE ALSO
mpar(1), join(1), mpld(1), mpnm(1), mpranlib(1), sed(1), sort(1), tsort(1)
NAME

mpnm — print MasPar program's name list

SYNTAX

mpnm [ options ] [ file ... ]

DESCRIPTION

The mpnm(1) command prints the name list (symbol table) of each object file in the argument list. If an argument is an archive, a listing for each object file in the archive is produced. If no file argument is given, the file a.out is used.

Unless the -a option is specified, each output symbol name is preceded by its value, which is blank if undefined, and a letter. The letter defines the symbol type:

N       nil storage, compiler internal usage
T       external text
t       local text
D       external initialized data
d       local initialized data
B       external zeroed data
b       local zeroed data
A       external absolute
a       local absolute
U       undefined external
G       external small initialized data
g       local small initialized data
S       external small zeroed data
s       local small zeroed data
R       external read only
r       local read only
C       common
E       small common
V       external small undefined

There is no distinction made between front end, ACU, or PE text or data symbols. For example, T can mean either front end or ACU text.

The output is sorted alphabetically.

If the -a option is used, the format provides an expanded listing with these columns:

Name       the symbol or external name
Value      the value field for the symbol or external, usually an address or interesting debugging information
mpnm(1)

Class the symbol type
Type the symbol’s language declaration
Size unused
Index the symbol’s index field
Section the symbol’s storage class

OPTIONS
-a Displays all symbols including debug symbol table.
-e Prints only global (external) symbols.
-f Displays all symbols including debug symbol table. (Same as -a)
-g Prints only global (external) symbols. (Same as -e)
-n Sorts numerically rather than alphabetically.
-o Prepends file or archive element name to each output line.
-p Prints symbolic table order and does not sort.
-r Sorts in reverse order.
-u Displays only undefined symbols.

DIAGNOSTICS
bad format The file is neither an archive file nor a .o file.
cannot open The file could not be opened.
invalid argument An invalid option was specified.
no name list The file does not contain a symbol table.
r un out of memory The string table in either the archive or the .o file is too big.
out of memory Internal processing ran out of memory.
old archive The archive format is obsolete.
no symbolic header The file does not contain a symbolic header.
no fdr table The file does not contain a file descriptor table.
no internal strings The file does not contain an internal strings section.
no internal symbols The file does not contain an internal symbols table.
no external strings The file does not contain an external strings section.
no external symbols The file does not contain an external symbols table.
no aux table The file does not contain an auxiliary table.
no rfdt table The file does not contain a relative file descriptor table.
Set and Range not supported
Set and Range features not required for C or Fortran are not supported.
type not supported Type not required for C or Fortran is not supported.
SEE ALSO
mpar(1), mpar(5)
NAME
mppe — MasPar Programming Environment

SYNTAX
mppe

DESCRIPTION
The mppe command invokes the MPPE (MasPar Programming Environment), which allows you to run and analyze programs written for the MasPar family of computers. Currently, the MPPE supports programs that have been compiled with mpfortran(1), mpl_fort(1), mpl_cc(1), or mpl(1).

For information on how to display and/or run the MPPE remotely (on a machine other than the MP-1 front end machine where your program is running), see the MPPE User Guide.

The mppe command creates a window (called the MPPE root window) in the lower right portion of your screen. To iconify, resize, send to back, etc., use the standard Motif mechanisms provided on your workstation. To get help from within the MPPE, press the HELP button in the title bar of the Root window.

OPTIONS
The mppe command has no options or arguments.

SEE ALSO
msd(1), mpfortran(1), mpl_fort(1), mpl_cc(1), mpl(1)
MPPE Quick Reference, MPPE User Guide
NAME
mprof — MasPar report and profile generation front end

SYNTAX
mprof [ controls ... ] [ modes ... ] file [ file-options ... ]

DESCRIPTION
The mprof command calls the mpreport program to generate a report of compile-time
messages for file and calls the mppe program, mppeback, to execute file and generate two
execution profiles for it.

The compile-time message report is created in a file with a .ct extension. The report contains
a summary section that shows the count of compile-time messages by category for each
routine and a statement detail section that shows the messages for each statement. The
optional mode switches can be used to tailor the content of the .ct file.

The two execution profiles are the same ones produced when mppe is requested to "print"
profiles. One profile is created in a file with a .ps extension. This file contains a postscript
profile that shows the profiles in bargraphs and displays the source text. The other profile is
created in a file with a .as extension. This file can be displayed on an ascii terminal and
contains more detailed profile information than the postscript form. For more details on the
profiles, see the MPPE documentation on profiling.

The following options are modes that specify the content of the compile-time message report
in the .ct file.

-threshold=n
   Include all compile-time messages with level less than or equal to n.
   The default is n=0. As a general rule, as the level number increases,
   the messages become less critical and may be more voluminous.
   Currently, the highest level used is 2.

-blocked
   Issue only the compile-time messages that indicate multi-layer array
   mappings that may affect blocked algorithms. Ignore the -threshold
   setting and print messages of all levels.

-storage
   Issue only the compile-time messages that indicate storage usage.
   Ignore the -threshold setting and print messages of all levels.

EXAMPLES
To obtain the default compile-time message report and the two execution profiles on file,
enter:

mprof file

To obtain only the default compile-time message report on file, enter:

mprof -noexecute file

To obtain all the compile-time messages on file, enter:

mprof -noexecute -threshold=2 file
mpprof(1)

RESTRICTIONS
    The program source files need to be in current directory in order to be included in the .ps profile.

SEE ALSO
    mppe(1)
NAME
mpq — MasPar job queue examination program

SYNTAX
mpq [-Zq] [hostname]

DESCRIPTION
The mpq command examines the shared memory segment maintained by dpumanager(8) which contains the list of MasPar DPU jobs in execution or waiting for execution.

For each job queued, mpq job's user name (USER), process id (PID), priority (PRIO), pmem resident time limit (LIMIT), cmem size (CSZ), pmem size (PSZ), run time (RUNTIME), pmem resident time (PMEMTIME), status (STATUS) and starting time (START).

The job status may be "active", meaning that it currently holds the DPU memory; "queued", meaning it is waiting for availability of the DPU memory; or, "swapped", meaning that the job has been swapped out. If the job is marked as "background" or "held", the status also shows a single-letter abbreviation "B" or "H". The user may use command mpctl to change job's time limit.

If dpumanager(8) is not running, an error indicating that the shared memory segment cannot be attached will be printed.

OPTIONS
-Zq Suppresses printing of the MasPar copyright notice.

RESTRICTIONS
Due to the dynamic nature of the queue, mpq may occasionally print erroneous information.

SEE ALSO
dpumanager(8), mpctl(1), mpstat(1), maspard(8)
NAME
mpranlib — MasPar archive table of contents builder

SYNTAX
mpranlib archive ...

DESCRIPTION
The mpranlib(1) command converts each archive to a form that the linker, mpld(1), can link more rapidly. The mpranlib(1) command does this by adding a table of contents called ___SYMDEF to the beginning of the archive.

The archive may contain a mixture of extended COFF (MIPS ULTRIX files) and MOFF (MasPar Object File Format) files.

RESTRICTIONS
The mpranlib(1) command uses mpar(1) to reconstruct the archive. Therefore, sufficient temporary file space for this operation must be available in the file system that contains the current directory.

Because generation of a library by mpar(1) and randomization of the library by mpranlib(1) are separate processes, phase errors are possible. The linker, mpld(1), warns you when the modification date of a library is more recent than the creation date of its dictionary. This means that you also get the warning even if you only copy the library.

SEE ALSO
ar(1), mpar(1), mpld(1), ranlib(1)
NAME
mpsise — print MasPar program's sizes

SYNTAX
mpsise object...

DESCRIPTION
The mpsise(1) command prints the decimal number of bytes required by the FE text, FE
data, FE bss, ACU text, ACU data, ACU bss, PE data, and PE bss portions, and their sum in
hex and decimal, the MOFF file version, and PE array size of each object-file argument.
Both extended COFF (MIPS ULTRIX) format and MasPar Object File Format (MOFF) are
supported. If no file is specified, a.out is used.

DIAGNOSTICS
invalid option Some option has be entered.
file not found The indicated file could not be opened.
cant read header Failed to read the a.out header.
bad a_magic in header The a.out header magic number is not one of the supported values
(AMAGIC 0407, NMAGIC 0410, ZMAGIC 0413,
RELOC_MAGIC 0422).
cant read string table size Failed to read the first four bytes of the string table, which should
contain the size of the string table. This information is necessary for
locating the extension header.
incorrect version number The MOFF version of the file is incompatible with this version of
the tools.
incorrect COFF header Failed to read the COFF header.
NAME
mpstat — print MasPar job accounting statistics

SYNTAX
mpstat [options]

DESCRIPTION
The mpstat command examines, and optionally clears, the accounting file generated by the
dpumanager(8) program. Depending on the options specified, it will print a list of all jobs run and/or a summary.

OPTIONS
-a Requests a chronological list of all the relevant jobs listed in the accounting file. The
job starting time(STARTING DATE/TIME), user name(USERNAME), process
id(PID), running time(RUNTIME), wall time(WALLTIME),
pnemetime(PMEMTIME), pnemsize(PMSZ), cmemsize(CMSZ) will be printed for
each job. Wall time is the time when the job started to the time when job exited.
Pnemetime is the time when the job is resident inside the PMEM. The pnemsize and
cmemsize are the smallest partitions of corresponding PMEM and CMEM the job
will take.
-c Requests the accounting file to be cleared at the end of the program.
-f file Specifies the name of an accounting file to be used in place of the default.
-s Requests a summary showing the number of jobs, the average and maximum running
times, and the average and maximum wall times. This is the default when neither -a
nor -c is specified.
-l job_count Specifies that only the tail job_count entries in the accounting log are of interest.
This allows you to see only the most recent log entries. Note that other filters (like
-u) are applied after this one, so you may see less than the specified number of
entries. This option may be used with both -a and -s.
-u userName Causes mpstat to only count jobs for the specified user (login name).
-Zq Suppresses printing of the MasPar copyright notice.

RESTRICTIONS
The accounting file may only be cleared when you are running mpstat as root.

FILES
/usr/adm/dpuacct.log

SEE ALSO
dpumanager(8), mpq(1)
NAME
mpstrip — MasPar object file stripper

SYNTAX
mpstrip name...

DESCRIPTION
The mpstrip command removes the symbol table, relocation bits, and other debugging information ordinarily attached to the output of the assembler and loader. This is useful to save space after a program has been debugged.

SEE ALSO
mpld(1), strip(1)
NAME
  mpswopt — print or set MasPar program's PE-array size requirements

SYNTAX
  mpswopt  [-opt]  executable  [ executable  ... ]

DESCRIPTION
  The mpswopt command is used to print or set the PE-array size requirements stored in a MasPar Object File Format (MOFF) executable file. If the required size is smaller than the physical PE array, the program's run-time code will adjust the system's SWOPT (SoftWare OPTion) to a value less than its HWOPT (HardWare OPTion). This will cause the program-visible PE array to appear smaller than its physical size. If the program has no required size, SWOPT will equal HWOPT during execution. If the required size is larger than the physical PE array, the program will not run.

OPTIONS
  -opt  Causes executable files to be patched so their SWOPT equals opt. Without this switch, PE-array size requirements are not changed, only printed. SWOPT values correspond to the following PE array sizes:

<table>
<thead>
<tr>
<th>opt</th>
<th>PEs</th>
<th>ROWs</th>
<th>COLs</th>
</tr>
</thead>
</table>
  | 0   | 0    | 0    | 0     | no required size
  | 1   | 1K   | 32   | 32    |
  | 2   | 2K   | 32   | 64    |
  | 3   | 4K   | 64   | 64    |
  | 4   | 8K   | 64   | 128   |
  | 5   | 16K  | 128  | 128   |

DIAGNOSTICS
  The previous and new size requirements are printed on standard output. They are shown in terms of required rows-by-columns.

  Exit status 0 indicates success, 1 indicates failure, and -1 indicates usage error. Usage and file-open error messages are printed on standard error.

RESTRICTIONS
  Do not use on MasPar Fortran objects.

FILES
  /usr/maspar(field)/bin resident directory for executable mpswopt

SEE ALSO
  mplode(1), mpsize(1), mplimit(1)
NAME
mpvast — MasPar VAST-2 driver

SYNTAX
mpvast [ -o output ] [ -l listing ] [ options ] input1 [ inputn ]
where:
output Is a compilable output file. The default output filename is the input filename prefixed by "V".
listing Is a MasPar VAST-2 listing file. No listing is produced if this parameter is not used.
options MasPar VAST-2 options are described below and in the MasPar VAST-2 User's Guide.
input1...inputn Are Fortran source files.

DESCRIPTION
The mpvast(1) command invokes MasPar VAST-2. MasPar VAST-2 converts Fortran 77 programs to Fortran 90, and Fortran 90 programs to Fortran 77.

OPTIONS
The mpvast command supports all the options described in the MasPar VAST-2 User Guide. When mpvast is used with no arguments, it prints a short summary of all the options possible printed to standard output. In addition to those options, mpvast accepts the following options:

-extend_source Extend the range of statement text from columns 1-72 to columns 1-132. The default is -noextend_source. (Same as -ec.)
-fS Disable all generation of FORALL constructs. By default, FORALL is generated when reasonable.
-inline Automatically expand called routines inline, searching the input file for expandable routines (same as -e78).
-scalarize Convert Fortran 90 code into Fortran 77 code (same as -e1)
-split_common Convert each common block declaration into two common blocks: one containing only scalars and one containing only arrays (same as -e12).
-qq Disable "sloshing" diagnostics in the listing.
-Zq Suppress the MasPar copyright and version number notice.

SEE ALSO
mpifortran(1)
MasPar VAST-2 User Guide
Name
pcmp — MasPar parallel compare two files

Syntax
pcmp [ —l | —s ] file1 file2 [ skip1 ] [ skip2 ]

Description
The pcmp command compares two files using the parallel array to do the comparisons. If either file1 or file2 is ‘—’, standard input is used for the file. With no options, pcmp makes no comment if the files are the same. If they differ, it reports the byte and line number at which the difference occurred to standard output. If one file is an initial subsequence of the other a message including the file name is written to standard error.

The optional skip1 and skip2 parameters are initial byte offsets into file1 and file2 respectively and may be either octal, by specifying a leading 0, or decimal. When using skip1 and skip2 the offset is treated as the start of the respective input file. Only one option may be specified at a time. Only one of the input files may be standard input at a time. Because the line number is not calculated when using either of the options the use of either flag will increase the speed of pcmp.

Options
—l Long format: prints the byte number (decimal) and the differing bytes (octal) for each difference.
—s Suppresses normal output and sets the exit code only.

Diagnostics
Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

See Also
comm(1), diff(1)
vmestat(1)

NAME
vmestat — Print status of MasPar VMEFS files

SYNTAX
vmestat [ -Zq ] <file list>

DESCRIPTION
The vmestat command prints out the VME address, size and address mode associated with a list of vmeefs(5) files. The "-Zq" option suppresses the MasPar copyright notice.

EXAMPLE
In the following example there is one file (named "foo") in the /vme directory, which is a mounted vmeefs(5) file system.

> vmestat /vme/*
Copyright (c) 1991 MasPar Computer Corp. All rights reserved
Version 3.0
/vme/foo 0xe8000000 0x10000 A32D32

The first column shows the file name. The second is the VME base address. The third is the size of the VME address region. The fourth is the VME address mode.

SEE ALSO
vmeaccess(3), vmeefs(5)
NAME
dpumanager — MasPar DPU job manager daemon

SYNTAX
etc/dpumanager [ options... ]

DESCRIPTION
The MasPar job manager daemon, dpumanager, maintains the queue for DPU jobs, determines which job has access to the DPU at any particular time and which job is swapped out, and assures that the DPU state is properly initialized between jobs. It uses a privileged access mode to the ACU driver in order to monitor calls to open(2) and close(2) and certain calls to ioctl(2). This allows it to determine which processes need access to the DPU.

When started, dpumanager puts itself in the background by forking a child process and exiting the parent. This behavior may be suppressed by using the -nodaemon option.

Jobs are queued in order of (a) priority and (b) time when the DPU was requested. The job manager assigns a queue priority based primarily on the job’s PE memory requirement and dpu resident time. In addition, jobs with a time limit of one minute or less get a small boost in priority. The priority is an integer from 0 to 10000, where a lower number is a higher priority (see mpq(1)). A job is queued in front of the first job that has a lower priority. A job’s priority increases every time another (higher priority) job skips in front of it, so it is not possible for small jobs to permanently block execution of a large job.

The first n jobs on the queue are loaded in DPU memory and share the machine depending on the kernel scheduling. The number n varies according to job requirements and memory availability. The maximum value for n is determined by the -jobs command line option. A job cannot share the machine if it needs all of memory or it is not running as diagnostic job.

When access to the DPU is granted, the job manager assigns one or more partitions of PE memory. The amount assigned is normally determined by fields in the object file header, which are set using the mplimit(1) command. The memory allocation cannot be dynamically increased, so it is important that the job request an adequate quantity. The default is determined by the -pmem option.

The job manager attempts to assure that an appropriate version of DPU microcode (and opcode map) is loaded for each job. When started, it loads the default microcode version from $MP_PATH/etc/mp*ucode.wo. At the beginning of each job it reloads the microcode if the wrong version is known to be loaded or if the machine does not respond as expected.

The job manager is responsible for loading and starting up the ACU kernel program. The ACU kernel is reloaded whenever microcode is reloaded or whenever the kernel does not appear to be operating correctly.

The job manager is also responsible for starting up the dpumld and dpushitchd processes. The dpumld process is a daemon for instruction memory loading and the dpushitchd is a daemon doing nothing but providing the kernel a context on which the kernel does the job switching schedule. If the dpumld process is died for whatever reason, the job manager will kill all the jobs and reload the ACUK kernel program and restart the process. If the dpushitchd is died for whatever reason, the job manager will simply restart the process again.
dpumanager(8)

Jobs may have a time limit set. A system maximum time limit is set by the -maxtime command line option, by the privileged ioctl(2) call, DPUOSYSTIMELIMIT, or using the command mpctl(1). The default is no time limit. When a time limit is exceeded, the job manager sends the offending process a SIGXCPU signal, which normally causes the user process to exit.

The job manager accounting file (/usr/adm/dpuacct.log) contains an entry for the beginning and end of every job. This binary file is read by mpstat(1).

The job manager maintains a shared memory segment containing the job queue structures. This may be examined using the mpq(1) command and mpctl(1) command.

The job manager also maintains a DPU configuration file (/usr/tmp/dpuconfig), which may be used by other programs.

The terminate (15) signal causes the job manager to send a hangup signal to all the jobs and then exit. The hangup (1) signal tells it to send a hangup signal to all the jobs but not exit. The quit (3) signal tells it to send a hangup signal just to the current active job.

OPTIONS

-Zq Suppress printing of the MasPar copyright notice.
-jobs n Specify the maximum number of jobs in memory and the number of memory partitions. The valid range is 1 to 16. The default is 1, which disables dpu sharing. A value of 4 is recommended when dpu sharing is desired.
-maxtime t Specify a system maximum time limit, in seconds, for all jobs.
-nodaemon Prevent the job manager from putting itself in background.
-pmem k Specify the default amount of PE memory (in K-bytes) that each job is assigned. The default is one partition, determined by the -jobs option.
-sdir path Specify the swap directory and enable the swap facility. If this option is unspecified, the dpumanager disable the swap facility. To get reasonable performance of swapin/swapout of jobs, it is recommended to use the MasPar disk array.
-stime t Specify the minimum "swap slice" of time. A regular job resident in the dpu memory longer than this time period may be eligible to be swapped to disk. The default amount of time each job is swapped to disk is equal to its partition size times this swap slice time. A regular job swapped out to disk longer than its disk swap slice may be eligible to be swapped back in from disk. If this option is unspecified, the dpumanager calculates the swap slice based on the machine size and whether MasPar disk array is used.

-trace Turn on the trace.

DIAGNOSTICS

The job manager log (/usr/adm/dpujobmgr.log) is a plain-text file containing a time-stamped entry for each significant event detected by the job manager. All faults detected by the job manager or the background diagnostic process are reported in this log file. In addition, an entry is made when the job manager starts up, shuts down, or loads microcode.
When a fault is detected that requires termination of a job, the job manager sends an IOT (6) signal to the job.

ENVIRONMENT
The job manager uses the environment variable MP_PATH to find the default microcode image (etc/mp*ucode.wo) and the ACU kernel (etc/acuk).

RESTRICTIONS
Pending jobs will hang if the job manager terminates abnormally. In this instance, it is necessary to kill all such jobs manually; they will never be granted access to the DPU even when the job manager is restarted.

The job manager must always be executed with root privileges.

FILES
/usr/adm/dpujobmgr.log — log file
/usr/adm/dpuacct.log — job accounting file
/usr/tmp/dpuconfig — DPU configuration file
$MP_PATH/etc/acuk — ACU kernel
$MP_PATH/etc/mp11ucode.wo — default microcode image for MP1100
$MP_PATH/etc/mp12ucode.wo — default microcode image for MP1200
$MP_PATH/etc/startup.wo — dummy microcode used during initialization

SEE ALSO
mpi(1), mpq(1), mpctl(1), mpstat(1), dpuDevice(3), acu(4), maspard(8), mplimit(1), swapping(8)
NAME
   mpd — MasPar system information daemon, Version 2.1

SYNTAX
   mpd [ config-file ] [ options... ]

DESCRIPTION
   The system daemon, mpd, runs on any machine with a DPU and supplies information to
   requesting programs about the machine configuration. The information program, mpi(1),
   may be used to query this information.

   In normal operation, mpd is started by the /etc/rc.maspar script and runs until the machine is
   shut down. It puts itself in background automatically unless the -nodaemon option is
   specified. It monitors network service port 1950 unless another port is specified with the -p
   option. It reads the file "/usr/tmp/dpuconfig" for configuration information unless an
   alternate file name is on the command line.

   When mpd receives a query, it reads the configuration file and formats a reply. If the
   configuration file is missing or invalid, mpd will not respond. The configuration file is
   normally maintained by the DPU job manager, dpumanager(8).

FILES
   /usr/tmp/dpuconfig
NAME
mpshutdowm — terminate the MasPar DPU job manager

SYNTAX
/etc/mpshutdowm

DESCRIPTION
The mpshutdowm command sends a termination signal to the DPU job manager daemon, dpumanager(8). When the job manager receives this signal, it should send a hangup signal to all pending DPU jobs and then exit.

This is a shell script which should be run as root.

DIAGNOSTICS
mpshutdowm will exit with a non-zero code if it is unable to kill the job manager, if the job manager is not present, or if it finds multiple instances of the job manager.

SEE ALSO
dpumanager(8)
NAME
  mptimelimit — set the MasPar DPU maximum job time limit

SYNTAX
  etc/mptimelimit [-Zq] -maxtime time

DESCRIPTION
  The mptimelimit command sets the maximum time limit for all MasPar DPU jobs. This limit
  remains in effect until the next mptimelimit command or until the job manager is restarted.
  The command is equivalent to the "-maxtime" argument on the dpumanager(8) command.
  This limit applies to the total wall time during which the job holds the DPU. Periods where
  the job is waiting in queue or swapped out do not count.

ARGUMENTS
  -maxtime time
    Specify the job time limit, in seconds. A value of zero disables the system limit.

  -Zq    Suppress the MasPar copyright notice.

RESTRICTIONS
  Mptimelimit requires system privileges and may be run only by the super-user (root).
  The DPU job manager must be active when this command is executed.

SEE ALSO
  dpumanager(8)
NAME
  mpzap — MasPar Binary Zapper, Version 3.0

SYNTAX
  mpzap [ -z value ] file

DESCRIPTION
  The mpzap command is used to inspect the value of a special constant in a binary executable and optionally to change this value. To change a value, you must run mpzap as root.

  Currently mpzap is only used to change the default machine size used by the MasPar Fortran compiler. The default machine size of 1K (1024) processors used by MasPar Fortran can be changed on a per site basis by the superuser as follows:

  mpzap -z value file

  where value is 1, 2, 4, 8, or 16 for a 1K, 2K, 4K, 8K or 16K machine, respectively. For example, if you have a 2K machine, you would enter

  mpzap -z 2 $MP_PATH/bin/mpfortran

  Any user wishing to know the value set in the site’s mpfortran program may do so by invoking mpzap with no options:

  mpzap $MP_PATH/bin/mpfortran

  If a value is changed or if no appropriate value is found, warning messages are produced.

OPTIONS
  -z value Specifies the value for the machine size to be used by the MasPar Fortran compiler.
abort 1-8
access 1-8
alloca 1-8
apropos iv

cc 1-4, 1-7
CC 1-7
Clear 2-2
Compilers
   cc 1-4, 1-7
   f77 1-3, 1-7
   makefiles 1-6
   mpfortran 1-3
   mpl 1-4
   mpl_cc 1-3, 1-7
   mpl_f77 1-3, 1-7
   with no .c.o rule 1-7
Compiling
   with -G 8 1-5
   .cshrc file 1-2

Data parallel ii
DEC documentation iii, 1-7
DEC Fortran 1-3
DECstation 1-2
DECstation 5000 i
Default path 1-1
DPU 1-3
   halt and fault conditions 1-9
dpumanager 1-3, 1-6

Environment variables

MP_PATH 1-1, 1-2
Examples iv, 1-1, 1-2
exec 1-8, 1-9
execve 1-8
exit 1-8

f77 1-3, 1-7
FC 1-7
Flat profiling 2-2
Font conventions iv
fork 1-8
Fortran
   See DEC Fortran, MPF
fprintf 1-8
fputs 1-8

getenv 1-8
Global pointers 1-5

Help iv
Hierarchical profiling 2-2

index 1-8
ioctl 1-8

Job queue 1-3
Job swapping 1-3

Libraries 1-7
Linking
   -G 8 compiled with -G 0 1-5
   libraries with global pointers 1-5
lseek 1-8
.m files 1-7
Makefiles 1-6
malloc 1-8
man iv
MasPar Architecture Specification ii
MasPar Commands manual ii
MasPar Fortran
See MPF
MasPar Fortran Reference manual
ii, 1-3
MasPar Fortran User Guide ii, iii, 1-3
MasPar Programming Language
See MPL
MasPar System Overview manual ii
mpar 1-8
mpconfig 1-6
MPDDL Reference manual ii
MPF ii, 1-3
mpfortran 1-3
MPIPL Reference manual ii
mpl 1-4
MPL ii, 1-3, 1-4, 1-7
MPI Reference manual ii, iii, 1-3
MPL User Guide ii, iii, 1-3
mplimit 1-5, 1-6
mpl_cc 1-4, 1-7
mpl_f77 1-3, 1-7
MPML Reference manual ii
MPPE 1-2
   displaying 1-2
   invoking 1-2
   unable to use on certain
   executables 1-7
MPPE User Guide ii
mpq 1-3
mpranlib 1-8
mpsize 1-6
mpstat 1-3
mptimelimit 1-3
MP_PATH 1-1, 1-2, 1-3

Notation conventions iv

Path, default 1-1
perror 1-8
Profiling 2-1
   flat 2-2
   hierarchical 2-2
putenv 1-8

read 1-8
Run-time environment 1-9
Run-time libraries 1-8

SIGEMT 1-9
signal 1-8
Signals 1-9
sigvec 1-8
SPARCstation 1-2
sprintf 1-8
sscanf 1-8
Statement Profiles menu 2-2
Stdin/Stdout window 1-2
strcat 1-8
strlen 1-8
strncmp 1-8

Ticks 2-2
two-byte offset 1-5
Typographic conventions iv

ULTRIX i
ULTRIX C 1-3
UNIX clock interrupt 2-2

VAXstation 1-2
vfork 1-8

X terminal 1-2