
# Cray T3E User's Guide

Juha Haataja and Ville Savolainen (eds.)

Center for Scientific Computing, Finland

2nd edition

ISBN 952-9821-43-3

http://www.csc.fi/oppaat/t3e/

# Preface

This is the second edition of a user's guide to the Cray T3E massively parallel supercomputer installed at the Center for Scientific Computing (CSC), Finland.

The first edition of this guide was written by Juha Haataja, Yrjö Leino, Jouni Malinen, Kaj Mustikkamäki, Jussi Rahola, and Sami Saarinen. The second edition was written by Juha Haataja, Jussi Heikonen, Yrjö Leino, Jouni Malinen, Kaj Mustikkamäki, and Ville Savolainen.

The following colleagues at CSC have provided invaluable feedback about this book: Juha Fagerholm, Erja Heikkinen, Tiina Kupila-Rantala, Peter Råback, Tomi Salminen, and Raimo Uusvuori.

The second edition is available only in PDF format, and can be loaded and printed freely from the WWW address `http://www.csc.fi/oppaat/t3e/`. The paper version of the first edition can still be ordered from CSC.

We are very interested in receiving feedback about this publication. Please send your comments to the e-mail address `Juha.Haataja@csc.fi`.

Espoo, 31st July 1998

*The authors*

# Contents

# Chapter 1

# Introduction

This chapter gives a short introduction of the Cray T3E system. We also describe the policies imposed on using the computer: application forms, scalability testing, and user quotas.

## 1.1 How to use this guide

This book is divided into ten independent chapters, and it can be used as a handbook. However, we recommend that you browse through at least the first four chapters which provide a general overview of the Cray T3E system.

This chapter gives a short introduction to the Cray T3E parallel supercomputer, and provides pointers to additional information. Chapter 2 describes how to log in to the system and how to compile and run your applications. Chapter 3 discusses the Cray T3E hardware. Chapter 4 provides information on the program development environment of the T3E system.

Chapters 5 and 6 give more detailed information about the Fortran and C/C++ compilers on the system. Chapter 7 shows how to parallelize your codes using the MPI or PVM message-passing libraries, the Cray data-passing library SHMEM, or the data-parallel HPF programming model.

Chapter 8 discusses the batch job system and how to submit your applications to the NQE system (Network Queuing Environment).

Chapter 9 illustrates Cray programming tools such as the TotalView debugger and the available profiling tools. Finally, Chapter 10 discusses some further topics, such as timing of programs or predicting the parallel performance of a code.

## 1.2 Usage policy

As the Cray T3E is a high-performance computational resource, CSC enforces a usage policy in order to guarantee an efficient and fair usage of the computer.

When applying for access to the Cray T3E you are supposed to have a user id also on some other computer at CSC. A T3E resource application form can be requested by contacting Ms. Paula Mäki-Välkkilä at CSC, tel. (09) 457 2718, e-mail `Paula.Maki-Valkkila@csc.fi`.

The project application form is handled by the CSC T3E contact person at CSC, who will contact the applicant if necessary. You are first granted an initial quota of 100 hours, which gives you a possibility to test and tune your code using 16 processors at maximum.

Within the limits of the initial quota, you have three months to demonstrate that your code scales up. The results of the scaling tests should be sent to CSC by the end of this time. Currently a typical requirement is to attain at least a speedup of 1.5 when doubling the number of processors allocated. The eventual quota and the maximum number of processors for production runs will be granted by CSC's Scientific Director when the results of the scaling tests are available.

To get a production quota, you are requested to explain how you have parallelized the program code. The aim of this procedure is to ensure that you have understood the specific features of the T3E system and that the code is parallelized efficiently.

The T3E system is intended only for parallel jobs. Single processor production runs are not allowed. The computer can be used both for interactive program development (testing and debugging) and for production runs. However, there is a limit of 16 processors for interactive use. If you want to use more processors, you have to use the batch job facilities.

When running a batch job, a set of processors will be dedicated for your job. If some of the processors become idle during the run, no other user is allowed to use them until the whole run is finished.

Your quota will be charged according to the number of processors that are assigned to the job multiplied by the elapsed wall clock time. For example, if you run a job on 32 processors for three hours, 96 hours will be deducted from your quota.

The usage policy and especially the configuration for batch jobs is likely to change in time. Current configuration and batch job limits are given in Chapter 8 (page 81).

## 1.3 Overview of the system

The Cray T3E system at CSC has currently 224 RISC processors for parallel applications. In addition, there are 16 processors for system services and for interactive use.

The T3E has a good user and programming environment. The system feels like any Unix computer. You log in to the Internet address `t3e.csc.fi` and end up in an interactive processor.

All processors share a common file system. The `ps` and `top` commands can be used to look at processes on any processor (see page 29 for details). The parallel nature of the machine is only manifested when running parallel codes.

The single-processor performance is a critical factor in the performance of parallel user codes. As on most RISC processors, user codes may typically obtain only 5–15% of the maximum performance on each processor. This is also true for the Cray T3E. However, in linear algebra operations, the BLAS kernels can reach over 65% and LAPACK kernels over 45% of the peak performance. See page 33 for more details.

The T3E has a remarkably fast communication network which makes the machine a well-balanced system. It is quite easy to write parallel codes that scale up to a hundred processors.

The Cray T3E is an IEEE-conformant system with 64-bit integer and floating point representation by default. To conserve memory, you can switch to 32-bit representation of integers or integers and floating point values together.

The Cray T3E hardware is described in Chapter 3, and code optimization is discussed in Chapters 5 and 6 (Fortran 90 and C, respectively).

The Cray T3E series is a product of Cray Research, which is a subsidiary of Silicon Graphics, Inc.

## 1.4 Programming environment

The Cray T3E system offers a versatile programming environment for the users. There is a high-quality Fortran 90 compiler which can, of course, be used to compile standard-conforming FORTRAN 77 programs as well. Also C and C++ compilers are available.

Parallelization can be done using the Cray implementation of MPI (Message Passing Interface), which has been optimized for the system. Also the PVM libraries (Parallel Virtual Machine) are available. MPI is discussed in Section 7.2 (page 63) and PVM in Section 7.3 (page 68).

Besides the portable MPI and PVM message-passing systems, the high-performance SHMEM library is available. This is a Cray-specific library for parallelization using the "data-passing" or one-sided communication paradigm. See page 70 for further details.

In addition to the message-passing and data-passing methods for parallelization, there is a possibility for data-parallel programming on the Cray T3E. The HPF (High Performance Fortran) programming model is a data-parallel extension of the Fortran 90 programming language (see Section 7.5 on page 78 for details).

## 1.5 Programming tools and libraries

In addition to the previously mentioned compilers and parallelization tools, there are additional programming tools available on the T3E. The MPP Apprentice and PAT (the Performance Analysis Tool) profiling tools make it possible to locate performance bottlenecks in a parallel code. The Cray TotalView debugger makes finding bugs in a parallel program easier. You may also use the VAMPIR software for tracing message passing of MPI codes.

The Cray T3E system also offers some standard numerical libraries, such as the Cray Libsci library, which contains high-performance versions of the BLAS and LAPACK libraries. In addition, the Libsci library offers tuned routines for, e.g., FFT operations on large datasets.

The IMSL and NAG general-purpose numerical libraries are also available. At the moment, these packages contain only single-processor routines, but in the future some of the IMSL and NAG routines will be parallelized. However, the ScaLAPACK library already offers some parallel routines for linear algebra operations. In addition, some of the FFT routines in Libsci are parallelized.

See page 33 for more information on the Libsci library. If you are interested in using the NAG or IMSL libraries, see pages 37 or 38, respectively.

## 1.6 Notation used in this guide

The `teletype` font indicates a command or a file name, or the output of a program. To distinguish between user commands and the response of the computer, the following fonts are used:

```
t3e% pwd
/csc
t3e% echo $ARCH
t3e
```

Here the prompt and response of the machine have been typeset with the `teletype` font, and the user commands are shown in **boldface**.

The generic names given to the commands are indicated with a *slanted* font type:

```
rm file
```

The optional parts of a command are written inside brackets:

```
more [options] [file]
```

Some commonly used names have been written in the same way as Unix. To introduce new terms, an *emphasized* text type is used.

## 1.7 Sources for further information

A general introduction to the Unix programming environment is given in Finnish in *Metakoneen käyttöopas* (Metacomputer Guide) [Lou97]. For a short introduction to the CSC metacomputer in English, see *CSC User's Guide* [KR97].

There is a good on-line reference for the Unix operating system at the WWW address

```
http://unixhelp.ed.ac.uk/index.html
```

CSC has published textbooks on MPI and PVM in Finnish [HM97, Saa95]. CSC has also published textbooks on Fortran 90 [HRR96] and numerical methods [HKR93] (in Finnish).

There is a mailing list `t3e-users@csc.fi` for CSC's Cray T3E users. This is the most rapid and flexible means for CSC's personnel to reach the T3E customers. Hints and tips on T3E usage are also submitted via the list.

Since the number of users is still growing, we keep a backlog of the messages for new users and occasional review at

```
http://www.csc.fi/oppaat/t3e/t3e-users/archive/
```

There are several types of MPI handbooks. At least the following books are useful:

- *MPI: A Message-Passing Interface Standard* [For95]
- *MPI: The Complete Reference* [SOHL+96]
- *Using MPI: Portable Parallel Programming with the Message-Passing Interface* [GLS94]
- *Parallel Programming with MPI* [Pac97]

The basics of parallel programming are discussed in the textbook *Designing and Building Parallel Programs* [Fos95]. Another good textbook is *Introduction to Parallel Computing — Design and Analysis of Algorithms* [KGGK94].

CSC maintains a WWW service called *CSC Program Development*, which contains examples of parallel codes, an English-Finnish parallel computing dictionary, and some other information. The WWW address is

```
http://www.csc.fi/programming/
```

# Chapter 2

# Using the Cray T3E at CSC

This chapter helps you to start using the Cray T3E at CSC: how to log in, where to store files, how to to use the compiler and run your codes etc. The usage policy of the machine is discussed in Section 1.2 on page 8.

## 2.1 Logging in

When logging into the Cray T3E, you will actually get connected to one of the *command processors*. These are the processing elements (PEs) that are responsible for Unix command processing.

In order to log in, you normally have to first log into a local Unix computer (at your university or some other site on the Internet) and then use a `ssh`, `telnet` or `rlogin` command, giving `t3e.csc.fi` as argument. For example, logging in from `cypress.csc.fi`:

```
cypress% ssh t3e.csc.fi
```

The same can be done using `telnet`:

```
cypress% telnet t3e.csc.fi
```

Or `rlogin`:

```
cypress% rlogin t3e.csc.fi
```

If you use the `ssh` or `rlogin` command and your user id on the Cray T3E is different from your user id on your local computer, you must give the `-l` option to specify the user id on the Cray T3E. This option is added after the computer name, for example:

```
cypress% ssh t3e.csc.fi -l user_id
```

`ssh` is the preferred way to connect to T3E as well as to all CSC machines

because it uses a secure way to authenticate oneself to the host machine.

If you are using an X terminal or an equivalent (a workstation or a microcomputer with software supporting the X Window System), you can establish an X Window System connection to the Cray T3E directly.

An easy way to use X Window System connection to the T3E is an `ssh` connection running in local `xterm`. To establish an X Window System connection to the T3E with environment settings correct, type:

```
localhost% xterm -e ssh t3e.csc.fi
```

Using `rlogin` or `telnet` connections, the procedure differs somewhat from `ssh`. Once logged in, the appropriate value of the `DISPLAY` environment variable has to be set, if any X applications are to be run:

```
t3e% setenv DISPLAY your_x_terminal:0.0
```

Sometimes the Cray T3E may not recognize the string *your_x_terminal* and you have to give the numerical Internet address instead:

```
t3e% setenv DISPLAY 128.256.512.64:0.0
```

If the Internet address is not known, it can be found by the following command:

```
t3e% nslookup your_x_terminal
```

When a connection has been established to the Cray T3E using `telnet`, you have to enter your user id and password. A typical `telnet` session starts as follows:

```
Trying 128.214.248.31...
Connected to t3e.csc.fi.
Escape character is '^]'.

Cray UNICOS/mk (t3e) (ttyp007)

login: user_id
Password: jameS5#e          (the password does NOT show up)

t3e%
```

After displaying the prompt (`t3e%`), the Cray T3E is ready to execute your commands.

## 2.2 Files

Your home directory (`$HOME`) is located on the disk server. Therefore, your files are shared between the T3E and other computer systems at CSC.

However, for performance reasons it is highly recommended to copy

all files before running a job from the home directory tree to the local T3E disk described below. The home directory is suitable only for small initialization files and frequently used small programs. It is not intended for extensive I/O operations or for large data sets.

There are three file storage areas available for users. Usually you need not (and should not) refer to directories with full path names. Instead, use the symbolic names (environment variables) listed in the following table.

| *Symbol* | *Where* | *Lifetime* | *Backup* |
|---|---|---|---|
| `$HOME` | Home directory (NFS-mounted) | Unlimited | Yes |
| `$TMPDIR` | `/tmp/$LOGNAME` | One day | No |
| | `/tmp/jtmp.`*session-id* | Interact. session | No |
| | `/nqstmp/nqs.`*job-id* | Batch job | No |
| `$WRKDIR` | `/wrk/$LOGNAME` | Seven days | No |

The home directory (`$HOME`) tree is backed up regularly. This directory is meant for permanent files, with a maximum total size of a few megabytes. It is a typical repository for source codes and small input files.

The temporary directory (`$TMPDIR`) should be used by programs which produce temporary, run-time files. Unless changed in the login or run script, all files will typically be deleted upon the exit of a job. The size of the disk storage is typically a few gigabytes and no backups are taken.

The environment variable `$TMPDIR` can have three different values depending on the execution mode, or on settings in your login scripts.

An interactive session gets a unique session id. The variable `$TMPDIR` points to the directory `/tmp/jtmp.`*session-id*, which can be used to store temporary files. The files in this directory are deleted upon the end of the session. Thus, you may find it more convenient to redeclare this environment variable to be `/tmp/$LOGNAME`, where `$LOGNAME` is your username.

In a batch job, the directory `$TMPDIR` gets its unique value from the identification string of the job, and this directory is removed at the end of the job.

The working directory (`$WRKDIR`) differs from the temporary directory in the storage time. However, since no backup is taken, a disk crash may destroy its contents. Untouched files will be deleted after seven days.

## 2.3 Editing files

You can use the Emacs or `vi` editors on the T3E. To start Emacs, give the command

```
emacs [options] [filename]...
```

Here is an example:

```
emacs -nw main.f90
```

However, because your home directory is shared with other computers at CSC, you can do your editing on some other system. We recommend this approach, because it minimizes the interactive load on the T3E.

You get a short introduction to Emacs in Finnish by giving the command

```
help emacs
```

## 2.4 Compiling and running applications

Parallel programs on the Cray T3E can be either *malleable* or *non-malleable*. Malleable executables can be run on any number of processing elements using the `mpprun` command. Non-malleable executables are fixed at compile time to run on a specific number of processors.

If a program is to be non-malleable, it has to be compiled and linked with the option `-Xn` (or `-X n`) indicating the number *n* of PEs. A program with a fixed number of PEs can be started directly. For example, in the following we use ten processors:

```
t3e% f90 -X 10 -o prog.x prog.f90
t3e% ./prog.x
```

To produce a malleable program, the source code can be compiled and linked with the option `-Xm`, but since this is the default, you can usually omit the flag. To choose the number of processors, the executable has to be run using the `mpprun` command.

The following example compiles, links and executes twice a program called `prog.x`. The first invocation uses five (5) processors and the second twelve (12) processors:

```
t3e% f90 -o prog.x prog.f90
t3e% mpprun -n  5 ./prog.x
t3e% mpprun -n 12 ./prog.x
```

Note: if the program is executed only on one processor, it will not be run on the application nodes, and thus it might be interrupted by other activities.

Interactive jobs can use at maximum 16 processors and 30 min parallel CPU time.

## 2.5 Executing in batch mode

The batch jobs on all CSC's computers are handled by the NQE system (Network Queuing Environment). A more detailed description of this system is given in Chapter 8 (page 81).

You can run a batch job by submitting an NQE request, which is a shell script that contains NQE commands and options, shell commands, and input data. At the moment, batch requests have to be submitted locally on the Cray T3E with the command:

```
qsub [-l mpp_p=number]
     [-l mpp_t=time]
     [-l p_mpp_t=ptime]
     [options] script
```

Resource specifications (option `-l`) are:

| *Option* | *Meaning* |
|---|---|
| `-l mpp_p=`*number* | Number of PEs to be used within a job |
| `-l mpp_t=`*time* | Maximum execution time of all the programs in the script. The time should be given in the formats `hh:mm:ss`, `mm:ss` or `ss`. |
| `-l p_mpp_t=`*ptime* | Maximum processing time of any single program in the script. |

Other typical options are:

| *Option* | *Meaning* |
|---|---|
| `-r` *request_name* | Specific name for the batch job request |
| `-q` *queue_name* | Batch queue name where the job should be run |
| `-lT` *time* | Maximum single processor time for the job to be run (`hh:mm:ss`). |
| `-eo` | Concatenate stderr output to stdout |
| `-o` *filename* | Specific script output file |

The argument *script* of the `qsub` command is a file, which contains the Unix commands to be executed. If omitted, standard input will be used.

Before executing the commands in the script, the batch system uses your default shell to log in to the T3E. This sets up your normal user

environment. After this, the commands in the script are executed using `/bin/sh`. This can be overridden using the option `-s` *shell_name*.

Here follows an example script, which is written into a file called `t3e.job`. The request name is set to `simulation`. The job file reserves at maximum six processors (option `-l mpp_p=6`), and the approximate maximum wall clock time is 600 seconds (`-l mpp_t=600`). Standard error is concatenated with the standard output (option `-eo`).

```
#QSUB -r simulation
#QSUB -l mpp_p=6
#QSUB -l mpp_t=600
#QSUB -eo
#QSUB

cd $TMPDIR
cp $HOME/data/inputfile .
cp $HOME/src/a.out .
mpprun -n $NPES a.out
```

First, the script changes the current directory to the temporary directory. Thereafter, the input file `inputfile` and the executable program `a.out` are copied there. Finally, the `mpprun` command triggers the actual parallel execution.

If the time limit of 600 seconds is reached, the job will be terminated.

The environment variable `$NPES` indicates the number of processors actually allocated for the job. This is a local feature of the T3E at CSC.

To submit the previous job file to the `prime` queue, use the command

```
t3e% qsub -q prime t3e.job
```

You can use the command `qstat` for checking out the status of your batch job. You get a listing of the current processes with the commands `top` and `ps -PeMf`. Use the command `qdel` to delete a batch job from the queue. See Chapter 8 for more details.

## 2.6 More information

There are normal Unix-style `man` pages available on the Cray T3E. In addition to this, CSC's `help` system is available on the Cray T3E. For example, you can look up how to use the IMSL libraries:

```
help imsl
```

The guide *Metakoneen käyttöopas* (Metacomputer Guide) [Lou97] describes (in Finnish) the CSC environment in detail. See also the Web address

```
http://www.csc.fi/metacomputer/english/crayt3e.html
```

for some information in English.

Cray has published several manuals, which help in using the T3E. On-line versions of the manuals are found at the Web address

```
http://www.csc.fi:8080
```

The most useful manuals are the following:

- *CF90 Commands and Directives Reference Manual* [Craa]
- *Cray T3E Fortran Optimization Guide* [Crac]
- *Cray C/C++ Reference Manual* [Crab].

# Chapter 3

# The Cray T3E system

This chapter reviews the Cray T3E hardware and operating system.

## 3.1 Hardware overview

The Cray T3E system consists of the following hardware components:

- processing elements
- interconnect network
- I/O controllers
- external I/O nodes.

This section briefly presents each of the system components and their interactions.

The current configuration at CSC is as follows:

- For parallel programs there are 224 application processing elements (PEs) which contain 375 MHz processors.
- In addition, there are 16 command and operating system processing elements.
- Each processing element has 128 MB of local memory.
- The total memory in the application PEs is 28 GB.
- The total theoretical peak performance of the application processors is 168 Gflop/s ($= 224 \times 750$ Mflop/s).
- The local disk space is over 300 GB.

This configuration may change in the future. Use the command `grmview` to find out the current situation.

## 3.2 Distributed memory

The T3E has a physically distributed and a logically shared memory architecture. Access to the local memory inside current processing element is faster than to the remote memory.

Essentially, the T3E is a MIMD (Multiple Instruction, Multiple Data) computer although it supports SIMD (Single Instruction, Multiple Data) programming style.

The operating system software of the Cray T3E system is functionally distributed among the PEs. For every 16 PEs dedicated to user computation, there is, on average, one additional system PE. System PEs are added to provide operating system services and to handle the interactive load of the system, e.g., compiling and editing...

## 3.3 Processing elements

The T3E at CSC is physically composed of 224 + 16 = 240 nodes. Each node in the T3E consists of a processing element (PE) and interconnection network components. Each PE contains a DEC Alpha 21164 RISC microprocessor, local memory and support circuitry. Figure 3.1 illustrates the components inside one node.

Each PE has its own local memory. The global memory consists of these local memories.

The Cray T3E memory hierarchy has several layers: registers, on-chip caches (level 1 and level 2), local memory and remote memory. The processor bus bandwidth is in the range of 1 GB/s but the local memory bus speed is limited to 600 MB/s.

To enhance the performance of the local memory access, there is a mechanism called *stream buffers* or *streams* in the Cray T3E. Six streams fetch data in advance from the local memory when small-strided memory references are recognized.

The consequences of simultaneous remote memory operations (see Section 3.6) and streamed memory access to the same location in memory can be fatal. There is a possibility of data corruption and even of a system hang. Therefore it is very important to synchronize local and remote memory transfers or to separate memory areas for remote transfers.

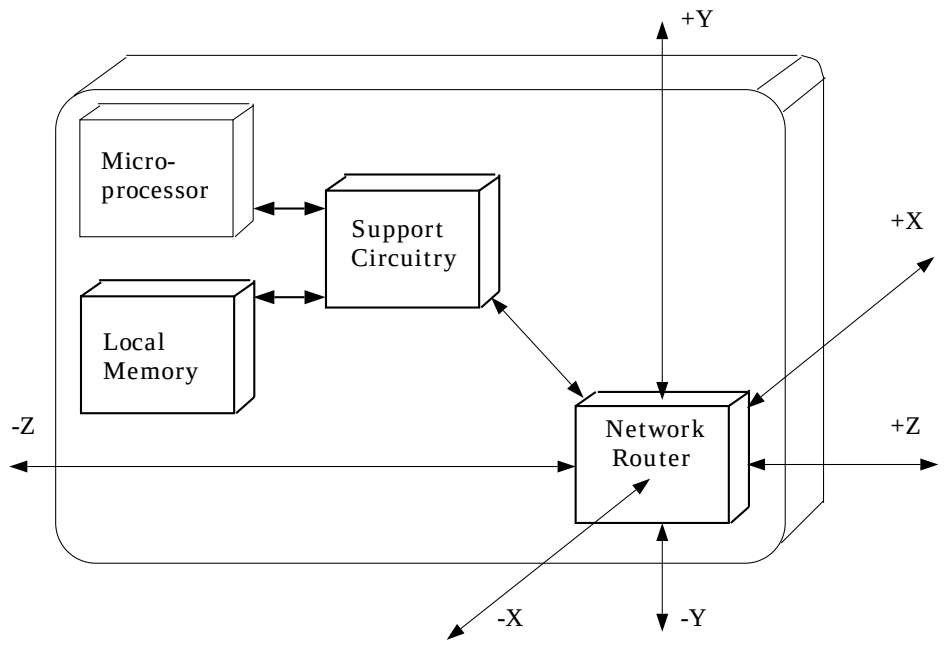



Figure 3.1: The components of Cray T3E node.

This is a problem only if you are using the SHMEM library for communication. The MPI library, for example, handles the streams mechanism properly.

The streams mechanism can be disabled or enabled on the user level by using the environment variable `$SCACHE_D_STREAMS`. To enable streams, give the command

```
setenv SCACHE_D_STREAMS 1
```

before executing your program. To disable streams, set the environment variable to 0. Use the command `udbsee` to see whether you have rights to use the streams mechanism.

You can also set the streams using the C/C++ function

```
#include <mpp/rastream.h>
void set_d_stream(int ss);
```

or the Fortran routine

```
INTEGER :: ss
CALL SET_D_STREAM(ss)
```

See the manual pages for more details (`man intro_streams`).

| *Attribute* | *Value* |
|---|---|
| Processor type | DEC Alpha 21164 |
| Physical address base | 40 bits |
| Virtual address base | 43 bits |
| Clock rate on the T3E | 375 MHz |
| Peak floating-point rate | 750 Mflop/s |
| Peak instruction issue rate | 4 (2 floating-point + 2 integer) |
| Size of the on-chip instruction cache | 8 kB |
| Size of the on-chip level 1 data cache | 8 kB |
| Size of the on-chip level 2 data cache | 96 kB |

Table 3.1: Characteristics of the DEC Alpha 21164 processor.

## 3.4 Processor architecture

The microprocessor in each of the T3E nodes is a DEC Alpha 21164, a RISC processor manufactured by COMPAQ/Digital. This 64-bit processor is cache-based, superscalar, and has pipelined functional units. It supports the IEEE standard for 32-bit and 64-bit floating point arithmetics.

The range of a 64-bit floating point number is

$$2.2250738585072014 \cdot 10^{-308} \ldots 1.7976931348623157 \cdot 10^{+308}.$$

The mantissa contains 53 bits, and therefore the precision is about 16 decimal numbers.

Correspondingly, 32-bit floating point numbers are between

$$1.17549435 \cdot 10^{-38} \ldots 3.40282347 \cdot 10^{+38}$$

The mantissa contains 24 bits (the leading bit is not stored), and the precision is about 7 decimal numbers.

Specific characteristics of the processor are presented in Table 3.1. The structure of the processor is illustrated in Figure 3.2.

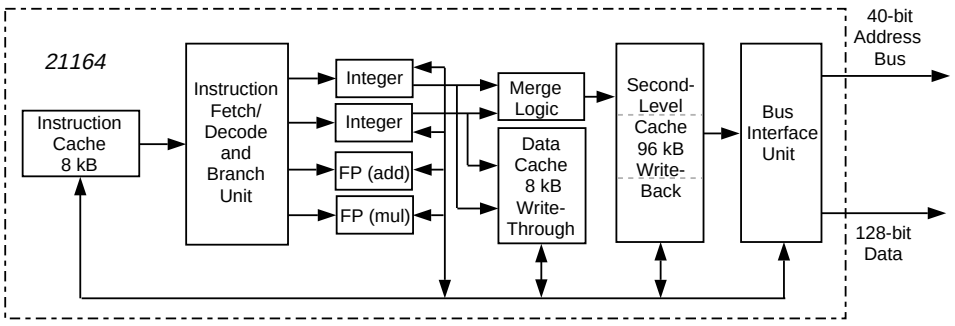



Figure 3.2: The DEC Alpha 21164 processor architecture.

## 3.5 Local memory hierarchy

The local four-level memory hierarchy of the processing elements is shown in Figure 3.3. Nearest to the execution units are the registers. Caches for instructions and data (ICACHE and DCACHE) are each of size 8 kB. The second-level cache, SCACHE (96 kB in total), is on the Alpha chip. The fourth level of the memory hierarchy is the main (DRAM) memory (128 MB).

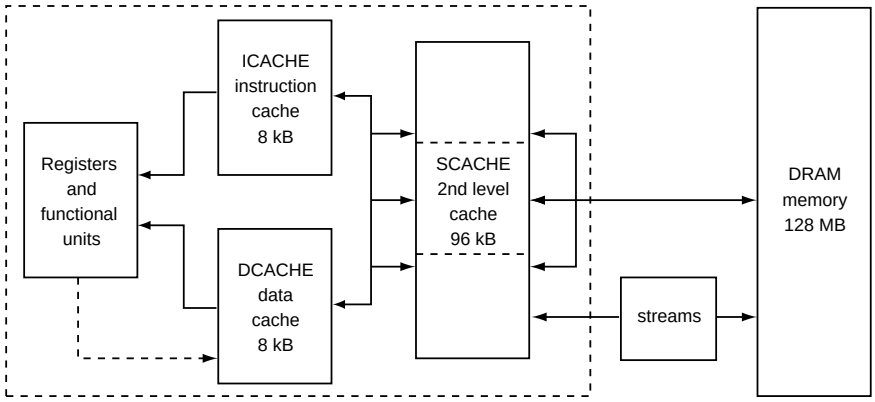



Figure 3.3: The local memory hierarchy.

It takes 2 clock periods (cp) to start moving a value from the first level data cache DCACHE to registers. The bandwidth is 16 bytes in a cp.

The size of the DCACHE is 8 kB, or 1024 words of 8 bytes. The cache is divided into 256 lines of 32 bytes each. Each read operation allocates one line in DCACHE for moving data from the 2nd level cache (SCACHE) or the main memory. This means that four consecutive 64 bit words are read at a time. Therefore, *arrays should always be indexed using the stride of one!*

For example, if you have a loop which indexes array elements which are 8 kB apart in memory, all the elements will be stored to the same DCACHE position. Therefore the data has to be fetched from a lower level of the memory hierarchy each time. This kind of memory reference pattern slows down the program considerably.

The second level cache (SCACHE) is of size 96 kB. This cache is three-way set-associative, which means that each location in the central memory can be loaded to three different locations in the SCACHE. This mapping is random and the programmer can not dictate it. Therefore, from the programmer's point of view, the SCACHE is actually of size 32 kB or a third of the physical size.

Each part of the set-associative SCACHE is direct-mapped to the memory in the same way as DCACHE is. You can fit 4096 words (each 8 bytes) to each of the three parts of the SCACHE. The latency of SCACHE is 8 cp for moving data to the DCACHE. The bandwidth is 16 bytes in a cp, or

two words in each cp. An SCACHE line is 64 bytes. Therefore, data is moved in consecutive blocks of 64 bytes from the main memory.

When you are optimizing your code, the most important thing is to optimize the usage of the DCACHE. Almost as important is to optimize the usage of the SCACHE.

Because of the reasons mentioned above, try to avoid step sizes of 8 kB or 32 kB when you are referencing memory. The most optimal way is to use stride one, which in the case of Fortran means changing the first index of arrays with a step size of one.

Here is a simple example of memory references:

```
REAL, DIMENSION(n) :: a, b
REAL, DIMENSION(n,n) :: c
INTEGER :: i, j

DO i = 1, n
  DO j = 1, n
    c(i,j) = c(i,j) + a(i)*b(j)
  END DO
END DO
```

If the constant `n` is of size 1024, the code runs very slowly due to the memory references `c(1,1)`, `c(1,2)`, `c(1,3)` etc., which are 8 kB apart in memory. You should rearrange the loops as follows to get better performance:

```
DO j = 1, n
  DO i = 1, n
    c(i,j) = c(i,j) + a(i)*b(j)
  END DO
END DO
```

## 3.6 Interprocessor communication

The system PEs of the Cray T3E are connected through a high-speed, low-latency interconnection network. The peak data-transfer speed between processors is 480 MB/s in every direction through the bi-directional 3D torus network. The hardware latency is less than 1 $\mu$s.

The T3E system interconnection network operates asynchronously and independently from the PEs to access and redistribute global data. The 3D torus topology ensures short connection paths. The bisectional bandwidth is also high (measured by splitting the machine in half and finding out the maximum transfer rate between these parts). The topology has also the ability to avoid failed communication pathways.

An example of routing through the interconnection network is presented in Figure 3.4.

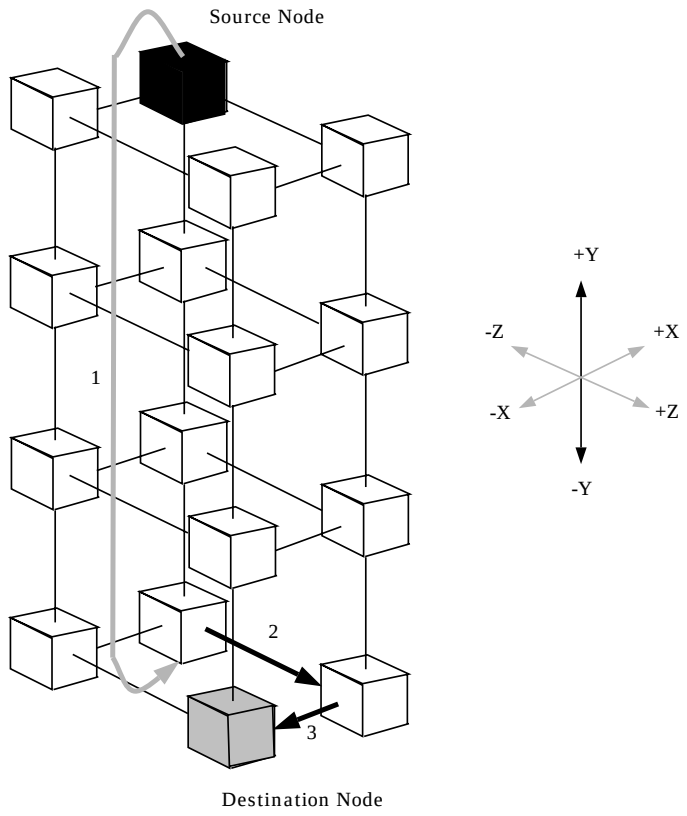



Figure 3.4: A routing example through the 3D torus network of the T3E.

Addressing of remote memory is managed by the External Register Set, or *E-registers*. Latency hiding and synchronization are integrated in 512 + 128 off-chip memory-mapped E-registers. The E-registers consist of a general set of 512 external registers that manage asynchronous data transfer between nodes by providing the destinations and/or sources of all remote references. The second set of 128 registers are reserved for the operating system. E-registers are used by predefined op-codes.

Each PE has 32 Barrier/Eureka Synchronization Units (BESUs), used to implement barrier and eureka type synchronization and atomic operations. Barriers may be used, among other things, to execute SIMD codes efficiently. Eureka operations can be used to indicate, for example, that one PE has found a solution.

The virtual synchronization networks have higher priority for the physical channel between nodes than any other traffic. Therefore the global synchronization is very efficient.

## 3.7 External I/O

The T3E system has four processing elements per one I/O controller, while one out of every two I/O controllers is connected to a GigaRing controller. These controllers can be connected to external I/O clients through high-speed GigaRing channels. Figure 3.5 illustrates the I/O path from a PE to an external disk device.

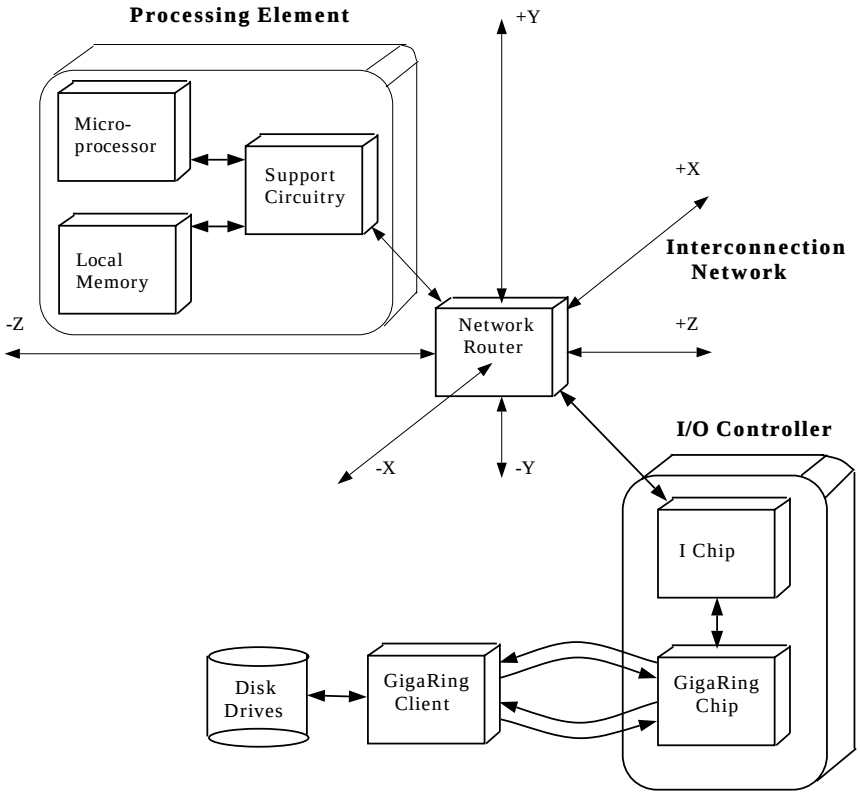



Figure 3.5: A Cray T3E node with external I/O.

The GigaRing architecture is implemented using a dual-ring design, with data in the two rings traveling in opposite directions. The raw data bandwidth of one ring is up to 600 MB/s which gives a total of 1200 MB/s per channel. The peak data bandwidth is 800 MB/s per channel for half-duplex connections and $2 \times 700$MB/s for full-duplex connections between two GigaRing nodes. The data bandwidth from a T3E node is limited by the connection through the Network Router and the I/O controller. Thus, the bandwidth is up to 500 MB/s.

GigaRing channels can be configured with multiple nodes of different type, a Multi Purpose Node (MPN) or a Single Purpose Node (SPN). In an MPN several types of I/O controllers can be installed: FDDI, Ethernet, ATM or SCSI.

## 3.8 The UNICOS/mk operating system

The Cray T3E has a distributed microkernel based operating system. This provides a single system image of the global system to the user. UNICOS/mk is a Unix-like operating system based on Cray's UNICOS system, which runs on parallel vector processor (PVP) platforms such as the Cray C90.

The microkernel is based on the CHORUS technology. It provides basic hardware abstraction, memory management, thread scheduling and interprocessor communication between the processes.

Several processors offer operating system (OS) services, "servers". These servers look like normal processes running on top of the microkernel. The most important OS servers are listed in Table 3.2. Additionally, various servers manage logging, devices, and other operating system services.

| *Server* | *Function* |
|---|---|
| Process Manager (PM) | Manages processes local to its PE |
| Global Process Manager (GPM) | Manages list of all known processes |
| Global Resource Manager (GRM) | Tracks and allocates the resource management |
| Configuration Server (CS) | Processes requests for system configuration data |
| File Server | Provides file system services |
| File Server Assistant | Provides file system services locally |

Table 3.2: Some UNICOS/mk high-level OS servers.

The user communicates with the servers using normal Unix-type application programming interfaces (APIs) like in any other Unix system, i.e., using libraries and system calls.

Scalability is an important issue in a system like the Cray T3E. When the system size is increased, not only the number of application processors is affected, but also the number of command and OS processors. The command processors run interactive user jobs and the operating system processors run OS servers.

## 3.9 File systems

The Cray T3E running UNICOS/mk has a similar file system structure as many other Unix systems. These include the file systems / (root directory), `/usr/` and `/tmp/`.

The T3E file systems at CSC are located on striped FiberChannel disks residing in one GigaRing, which is attached to a Multi Purpose Node (MPN). The total disk capacity is over 300 GB. Most of the space is allocated for paging (swapping), `$TMPDIR` and `$WRKDIR`.

## 3.10 Resource monitoring

The most useful commands for viewing the global configuration and status of the Cray T3E system are `grmview` and `top`.

The `grmview` command displays information on the Global Resource Manager (GRM) regarding the current PE configuration (PE map), applications currently running on the PEs and applications waiting to run on the PEs.

Here is an extract of the output of the command `grmview -l`:

```
PE Map: 240 (0xf0) PEs configured
              Ap. Size  Number Aps.  Abs.
 Type    PE  min  max running limit limit    x  y  z Clock UsrMem FreMem
+ APP     0    2  192       1     1     2    0  0  0   375    118     19
+ APP   0x1    2  192       1     1     2    1  0  0   375    118     20
+ APP   0x2    2  192       1     1     2    0  1  0   375    118     20
+ APP   0x3    2  192       1     1     2    1  1  0   375    118     20
+ APP   0x4    2  192       1     1     2    0  2  0   375    118     20
+ APP   0x5    2  192       1     1     2    1  2  0   375    118     20
+ APP   0x6    2  192       1     1     2    0  3  0   375    118     20
+ APP   0x7    2  192       1     1     2    1  3  0   375    118     20
+ APP   0x8    2  192       1     1     2    2  0  0   375    118     20
+ APP   0x9    2  192       1     1     2    3  0  0   375    118     20
...
+ APP  0x4a    2  192       1     1     2    2  1  2   375    118     69
+ APP  0x4b    2  192       1     1     2    3  1  2   375    118     69
+ APP  0x4c    2  192       1     1     2    2  2  2   375    118     69
+ APP  0x4d    2  192       1     1     2    3  2  2   375    118     69
+ APP  0x4e    2  192       1     1     2    2  3  2   375    118     69
+ APP  0x4f    2  192       1     1     2    3  3  2   375    118     69
+ APP  0x50    2  192       0     1     2    4  0  2   375    118    118
+ APP  0x51    2  192       0     1     2    5  0  2   375    118    118
+ APP  0x52    2  192       0     1     2    4  1  2   375    118    118
+ APP  0x53    2  192       0     1     2    5  1  2   375    118    118
...
+ APP  0xd0    2  192       0     1     2    5  1  6   375    118    118
+ APP  0xd1    2  192       0     1     2    4  2  6   375    118    118
+ APP  0xd2    2  192       0     1     2    5  2  6   375    118    118
+ APP  0xd3    2  192       0     1     2    4  3  6   375    118    118
+ APP  0xd4    2  192       0     1     2    5  3  6   375    118    118
+ APP  0xd5    2  192       0     1     2    6  0  6   375    118    118
+ APP  0xd6    2  192       0     1     2    7  0  6   375    118    118
+ APP  0xd7    2  192       0     1     2    6  1  6   375    118    118
+ APP  0xd8    2  192       0     1     2    7  1  6   375    118    118
+ APP  0xd9    2  192       0     1     2    6  2  6   375    118    118
+ APP  0xda    2  192       0     1     2    7  2  6   375    118    118
+ APP  0xdb    2  192       0     1     2    6  3  6   375    118    118
+ APP  0xdc    2  192       0     1     2    7  3  6   375    118    118
```

```
+ APP  0xdd    2  192       0     1     2    0  0  7   375    118    118
+ APP  0xde    2  192       0     1     2    1  0  7   375    118    118
+ APP  0xdf    2  192       0     1     2    0  1  7   375    118    118
+  OS  0xe0    0    0       0     0     0    1  1  7   375     92     40
+ CMD  0xe1    1    1       0 unlim unlim    0  2  7   375    115     42
+ CMD  0xe2    1    1       0 unlim unlim    1  2  7   375    115     92
+ CMD  0xe3    1    1       0 unlim unlim    0  3  7   375    116     87
+ CMD  0xe4    1    1       0 unlim unlim    1  3  7   375    116     97
+ CMD  0xe5    1    1       0 unlim unlim    2  0  7   375    116    100
- NUL  0xe6    0    0       0     0     0    0  3  6     0      0      0
- NUL  0xe7    0    0       0     0     0    1  2  6     0      0      0
+ CMD  0xe8    1    1       0 unlim unlim    3  0  7   375    117    100
+ CMD  0xe9    1    1       0 unlim unlim    2  1  7   375    117     99
+ CMD  0xea    1    1       0 unlim unlim    3  1  7   375    117    100
+ CMD  0xeb    1    1       0 unlim unlim    2  2  7   375    117    113
+ CMD  0xec    1    1       0 unlim unlim    3  2  7   375    117     99
+ CMD  0xed    1    1       0 unlim unlim    2  3  7   375    116     99
+ CMD  0xee    1    1       0 unlim unlim    3  3  7   375    116    113
+  OS  0xef    0    0       0     0     0    0  0  3   375    105    105

Exec Queue: 4 entries total. 4 running, 0 queued
  uid   gid  acid    Label Size  BasePE   ApId Command           Note
 nnnn  nnnn  nnnn      -      16    0x68  11839 ./prog1             -
 nnnn  nnnn  nnnn      -      64       0  16931 ./prog2             -
 nnnn  nnnn  nnnn      -      16    0x40  36477 ./prog3             -
 nnnn  nnnn  nnnn      -      64    0x78  38718 ./prog4             -
```

The listing indicates that 224 processors are application nodes (APP), two are operating system nodes (OS), and 12 are command nodes (CMD). This listing also shows that two processors were non-operational. Four parallel jobs were running using 16–64 processors.

The listing also shows that all processors have the clock rate of 375 MHz. Earlier there were processors having different clock rates, but now all are running at the same speed.

The `grmview` command also shows the coordinates of the PEs in the 3D torus. You can see that the size of the torus is $8 \times 4 \times 7$, so the torus is not a complete cube.

The `top` command gives a global picture of the system status at a glance. Here is an example of the output:

```
last pid:     5;  load averages:  0.00,  0.00,  0.00              09:37:10
116 processes: 110 sleeping, 6 running
CPU states: 31.9% idle, 66.7% user,  0.0% syscall,  1.4% kernel,  0.0% wait
Memory: 30464M physical, 28025M usrmem, 2428M sysmem, 16447M free

  PID USERNAME PRI NICE   RES STATE   TIME     CPU NPE @PE COMMAND
 1406 user1     24    0  653M run    79.9H  100.0%  32  64 prog1.x
 1688 user1     -5    0 1861M run    69.7H  100.0%  32  96 prog2.x
 9985 user2     -5    0 4582M run    24.3H   99.6%  64 128 prog3.x
 9864 user3     34    0  245M run   391:11   97.3%  16  20 prog4.x
 8817 user4     -5    0 1062M run   707:08   94.9%  16   0 prog5.x
10501 jhaataja  34    4 7120K sleep   0:04   25.5%   1 195 top
```

This listing shows that currently about 67% of the computer is used for computing. The `top` command also shows the total memory usage and the total memory requirements of the running processes.

You can also use the command

```
ps -PeMf
```

to see what parallel processes are running. Here is an extract from the output:

```
F S      UID   PID  PPID  ...  STIME TTY   TIME CMD
1 R    user1  1406  1386       11:17 ?    147:3 ./prog1.x
1 R    user1  1688  1617       11:36 ?    129:5 ./prog2.x
1 R    user2  8817  8809       13:03 ?    44:26 ./prog3.x
1 R    user3  9864  9855       13:22 ?    24:04 ./prog4.x
1 R    user4  9985  9922       13:24 ?    23:15 ./prog5.x
```

You can compare this with the output of the `top` command shown above.

## 3.11 More information

To get more information about the Cray T3E hardware architecture and the system software, a good place to start are the WWW pages of Cray Research, Inc.:

```
http://www.cray.com
```

The current configuration of the T3E at CSC can also be found on the WWW pages at CSC:

```
http://www.csc.fi/metacomputer/crayt3e.html
```

# Chapter 4

# Program development

This chapter shows how to compile and run your programs on the Cray T3E at CSC. Fortran programming is discussed in more detail in Chapter 5 and C/C++ in Chapter 6. Parallel programming (message passing etc.) is discussed in Chapter 7.

## 4.1 General overview

The Cray T3E environment for program development is automatically initialized upon logging in or startup of a batch job. This environment consists of a Fortran 90 compiler (`f90`) and ANSI C (`cc`) and C++ (`CC`) compilers. You can also use efficient mathematical and scientific libraries, e.g., to obtain good performance in linear algebra operations.

You can do parallelization with native MPI and PVM message-passing libraries, or with SHMEM, the Cray-specific one-sided communication library. The HPF data-parallel language is also available.

The system offers the Cray TotalView parallel debugger and the performance monitoring tools MPP Apprentice and PAT as help to program development. The VAMPIR software package can be used to trace and profile message-passing programs visually.

## 4.2 Compiling and linking programs

Both the Fortran 90 compiler (`f90`) and the C/C++ compilers (`cc` and `CC`) accept a few common compiler options.

The option `-Xn` or `-X n` is used to indicate how many processors you want for your application. If you do not provide this option, the program can be run on any number of processors using the `mpprun` command. This kind of executable is called *malleable*.

Here is a typical example of generating and running a *non-malleable* executable, which has to be run on a fixed number of PEs:

```
t3e% f90 -X 16 -o prog.x prog.f90
t3e% ./prog.x
```

Here we are using 16 processors for our application. However, we can also generate and run a malleable executable `prog.x`:

```
t3e% f90 -o prog.x prog.f90
t3e% mpprun -n 16 ./prog.x
t3e% mpprun -n 8 ./prog.x
```

Here the number of processors was given to the `mpprun` command and the option `-X` was omitted. The program was first run on 16 processors and then on eight processors.

The same applies to the C/C++ compilers. Here is an example of compiling and running a C program:

```
t3e% cc -o prog.x prog.c
t3e% mpprun -n 8 ./prog.x
```

Here we used eight processors for running our malleable executable `prog.x`.

*The MPI, PVM, and SHMEM libraries are automatically linked to your application, when needed.* Therefore, you do not need to provide any special flags to be able to use, e.g., MPI calls. The Cray scientific library (Libsci) is also automatically linked to your program.

The option `-O` indicates the optimization level used in the compilation. If you are running production jobs, you should always turn code optimization on! The default is moderate optimization, but you can request more aggressive optimization.

The Fortran 90 compiler is discussed in more detail in Chapter 5, and the C/C++ compiler in Chapter 6.

## 4.3 Libsci — Cray scientific library

Libsci is a collection of various mathematical subroutines. Most of the routines solve some specific problem of linear algebra, but there are a few routines for fast Fourier transforms as well.

Libsci is Cray's own package of subroutines for computational tasks. Libsci is divided into following sublibraries: BLAS 1,2,3, LAPACK, BLAS_S,

PBLAS, ScaLAPACK, BLACS, and FFT. The most straightforward way to obtain more information on these libraries is through the `man` command as follows:

```
man intro_lapack
```

There is a manual page for almost all subroutines in Libsci, the most notable exception being the routines under the PBLAS library. Unfortunately, there are also manual pages for some non-existent routines such as those solving sparse linear systems.

For more information, you can look up the following WWW addresses:

| *Package* | *WWW address* |
|---|---|
| BLAS | `http://www.netlib.org/blas/index.html` |
| LAPACK | `http://www.netlib.org/lapack/` |
| ScaLAPACK | `http://www.netlib.org/scalapack/` |
| PBLAS | `http://www.netlib.org/scalapack/html/pblas_qref.html` |
| BLACS | `http://www.netlib.org/blacs/index.html` |

The Libsci library is automatically linked when programs are loaded.

### 4.3.1 BLAS

BLAS (Basic Linear Algebra Subroutines) is the first in a series of subroutine packages designed for solving efficiently computational problems in linear algebra. As the name indicates, the tasks that the BLAS routines perform are of the most fundamental kind: adding and multiplying vectors and matrices.

BLAS is divided into three levels: level 1 routines handle operations between two vectors, level 2 routines take care of operations between a vector and a matrix, and, finally, the routines at level 3 can manipulate two or more matrices. For instance, the routine SDOT computes the dot product (inner product) of two vectors. This routine belongs to the level 1 BLAS, whereas the SGEMV routine multiplies a vector by a matrix and is thus a level 2 routine.

### 4.3.2 LAPACK

LAPACK (Linear Algebra PACKage) is a collection of subroutines aimed for more complicated problems such as solving a system of linear equations or finding the eigenvalues of a matrix. LAPACK is built on top of BLAS.

### 4.3.3 BLACS

Both BLAS and LAPACK are developed for single processor computations. In order to solve problems of linear algebra on parallel machines where matrices can be distributed over several processors, we need to communicate data between the processors. For this purpose there is a special library called BLACS (Basic Linear Algebra Communication Subroutines).

The routines in BLACS can be divided into three classes. First, there are communication routines for sending and receiving parts of matrices between two or more processors. Second, there are global reduction routines in which all processors take part. An example of these is finding the element of the largest absolute value in a distributed matrix. Third, there are a few general support routines for setting up the communication network.

### 4.3.4 PBLAS and ScaLAPACK

PBLAS (Parallel BLAS) and ScaLAPACK (Scalable LAPACK) are parallelized versions of BLAS and LAPACK, respectively. The names of the multiprocessor routines in these libraries are almost the same as the ones used for the corresponding single processor routines, except for an initial P for “parallel”.

For some obscure reason, Cray has not documented the PBLAS library at all, except for a short notice on the ScaLAPACK manual pages about PBLAS being supported. For ScaLAPACK, the situation is somewhat better, since for all available routines there is a manual page. On the other hand, the current implementation of ScaLAPACK on the T3E does not support all routines available in the public domain version. See Table 4.1 for the existing ScaLAPACK routines.

### 4.3.5 Details

All of the above mentioned libraries follow a naming convention. This dictates that any subroutine operating with single precision floating-point numbers should be given a name beginning with S. Correspondingly, those routines accepting double precision floating point numbers as arguments have a name beginning with D (not counting the letter P for parallel versions, which precedes the actual name).

However, since the single precision floating point numbers on the T3E have 8 bytes, which on most other computers corresponds to the double precision, you should make sure that you change not only the type definitions of the variables but also all calls to BLAS etc. accordingly. Note that on the T3E there are no BLAS routines starting with the letter D.

| *Routines* | | *Explanation* |
|---|---|---|
| PSGETRF | PCGETRF | LU factorization and solution of linear general |
| PSGETRS | PCGETRS | distributed systems of linear equations |
| PSTRTRS | PCTRTRS | |
| PSGESV | PCGESV | |
| PSPOTRF | PCPOTRF | Cholesky factorization and solution of real |
| PSPOTRS | PCPOTRS | symmetric or complex Hermitian distributed |
| PSPOSV | PCPOSV | systems of linear equations |
| PSGEQRF | PCGEQRF | QR, RQ, QL, LQ, and QR with column pivoting |
| PSGERQF | PCGERQF | for general distributed matrices |
| PSGEQLF | PCGEQLF | |
| PSGELQF | PCGELQF | |
| PSGEQPF | PCGEQPF | |
| PSGETRI | PCGETRI | Inversion of general, triangular, real symmetric |
| PSTRTRI | PCTRTRI | positive definite or complex Hermitian positive |
| PSPOTRI | PCPOTRI | finite distributed matrices |
| PSSYTRD | PCHETRD | Reduction of real symmetric or complex Hermitian matrices to tridiagonal form. |
| PSGEBRD | PCGEBRD | Reduction of general matrices to bidiagonal form |
| PSSYEVX | PCHEEVX | Eigenvalue solvers for real symmetric or complex Hermitian distributed matrices |
| PSSYGVX | PCHEGVX | Solvers for generalized eigenvalue problem with real symmetric or complex Hermitian distributed matrices |
| INDXG2P | | Computes the coordinate of the processor in the two-dimensional (2D) processor grid that owns an entry of the distributed array |
| NUMROC | | Computes the number of local rows or columns of the distributed array owned by a processor |

Table 4.1: The ScaLAPACK routines on the Cray T3E.

In an exactly similar fashion, the subroutines for complex arithmetics have always a C as their first letter, never a Z. Thus, you should call CGEMV, not ZGEMV.

BLACS, PBLAS and ScaLAPACK libraries all share the same method to distribute matrices and vectors over a processor grid. This distribution is controlled by a vector called the descriptor. The descriptor in the ScaLAPACK implementation in T3E used to differ from the one specified in the manuals, but this is no longer true. Thus, with respect to the composition of the descriptor, ScaLAPACK codes should be portable to other machines.

## 4.4 The NAG subroutine library

The NAG library is a comprehensive mathematical subroutine library that has become a *de facto* standard in the field of numerical programming. NAG routines are *not* parallelized on the T3E. The implemented single PE version is Mark 17 (July 1998).

NAG routines cover several branches of numerical mathematics including ordinary and partial differential equations, integral equations, interpolation, finding the extreme values and zeros of a function, statistical analysis, and linear algebra.

Because of Cray's precision conventions explained in the previous section, all routines should use formally single precision (real) arithmetic, which corresponds to double precision on most other computers. Thus the names of the NAG routines should end in the letter E, e.g., F06ABE instead of F06ABF.

The linear algebra subroutines in Chapter F07 of the NAG library as well as the least-squares algorithms in Chapter F08 call LAPACK routines in Cray's Libsci.

A program containing calls to NAG routines is compiled with the option `-lnag`, e.g.,

```
f90 -o prog prog.f90 -lnag
```

For a thorough introduction to the NAG library, it is necessary to browse the manuals, if available at your local computer center. Alternatively, you may read the file `$DOC/nagdoc/essint` where the essential principles are described.

If you are already familiar with NAG, you can try to decide which routine to use by studying the file `$DOC/nagdoc/summary`. On the T3E, you can find more information about NAG with the command

```
man nag_fl_un
```

These manual pages give mainly T3E dependent details. Files

```
$DOC/nagdoc/called
$DOC/nagdoc/calls
```

contain information about the calls between various NAG routines. The most important features of Mark 17 are collected in the file

```
$DOC/nagdoc/news
```

See also the file `$DOC/nagdoc/replaced` if you are interested in the differences between Mark 17 and previous releases.

There is a collection of almost 1000 example codes in the directory `$DOC/nag_examples/source/` with the associated data files in the directory `$DOC/nag_examples/data/`. The correct results are stored in the directory `$DOC/nag_examples/results/`.

You can also use the NAG on-line documentation on Cypress by the command

```
naghelp
```

## 4.5 The IMSL subroutine library

The IMSL library is another general purpose mathematical subroutine library with two separate parts, MATH/LIBRARY and STAT/LIBRARY. The release installed on the T3E is the IMSL FORTRAN 90 MP Library version 3.0.

What has been stated about the precision of arithmetic operations above applies here as well: programs should introduce nominally only single precision (real) variables, even though all calculations are carried out in 8 byte operations.

Before running any application programs which use the IMSL routines you must give the initialization command

```
use imsl
```

After this, compilation and linking take place as follows:

```
f90 -o prog prog.f90 -p$MODULES_F90 $LINK_F90 -Xm
```

Note: when calling IMSL routines one *must* specify the non-malleable option `-Xm`, because the IMSL library is compiled for fixed one processor.

IMSL is documented in a four-volume manual. Unfortunately, IMSL offers no on-line documentation on the T3E. However, one can access the the hypertext help system for IMSL on Caper Cypress and Cypress2 with the command

```
imsl.help
```

and the text-based help system on Cray C94 by

```
imsl.idf
```

There is also a large collection of example codes in the directory

```
$DOC/imsl_examples/
```

## 4.6 More information

Chapter 5 discusses Fortran programming in more detail and Chapter 6 C and C++ programming.

The manual *Introducing CrayLibs* [Crad] contains a summary of Cray scientific library routines.

You can use `help` to get some information about the IMSL and NAG libraries in the CSC environment with the commands

```
help imsl
help nag
```

You can also use the NAG and IMSL help systems on other computers at CSC as described in Sections 4.4 and 4.5.

# Chapter 5

# Fortran programming

The Cray T3E offers a Fortran 90 compiler which can be used to compile standard-conforming FORTRAN 77 programs as well. This chapter discusses the most essential compiler features. Parallel programming is described in Chapter 7. Programming tools are discussed in Chapter 9.

## 5.1 The Fortran 90 compiler

The Cray T3E Fortran 90 compiler (CF90) supports a full implementation of the ANSI and ISO Fortran 90 standard. The compiler also includes many traditional Cray-specific features, such as Cray-style pointers.

A separate FORTRAN 77 compiler is not (and will not be) available on the Cray T3E. Because the FORTRAN 77 standard is included in the Fortran 90 programming language, you can also compile FORTRAN 77 programs with the Cray CF90 compiler. You should note, however, that some Fortran programs contain vendor-specific extensions which may or may not be compatible with the Fortran 90 standard.

Note that the Cray Fortran 90 compiler is unable to parallelize your code automatically. Instead, you must use explicit methods such as message-passing (MPI or PVM) libraries, or the data-passing library (SHMEM).

The data-parallel programming language HPF (High Performance Fortran) with HPF_CRAFT extensions is also available. This programming model, supporting HPF directives in the Fortran source code, is discussed in Section 7.5.

## 5.2 Basic usage

The CF90 compiler is invoked using the command `f90` followed by optional compiler options and the filenames to be compiled:

```
t3e% f90 [options] filenames
```

If the `-c` option is not specified, the `f90` command will automatically invoke the linker to create an executable program.

You can compile and link in a single step:

```
t3e% f90 -o prog.x prog.f90 sub.f90
```

Here the source code files `prog.f90` and `sub.f90` were compiled into the executable program `prog.x`. The compilation and linking can also be done in several steps using the `-c` option:

```
t3e% f90 -c prog.f90
t3e% f90 -c sub.f90
t3e% f90 -o prog.x prog.o sub.o
```

This way, only the changed program units have to be compiled before linking the `.o` files.

The malleable program `prog.x` can now be executed using the `mpprun` command:

```
t3e% mpprun -n 16 ./prog.x
```

Here we used 16 processors.

The program can also be compiled and linked into a non-malleable executable by:

```
t3e% f90 -X 16 -o prog.x prog.f90 sub.f90
```

## 5.3 Fixed and free format source code

The Fortran compiler uses filename extensions to distinguish different types of files. The compiler interprets the extensions `.f` and `.F` to mean the traditional *fixed form* of source code ("FORTRAN 77 style"). The extensions `.f90` and `.F90` imply the new *free form* of source code. You can override these defaults using the options `-f fixed` and `-f free`. Table 5.1 illustrates the use of the file extensions.

| *File extension* | *Type* | *Notes* |
|---|---|---|
| `.f` | Fixed source form (72 columns) | No preprocessing |
| `.f90` | Free source form (132 columns) | No preprocessing |
| `.F` | Fixed source form (72 columns) | Preprocessing |
| `.F90` | Free source form (132 columns) | Preprocessing |
| `.o` | Object file | Passed to linker |
| `.a` | Object library file | Passed to linker |
| `.s` | Assembly language file | Passed to assembler |

Table 5.1: The interpretation of some filename extensions.

## 5.4 Compiler options

You can control the compilation process using compiler options. The most common situation is to increase the optimization level. The following command sequence illustrates a typical compilation process and creation of an executable (`master.x`). The source code is in the files `master.f90` and `shallow.f90`.

```
t3e% f90 -c master.f90
t3e% f90 -c -O3 -Obl,aggress,split2,unroll2 shallow.f90
t3e% f90 -o master.x master.o shallow.o
```

In the previous example we used the option `-O3`. This normally generates faster programs with reduced turnaround time. The cost of code optimization is increased compilation time which can sometimes be excessive. The size of the executable can also increase.

You may also request more information about the optimizations made by the compiler in the form of a listing file. Compiler options can also be used to activate debugging or performance tracing (see Chapter 9).

The default size of `REAL` and `INTEGER` variables is 8 bytes or 64 bits, which can be changed to 32 bits with the option `-s default32`.

Table 5.2 lists the most important CF90 compiler options.

Without explicit compiler options, the compiler assumes conservative optimization levels, which do not introduce side effects. Some features can be enabled or disabled with the `-e` and `-d` options, see Table 5.3.

## 5.5 Optimization options

It is very important to note that single-CPU code optimization is essential in getting good performance on the Cray T3E. If the speed of your code is only 10 Mflop/s per processor, compared to the peak performance of

| *Option* | *Explanation* |
|---|---|
| `-c` | Compile only, do not attempt to link |
| `-r2` | Request for standard listing file (`.lst`) |
| `-r6` | Request for full listing |
| `-rm` | Request for listing with loopmarks |
| `-i32` | Treat `INTEGER` as 32-bit (4 bytes) |
| `-s default32` | Treat `INTEGER`, `REAL` and `LOGICAL` as 32-bit |
| `-dp` | Treat `DOUBLE PRECISION` as `REAL` (default) |
| `-On` | Set the optimization level *n* (0,1,2,3) |
| `-O3` | Aggressive optimization |
| `-Osplit2` | Automatic loop splitting (check all loops) |
| `-Ounroll2` | Automatic loop unrolling (2 copies) |
| `-Oaggress` | Increases internal table limits for better optimization |
| `-Obl` | Enables bottom loading of scalar operands in loops |
| `-g` | Enable debugging of code with TotalView |

Table 5.2: Some CF90 compiler options. For further details about the options see the manual pages using the command `man f90`.

750 Mflop/s per processor, the parallel performance of the code will be poor even if the code parallelizes perfectly.

The optimization of Fortran codes is discussed in detail in the Cray publication *Cray T3E Fortran Optimization Guide* [Crac]. Therefore we only present a short review of the subject here.

The first step in the performance optimization is the selection of a robust and an efficient algorithm with regard to the parallel implementation. When available, tested and efficient library routines should be used. The code should be optimized first via compiler options and later, if necessary, by manual intervention. Furthermore, one should begin the optimization of a program from the parts that take most resources. This requires profiling the program that is discussed in Chapter 9.

The `-O` option of the `f90` command can be used to do several different types of code optimization. The option `-On`, where *n* is 0, 1, 2 or 3, is the basic way to select the optimization level.

Instead of giving the numerical optimization level, you can also request a specific type of optimization. For example, the option

```
-O aggress,scalar3,bl,unroll2,split2
```

specifies aggressive optimization with extensive scalar optimization and several techniques for loop optimization (bottom loading, unrolling and loop splitting).

The specific optimization types can also be selected by compiler directives in the source code (see Section 5.7 on page 45).

| *Option* | *Explanation* |
|---|---|
| -dn, -en | Report nonstandard code |
| -dp, -ep | Use double precision |
| -er, -dr | Round multiplication results |
| -du, -eu | Round division results upwards |
| -dv, -ev | Static storage |
| -dA, -eA | Use the Apprentice tool |
| -dI, -eI | IMPLICIT NONE statement |
| -dR, -eR | Recursive procedures |
| -dP, -eP | Preprocessing, no compilation |
| -dZ, -eZ | Preprocessing and compilation |

Table 5.3: Enabling or disabling some compiler features. The default option is listed first.

## 5.6 Optimizing for cache

The Cray T3E memory hierarchy is discussed in Section 3.5 on page 24. Here is an example of a poorly performing code fragment:

```
INTEGER, PARAMETER :: n = 4096
REAL, DIMENSION(n) :: a, b, c
COMMON /my_block/ a, b, c
INTEGER :: i

DO i = 1, n
  a(i) = b(i) + c(i)
END DO
```

Here the COMMON statement is used to ensure that the arrays a, b and c are in consecutive memory positions. Because of this, the elements a(1) and b(1) are 4096 words or 32 kB apart in memory, and they are thus mapped to the same line of the SCACHE. The same applies to b(1) and c(1). Because the elements are also a multiple of 1024 words apart, they also map to the same DCACHE line, which is even worse.

The size of the DCACHE is 8 kB, and the size of the SCACHE is effectively 32 kB. A DCACHE line is 32 bytes or 4 words, and a SCACHE line is 64 bytes or 8 words.

Because the array elements b(i) and c(i) map to the same cache line both in the DCACHE and in the SCACHE, each load operation of c(i) replaces the previously loaded b(i) value.

Since a complete cache line is read from memory at a time, also the adjacent memory locations are replaced. This causes *a lot of* unnecessary memory traffic.

You can improve the performance by padding the arrays so that the corresponding elements do not map to the same cache lines:

```
INTEGER, PARAMETER :: n = 4096, pad = 8
REAL, DIMENSION(n+pad) :: a, b
REAL, DIMENSION(n) :: c
COMMON /my_block/ a, b, c
```

The rest of the code is identical. The padding can also be done using extra arrays:

```
INTEGER, PARAMETER :: n = 4096, pad = 8
REAL, DIMENSION(n) :: a, b, c
REAL, DIMENSION(pad) :: temp1, temp2
COMMON /my_block/ a, temp1, b, temp2, c
```

After this, the read operations for arrays `a` and `b` do not map to the same DCACHE and SCACHE lines, and the write operations of the `c(i)` elements do not map to these cache lines. This makes the code run a lot faster!

Similar techniques can also be used with arrays of two or more dimensions.

## 5.7 Compiler directives

In addition to using compiler options, you can use the so-called *compiler directives* to control the compilation process. There are directives which help in code optimization, memory usage, checking array bounds etc.

Table 5.4 lists the most useful compiler directives.

| *Directive* | *Explanation* |
|---|---|
| `free`, `fixed` | Specifying source form |
| `[no]bounds` [*array*] | Array bounds checking |
| `integer=`*n* | Specifying integer length |
| `name` (*fortran_name*="*ext_name*") | Naming external routines |
| `[no]bl` | Bottom loading operands |
| `[no]split` | Loop splitting |
| `[no]unroll` [*n*] | Loop unrolling (*n* copies) |
| `cache_align` *var* | Align on cache line boundaries |
| `symmetric` [*var, ...*] | Declaring local addressing |

Table 5.4: Some compiler directives for the `f90` command.

Directives are written into the source code as special comments, and the CF90 compiler interprets them in the compilation phase.

Here is a short example:

```
!dir$ split
DO i = 1, 1000
  a(i) = b(i) * c(i)
  t = d(i) + a(i)
  e(i) = f(i) + t * g(i)
  h(i) = h(i) + e(i)
END DO
```

The directive is marked with the characters `!dir$`. If the source code is written using the fixed source form, these characters must be at the beginning of the line.

Due to the directive `split`, the compiler will split the above loop in two as follows:

```
DO i = 1, 1000
  a(i) = b(i) * c(i)
  ta(i) = d(i) + a(i)
END DO
DO i = 1, 1000
  e(i) = f(i) * ta(i) * g(i)
  h(i) = h(i) + e(i)
END DO
```

This may make the code faster by reducing memory bandwidth. Instead of loading and storing a lot of data in an iteration of the loop, the split loop gives a better balance between computation and memory operations. This also improves the performance if we are using the streams mechanism.

Here is an example of *loop unrolling*:

```
!dir$ unroll 2
DO i = 1, 10
  DO j = 1, 100
    a(j,i) = b(j,i) + 1
  END DO
END DO
```

This results in the following unrolled loops (into two copies of the inner loop):

```
DO i = 1, 10, 2
  DO j = 1, 100
    a(j,i) = b(j,i) + 1
  END DO
  DO j = 1, 100
    a(j,i+1) = b(j,i+1) + 1
  END DO
END DO
```

The compiler may also fuse the two inner loops together to produce the following final code:

```
DO i = 1, 10, 2
  DO j = 1, 100
```

```
      a(j,i)   = b(j,i) + 1
      a(j,i+1) = b(j,i+1) + 1
    END DO
  END DO
```

Here we used the inverse operation of loop splitting to decrease the overhead due to loop control.

*Bottom loading* is an effective technique for overlapping loop control and loading of operands for the next iteration of the loop. Here is an example:

```
DO i = 1, 100
  a(i) = a(i) + b(i)*c(i)
END DO
```

After each iteration, one has to check whether to do further iterations, or to continue from the next statement after the loop. When the statement

```
  a(i) = a(i) + b(i)*c(i)
```

is executed, one has to issue load operations for the `a(i)`, `b(i)` and `c(i)` values, which can take some time. Therefore, one could start the load operations for the next iteration *first*, and only after this check if we should do another iteration.

Bottom loading can cause a program error if we try to load, for example, the value `c(101)`, which could be outside the memory allocated to the program. In practice this never occurs, except in cases where the loop has a large increment:

```
DO i = 0, 10000, 1000
   a(i) = a(i) + b(i)*c(i)
END DO
```

Here we could load the value `c(11000)`, which could be outside the memory bounds.

The `cache_align` directive can be used to align arrays or `COMMON` blocks on cache line boundaries:

```
REAL, DIMENSION(50) :: a, b
REAL, DIMENSION(10) :: c
COMMON /my_block/ a, b
!dir$ cache_align /my_block/, c
```

Here both the contents of the `COMMON` block `my_block` and the array `c` were aligned on the cache line boundary. Therefore the array elements `a(1)` and `c(1)` map to the first word of a cache line.

The directives `bounds` and `nobounds` tell the compiler to check specific array references for out-of-bounds errors:

```
!dir$ bounds [array_name [, array_name]...]
!dir$ nobounds [array_name [, array_name]...]
```

If the array names are not supplied, the directive applies to all arrays.

The `symmetric` directive is useful when using the SHMEM communications library.

```
!dir$ symmetric [var [, var]...]
```

This directive declares that a PE-private stack variable has the same local address on all PEs. For more information on the SHMEM library routines, issue the command `man intro_shmem`. See also Section 7.4 on page 70.

The directives

```
!dir$ free
!dir$ fixed
```

allow you to select the form of the source code within a file. This possibility is an extension to the Fortran 90 standard.

## 5.8 Fortran 90 modules

One of the strongest features of the Fortran 90 programming language are *modules*, which can be used to encapsulate data and procedures. In this way, one can define *abstract data types* which hide implementation details, and only the interface is public.

A module must be compiled before it can be used in a program unit. In T3E the module definitions are placed in an object file with the suffix `.o`. During the compilation of the main program the compiler looks for module definitions in all the `.o` files and `.a` archives in the present or specified directories.

As an example, suppose that a program consists of three modules called `myprec` (file `myprec.f90`), `matrix` (file `matrix.f90`), and `cg` (file `cg.f90`) together with the main program (`iterate.f90`). Assume that `matrix` uses `myprec`, `cg` uses `myprec` and `matrix`, and the main program uses all three modules. Then we can compile the program as follows:

```
t3e% f90 -c myprec.f90
t3e% f90 -c matrix.f90
t3e% f90 -c cg.f90
t3e% f90 -o iterate iterate.f90 myprec.o matrix.o cg.o
```

The resulting executable program is called `iterate`. Note that the call hierarchy of the modules is reflected in the order in which they are compiled.

Using the `make` system is a convenient way to handle the compilation and linking. In the `makefile` one must specify the dependencies between the modules and other program units (see Section 9.1). The following makefile handles the above example.

```
OBJS= iterate.o myprec.o matrix.o cg.o
OPTS= -c
```

```
F90= f90

iterate: $(OBJS)
        $(F90) -o $@ $(OBJS)

iterate.o: myprec.o cg.o matrix.o

cg.o: myprec.o matrix.o

matrix.o: myprec.o

.SUFFIXES: .f90

.f90.o:
        $(F90) $(OPTS) $<

clean:
        rm -f *.o iterate
```

If the module files (`.o` or `.a`) are not in the current directory, one can use the `-p` *path* option of the `f90` command to include additional search paths and/or module files.

It is common to place the modules in an archive so that they can be used in several programs. As an example we compile the previous modules as before and form a library called `libmod.a`:

```
t3e% ar rv libmod.a myprec.o matrix.o cg.o
t3e% rm myprec.o matrix.o cg.o
```

Suppose that `libmod.a` is in the subdirectory `lib`. Then we can compile and link the main program with

```
t3e% f90 -o iterate -p lib iterate.f90
```

The compiler option `-p` may take as an argument a directory name, when all archive files in it are search, or a single file name, e.g., `-p lib/libmod.a`.

## 5.9 Source code preprocessing

Source code preprocessing is activated if the filename extension is `.F` or `.F90`. Preprocessing directives (like `#ifdef...#else...#endif`) can help to isolate computer system specific features. This helps in maintaining a single version of source code in one source file.

On Cray systems it is often necessary to use the option `-F` to make certain macro expansions work (`#define`). The option `-D` can be used to define macros directly from the compiler command line. One can also use the compiler options `-eP` (only preprocessing) or `-eZ` (preprocessing and compilation) to preprocess source codes.

As an example, consider a simple code that computes and prints a root of a polynomial using IMSL routines ZREAL and WRRRN. The code is written so that it can be run on both T3E and Caper (DEC AlphaServer at CSC), on which the preprocessor replaces the single precision IMSL calls with the corresponding double precision versions by defining macros. Moreover, the variable `info` is printed if `INFO` is defined.

```
#ifdef __alpha
#define zreal dzreal
#define wrrrn dwrrrn
#endif

PROGRAM root
  IMPLICIT NONE
  INTEGER, PARAMETER:: prec=SELECTED_REAL_KIND(12,100)
  REAL (prec):: eabs=1.0e-5, erel=1.0e-5, eps=1.0e-5, &
                eta=1.0e-2, xi=1.0, x
  INTEGER:: nr=1, imax=100, info, one=1, zero=0
  CHARACTER(9):: title='A root is'
  REAL (prec), EXTERNAL:: fun

  CALL zreal(fun,eabs,erel,eps,eta,nr,imax,xi,x,info)
  CALL wrrrn(title,one,nr,x,one,zero)

#ifdef INFO
PRINT *, info
#endif

END PROGRAM root

FUNCTION fun(x) RESULT(value)
  IMPLICIT NONE
  INTEGER, PARAMETER:: prec=SELECTED_REAL_KIND(12,100)
  REAL (prec):: x, value

  value=x**2-2

END FUNCTION fun
```

To compile and link on T3E use

```
t3e% f90 -eZ -F root.f90 -p$MODULES_F90 $LINK_F90 -DINFO
```

Here the preprocessor was invoked with `-eZ`, `-F` enables macro expansion, and `-DINFO` defines `INFO`. On Caper the preprocessor is activated with `-cpp`:

```
caper% f90 $FFLAGS -cpp root.f90 $LINK_FNL -DINFO
```

In both cases the output is

```
 A root is
   1.414
 4
```

and four iterations were performed.

## 5.10 More information

CSC has published a textbook on Fortran 90 [HRR96]. A general introduction to the Unix programming environment is given in the Metacomputer Guide [Lou97]. Both books are written in Finnish.

Code optimization is discussed in the Cray manual *Cray T3E Fortran Optimization Guide* [Crac]. Compiler directives are explained in the manual *CF90 Commands and Directives Reference Manual* [Craa]. The WWW address

```
http://www.csc.fi:8080
```

contains on-line versions of the Cray manuals.

Compiling Fortran 90 modules at CSC is discussed in the @CSC magazine 4/97, and using preprocessing to write portable code is considered in @CSC 2/97.

# Chapter 6

# C and C++ programming

This chapter discusses C and C++ programming on the Cray T3E. Parallel programming is described in Chapter 7 and programming tools are discussed in Chapter 9.

## 6.1 The Cray C/C++ compilers

The Cray C++ Programming Environment contains both the Cray Standard C and the Cray C++ compilers. The Cray Standard C compiler conforms with the ISO and ANSI standards. The Cray C++ compiler conforms with the ISO/ANSI Draft Proposed International Standard.

Because both the Cray Standard C and Cray C++ compilers are contained within the same programming environment, programmers writing code in C should use the `cc` or `c89` commands to compile their source files. The command `c89` is a subset of `cc` and conforms to the POSIX standard. Programmers writing code in C++ should use the `CC` command.

Note that the C/C++ compiler is unable to parallelize automatically your code. Instead, you must use explicit methods like message-passing (MPI or PVM) libraries, or the data-passing library (SHMEM).

The following commands are included in the C/C++ programming environment on the T3E:

| *Command* | *Description* |
|---|---|
| `cc` | Cray Standard C compiler |
| `c89` | Cray Standard C compiler |
| `CC` | Cray C++ compiler |
| `cpp` | Preprocessor of the C compiler |

The compilation process, if successful, creates an absolute object file, named `a.out` by default. This binary file, `a.out`, can then be executed.

For example, the following sequence compiles the source file `myprog.c` and executes the resulting malleable program `a.out` with eight processors:

```
t3e% cc myprog.c
t3e% mpprun -n 8 ./a.out
```

Compilation can be terminated with the appropriate options to produce one of several intermediate translations, including relocatable object files (option `-c`), assembly source expansions (option `-S`), or the output of the preprocessor phase of the compiler (option `-P` or `-E`).

In general, the intermediate files can be saved and later resubmitted to the `CC`, `cc`, or `c89` commands, with other files or libraries included as necessary. By default, the `CC`, `cc`, and `c89` commands automatically call the loader, `cld`, which creates an executable file.

The program can also be compiled and linked into a non-malleable executable by:

```
t3e% cc -X 8 -o myprog.c
```

## 6.2 The C compiler

The Cray Standard C compiler consists of a preprocessor, a language parser, an optimizer and a code generator. The Cray Standard C compiler is invoked by commands `cc` or `c89`.

The `cc` command accepts C source files that have the `.c` and `.i` suffixes, object files with the `.o` suffix, library files with the `.a` suffix and assembler source files with the `.s` suffix. The `cc` command format is generally as follows:

```
t3e% cc [compiler_options] files
```

The sizes of the C datatypes on the T3E are:

- `float`: 4 bytes
- `double`: 8 bytes
- `long double`: 8 bytes
- `int`: 8 bytes
- `long`: 8 bytes
- `long long`: 8 bytes

## 6.3 Calling Fortran from C

Sometimes you need to call Fortran routines from C programs. In the following, we calculate a matrix product using the routine SGEMM from the Libsci library:

```
#include <stdio.h>
#include <fortran.h>

#define DGEMM SGEMM

#define l 450
#define m 500
#define n 550

main()
{
  double a[n][l], b[l][m], ct[m][n];
  int ll, mm, nn, i, j, k;
  double alpha = 1.0;
  double beta = 0.0;
  void DGEMM();
  char *transposed = "t";
  _fcd ftran;

  /* Initialize */

  for (i = 0; i < n; i++)
    for (j = 0; j < l; j++)
      a[i][j] = i-j+2;
  for (i = 0; i < l; i++)
    for (j = 0; j < m; j++)
      b[i][j] = 1/(double)(i+2*j+2);

  ftran = _cptofcd(transposed, strlen(transposed));

  ll = l; mm = m; nn = n;
  DGEMM(ftran, ftran, &nn, &mm, &ll, &alpha, a, &ll,
        b, &mm, &beta, ct, &nn);

  printf("%.6f\n", ct[10][10]);

  exit(0);
}
```

Note that on the T3E, the SGEMM routine performs the calculation using 64-bit real numbers, corresponding to the `double` type in C. Before making the Libsci call, we need to convert the C strings into Fortran strings. This is done with the function `_cptofcd`. We also use the type `_fcd` defined in the header file `fortran.h`.

The fact that Fortran stores arrays in reverse order compared to C needs to be taken into account. Therefore, the array `ct` contains the transpose of the result of the matrix multiplication.

This program takes about one second to execute on a 375 MHz processor, which corresponds to the execution speed of about 240 Mflop/s.

## 6.4 C compiler options

The most typical compiler options are given in the Table 6.1. Many of the options have corresponding compiler directives, which can be included in the source code.

| *Compiler option* | *Meaning* |
|---|---|
| `-c` | Compile only, do not attempt to link |
| `-O`*n* | Choose optimization level *n* (0,1,2,3) |
| `-h`*option* | Enable specific compiler actions |
| `-haggress` | Aggressive optimization |
| `-hunroll` | Enable loop unrolling |
| `-hscalar`*n* | Choose scalar optimization level *n* (0,1,2,3) |
| `-hstdc` | Strict conformance of the ISO C standard |
| `-hsplit` | Split loops into smaller ones |
| `-happrentice` | Compile for MPP Apprentice |
| `-lapp` | Link with MPP Apprentice library |
| `-g` | Compile for the Cray TotalView debugger |
| `-Gf` | Debugging with full optimization |
| `-Gp` | Debugging with partial optimization |
| `-Gn` | Debugging with no optimization (same as `-g`) |
| `-X`*npes* | Compile for *npes* processors |
| `-D`*macro*`[=`*def*`]` | Define a `cpp` macro |
| `-U`*macro* | Undefine a `cpp` macro |
| `-V` | Display the version number of the compiler |
| `-W`*phase*`[,"`*options*`"]` | Pass *options* to *phase* |
| `-I`*incpath* | Search include files also from *incpath* |
| `-L`*libpath* | Search libraries also from *libpath* |
| `-l`*name* | Link also with library `lib`*name*`.a` |

Table 6.1: Typical compiler options.

## 6.5 C compiler directives (#pragma)

The #pragma directives are used within the source program to request certain kinds of special processing. The #pragma directives are extensions to the C and C++ standards. They are classified according to the following types:

- general
- template instantiation (Cray C++ only)
- scalar
- tasking
- inlining.

You can control the compiler analysis of your source code by using #pragma directives. The #pragma directives have the following form:

```
#pragma [_CRI] identifier [arguments]
```

In the specification, the macro expansion is applied only to arguments. The _CRI specification is optional and ensures that the compiler will issue a message concerning any directives that it does not recognize. Diagnostics are not generated for any directives that do not contain the _CRI specification.

To ensure that your directives are seen only by Cray Research compilers, you should use the following coding technique, where *identifier* represents the name of the directive:

```
#if _CRAYC
  #pragma _CRI identifier
#endif
```

The following sections describe the most useful #pragma directives in the Cray T3E environment. They are all classified as scalar directives and used for code optimization.

### cache_align

The cache_align directive aligns a variable on a cache-line boundary. This is useful for frequently referenced variables.

The first-level cache (DCACHE) line consists of four 64-bit words which are loaded from the memory to the cache whenever any of the words is referenced. By using the directive you can be sure that a specified memory location is loaded to the first word of a cache-line.

The effect of the cache_align directive is independent of its position in the source code. It can appear in global or local scope. The format of the directive is as follows:

```
#pragma _CRI cache_align var_list
```

In the previous format, *var_list* represents a list of variable names separated by commas. In C, the `cache_align` directive can appear before or after the declaration of the named objects. In C++, it must appear after the declaration of all named objects.

### noreduction

The `noreduction` compiler directive tells the compiler not to optimize the loop that immediately follows the directive as a reduction loop. If the loop is not a reduction loop, the directive is ignored.

You may choose to use this directive when the loop iteration count is small or when the order of evaluation is numerically significant. In the latter case normal optimization may change the result of a reduction loop, because it rearranges the operations.

The format of this directive is:

```
#pragma _CRI noreduction
```

The following example illustrates the use of the `noreduction` compiler directive:

```
sum = 0;
#pragma _CRI noreduction
for (i=0; i<n; i++) {
  sum += a[i];
}
```

Here we know that `n` will be a small number and therefore we do not want to optimize this loop as a reduction loop.

### split

The `split` directive instructs the compiler to attempt to split the following loop into a set of smaller loops. Such a loop splitting improves single processor performance by making the best use of the six stream buffers of the Cray T3E system. The stream buffers reduce memory latency and increase memory bandwidth by prefetching for long, small-strided sequences of memory references.

The `split` directive may avoid performance problems with the stream buffers by splitting an inner loop into a set of smaller loops, each of which allocates no more than six stream buffers.

The `split` directive has the following form:

```
#pragma _CRI split
```

This compiler directive should be placed immediately before the loop to be split. It should immediately precede a `for`, `while`, `do` or `label` statement, but it should not appear in any other context.

The `split` directive merely asserts that the loop can profit by splitting. It will not cause incorrect code.

The compiler splits the loop only if it is safe. Generally, a loop is safe to split under the same conditions that a loop is vectorizable. The compiler only splits inner loops, but it may not split loops with conditional code.

The `split` directive also causes the original loop to be stripmined, and therefore the data is processed in blocks small enough to fit in the cache.

Loop splitting can reduce the execution time of a loop by as much as 40%. Even loops with as few as 40 iterations may be split. The loops must contain more than six different memory references with strides less than 16.

Note that there is a slight risk on increasing the execution time of certain loops. Loop splitting also increases compilation time, especially when loop unrolling is also enabled.

Here is an example of loop splitting:

```
#pragma _CRI split
for (i=0; i<1000; i++) {
  a[i] = b[i] * c[i];
  t = d[i] + a[i];
  e[i] = f[i] + t * g[i];
  h[i] = h[i] + e[i];
}
```

First, the compiler generates the following loop:

```
for (i=0; i<1000; i++) {
  a[i] = b[i] * c[i];
  ta[i] = d[i] + a[i];
}
for (i=0; i<1000; i++) {
  e[i] = f[i] * ta[i] * g[i];
  h[i] = h[i] + e[i];
}
```

Finally, the compiler stripmines the loops to increase the potential for cache hits and reduces the size of arrays created for scalar expansion:

```
for (i1=0; i1<1000; i1+=256) {
  i2 = (i1+256 < 1000) ? i1+256 : 1000;
  for (i=i1; i<i2; i++) {
    a[i] = b[i] * c[i]
    ta[i-i1] = d[i] + a[i]
  }
  for (i=i1; i<i2; i++) {
    e[i] = f[i] * ta[i-i1] * g[i]
    h[i] = h[i] + e[i]
  }
}
```

### symmetric

The `symmetric` directive declares that an `auto` or `register` variable has the same local address on all processing elements (PEs). This is useful for global addressing using the SHMEM library functions. The format for this compiler directive is:

```
#pragma _CRI symmetric var...
```

The `symmetric` directive must appear in local scope. Each variable listed on the directive must:

- be declared in the same scope as the directive
- have `auto` or `register` storage class
- not be a function parameter.

Because all PEs must participate in the allocation of symmetric stack variables, there is an implicit barrier before the first executable statement in a block containing symmetric variables.

### unroll

The `unroll` directive allows the user to control unrolling for individual loops. Loop unrolling can improve program performance by revealing memory optimization opportunities such as read-after-write and read-after-read. The effects of loop unrolling also include:

- an improved loop scheduling by increasing the basic block size
- a reduced loop overhead
- improved chances for cache hits.

The format for this compiler directive is:

```
#pragma _CRI unroll [n]
```

Item *n* specifies the total number of loop body copies to be generated. The value of *n* must be in the range of 2 through 64. If you do not specify a value for *n*, the compiler attempts to determine the number of copies to be generated based on the number of statements in the loop nest.

Warning: If placed prior to a non-innermost loop, the `unroll` directive asserts that the following loop has no dependencies across iterations of that loop. If dependencies exist, incorrect code could be generated.

The `unroll` compiler directive can be used only on loops whose iteration counts can be calculated before entering the loop.

The compiler can be directed to attempt to unroll all loops generated for the program with the command-line option `-hunroll`.

The amount of unrolling specified on the `unroll` directive overrides those chosen by the compiler when the command-line option `-hunroll` is specified.

In the following example, assume that the outer loop of the following nest will be unrolled by two:

```
#pragma _CRI unroll 2
  for (i=0; i<10; i++) {
    for (j=0; j<100; j++) {
      a[i][j] = b[i][j] + 1;
    }
  }
```

With outer loop unrolling, the compiler produces the following nest, in which the two bodies of the inner loop are adjacent to each other:

```
for (i=0; i<10; i+=2) {
  for (j=0; j<100; j++) {
   a[i][j] = b[i][j] + 1;
  }
  for (j=0; j<100; j++) {
   a[i+1][j] = b[i+1][j] + 1;
  }
 }
```

The compiler then fuses the inner two loop bodies, producing the following nest:

```
for (i=0; i<10; i+=2) {
  for (j=0; j<100; j++) {
    a[i][j] = b[i][j] + 1;
    a[i+1][j] = b[i+1][j] + 1;
  }
}
```

Outer loop unrolling is not always legal because the transformation can change the semantics of the original program. For example, unrolling the following loop nest on the outer loop would change the program semantics because of the dependency between array elements `a[i][...]` and `a[i+1][...]`:

```
/* Directive will cause incorrect code due to dependencies */
#pragma _CRI unroll 2
 for (i=0; i<10; i++) {
  for (j=1; j<100; j++) {
   a[i][j] = a[i+1][j-1] + 1;
  }
 }
```

## 6.6 The C++ compiler

The Cray C++ compiler conforms with the ISO/ANSI Draft Proposed International Standard. A revised version of the standard has recently been accepted as the ISO/ANSI standard.

The Cray C++ compiler is invoked by the command `CC`. The compiler consists of a preprocessor, a language parser, a prelinker, an optimizer and a code generator.

The compiler supports templates, run time type identification (RTTI), member templates, partial specialization and namespaces. Moreover, the Silicon Graphics Standard Template Library (STL) is supported.

Cray C++ Tools and Mathpack libraries are installed on T3E. The libraries are Cray versions of Rogue Wave C++ libraries. The Tools library corresponds to Tools.h++ library and the Mathpack is equivalent to the combination of Math.h++ and LAPACK.h++.

The Cray C++ Compiler options and directives are similar with those described in conjunction with the Cray Standard C compiler.

## 6.7 More information

See the Cray publication *Cray C/C++ Reference Manual* [Crab]. The WWW address

```
http://www.csc.fi:8080
```

contains on-line versions of the Cray manuals.

The 4/96 and 5/97 issues of the @CSC magazine contain information about C++ programming in CSC's metacomputer environment, including the T3E.

For the description of the compiler options, use the `man` command, e.g.:

```
man cc
```

# Chapter 7

# Interprocess communication

This chapter describes how to use the MPI or PVM message-passing libraries on the Cray T3E at CSC. In addition, the properties of the Cray data-passing library SHMEM are described in some detail. The data-parallel High Performance Fortran (HPF) programming model is introduced, too.

## 7.1 The communication overhead

Parallelization on the T3E can be done by three different approaches: message"-passing, data-passing and data-parallel programming. The defining feature of message passing is that the data transfer from the local memory of one process to the local memory of another requires operations to be performed by both processes. In contrast, data-passing routines for sending or receiving data are one-sided. In data-parallel model, the programmer specifies only the data distribution between processes, and the compiler generates a parallel executable code.

The fastest way to communicate between processors on the Cray T3E is using the SHMEM library (Shared Memory Library). This is a Cray-specific data-passing library and will not produce portable code to other architectures. It may be preferable to use MPI (Message Passing Interface), a standardized and portable message-passing library. Another portable message-passing library on the T3E is the somewhat older PVM (Parallel Virtual Machine), which is, in general, a bit slower than MPI. The Cray T3E system has also a data-parallel HPF (High Performance Fortran) compiler, supporting Cray's CRAFT data-parallel programming model, as well.

Message latency (or start-up time) is about 1 $\mu$s (microseconds) when using the SHMEM library. With MPI the latency is about 30 $\mu$s. The maximum bandwidth is in practice about 230 MB/s with both SHMEM

and MPI. Latency and bandwidth are not equally transparent to the HPF user, but in general HPF programs are slower than SHMEM and MPI applications.

The total bandwidth of the machine is very large due to six bi-directional communication links in each PE. It does not matter much where the computational nodes of your application are physically situated. The physical start-up time of message passing is about 100 clock periods, which is incremented by about 2 clock periods for each additional link between processors. However, the system allocates "neighboring" processors to your application to minimize the total communication overhead in the computer.

## 7.2 Message Passing Interface (MPI)

MPI (Message Passing Interface) is a standardized message-passing library defined by a wide community of scientific and industrial experts. Portability is the main advantage of establishing a message-passing standard. One of the goals of MPI is to provide a clearly defined set of routines that can be implemented efficiently on many types of platforms.

MPI is also easier and "cleaner" to use than the somewhat older PVM library. In addition, the MPI library on the T3E is usually about 30% faster than the PVM library.

Note that you do not need to use any special linker options to use MPI, because the MPI libraries are linked automatically on the T3E. MPI routines may be called from FORTRAN 77, Fortran 90, C or C++ programs. The version of the MPI standard available on the T3E is MPI-1, not MPI-2.

### 7.2.1 Format of the MPI calls

The format of the MPI calls for Fortran programs (with few exceptions) is as follows:

```
SUBROUTINE sub(...)
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  INTEGER :: return_code
  ...
  CALL MPI_ROUTINE(parameter_list, return_code)
  ...
END SUBROUTINE sub
```

In Fortran 90 programs, it is often convenient to place the definitions in `MODULE mpi` which is taken into use in other modules by the command `USE mpi`.

Correspondingly, for C/C++ programs the format is:

```
#include <mpi.h>

void sub(...)
{
   int return_code;
   ...
   return_code = MPI_Routine(parameter_list);
}
```

### 7.2.2 Some MPI routines

The MPI standard includes more than 120 routines. However, one needs only a few of them for efficient message passing and, at minimum, one can do with six MPI routines. The most important MPI routines are listed in Table 7.1 (the Fortran syntax is shown).

The variable `comm` is often set to the value `MPI_COMM_WORLD` after initialization. For most applications this is the only *communicator*. It binds all processes of a parallel application into a single group. The value of `MPI_COMM_WORLD` is defined in the MPI header file `mpif.h`.

`MPI_BCAST` and `MPI_REDUCE` are examples of collective operations. MPI includes advanced features such as defining application topologies and derived datatypes.

For more information about a particular MPI routine, issue the command

```
man mpi_routine
```

For example, give the command `man mpi_send` to find documentation for the `MPI_SEND` routine. The manual pages show the C language syntax.

### 7.2.3 An example of using MPI

Below is a short MPI example program which uses collective communication to calculate the global sum of task id numbers:

```
PROGRAM example
  IMPLICIT NONE
  INCLUDE 'mpif.h'
  INTEGER :: ntasks, id, rc, data, s
  CALL MPI_INIT(rc)
  IF (rc /= MPI_SUCCESS) THEN
    WRITE(*,*) 'MPI initialization failed'
    STOP
  END IF
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, rc)
```

| *Fortran syntax* | *Meaning* |
|---|---|
| `MPI_INIT(rc)` | Initialize the MPI session.<br>This should be the very first call. |
| `MPI_FINALIZE(rc)` | Terminate the MPI session.<br>This should be the very last call. |
| `MPI_COMM_SIZE(comm, nproc, rc)` | Get the number of processes in `comm`. |
| `MPI_COMM_RANK(comm, myproc, rc)` | Get my task id in `comm`. |
| `MPI_SEND(buf, buflen, datatype, dest, tag, comm, rc)` | Sends data `buf` to process `dest`. |
| `MPI_SSEND(buf, buflen, datatype, dest, tag, comm, rc)` | Sends data `buf` to process `dest` (synchronous send). |
| `MPI_RECV(buf, buflen, datatype, src, tag, comm, status, rc)` | Receives data to `buf` from `src`. |
| `MPI_BCAST(buf, buflen, datatype, root, comm, rc)` | Broadcast data from `root` to other processes in `comm`. |
| `MPI_REDUCE(sbuf, rbuf, buflen, datatype, oper, root, comm, rc)` | Performs global operation (sum, max, ...) from `sbuf` to `rbuf`. |
| `MPI_ISEND(buf, buflen, datatype, dest, tag, comm, request, rc)` | Sends data `buf` to process `dest`, but does not wait for completion (non-blocking send). |
| `MPI_IRECV(buf, buflen, datatype, src, tag, comm, request, rc)` | Receives data to `buf` from `src`, but does not wait for completion (non-blocking receive). |
| `MPI_WAIT(request, status, rc)` | Checks whether a request has been completed. |

Table 7.1: A list of important MPI routines.

```
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, id, rc)
  data = id
  CALL MPI_REDUCE(data, s, 1, MPI_INTEGER, &
      MPI_SUM, 0, MPI_COMM_WORLD, rc)
  CALL MPI_BCAST(s, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, rc)
  WRITE(*,*) 'data:', data, 'sum:', s
  CALL MPI_FINALIZE(rc)
END PROGRAM example
```

If this program is in the file `collect.f90`, it can be compiled and run interactively as follows:

```
t3e% f90 -o collect.x collect.f90
t3e% mpprun -n 8 ./collect.x
 data: 0 sum: 28
 data: 4 sum: 28
 data: 1 sum: 28
 data: 5 sum: 28
 data: 3 sum: 28
 data: 7 sum: 28
 data: 6 sum: 28
 data: 2 sum: 28
t3e% mpprun -n 3 ./collect.x
 data: 0 sum: 3
 data: 1 sum: 3
 data: 2 sum: 3
```

The program was first run on eight processors, and thereafter on three processors. Chapter 8 discusses running batch jobs.

Here is a C language version of the same program:

```
#include <stdio.h>
#include <mpi.h>

main(int argc, char *argv[])
{
  int ntasks, id, rc, data, s;
  rc = MPI_Init(&argc, &argv);
  if (rc != MPI_SUCCESS) {
    printf("MPI initialization failed\n");
    exit(1);
  }
  rc = MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &id);
  data = id;
  rc = MPI_Reduce(&data, &s, 1, MPI_INT, MPI_SUM, 0,
                  MPI_COMM_WORLD);
  rc = MPI_Bcast(&s, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("data: %d sum: %d\n", data, s);
  rc = MPI_Finalize();
  exit(0);
}
```

If the program is in the file `collect.c`, it can be compiled as follows:

```
t3e% cc -o collect.x collect.c
```

The program may be executed as in the Fortran 90 case above.

### 7.2.4 Reducing communication overhead in MPI

On the T3E, it is in some cases faster to use the synchronous send routine `MPI_SSEND` instead of the standard routine `MPI_SEND`. The synchronous routine avoids some overhead in buffering the messages, but may cause load inbalance due to synchronization.

You can also post an `MPI_IRECV` call, which initiates a receive operation, and compute before checking for the arrival of a message. You could also issue an `MPI_IRECV` call before sending your data, which helps in avoiding possible deadlock situations.

If the communication speed of MPI seems to be too slow, for example due to many small messages, you can try to use the SHMEM library (see Section 7.4 on page 70). You can check your code for possible communication bottlenecks with the MPP Apprentice tool (see page 95) or VAMPIR products (see page 100).

### 7.2.5 MPI data types

MPI introduces a datatype argument for all messages sent and received. The predefined MPI datatypes correspond directly to Fortran 77 and C datatypes. On the T3E, the sizes of the most important MPI datatypes are:

- `MPI_INTEGER`: 8 bytes
- `MPI_REAL`: 8 bytes
- `MPI_DOUBLE_PRECISION`: 8 bytes
- `MPI_INT`: 8 bytes
- `MPI_LONG`: 8 bytes
- `MPI_FLOAT`: 4 bytes
- `MPI_DOUBLE`: 8 bytes

### 7.2.6 Further information about MPI

CSC has published an MPI textbook in Finnish [HM97], which discusses MPI in more detail. The book is also available on the WWW pages at

```
http://www.csc.fi/oppaat/mpi/
```

You should also acquire a handbook of MPI [For95, SOHL+96] to be able to use the system efficiently. There are also many tutorials of MPI in English [GLS94, Pac97].

Some examples of MPI programs are available in the WWW system, see the address

```
http://www.csc.fi/programming/examples/mpi/
```

## 7.3 Parallel Virtual Machine (PVM)

PVM (Parallel Virtual Machine) is a message-passing library that is well-suited for heterogeneous computing. It is somewhat older and clumsier to use than MPI.

### 7.3.1 Using PVM on the T3E

You do not need to use any special linker options to use PVM calls in your program. Please note that the *spawn* routines should not be used on the T3E, because you would be running one process in each processing element and the number of processing elements is fixed during the run.

You can use the routine `pvm_get_PE` to find out the id number of each task. An example is given below. This routine is a T3E specific feature and is not included in most other PVM implementations.

### 7.3.2 An example of a PVM program

The following example shows a simple PVM program on the Cray T3E. In the example, the number of processors specified by the command `mpprun`, is passed to the variable `nproc`. The process number 0 sends a distinct message to all nodes, and these print out what they received. Due to buffering, the master node 0 can send a message to itself, too.

```
PROGRAM main
  IMPLICIT NONE
  INCLUDE 'fpvm3.h'
  INTEGER, PARAMETER :: tag = 100, msglen = 1, stride = 1
  INTEGER :: mytid, mype, nproc, j, to, rc, &
      from, message

  CALL PVMFmytid(mytid)
  CALL PVMFgetpe(mytid, mype)
  CALL PVMFgsize(PVMALL, nproc)

  WRITE (*,*) 'PE#',mype,': tid=',mytid,' nproc=',nproc

  IF (mype == 0) THEN
    DO j = 0, nproc-1
      CALL PVMFinitsend(PvmDataRaw, rc)
```

```
        CALL PVMFpack(INTEGER8, j, msglen, stride, rc)
        to = j
        CALL PVMFsend(to, tag, rc)
      END DO
    END IF

    from = 0
    CALL PVMFrecv(from, tag, rc)
    CALL PVMFunpack(INTEGER8, message, msglen, stride, rc)

    WRITE (*,*) 'PE#',mype,': message=',message
  END PROGRAM main
```

Compile, link and run the program as follows (on three processors):

```
t3e% f90 -o pvmprog.x pvmprog.f90
t3e% mpprun -n 3 ./pvmprog.x
 PE# 2 : tid= 393218  nproc= 3
 PE# 0 : tid= 393216  nproc= 3
 PE# 1 : tid= 393217  nproc= 3
 PE# 0 : message= 0
 PE# 2 : message= 2
 PE# 1 : message= 1
```

The same program in C is as follows:

```
#include <stdio.h>
#include <pvm3.h>

main()
{
  int mytid = pvm_mytid();
  int mype  = pvm_get_PE(mytid); /* CRAY MPP specific */
  int nproc = pvm_gsize(NULL);   /* Default group */
  int tag, len, stride;
  int from, message;

  printf("PE#%d: tid=%d nproc=%d\n", mype, mytid, nproc);
  if (mype == 0) {
     int to, j;
     for (j=0; j<nproc; j++) {
        pvm_initsend(PvmDataRaw);
        pvm_pkint(&j,len=1,stride=1);
        pvm_send(to=j,tag=100);
     }
  }
  pvm_recv(from=0, tag=100);
  pvm_upkint(&message, len=1, stride=1);
  printf("PE#%d: message=%d\n", mype, message);
}
```

Compile, link and run the program as follows (3 processors):

```
t3e% cc -o pvmprog.x pvmprog.c
t3e% mpprun -n 3 ./pvmprog.x
```

```
PE#2: tid=393218 nproc=3
PE#0: tid=393216 nproc=3
PE#1: tid=393217 nproc=3
PE#0: message=0
PE#2: message=2
PE#1: message=1
```

### 7.3.3 Further information about PVM

CSC has published a textbook on PVM in Finnish [Saa95].

The Cray implementation of PVM is described in the publication *Message Passing Toolkit: PVM Programmer's Manual* [Crah].

Some examples of PVM programs are available in the WWW system, see the address

```
http://www.csc.fi/programming/examples/pvm/
```

## 7.4 Shared Memory Library (SHMEM)

In addition to the standard message-passing libraries MPI and PVM, there is a package of very fast routines for transferring data between the local memories of the PEs. The collection of these routines is known as the *Shared Memory Library*, usually referred to as the *SHMEM* library. SHMEM routines can be used either as an alternative for MPI or PVM or together with them in the same program.

Examples of the SHMEM routines available on the T3E are listed in Table 7.2. For a complete list and manual summary pages give the command

```
man intro_shmem
```

Because of their low-level nature SHMEM routines have minimal overhead and latency. Thus they offer almost always the fastest way to carry out operations involving one or several remote memories. The routines are *one-sided*, which means that there is no pairing of *send* and *receive* calls as in MPI or PVM, but instead just one call from the PE that reads from or writes into the memory of another PE.

As a drawback, the programmer must pay special attention to data coherence, i.e., that the data actually transferred is also the data that was meant to be transferred in the first place, or that no relevant data is overwritten in the remote memory. Furthermore, the programs using SHMEM routines are not portable to other systems.

To have all the necessary constants and structures at your disposal you must give an include command at the beginning of the program. For the

| *Routine* | *Description* |
|---|---|
| `num_pes` | Returns the total number of PEs. |
| `shmem_add` | Performs an atomic add operation on a remote data object. |
| `shmem_barrier` | A barrier routine for synchronization purposes. |
| `shmem_broadcast` | Sends a local variable to all other PEs. |
| `shmem_collect` | Concatenates data from several PEs to each of them. |
| `shmem_fence` | An auxiliary routine for ordering calls to `shmem_put`. |
| `shmem_get` | Reads from a remote (another PE) memory. |
| `shmem_lock` | An auxiliary routine for protecting a part of the memory from simultaneous update by multiple tasks. |
| `shmem_max` | A collective routine for finding the maximum. value of a symmetric variable between all PEs. |
| `shmem_min` | A collective routine for finding the minimum. value of a symmetric variable between all PEs. |
| `shmem_my_pe` | Returns the identity number of the calling PE. |
| `shmem_prod` | A reduction routine for calculating the product of one or several variables from every PE. |
| `shmem_put` | Writes into a remote (another PE's) memory. |
| `shmem_sum` | A reduction routine for summing up the values of one or several variables from every PE. |
| `shmem_swap` | Changes variables between two PEs. |
| `shmem_wait` | Waits for a variable on the local PE to change. |

Table 7.2: Some SHMEM routines.

C language this is

```
#include  <mpp/shmem.h>
```

and for Fortran 90:

```
INCLUDE 'mpp/shmem.fh'
```

The Fortran compiler in T3E knows automatically where to search for SHMEM constants and thus the `INCLUDE` command is not obligatory for Fortran programs.

Use `man` *shmem_command* for finding instructions on how to use the SHMEM library. A good way to start is by giving the command `man intro_shmem`.

### 7.4.1 Using the SHMEM routines

SHMEM routines can be divided into a few basic categories according to their respective tasks. The *point-to-point communication* routines transfer data between two PEs, whereas *collective* routines involve data transfer between several PEs. *Reduction* routines are used to find out certain properties of data stored in the memories of a group of PEs. *Synchronization* routines give the programmer a possibility to control the order of calls to other SHMEM routines. Finally, there are some *cache management* routines for taking care of data coherence.

### 7.4.2 Data addresses

Data objects are passed by address to SHMEM routines. This means that the address of the remote data object must be known to the PE calling a SHMEM routine. This is typically realized by having a corresponding data object in the local memory. The data objects are then called *symmetric*. The following data objects are symmetric on the T3E:

- Fortran data objects in common blocks or with the `SAVE` attribute
- Fortran arrays allocated with `shpalloc`
- Fortran stack variables declared with a `!dir$ symmetric` directive.
- non-stack C and C++ variables
- C and C++ data allocated by `shmalloc`
- C and C++ stack variables declared with a `#pragma symmetric` directive

There is also another possibility besides having symmetric data objects on different PEs, namely passing the address of a remote data object to the calling PE before the actual call to a SHMEM routine is carried out. In this case the remote data object is called *asymmetric accessible*. The following data objects are asymmetric accessible on the T3E:

- C and C++ data allocated by `malloc` and C++ data allocated by the `new` operator
- C and C++ variables with the `automatic` or `register` storage class
- Fortran arrays allocated with `hpalloc`
- Fortran PE-private data objects on the stack.

### 7.4.3 Point-to-point communication

Point-to-point communication is the most widely occuring form of data transfer in parallel computing in shared-memory computers. There are two basic routines for this purpose in the SHMEM library, `shmem_get` and `shmem_put`. Since SHMEM routines are one-sided, only one of these is needed to transfer data.

#### shmem_get

The C language syntax of a call to `shmem_get` is

```
void shmem_get(void *target, void *source, int len, int pe);
```

For Fortran 90, the syntax is

```
INTEGER :: len, pe
CALL SHMEM_GET(target, source, len, pe)
```

The routine `shmem_get` copies data of length `len` (in 8-byte words) from the memory of PE `pe`. The copying is started at address `source` in this PE, and the data is moved to the address `target` in the calling PE's memory. The calling PE is blocked during the transfer, i.e., it waits until the transfer is finished before moving on to the next command in the program.

#### shmem_put

The C syntax of a call to shmem_put is

```
void shmem_put(void *target, void *source, int len, int pe);
```

For Fortran 90 the syntax is

```
INTEGER :: len, pe
CALL SHMEM_PUT(target, source, len, pe)
```

The routine `shmem_put` copies data of length `len` (in 8-byte words) from the memory of the calling PE starting at address `source` to address `target` in the memory of PE `pe`. The calling PE is *not* blocked during the transfer, but it continues with the program as soon as the transfer is initiated. Therefore, the programmer should consider using some synchronizing routine such as `shmem_wait`.

In terms of efficiency, there is not much difference between `shmem_put` and `shmem_get`. For faster performance one should try to concentrate data transfers into as few and large blocks of data as possible. Having `len` a multiple of 8 is optimal for these routines.

#### Other point-to-point routines

The routines shmem_put and shmem_get operate correctly when the data being transferred consists of items with a size of 8 bytes. If the size of a single data item is only 4 bytes (32 bits), one must call instead the routines shmem_put4 or shmem_get4 in Fortran or shmem_put32, shmem_get32 in C and C++.

Another crucial restriction for the basic versions of shmem_put and shmem_get is that they accept only consecutive data items. To move data with a non-unit stride from a PE to another (or to itself, for that matter), there are extended versions shmem_iget and shmem_iput.

The C language syntax for shmem_iget is

```
void shmem_iget(void *target, void *source, int target_inc,
                int source_inc, int len, int pe)
```

For Fortran 90 this is

```
CALL SHMEM_IGET(target, source, target_inc, source_inc, &
    len, pe)
```

The syntax is similar for the routine shmem_iput and the corresponding 4-byte versions. The integer valued arguments target_inc and source_inc contain the strides between consecutive data items in the target and the source data, respectively.

#### Atomic operations

SHMEM library contains several fast routines for updating or checking the value of a single variable on a remote PE. These routines are called *atomic operations*. These include, e.g., shmem_inc for incrementing the value of a remote data object and shmem_swap for exchanging the values of a single remote and local variable.

### 7.4.4 Reduction routines

A reduction routine computes a single value from a data object distributed over several PEs' memories, e.g., the sum of the elements of a vector. SHMEM contains the following reduction routines: shmem_and, shmem_max, shmem_min, shmem_or, shmem_prod, shmem_sum and shmem_xor. A general call to these routines in C is of the form

```
void shmem_type_op_to_all(type *target, type *source,
    int nreduce, int PE_start, int logPE_stride, int PE_size,
    type *pWrk, long *pSync);
```

where *type* is one of {short, int, float, double, complex, complexd} and *op* is one of {sum, prod, min, max, and, or, xor}. In Fortran, the cor-

responding call would be

```
CALL SHMEM_type_op_TO_ALL(target, source, nreduce, &
    pe_start, logpe_stride, pe_size, pwrk, psync)
```

and here *type* is one of {INT8, INT4, REAL8, REAL4, COMP8, COMP4}, and *op* is one of the operations already mentioned.

The call above applies reduction operation *op* on data of type *type* at address source in the memories of all PEs involved. The result is stored at address target. The argument nreduce tells on how many consecutive data items the reduction operation is to be performed.

Let us suppose that we have two PEs both of which store a vector of 4 integer elements. If we call shmem_int8_sum_to_all with nreduce = 1, the result will be one integer which equals the sum of the first elements of the vectors. If nreduce equals 4, we get an array of 4 integers, and each element in this array is the sum of the corresponding elements in the original vectors. Thus, if the total sum of all elements in both vectors is to be calculated, one must first call a SHMEM routine to form an array of partial sums, and then finish the calculation by summing up the elements in the resulting array with, e.g., a BLAS routine.

The triple pe_start, logpe_stride, pe_size is used to define the so called *active set*, which includes the PEs taking part in the reduction operation. The value of pe_start is simply the number of the first PE in the active set. The value of logpe_stride is the logarithm (in base 2) of the stride between the PEs, and pe_size is the number of PEs in the active set. Thus {pe_start, logpe_stride, pe_size} = $\{0, 1, 5\}$ indicates that the active set consists of the PEs $0, 2, 4, 6, 8$. As another example, {pe_start, logpe_stride, pe_size} = $\{0, 0, n\}$ indicates that the active set is PEs $\{0, 1, \ldots, n-1\}$.

Note: all the PEs in an active set and only these should call a collective routine!

Finally, pwrk and psync are symmetric work arrays. The argument psync is of integer type, and of size shmem_reduce_sync_size (this constant is defined in the file mpp/shmem.h or in the file mpp/shmem.fh. They should be included at the beginning of a code utilizing SHMEM library). The variable psync must be initialized so that the value of each entry is equal to shmem_sync_value. After initialization it is a good idea to call a barrier routine to guarantee synchronization before using psync.

The argument pwrk should be of the same type as the reduction routine and of size max(nreduce/2 + 1, shmem_reduce_min_wrkdata_size).

### 7.4.5 Other important routines

There are two very important routines which a parallel program on T3E will almost certainly call, namely `shmem_my_pe` and `shmem_n_pes`. The former reveals the calling PE its identity, while the latter returns the total number of PEs in use. Their syntaxes are as follows:

```
int shmem_my_pe(void);
int shmem_n_pes(void);
```

For Fortran 90 the syntax is

```
INTEGER :: mype, npes
INTEGER, EXTERNAL :: SHMEM_MY_PE, SHMEM_N_PES
mype = SHMEM_MY_PE()
npes = SHMEM_N_PES()
```

In some cases it may be necessary to stop the execution of a program and wait until all other PEs have performed some critical tasks. For these situations, there is a routine called `shmem_barrier`. However, on the T3E there is a simpler (and faster!) routine for this purpose called simply *barrier*:

```
void barrier()
```

In Fortran 90 the call is

```
CALL BARRIER()
```

The most important difference between `shmem_barrier` and `barrier` is that with `shmem_barrier` it is possible to interrupt temporarily the action of just some of the PEs in use. Because of its speed, we suggest that the `barrier` routine be preferred whenever all PEs are halted. See `man shmem_barrier` for more information.

### 7.4.6 Example of using SHMEM

The following simple code illustrates the use of some of the routines discussed above. Each PE has two four-component vectors of integers, called `source` and `target`, and PE number 0 copies its `source` vector into the location of `target` vector of PE number 1. Finally, `target` vector is used to compute the values of vector `c`.

Note that in the Fortran 90 version the attribute `SAVE` is assigned to both `source` and `target` vectors in order to make them symmetric data objects. The call to `barrier` routine is necessary, because without it, PE number 1 might use the original `target` vector while computing the value of `c`, not the one passed by PE number 0.

Here is the example program in Fortran 90:

```
PROGRAM shmemex
  IMPLICIT NONE
```

```
  INCLUDE 'mpp/shmem.fh'

  INTEGER, PARAMETER :: n = 4
  INTEGER, DIMENSION(n), SAVE :: &
      source_pe = (/1,2,3,4/), &
      target_pe = (/5,6,7,8/)
  INTEGER, DIMENSION(n) :: c
  INTEGER :: i, mype
  INTEGER, EXTERNAL :: shmem_my_pe

  mype = shmem_my_pe()
  IF (mype == 0) THEN
    CALL shmem_put(target_pe, source_pe, n, 1)
  ENDIF
  CALL barrier()
  DO i = 1, n
    c(i) = 2*target_pe(i)
  END DO
  WRITE (*,'(i2,a7,8i3)') mype, ' : c = ', c
END PROGRAM shmemex
```

Here is the same example program in C:

```
#include <stdio.h>
#include <mpp/shmem.h>

main() {
  static long source[4] = {1,2,3,4};
  static long target[4] = {5,6,7,8};
  long c[4];
  int i;

  if(_my_pe() == 0)
    shmem_put(target,source,4,1);
  barrier();
  for(i=0; i<4; ++i)
    c[i] = 2*target[i];
  printf("PE:%d c is: %d %d %d %d \n",
    _my_pe(), c[0], c[1], c[2], c[3]);
}
```

Above, we have used the function `_my_pe` to find out the task id number. The Fortran 90 program can be compiled with the command

```
f90 -o shmemex shmemex.f90
```

and the C program with the command

```
cc -o shmemex shmemex.c
```

After this, the program is started on two PEs by typing

```
mpprun -n 2 ./shmemex
```

The result of the Fortran 90 program will be

```
0 : c =  10 12 14 16
1 : c =   2  4  6  8
```

The output of the C program is similar.

## 7.5 High Performance Fortran (HPF)

The Cray T3E system at CSC has a High Performance Fortran (HPF) compiler. HPF is a developing standard agreed by several computer and software vendors. The Portland Group HPF (PGHPF) version 2.3 on T3E also supports the CRAFT programming model used on Cray T3D systems.

Both HPF and CRAFT are implicit Fortran programming models where the user writes a Fortran 90 program and specifies how arrays are to be distributed among the processors. The compiler then analyzes the data dependencies in the code and generates a parallel executable version of the code. In these *data-parallel* programs individual processes execute the same operations on their respective parts of the distributed data structures, and the parallelism is mainly on the loop level. Since both the HPF and CRAFT directives are actually Fortran 90 comments, the parallel programs can be compiled using the standard Fortran 90 compilers for serial execution.

These implicit programming models provide a much faster and simpler way to parallelize programs than the explicit message passing libraries. On the other hand, in many cases hand-tuned message passing codes can outperform the HPF compilers. Typically the HPF version is at most two times slower. Moreover, the HPF language currently has little support for problems with irregular data structures, but this will change in future revisions of the language. HPF is well suited for prototyping at the development phase, while later critical parts of the code can be parallelized with the message passing libraries or the shared memory library on the T3E.

With the CRAFT programming model, the user can specify also the distribution of work in more detail. It also supports parallel I/O, private data and task parallelism in MIMD style.

On Cray T3D systems the CRAFT programming model was the only implicit programming model available. On T3E systems the CRAFT model is incorporated into the HPF compiler with some syntactical changes. The data distribution directives of the original CRAFT model have been changed to conform with the HPF standard in the following way:

- The directive `!DIR$` has been changed to `!HPF$`.
- The directive `DOSHARED` has been changed to `INDEPENDENT`.

- The directive SHARED has been changed to DISTRIBUTE.
- The distribution specification : for a degenerate distribution has been changed to *.
- The distribution specification :BLOCK has been changed to BLOCK.
- The intrinsic function called IN_DOSHARED is called (in HPF_CRAFT) IN_INDEPENDENT.

The PGHPF 2.3 compiler conforms with HPF standard version 1.1. Fortran 90 internal procedures and recursion are not supported and pointers are supported with certain restrictions.

The following is a simple example of an HPF code:

```
PROGRAM dot
  IMPLICIT NONE
  INTEGER :: n
  REAL, DIMENSION(:), ALLOCATABLE :: a, b
  REAL :: d
  !HPF$ DISTRIBUTE A(BLOCK)
  !HPF$ ALIGN B(:) WITH A(:)

  WRITE (*,*) 'Input the number of points:'
  READ (*,*) n
  ALLOCATE (a(n), b(n))

  a = 1.0
  b = 2.0
  d = SUM(a*b)

  WRITE (*,*) d
END PROGRAM dot
```

The code is standard Fortran 90 code except for the two directive lines starting with !HPF$. The line

```
!HPF$ DISTRIBUTE A(BLOCK)
```

instructs the compiler that the array a should be so distributed along the processors that each processor gets a contiguous block of the array.

The directive

```
!HPF$ ALIGN B(:) WITH A(:)
```

says that the array b should also be distributed and the corresponding elements of b and a should reside in the same processor.

For the line d = SUM(a*b) the compiler generates code that computes the local partial sum in each processor and then gathers the partial sums into a global sum.

The PGHPF compiler must be initialized with the command

```
t3e% module load pghpf
```

The compiler is invoked with the command `pghpf`. It accepts files ending with `.hpf`, `.f`, `.F`,`.for` or `.f90`. Files ending with `.F` are processed using the C preprocessor. Suppose that the previous program is in the file `dot.f90`. The program can be compiled and run with the following commands:

```
t3e% pghpf -Minfo -Mstats -Mautopar dot.f90 -o dot
t3e% mpprun -n 4 dot -pghpf -stat alls
```

The compiler option `-Minfo` produces messages about parallelization of do loops and arrays that are not distributed. The option `-Mstats` enables collection of performance data during the run and `-Mautopar` parallelizes the parallelizable `DO` loops without the directive `independent`. The options `-pghpf -stat alls` print statistics about timing, memory use and message passing after the run. These options are not necessary, but they provide useful information about the program.

An introduction to HPF with a plenty of examples is presented in *The High Performance Fortran Handbook* [KLS+94]. In 3/97 and 5/97 issues of the @CSC magazine there are articles considering HPF on the T3E. The manual pages of the compiler are available with the command `man pghpf` and the complete documentation for PGHPF 2.3 can be accessed at `http://www.csc.fi/programming/doc/pghpf/2.3`. More information on the HPF standard can be obtained at the WWW address `http://www.mhpcc.edu/doc/hpf/hpf.html`.

# Chapter 8

# Batch queuing system

The batch queuing system ensures an optimum load on the computer and a fair distribution of resources for the users. On the Cray T3E the queuing system is called *Network Queuing Environment*, *NQE*.

## 8.1 Network Queuing Environment (NQE)

The NQE system makes it possible to submit batch jobs from different computers (*client computers*) to the target computer (*execution server*). NQE takes the submitted job script and reserves the requested resources. Thereafter the commands in the script file are executed and the results are returned to the user. The configuration of the NQE on the T3E ensures that the distribution of jobs is as fair as possible.

## 8.2 Submitting jobs

The sequence of commands or programs that you want to run on a computer is called a job. In the NQE system the jobs are sent to a *queue* which handles the job and executes it with appropriate resources. In the NQE system the jobs are sent to a *pipe queue* which redirects the job to a *batch queue*.

To submit jobs on the T3E, you must first create a job script file in normal shell script format. An example is given below:

```
#!/bin/ksh
# QSUB -q prime
# QSUB -l mpp_p=16
# QSUB -l mpp_t=7000
```

```
# QSUB -l p_mpp_t=7000
# QSUB

cd $HOME/sn6309
mpprun -n $NPES ./mmloop 6000 6000
```

First, the given command shell (here `/bin/ksh`) is used to run the script. The default shell is `/bin/sh`. These two shells are recommended.

The option `-q` tells the NQE system which queue the job should be sent to. Here we have requested the `prime` queue. At the moment this is the only queue a normal user on the T3E can submit jobs to. This queue redirects the request to the batch queue `small` (maximum limits:64-PE and 2 hours), `medium` (64-PE, 12 hours) or `large` (128-PE, 12 hours). The option `-l` specifies certain limits for the batch job: `mpp_p` is the maximum number of processors, `mpp_t` is the maximum execution time for all the programs in the script, and `p_mpp_t` is the maximum process time for any program in the script. It is mandatory to specify `mpp_p` and `mpp_t` either in the T3E job script or as `qsub` command line options.

These options are covered in more detail in Table 8.1.

| *qsub option* | *Meaning* |
|---|---|
| `-q` *queue* | The job is run in the given batch queue. |
| `-r` *name* | The name of the batch job. |
| `-lT` *time* | Maximum single processor time for the job to be run. Should be at least same as `mpp_t`. |
| `-l mpp_p=`*xx* | The number of PEs requested for the job. This number is available later in the script in the environment variable `$NPES`. |
| `-l mpp_t=`*yyy* | Maximum execution time of all the programs in the script. |
| `-l p_mpp_t=`*yyy* | Maximum processing time of any single program in the script. |
| `-e` *file* | Name of the error file. Default name of the error file is *name*`.e`*request-id*. |
| `-o` *file* | Name of the output file. Output to terminal is redirected to this file. Default output file name is *name*`.o`*request-id*. |
| `-eo` | Combines error messages to the output file. |
| `-s` *shell* | Selects the command shell. |

Table 8.1: Some `qsub` options.

After the options you have to specify the commands to be executed. In the example the first command (`cd`) changes the current directory to `$HOME/sn6309`. The parallel program `mmloop` is run with the `mpprun` command. Here the environment variable `$NPES` indicates the number of PEs allocated for the batch job (16 in the case above).

The batch job is submitted with the command

```
qsub [options] jobfile
```

The output from the command looks like this:

```
nqs-181 qsub: INFO
  Request <2227.sn6309>: Submitted to queue <prime> by <jbond(007)>.
```

The identifier `2227.sn6309` is the *request-id* of the job. This can be used to check the status of the job with the `qstat` command.

The most often encountered error is the following:

```
nqs-4517 qsub: CAUTION
  No such queue <pirme> at local host.
```

which usually means that the name of the queue is not written correctly (here `pirme` instead of `prime`).

The most important and frequent options to the `qsub` command are shown in Table 8.1. Options can be given in a script file as in the example above or in a command line. The options in the script file are written on pseudocomment lines starting with the characters `#QSUB`.

Options can be mixed so that some of them are given in the script file and some in the command line. If the same options are given, the options in the command line override the options in the script file.

## 8.3 Status of the NQE job

When you have submitted a job to be executed, you usually want to know how the job is running or if it is running at all. There are several different commands for checking the status of jobs. The easiest to use is the `cqstat` command, which starts a graphical tool to check out the status of jobs. Figure 8.1 shows an example of a `cqstat` session.

You have access only to those jobs that are owned by you. With the `cqstat` command you can also delete running jobs. This is done by first selecting the job to be deleted and after that selecting the *Delete* command from the *Action* menu. The `cqstat` command needs an X Window System to work.

Another command to get the status of batch jobs is `qstat`. This is a text based tool and it does not require the X Window System. The `qstat` command has special options for MPP machines only. Table 8.2 lists the most used options of the `qstat` command. These options accept either a queue name or a request id. If both are missing, the command shows requested data from all queues. Listing 8.1 shows an example of the output of command `qstat -a`. Listing 8.2 shows an example of the output of the command `qstat -f` *request-id*.

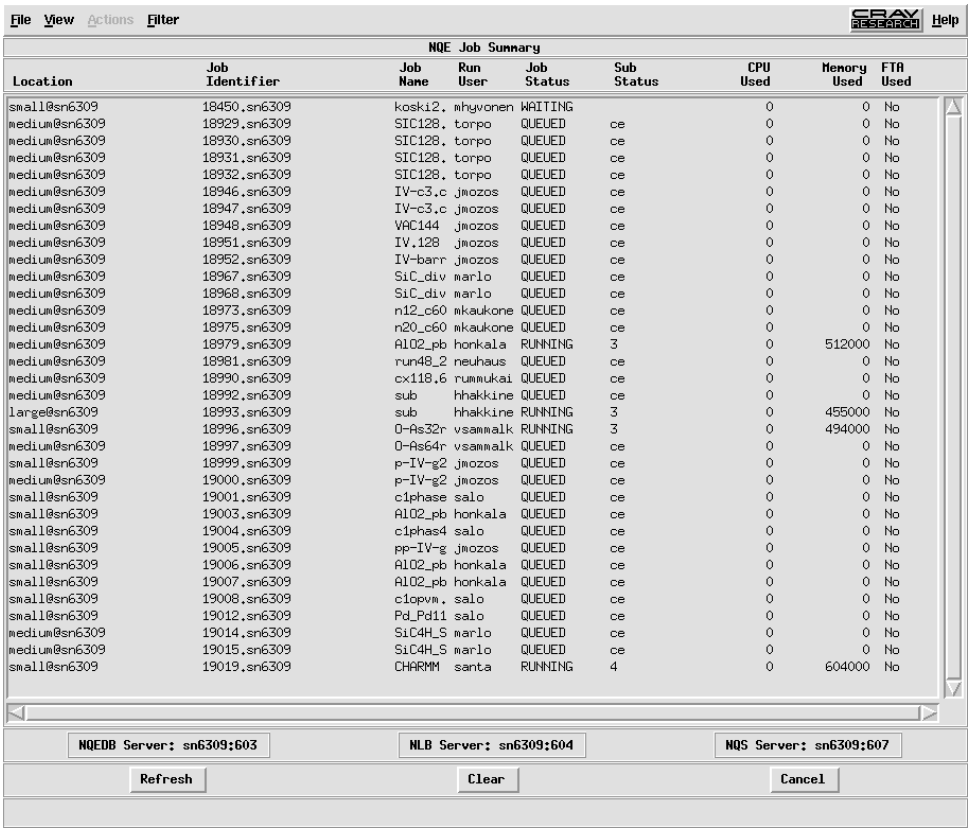



Figure 8.1: An example of a `cqstat` session.

```
---------------------------------
NQS 3.3.0.4 BATCH REQUEST SUMMARY
---------------------------------
IDENTIFIER    NAME    USER     LOCATION/QUEUE         JID  PRTY REQMEM REQTIM ST
------------- ------- -------- ---------------------- ---- ---- ------ ------ ---
18996.sn6309  O-As32r vsammalk small@sn6309          38633   20    494   7200 R03
19021.sn6309  CHARMM  santa    small@sn6309          38730   20    604   1000 R04
18999.sn6309  p-IV-g2 jmozos   small@sn6309                999 262144   7200 Qce
18979.sn6309  AlO2_pb honkala  medium@sn6309         37966   20    512  43200 R03
18929.sn6309  SIC128. torpo    medium@sn6309               999 262144  43200 Qce
18930.sn6309  SIC128. torpo    medium@sn6309               999 262144  43200 Qce
18993.sn6309  sub     hhakkine large@sn6309          37763   20    455  43200 R03
```

Listing 8.1: An example of a `qstat -a` command.

| `qstat` *option* | *Meaning* |
|---|---|
| `-a` | Display summary information for all jobs. |
| `-b` | Display summary information for batch jobs. |
| `-r` | Display summary information for running jobs. |
| `-m` | Display information about MPP queue limits. |
| `-u` *user* | Display information about *user*'s jobs. |
| `-f` | Display full information about queues or requests. |

Table 8.2: Some `qstat` options.

```
----------------------------------------
NQS 3.3.0.4 BATCH REQUEST: espy.sn6309        Status:         RUNNING
----------------------------------------                 3 Processes
                                                                Active
        NQE Task ID:    --
        NQS Identifier: 2230.sn6309             Target User:    jbond
                                                Group:          csc
        Account/Project: <csc007>
        Priority:        ---
        URM Priority Increment: 1
        Job Identifier: 52                      Nice Value:     25

        Created:        Thu Apr 24 1997         Queued:         Thu Apr 24 1997

<LOCATION/QUEUE>
        Name:           small@sn6309            Priority:       30
<RESOURCES>
                                PROCESS LIMIT   REQUEST LIMIT   REQUEST USED
        CPU Time Limit            <7200sec>       <7200sec>            0sec
        Memory Size                  <256mw>         <256mw>          614kw
        Permanent File Space     <unlimited>           <0>              0kw
        Quick File Space              <0>              <0>              0kw
        Type a Tape Drives                             <0>              0
        Type b Tape Drives                             <0>              0
        ...
        Type h Tape Drives                             <0>              0
        Nice Increment                <0>
        Temporary File Space          <0>              <0>
        Core File Size           <unlimited>
        Data Size                <unlimited>
        Stack Size               <unlimited>
        Working Set Limit        <unlimited>
        MPP Processor Elements                          8               8
        MPP Time Limit               7100sec         7100sec          0sec
        Shared Memory Limit                            <0>              0kw
        Shared Memory Segments                         <0>              0
<FILES>                 MODE            NAME
        Stdout:         spool           sn6309:/mnt/mds/.../espy.o2230
        Stderr:         spool           sn6309:/mnt/mds/.../espy.e2230
        Job log:        spool           sn6309:/mnt/mds/.../espy.l2230
        Restart:                        <UNAVAILABLE>
<MAIL>
        Address:                jbond@sn6309         When:
<PERIODIC CHECKPOINT>
        System:         off                     Request:        System Default
        Cpu time:       on     60 Min           Cpu time:       def <Default>
        Wall clock:     off   180 Min           Wall clock:     def <Default>
        Last checkpoint:None
<SECURITY>
        Submission level:               N/A
        Submission compartments:        N/A
        Execution level:                N/A
        Execution compartments:         N/A
<MISC>
        Rerunnable      yes                     User Mask:      007
        Restartable     yes                     Exported Vars:  basic
        Shell:          DEFAULT
        Orig. Owner:    007@sn6309
```

Listing 8.2: An example of a `qstat -f` command.

## 8.4 Deleting an NQE batch job

Sometimes it is necessary to delete a job before it is finished. For example, the input may be erroneous and you do not want to waste any CPU time. A job is deleted with the command `qdel`. The most usual way to use the `qdel` command is

`qdel` *request-id*

You can ensure the deletion of a job by sending a `SIGKILL` signal to the running job. Use the option `-k` of the `qdel` command:

`qdel -k` *request-id*

## 8.5 Queues

The pipe queue available on the T3E is called `prime`. This *pipe queue* redirects the jobs to the appropriate batch queues.

To see the current batch queues and their limits use the command `qstat -m`, which displays the names of batch queues and the time limits. Here is an extract from the output:

```
---------------------------------
NQS 3.3.0.4 BATCH QUEUE MPP LIMITS
---------------------------------
QUEUE NAME                RUN     QUEUE-PE'S    R-PE'S  R-TIME  P-TIME
                        LIM/CNT     LIM/CNT     LIMIT   LIMIT   LIMIT
----------------------- --- ---  ------ ------  ------  ------  ------
csc                       2/0       224/0        224   14400   14400
fmi                       4/0       256/0        128    7200    7200
fmi192                    4/0       192/0        192    7200    7200
small                    10/2       128/36        64    7200    7200
medium                   10/1       160/32        64   43200   43200
large                     1/1        **/128       128   43200   43200
----------------------- --- ---  ------ ------  ------  ------  ------
sn6309                   10/4        **/196
----------------------- --- ---  ------ ------  ------  ------  ------
```

To see which pipe queues are redirected to which batch queues, use the command `qstat`. Here is an extract of the command:

```
------------------------------
NQS 3.3.0.4 PIPE QUEUE SUMMARY
------------------------------
QUEUE NAME              LIM TOT ENA STS QUE ROU  WAI HLD ARR DESTINATIONS
----------------------- --- --- --- --- --- ---  --- --- --- ------------
nqebatch                  1   0 yes  on   0   0    0   0   0 input
csc_pipe                  1   0 yes  on   0   0    0   0   0 csc
prime                     1   0 yes  on   0   0    0   0   0 input
input                     1   0 yes  on   0   0    0   0   0 small
                                                             medium
                                                             large
----------------------- --- --- --- --- --- ---  --- --- --- ------------
```

```
sn6309                    5   0           0   0    0   0   0
----------------------- --- --- --- --- --- ---  --- --- --- ------------
```

## 8.6 More information

More information on NQE is available in the CSC help system:

```
help nqe
```

See also the manual pages of the commands `qsub`, `qstat` and `qdel`. Another good reference to check out is *CSC's T3E Users' Information Channel* in the WWW address:

```
http://www.csc.fi/oppaat/t3e/t3e-users/archive/
```

# Chapter 9

# Programming tools

## 9.1 The make system

The make utility executes commands in a makefile to update one or more targets, which typically are programs. The make system is mainly used to maintain programs consisting of several source files. When some of the source files are modified, the make system recompiles only the modified files (and those files that depend on the modified files).

Here is a typical makefile:

```
OBJECTS= func1.o func2.o
OPTS= -O
LIBS= -lnag

all: myprog

myprog: $(OBJECTS)
        $(CC) $(LDFLAGS) $(OBJECTS) $(LIBS) -o $@

.c.o:
        $(CC) -c $(OPTS) $<

clean:
        rm -f $(OBJECTS)
        rm -f core
        rm -f myprog
```

Each indented line of the makefile should start with a Tab character. There should also be an empty line at the end.

The Unix command

```
make
```

compiles the source codes `func1.c` and `func2.c`, and links them with the NAG library, producing an executable file `myprog`.

The line `.c.:` in the example specifies that `.c` files should be compiled into `.o` files using the command on the following tabulated line:

```
$(CC) -c $(OPTS) $<
```

The symbol `$(CC)` is already defined by the `make` system, but you could have redefined it to the appropriate compiler in the beginning of the makefile. The symbol `$(OPTS)` was defined as follows:

```
OPTS= -O
```

Therefore, this symbol is replaced by the string `-O`, which means code optimization. The symbol `$<` refers to the actual `.c` file. Thus, if we need to produce the file `func1.o`, the symbol `$<` will be replaced by the filename `func1.c`.

The dependencies and compiler options for the executable program `myprog` are introduced by the lines

```
myprog: $(OBJECTS)
        $(CC) $(LDFLAGS) $(OBJECTS) $(LIBS) -o $@
```

Here we specify that the program `myprog` depends on the files defined by the symbol `$(OBJECTS)` (e.g., the files `func1.o` and `func2.o`). The compilation command used the default linker options, which are given by the symbol `$(LDFLAGS)`. The symbol `$(LIBS)` is replaced by the string `-lnag`, which means that the NAG library will be linked with our program. The symbol `$@` refers to the name of the target file, here the name is `myprog`.

The command

```
make clean
```

can be used to clean up the directory, that is, to remove all the executable and object files and a possible core file.

You find more information on the make system with the command

```
man make
```

## 9.2 Program Browser

The Program Browser, Xbrowse, provides an interactive environment in which to view and edit Cray Fortran 90, FORTRAN 77 as its subset and Cray Standard C codes. Cross-reference information is given about aspects of the code being browsed and is updated when the code is

changed. The browser may act upon a routine, a file, or an entire program, which is composed of one or more distinct files, but treated by the browser as a single unit.

Xbrowse also acts as a base for other Cray Research tools that reference source code. To display a list of available tools, use the left mouse button to click on the *Tools* menu button.

Suppose you want to obtain information about all of your C code. You can start Xbrowse by entering the following command:

```
xbrowse *.c &
```

This command causes Xbrowse to run in the background. It displays the Xbrowse main window on your screen.

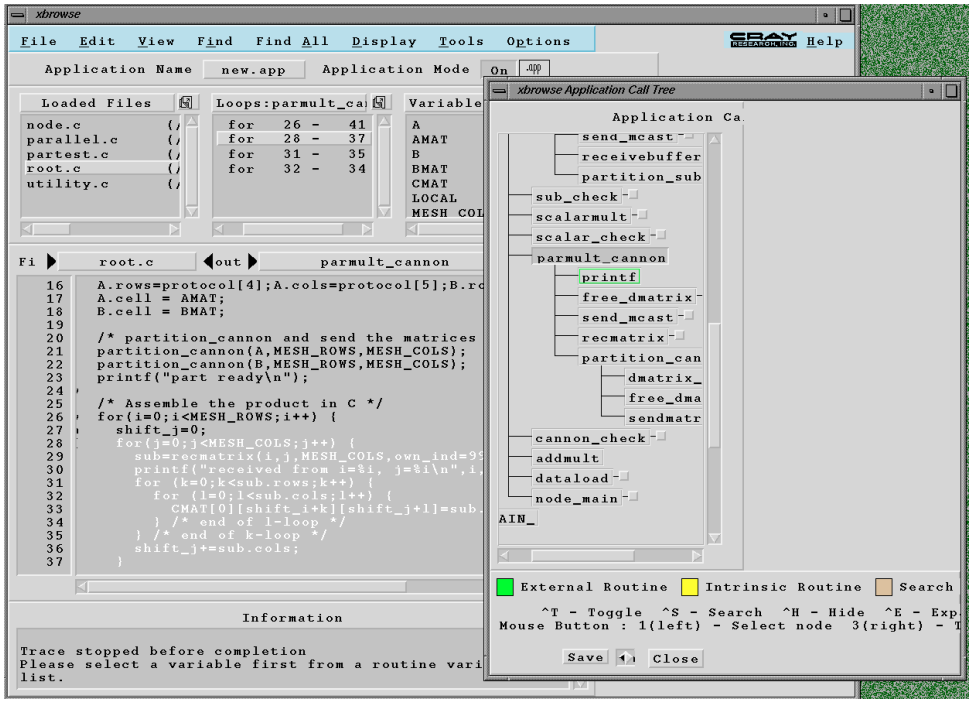



Figure 9.1: An example of an Xbrowse session.

The main window is composed of the following segments:

- The menu bar is located at the top of the main window. It displays buttons that open Xbrowse menus. To open a menu, position the cursor on the menu name and press the left mouse button.
- The upper display pane is composed of three separate information displays that list names of loaded files, routines, common blocks, and so forth. All lists have vertical scroll bars on the right. If information in a list exceeds the width of the listing area, a horizontal scroll bar is also displayed.

- The source code pane, located in the middle of the Xbrowse window, is the largest area of the window. This pane displays the current source code.
- The information pane is located at the bottom of the main Xbrowse window and provides information about the status of Xbrowse. You can also type equivalents of the Xbrowse commands for many menu options in this pane. (A list of these commands is available through the Help menu option.)

To open a file, position the cursor on the *File* menu button and press the left mouse button to open the menu. With the menu displayed, click on the *Open* option. A window is displayed on which you enter the name of the file (or files) to be opened.

A common activity while browsing code is to look for various types of objects. The following list names a few of the objects you can locate by Xbrowse:

- calls and callers
- common blocks (Fortran only)
- loops
- routines
- variables.

You can look for routine-based information with the commands in the *Find* menu. These commands let you look for file-based or application-based information. Most commands work in a similar manner, so trying a couple of searches will give you an idea of how to find objects in your code.

Call trees make it easy to see the structure of your code through a clear graphical format. The *Display* menu provides the following two commands for generating call trees:

- Call Tree
- Caller/Call Tree

Xbrowse gives you visual clues to help you identify various parts of your code within the call trees. External routines are shown in green and intrinsic routines are shown in yellow. An external call is one for which Xbrowse does not have the source code loaded.

The command *Call Tree* produces a static call tree of loaded source code. The name of the application or the file used by this option is shown at the top of the display. The called routines and subprograms are displayed in the middle.

The small, empty boxes following some nodes (routine names) on the tree indicate subtrees that stem from these nodes. To open the tree one

level position the cursor on the box and press the left mouse button. To close the tree one level position the cursor on the node name and press the right mouse button. When you click on a node, that node becomes the current node and is displayed in the main Xbrowse window.

The command *Caller/Call Tree* displays a static call tree of routines that call a specified subprogram and, in turn, displays any subprograms called by the specified subprogram. Selecting a node on the tree highlights the subprogram. The name of the source file (or the application when in application mode) used by this option is shown at the top of the window. Calling sequences are displayed in the middle of the window.

You find more information on the Program Browser with the command `man xbrowse`, from the help system of the Program Browser itself, or from the manual *Introducing the Program Browser* [Crag], available online at `http://www.csc.fi:8080`.

## 9.3 Debugging programs

### 9.3.1 Cray TotalView parallel debugger

The Cray TotalView is a symbolic debugger for parallel (or non-parallel) programs written in Fortran, C, C++ or HPF. The principal difference between conventional Unix debuggers and TotalView is that TotalView can debug multiple processes of parallel programs simultaneously and synchronously.

The Cray TotalView is available in both X Window System and line-mode versions. To use the Cray TotalView debugger, compile and link programs with the option `-g`. The option `-G` in Fortran, C and C++ compilers may also be used.

You may inspect either a malleable or non-malleable code by TotalView. The following example illustrates usage of TotalView with a fixed number of 16 processors:

```
t3e% f90 -X16 -g -o prog.x prog.f90
t3e% totalview ./prog.x &
```

When debugging a malleable code, use the TotalView option `-X`, e.g.,

```
t3e% totalview -X 16 ./prog.x &
```

The space before the number of processors is mandatory.

The TotalView graphical interface shows the program code, the call sequence, local variables and TotalView messages. Program execution can be controlled by pressing execution buttons or by typing in commands. The currently active section of source code is always displayed. When a subroutine is called, the debugger opens corresponding source files.

The user can move to the source of a subroutine by clicking on the name of the subroutine with the right mouse button.

TotalView has two execution modes: “all” or “single-processor”. In the execution mode “all”, the breakpoints and execution commands are applied to all processors simultaneously. In the single-processor execution mode, the user can set breakpoints individually for each processor. The execution mode is selected from the *PSet* menu. In the single-processor mode the active processor is selected from a slider marked *PE* just below the *Pset* menu.

The value of any variable can be examined by pressing the *Print...* button in the middle of the TotalView window. This brings up another window with the name of the variable as input. The actual values are shown in a third window, where the processor can be selected using a slider. The value can also be displayed by choosing the variable name in the program listing or the variable listing with the right-hand mouse button. There is also a graphical browser for arrays.

Figure 9.2 shows an example of a TotalView session. The two uppermost window panes show the call sequence and the values of local variables. Below these panes are buttons to control execution. Underneath is the program code.

The user can press the oval buttons on the left to define breakpoints. The current position is indicated with an arrow. The bottom pane in the large window shows messages from the debugger and can be used to issue commands manually. The small window at the bottom shows the value of the variable `mehigh` for the processor number 1.

A typical debugging session could consist of the following steps:

- Set the initial breakpoint either with the mouse or select *Set Breakpoint in Function* or *Set Breakpoint at Line* from the *Events* menu.
- Run the program to the breakpoint with the *Run* button. If the programs uses command-line arguments, use the *Run (args)* command from the *Control* menu.
- Examine the values of some variables with the *Print* button or by selecting the variable name with the right mouse button.
- Step through the code using the *Step* button.
- Step through the code without going into the called subroutines with the *Next* button.

### 9.3.2 Examining core files

When a program terminates with an error condition, a core file is often produced. This file contains the status of the program at the time of the

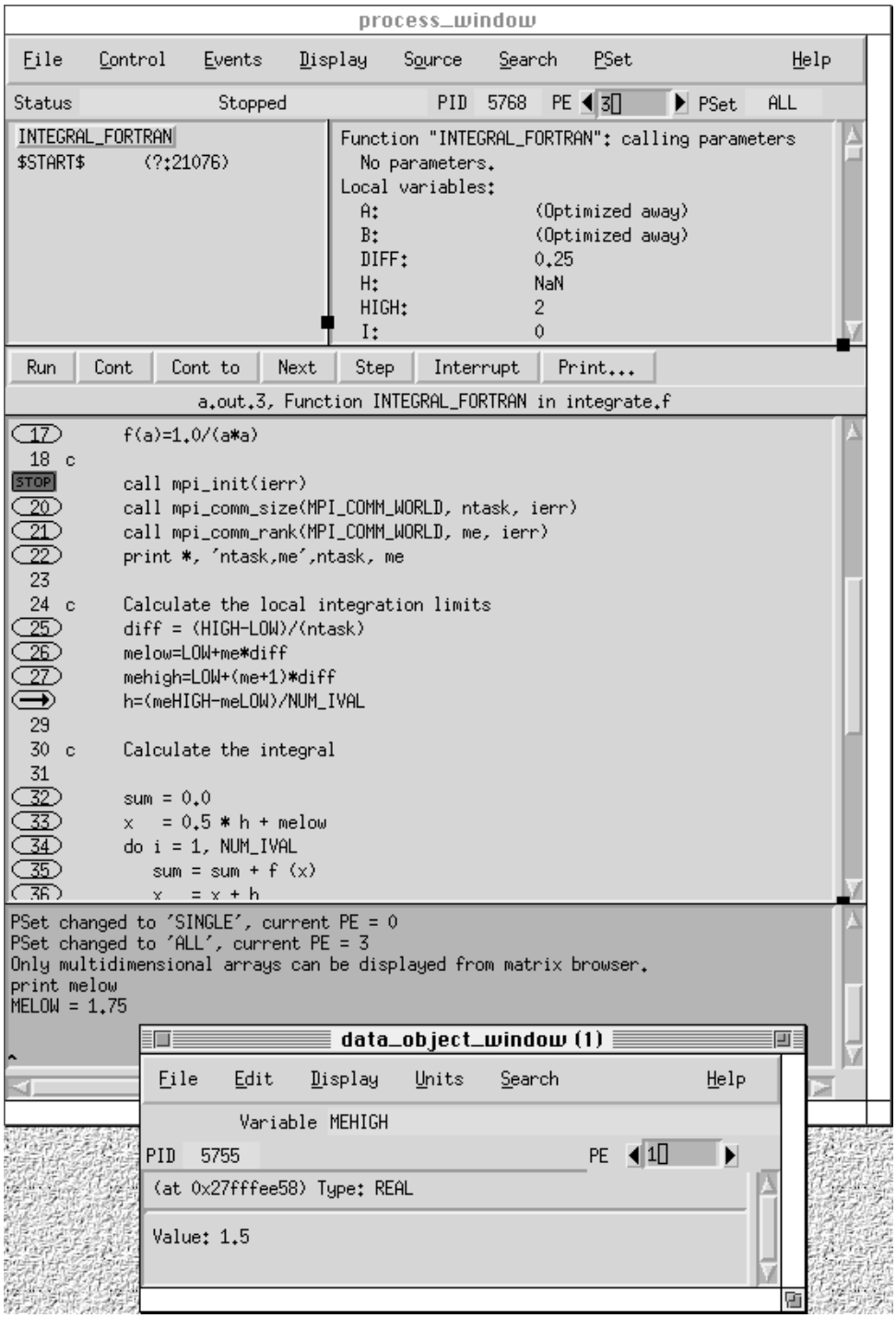



Figure 9.2: An example of a Cray TotalView session.

error, and it can be used to determine the cause of the problem.

The TotalView debugger can be used to examine core files. Start the debugger with the executable name (`prog.x` in the example) and the core file name:

```
totalview prog.x core
```

After this the debugger shows where each process has been stopped and you can examine the values of the variables. Some of the processes may have been stopped in an assembly routine. To get back to a user subroutine, select the subroutine name in the call sequence pane located at the upper left corner.

More information on the TotalView debugger can be obtained with the command `man totalview`, from the help system of the TotalView program itself or from the manual *Introducing the Cray TotalView Debugger* [Crae], available online at the WWW address `http://www.csc.fi:8080`.

## 9.4 Obtaining performance information

You can use the MPP Apprentice software to inspect possible performance bottlenecks in your program. There is also a text-based command `appview` for viewing the same profiling data. The Performance Analysis Tool (PAT) is a low-overhead profiling tool available on the T3E.

### 9.4.1 MPP Apprentice

MPP Apprentice is a postexecution performance analysis tool that helps you to locate bottlenecks on Fortran 90, C, C++ and HPF programs on the Cray T3E. It can be applied to both single processor and multiprocessor programs. MPP Apprentice reports, for example, time statistics summed across all processing elements for the whole program, as well as each `DO` loop, conditional clause or other statement groups in a program. It shows the total execution time, synchronization and communication time, the time to execute a subroutine and the number of instructions executed. It does *not* record time-stamped events, but collects summary information of the program execution.

MPP Apprentice works for both optimized and non-optimized code. It offers suggestions for improving performance and how to get rid of the bottlenecks. Apprentice works under the X Window System.

You have to use specific compiler options to generate a compiler information file (CIF) at compile time and a run-time information file (RIF)

when the program is executed. The files are passed to the Apprentice tool for graphical examination.

MPP Apprentice is used by the following steps. Fortran 90 programs are compiled with the option `-eA` and object codes linked with the MPP Apprentice library using the option `-lapp`:

```
t3e% f90 -c -eA prog.f
t3e% f90 -o prog.x prog.o -lapp
```

The compiler option `-eA` and the linker option `-lapp` work also with the PGHPF High Performance Fortran compiler. ANSI C and C++ programs are compiled with `-happrentice` option and object codes linked with the MPP Apprentice library `libapp.a`:

```
t3e% cc -c -happrentice prog.c
t3e% cc -o prog.x prog.o -lapp
```

The corresponding commands for C++ are:

```
t3e% CC -c -happrentice prog.c
t3e% CC -o prog.x prog.o -lapp
```

The next step is to run the parallel program. Be aware that the execution time of the instrumented code can now be considerably longer than without the MPP Apprentice hooks. After the execution you will have a run-time information file (RIF) called `app.rif` in your directory. After the program has been run start the MPP Apprentice tool with the command `apprentice`.

```
t3e% mpprun -n 4 ./prog.x
t3e% apprentice app.rif &
```

An example session with Apprentice is show in Figure 9.3. The top pane shows execution time for each subroutine. You can exclude or include the time spent in called subroutines with the buttons on top of the pane. You can click on the triangles to the right of the subroutine names to show or hide internal information about called subroutines.

The lower pane shows the number of instructions for a subroutine or a loop that has been selected in the upper pane. In some cases you can also get information on the shared memory usage or message passing usage for the selected routine.

The Apprentice tool can also provide textual reports on the performance of the code. Choose either *Observations* or *Reports* from the *Displays* menu to get the reports for the currently selected subroutine.

More information on the Apprentice tool can be obtained with the command `man apprentice`, from the help system of the Apprentice tool itself or from the manual *Introducing the MPP Apprentice Tool* [Craf], available online at the WWW address `http://www.csc.fi:8080`.

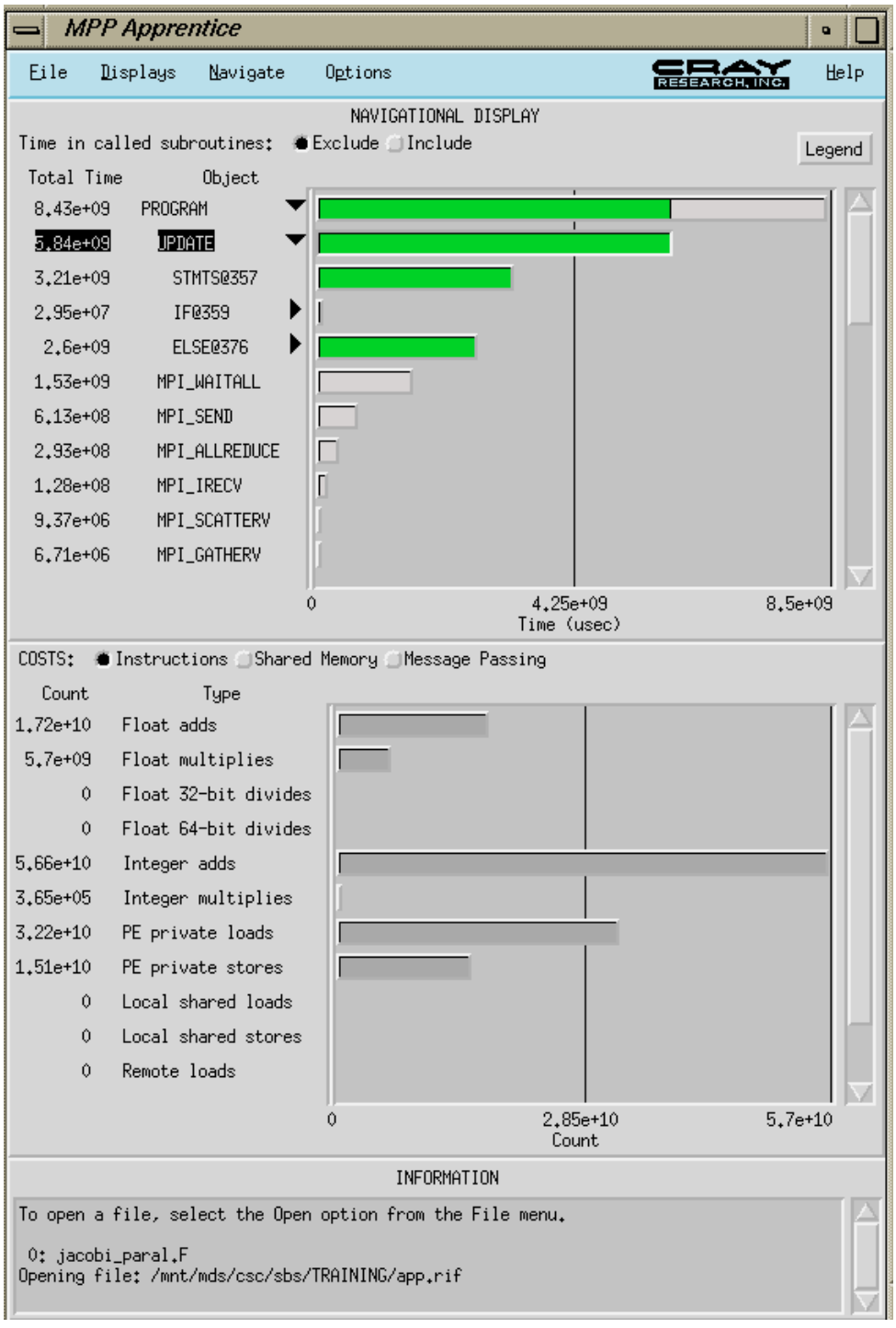



Figure 9.3: An example of an MPP Apprentice session. The upper pane shows timing statistics and the lower pane shows the number of instructions for the selected subroutine.

### 9.4.2 The appview command

In addition to MPP Apprentice, the `appview` command that a quick summary of the profiling data. Its output is similar to the output from the conventional Unix profiler `prof`. The `appview` command was developed at CSC and it relies on a few scripts that extract and sort information from the textual report produced by the command `apprentice -r`.

The following example illustrates the usage. The command line is

```
appview app.rif | more
```

The default RIF file name is `app.rif` and can be omitted from the command line. The output looks like this:

```
                #                    #     #
               # #    #####   #####  #     #     #    ######  #    #
              #   #   #    #  #    # #     #     #    #       #    #
             #     #  #    #  #    # #     #     #    #####   #    #
             #######  #####   #####   #   #      #    #       # ## #
             #     #  #       #        # #       #    #       ##  ##
             #     #  #       #         #        #    ######  #    #

        Fri Feb 21 13:32:07 EET 1997


Total time            24.273 seconds.
No. of routines        8

Routine             Exclusive    (%)      Inclusive    in_Called     Parallel
=======             =========    ===      =========    =========     ========
COLL2                  14.968 ( 61.67)       14.968        0.000       14.946
_HSIN                   8.216 ( 33.85)  <not instrumented>
INIT                    1.062 (  4.38)        9.277        8.216        1.062
_FWF                    0.022 (  0.09)  <not instrumented>
f$init                  0.005 (  0.02)  <not instrumented>
COLL2_TEST              0.000 (  0.00)       24.250       24.250        0.000
BARRIER                 0.000 (  0.00)  <not instrumented>
$END                    0.000 (  0.00)  <not instrumented>
```

The columns `Exclusive` and `Inclusive` show the execution time without and with the time spent in called subprograms, respectively. The column `in_Called` show the time for called subprograms.

### 9.4.3 PAT

The Performance Analysis Tool (PAT) provides a low-overhead method for profiling programs and obtaining timing and performance counter information. PAT can also be used for determining load balance across processing elements, generating and viewing trace files, performing event traces, etc. These advanced features are not available in the MPP Apprentice. On the other hand, PAT lacks the graphical interface of the Apprentice. PAT is used in evaluating the performance and scaling new T3E projects at CSC.

PAT periodically samples the program counter to generate an execution-time profile and uses the processor performance counters to gather

performance information and instruction counts.

PAT is able to analyze programs written in Fortran 90, C, C++ and HPF. The executable only needs to relinked, no recompiling is necessary. The linker option `-l pat` along with the PAT specific cld file `pat.cld` are required.

As an example, suppose that a Fortran 90 program in the file `prog.f90` is to be analyzed. The following commands can be used:

```
t3e% f90 -c prog.f90
t3e% f90 prog.o -o prog -l pat pat.cld
```

C, C++ and HPF programs are linked similarly. A log file of the type `pdf.1234` is produced during the execution.

Timing information is then displayed with the command

```
t3e% pat -T prog pdf.1234
```

Sample output for a four PE run looks like

```
Elapsed Time              4.229 sec     4 PEs
User    Time (ave)        3.441 sec   81%
System  Time (ave)        0.023 sec    1%
```

The PAT option `-m` produces performance counter statistics:

```
t3e% pat -m prog pdf.1234
```

```
Performance counters for FpOps

      Values given are in MILLIONS.

PE    cycles  operations   ops/sec     dcache     misses/sec
                                       misses

 0    425.34  152.45       134.39      5.48       4.83
 1    1574.81 152.40       36.29       5.61       1.34
 2    1574.87 152.40       36.28       5.62       1.34
 3    1575.15 152.40       36.28       5.62       1.34
```

The column ops/sec contains the floating point performance given in Mflop/s for each PE. A high cache miss rate can be caused by less than optimal program design. Instead of floating point operations it is possible to measure integer performance by setting

```
t3e% setenv PAT_SEL INTOPS
```

Memory load and store operations can also be monitored.

Profile information can be obtained with the option `-p`. Normally only subroutine-level profile is available. To produce a line-level profile the

program must be compiled with the option `-g`, which disables optimization and thus increases the run time.

More information about PAT in general and on its advanced features can be found on the manual pages (`man pat`).

## 9.5 Tracing message passing: VAMPIR

VAMPIR *Visualization and Analysis of MPI Resources* is a profiling tool for MPI applications. It consists of two parts: the VAMPIRtrace library which is linked to the program to be profiled, and the VAMPIR visualization tool which is used to analyze the trace file after the program execution.

The VAMPIR trace library is installed on the Cray T3E. The trace files must, however, be inspected with the visualization tool on Caper.

### 9.5.1 The VAMPIRtrace library

VAMPIRtrace is an MPI profiling library that generates VAMPIR traces. It hooks into the MPI profiling interface which guarantees low tracing overhead. The effects of distributed clock drifts are corrected automatically. Tracing can be enabled or disabled during runtime. The profiling library is suitable for Fortran 90, FORTRAN 77, C and C++ programs.

Before using the VAMPIRtrace library the VAMPIR environment on the T3E must be initialized with the command `use vampir`. The VAMPIRtrace library can be linked to a user's program, without any amendments, with the options `-lVT -pmpi`. In practice, however, one needs to add calls to VAMPIRtrace API routines in the source code to facilitate the analysis. This is done most conveniently using the source code preprocessor to maintain a single version to be run with or without the VAMPIRtrace. When a parallel program linked with the trace library is executed, it generates a trace file with the suffix `.bvp`. The trace file may be so large that you should run the program in the temporary or work directory (`$TMPDIR` or `$WRKDIR`).

The following example shows how to generate a trace file from a Fortran 90 master-slave application. The source code in the example is divided into modules in files `loadbal.F90`, `pgamma.F90`, `mpi.f` and `vt.F90`, and can be found at the Web address `http://www.csc.fi/programming/examples/vampir_mod/`

```
t3e% use vampir
[vampir is now in use]
t3e% f90 -I $PAL_ROOT/include -o loadbal mpi.f vt.F90 \
loadbal.F90 pgamma.F90 -lVT -lpmpi -DUSE_VT
```

The source code includes the VAMPIRtrace API calls by the preprocessor macro `USE_VT`. It is recommended to include the definitions in the file `$PAL_ROOT/include/VT.inc` with the option `-I`. The program may be run now, e.g., by the command

```
t3e% mpprun -n8 loadbal < input
```

If the program is run as a batch job (see Chapter 8), the command `use vampir` has to be included in the jobfile.

By default, the trace file is generated between the calls `MPI_INIT` and `MPI_FINALIZE`. Tracing may also be disabled and enabled by the API calls `VTTRACEOFF` and `VTTRACEON`.

VAMPIRtrace gathers information of *activities*, i.e., process states with a start and stop time. Activities are identified by a *class* name and a *symbolic* name. A class can contain an arbitrary number of activities. Activities can also be nested like subroutines calls. MPI calls automatically belong to the class *MPI*. The class and symbolic names are subsequently used by the VAMPIR visualization tool.

The following Fortran code shows how to define a section of code to belong to the class *Application* with the symbolic name *initialization*. The integer value 1 is a user-chosen label which must be applied in a globally consistent manner with and only with the `initialization` tag. The variable `ierr` is an error indicator.

```
INTEGER :: ierr
CALL VTSYMDEF(1, 'initialization', 'Application', ierr)
CALL VTBEGIN(1, ierr)
... initialization code to be marked ...
CALL VTEND(1, ierr)
```

In Fortran 90, it is a good idea to define symbolic names for the tag integers, e.g., in a separate `MODULE vt` by

```
INTEGER, PARAMETER :: INIT=1
```

A corresponding C code segment is

```
VT_symdef(1, "initialization", "Application");
VT_begin(1);
... initialization code to be marked ...
VT_end(1);
```

### 9.5.2 The VAMPIR visualization tool

The trace file must be transferred to Caper for the analysis with the X Window System based VAMPIR visualization tool. The VAMPIR environment on Caper must be set up using the command

```
use vampir
```

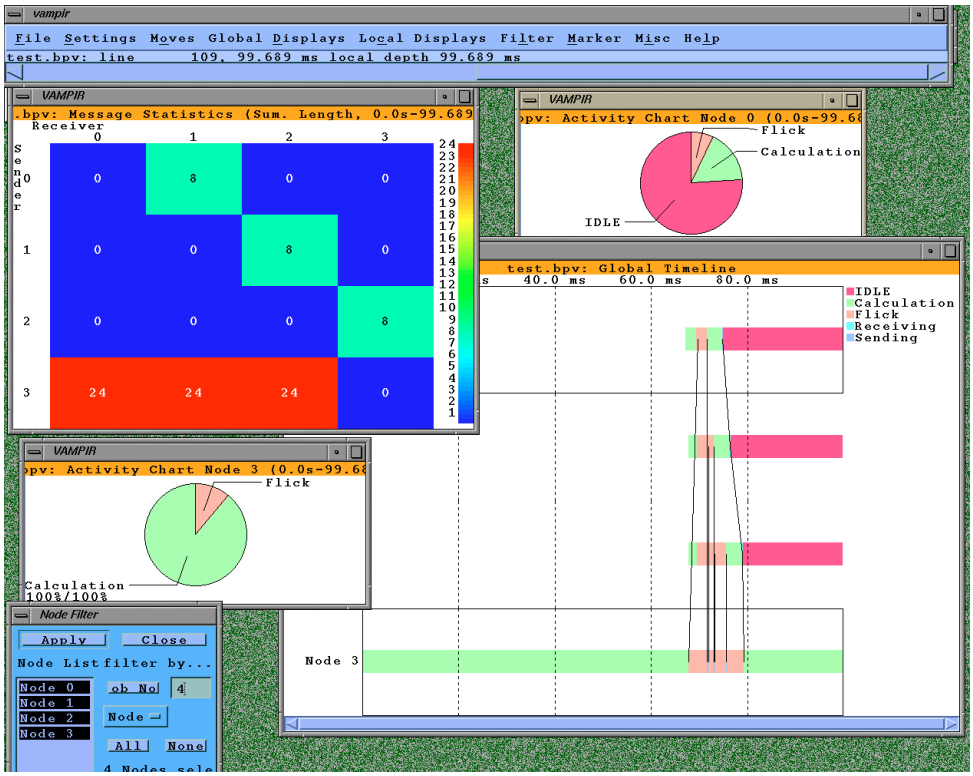



Figure 9.4: A sample of a VAMPIR session. On the lower right corner is the global timeline display showing the communication pattern. The communication statistics display on the upper left corner shows the number of messages sent between processors. The two pie charts show the activities of individual processors.

Before the VAMPIR visualization tool is used for the first time, the user should create a directory `.VAMPIR_defaults` in one's home directory and copy the configuration file `$PAL_ROOT/etc/VAMPIR.cnf` there.

The VAMPIR visualization tool is started with the command

```
vampir tracefile
```

The VAMPIR program can also open trace files that have been compressed using the `gzip` or `compress` commands.

There are three basic display modes to visualize the activities and message passing. These can be selected from the *Global Displays* menu:

- *Timeline* visualizes activities and message passing in a time interval along a time axis. The interval or a processor set can be selected from the display by drawing a rectangle with the left button of the mouse. The displayed messages may also be restricted or identified by the message tags, communicators or size.
- *Chart view* shows the time spent in different activities in a time

interval by selected processors.

- *Process view* shows the portion of time spent in a given activity class.

All the displays have several options which can be controlled through the pop-up menu from the right mouse button. These include

- Communication statistics: the total amount of communication between all processor pairs is shown. This is opened from the *Global timeline* display by selecting *Comm. Statistics*. The communication statistics display can be linked to the *Global timeline* display to show the statistics for the currently visible portion of time only. This is done in the communications statistics window by selecting *Use timeline portion* from the pop-up menu.
- Message lengths: the lengths of individual messages are shown. In the *Global timeline*, select *Identify Message* and point to a message line in the *Global timeline* display.
- Components/Parallelism: the number of processors engaged in a given activity is shown in the *Global timeline*.

Figure 9.4 shows a sample session of the VAMPIR visualization tool with the global timeline, communication statistics and processor activity chart displays.

### 9.5.3 More information

A compressed version of the user's guide of the VAMPIR visualization tool is available on Caper as `$DOC/VAMPIR-userguide.ps.gz`. Copy this file to a temporary directory and use the command `gunzip` to uncompress the file. A user's guide for the VAMPIRtrace library is available on Caper as `$DOC/VT-userguide.ps`.

# Chapter 10

# Miscellaneous notes

This chapter discusses some additional topics, such as timing of programs and defining the scalability of parallel programs.

## 10.1 Obtaining timing information

The most useful measure of processing time in a parallel environment is the wall clock time. This is due to the fact that traditional CPU times are processor-based, whereas the wall clock time gives a global view of aggregate parallel performance. All CSC's Cray T3E processors run at 375 MHz.

### 10.1.1 The `timex` command

The easiest way to find out wall clock times is to use the Unix command `timex` in front of the executable name:

```
t3e% timex mpprun -n 16 ./prog.x
```

### 10.1.2 The wall clock timer

On all Cray platforms the following C routine can be used to return the elapsed wall clock time in seconds:

```
#include <unistd.h>
double SECS(void) {
  static long cpcycle = 0;
  /* Get cycle time in picoseconds */
  if (cpcycle == 0) cpcycle = sysconf(_SC_CRAY_CPCYCLE);
```

```
  /* Wall clock time in seconds */
  return (double) _rtc() * cpcycle * 1.0e-12;
}
```

This routine can be called either in C/C++ or Fortran. In C/C++ the syntax is as follows:

```
extern double SECS(void);
double t1, t2, dt;
t1 = SECS();
... perform calculations ...
t2 = SECS();
dt = t2 - t1;
printf("Elapsed time: %f\n",dt);
```

In Fortran 90:

```
REAL :: t1, t2, dt
REAL, EXTERNAL :: secs
t1 = secs()
... perform calculations ...
t2 = secs()
dt = t2 - t1
WRITE (*,*) 'Elapsed time: ', dt
```

To use the SECS routine from Fortran you have to first compile the C routine and then link it with your program:

```
t3e% cc -c secs.c
t3e% f90 -o prog.x prog.f90 secs.o
```

### 10.1.3 The CPU clock timer

You can use the library function `ICPUSED()` in Fortran codes. This function returns the CPU time of a task in real-time clock ticks:

```
INTEGER :: time

time = ICPUSED()

...computation...

time = ICPUSED() - time
WRITE (*,*) 'CPU time in user space = ', &
    time, ' clock ticks'
```

C programmers can use the function `cpused()`:

```
#include <time.h>

time_t before, after, utime;

before = cpused();
```

```
...computation...

after = cpused();
utime = after - before;
printf("CPU time in user space = %ld clock ticks\n",
    utime);
```

### 10.1.4 Example of timing

Here is an example of a C program, which computes the matrix product using the SGEMM routine from Libsci:

```
#include <stdio.h>
#include <fortran.h>
#include <time.h>
#include <unistd.h>

#define DGEMM SGEMM

#define l 450
#define m 500
#define n 550

main()
{
  double a[n][l], b[l][m], c[n][m], ct[m][n];
  int ll, mm, nn, i, j, k;
  double alpha = 1.0;
  double beta = 0.0;
  void DGEMM();
  char *transposed = "t";
  _fcd ftran;
  time_t before, after, utime;

  /* Initialize */

  for (i = 0; i < n; i++)
    for (j = 0; j < l; j++)
      a[i][j] = i-j+2;
  for (i = 0; i < l; i++)
    for (j = 0; j < m; j++)
      b[i][j] = 1/(double)(i+2*j+2);

  ftran = _cptofcd(transposed, strlen(transposed));

  ll = l; mm = m; nn = n;

  before = cpused();

  DGEMM(ftran, ftran, &nn, &mm, &ll, &alpha, a, &ll,
        b, &mm, &beta, ct, &nn);
```

```
  after = cpused();
  utime = after - before;

  printf("ct[10][10] is %.6f\n", ct[10][10]);
  printf("CPU time in user space = %ld clock ticks\n",
         utime);

  exit(0);
}
```

See Section 6.3 on page 54 for more details on calling Fortran routines (here SGEMM) from C.

Here is an example of compiling and executing this program:

```
t3e% cc matmul.c
t3e% timex mpprun -n 2 ./a.out
ct[10][10] is -345.015608
ct[10][10] is -345.015608
CPU time in user space = 371395739 clock ticks
CPU time in user space = 371379536 clock ticks

        seconds         "clocks"
real    3.979838        (1193951546)
user    2.063669        (619100700)
sys     0.254990        (76497000)
```

Here we executed the program identically on two processors.

## 10.2 Parallel performance prediction

Several different models can be used to measure the scalability of a parallel program. Depending on your application and preferences, you may want to use parallelism to decrease execution time, to run bigger models or to optimize the speedup of parallel processing.

The so-called *Amdahl's law* applies to a fixed model size when you are using different numbers of processors. This model supposes that you can split the program in two parts, sequential and parallel. The sequential part takes $W_1$ seconds in all cases. The parallel part takes $W_p/p$ seconds, where the $W_p$ is the size of the parallel task and $p$ is the number of parallel processors.

*Speedup* $S_p$ is defined as the ratio of the time on one processor divided by the time on $p$ processors:

$$S_p = \frac{W_1 + W_p}{W_1 + W_p/p}.$$

This equation can be normalized by setting $W_1 + W_p = 1$. Here $W_1 = \alpha$ (the sequential portion) and $W_p = 1 - \alpha$ (the parallel portion). Now you get

$$S_p = \frac{1}{\alpha + (1 - \alpha)/p}.$$

For example, if you have a program which contains a 10% sequential part the equation reads

$$S_p = \frac{1}{0.1 + 0.9/p}.$$

Setting $p \to \infty$, you get the maximum speedup, which is $1/0.1 = 10$. Therefore, *the sequential part starts to dominate, when you add more processors.*

*Efficiency e* measures how well the code is parallellized:

$$e = \frac{S_p}{p}.$$

In the best case the efficiency is 1 and we say that *the scalability is linear.*

Ambdahls' law gives a rather pessimistic picture of scalability. In many cases it is not necessary to keep the model size fixed when doing parallel computing. This way, the parallel part of the program $(1 - \alpha)$ can be increased at the same time you add more processors.

*Gustafson's law* specifies a different scalability concept: you do not keep the model size fixed — instead, you *keep the solution time fixed.* This means that you want to solve the largest problem size possible, given a time limit. A typical case is weather forecasting: you want to get a 24-hour forecast within one hour, since the value of this forecast decreases rapidly as time goes by.

Gustafson's scaling law can be expressed as follows:

$$S'_p = \frac{W_1 + pW_p}{W_1 + W_p}.$$

Note that the time on $p$ processors is compared to the time it would take to compute this task on one processor. By normalizing ($W_1 + W_p = 1$, as above) you find

$$S'_p = p - \alpha'(p - 1).$$

Suppose that you have 128 processors available. Now,

$$S'_{128} = 128 - \alpha'(128 - 1) = 128 - 127\alpha'.$$

If the sequential part is $\alpha' = 0.1$, you obtain a speedup of $128 - 12.7 = 115.3$. If $\alpha' = 0.05$, you get a speedup of about 122. However, note that the bigger model size might actually not fit in the memory of one processor, so you most probably are not able to do the comparison runs.

You can derive the following connection between the parameters $\alpha$ and $\alpha'$ in Amdahl's and Gustafson's laws:

$$\alpha = \frac{\alpha'}{p - \alpha'(p-1)},$$
$$\alpha' = \frac{\alpha p}{1 + \alpha(p-1)}.$$

Figure 10.1 shows how these scalability laws are connected. Figure 10.2 shows how the speed of the code scales (according to Amdahl's law) when $\alpha = 0.02$ and $\alpha = 0.002$.

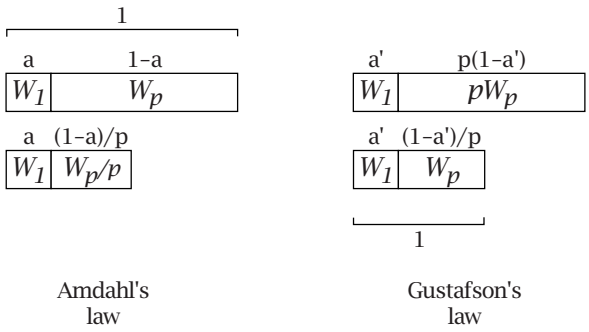



Figure 10.1: Illustration of Amdahl's and Gustafson's scalability laws.

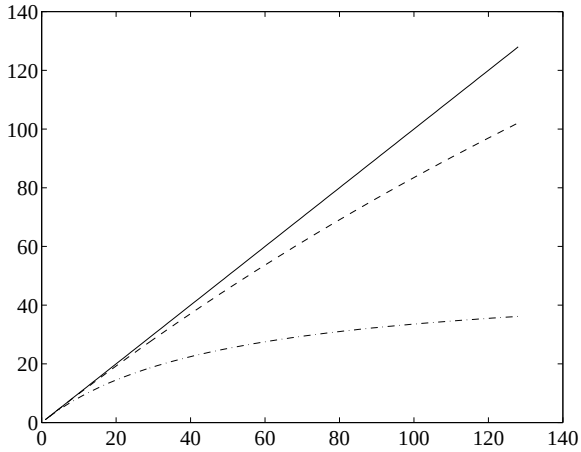



Figure 10.2: Illustration of Amdahl's scalability law for $\alpha = 0.02$ (−−) and $\alpha = 0.002$ (− · −).

In addition to Amdahl's and Gustafson's laws, there is also a model for *memory-bounded speedup*. In this case the actual constraint is the memory of the parallel machine, and you want to scale the program to use all available memory. A typical case of this is 3D fluid mechanics, where you usually want to solve large problems (dense grid) as efficiently as possible.

## 10.3 Scalability criteria at CSC

CSC imposes the following scalability criteria for Cray T3E applications:

> The speed of the application has to increase by 50%, when the number of processors is doubled.

For example, when doubling the processors from 8 to 16, the speed of the code should be 1.5 times as much.

You can use Gustafson's law for nice formulation of this criteria: compute the same calculation using $p$ and $p/2$ processors. Then you get the relative speedup $S_2'$ from Gustafson's law:

$$S_2' = 2 - \alpha'.$$

CSC's criteria is $S_2' > 1.5$, which corresponds to the criteria

$$\alpha' < 0.5$$

based on Gustafson's law. This can be also formulated as follows:

> In a large application, less than half of the time should be used for sequential processing.

## 10.4 More information

See the manual pages for more information about timing, e.g., `man timex` or `man ICPUSED`.

The basics of parallel programming are discussed in the textbook *Designing and Building Parallel Programs* [Fos95]. Another good textbook is *Introduction to Parallel Computing — Design and Analysis of Algorithms* [KGGK94].

# Appendix A

# About CSC

Center for Scientific Computing, or simply CSC, is a national service center that specializes in scientific computing and data communications. CSC provides modeling, computing and information services for universities, research institutes and industry. For example, Finland's weather forecasts are computed with the Cray supercomputers operated by CSC.

All services are based on versatile supercomputing environment, ultrafast FUNET data communications and high standard of expertise in various branches of science and information technology.

The premises of CSC are located in the building of TT-Tieto in Otaniemi, Espoo (street address: Tietotie 6) in the neighborhood of the Helsinki University of Technology (HUT) and the Technical Research Centre of Finland (VTT).

The mail address is

> Center for Scientific Computing (CSC)
> P.O. Box 405
> FIN-02101 Espoo
> Finland

The international phone number is +358-9-457 2001 and the fax number is +358-9-457 2302.

The best way to get help in problems is to use e-mail. You can send e-mail to customer advisers through Internet by using the address `helpdesk@csc.fi`.

CSC experts are available on the CSC premises and they can be reached by phone on weekdays from 9 am to 4 pm. Customers can also get customer information and answers to operational questions by calling the CSC Help Desk, tel. (09) 457 2821. The Help Desk answers in this service number on weekdays from 8 am to 4 pm. Outside the working

hours you can leave a message. The Help Desk registers the call, writes down the problem and tries to solve the problem immediately. If this is not possible, the problem is forwarded to the right experts to take care of it.

See the WWW pages at the address

```
http://www.csc.fi
```

for more information about CSC services.

# Appendix B

# Glossary

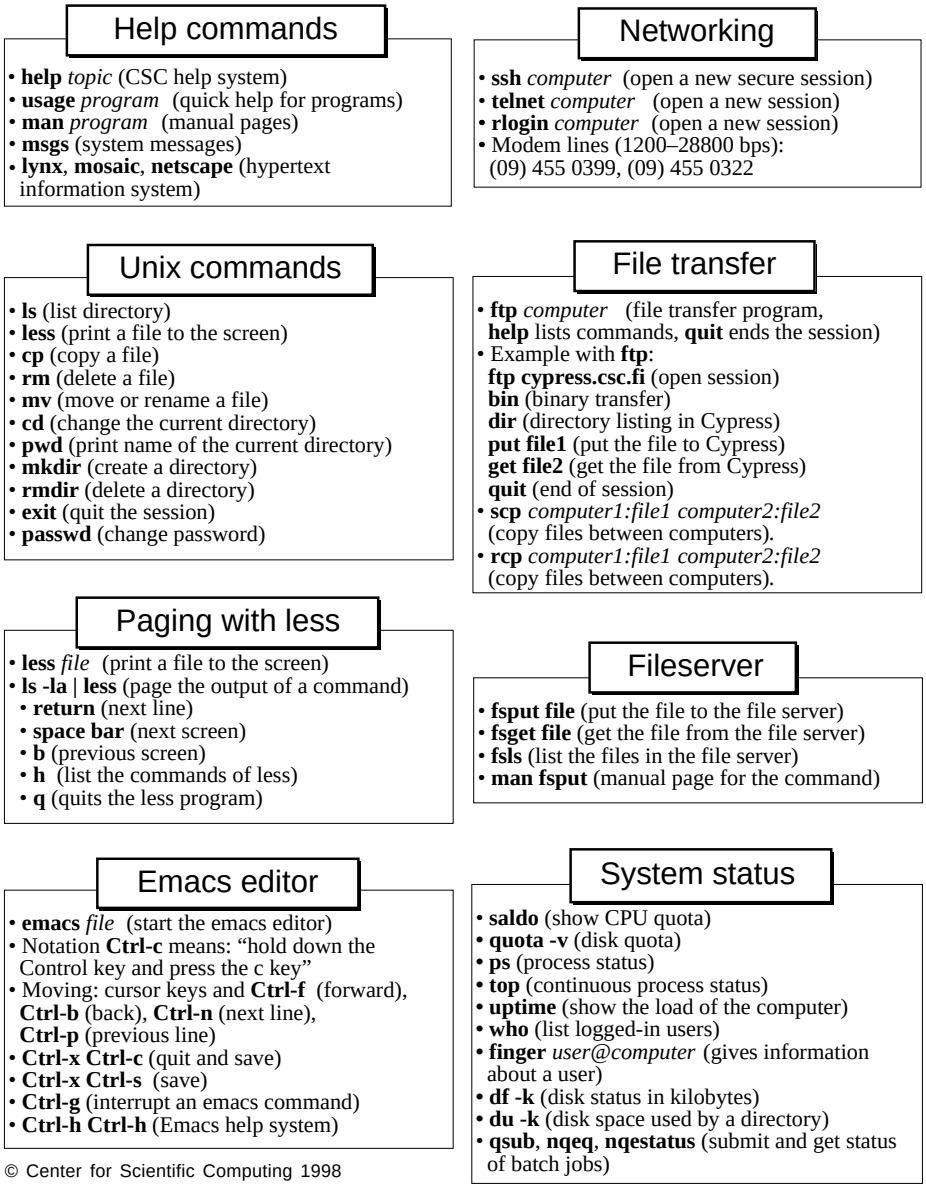
| | |
|---|---|
| ANSI | *American National Standards Institute*, organization deciding on the U.S. computer science standards. |
| Bandwidth | The amount of data that can be sent through a given communications circuit per second. |
| BLACS | *Basic Linear Algebra Communication Subroutines*, a subroutine package for interprocess communication in PBLAS and ScaLAPACK. |
| BLAS | *Basic Linear Algebra Subroutines*, a subroutine package for fundamental linear algebra operations. |
| Cache | A small fast memory holding recently accessed data, designed to speed up subsequent access to the same data. |
| CSC | *Center for Scientific Computing*, a national service center that specializes in a scientific computing and data communications. |
| Data-parallel | A SIMD programming style where the programmer specifies the data distribution between processes, and the compiler generates a parallel executable code. |
| Data passing | A communication technique between parallel processes where the routines for sending or receiving data are performed by only one of the processes. |
| Emacs | A popular screen editor used on Unix, VMS and other systems. |
| FUNET | *Finnish University and Research Network*, maintained by CSC. |

| | |
|---|---|
| HPF | *High Performance Fortran*, a data-parallel language extension to Fortran 90. |
| HTML | *Hypertext Markup Language*, a language for writing hypertext documents in the Web. |
| IEEE | *Institute of Electrical and Electronics Engineers*, the world's largest technical professional society, based in the USA. |
| IMSL | Fortran subroutine library for numerical and statistical computation. |
| LAPACK | *Linear Algebra PACKage*, a collection of subroutines for solving systems of linear equations and eigenvalue problems. |
| Latency | The time that it takes to start sending a package across the interconnection network. |
| Libsci | Cray's numerical subroutine library. |
| Malleable | Malleable programs can be run on any number of processors, specified at execution by the command `mpprun`. |
| Message passing | A communication technique between parallel processes where the data transfer from the local memory of one process to the local memory of another requires operations to be performed by both processes. |
| Microkernel | An operating system design which puts emphasis on small modules that implement the basic features of the system kernel and can be flexibly configured. |
| MIMD | *Multiple Instruction, Multiple Data*, a parallel computer architecture or programming style where many functional units perform different operations on different data. |
| MPI | *Message Passing Interface*, a standardized and portable de facto standard message-passing library. |
| NAG | Fortran subroutine library for numerical computation. |
| Netlib | An Internet archive accessed through e-mail to obtain, e.g., subroutine libraries. |
| Node | Processing element plus the interconnection network components. |

| | |
|---|---|
| Non-malleable | Non-malleable programs are fixed at compile time to run on a specific number of processors. |
| NQE | *Network Queueing Environment*, the batch queuing system on the T3E. |
| PBLAS | *Parallel BLAS*, parallelized version of BLAS. |
| PDF | *Portable Document Format*, a format defining the final layout of a document. The native file format for the Adobe Acrobat software package. |
| PE | *Processing Element*, consisting of a microprocessor, local memory and support circuitry. |
| PostScript | A widespread page description and printer language. |
| PVM | *Parallel Virtual Machine*, a standardized and portable message-passing library that is somewhat older than MPI. |
| RISC | *Reduced Instruction Set Computer*, a processor whose design is based on the rapid execution of a sequence of simple instructions rather than on a large variety of complex instructions. |
| Scalability | A measure of how efficiently a parallel program will work when the number of processors is increased. |
| ScaLAPACK | *Scalable LAPACK*, parallelized version of LAPACK. |
| SHMEM | *Shared Memory Library*, Cray's data-passing library. |
| SIMD | *Single Instruction, Multiple Data*, a parallel computer architecture or programming style where many functional units perform the same operations on different data. |
| Ssh | *Secure Shell*, a program for encrypted communication between two hosts over an insecure network. |
| Streams | Stream buffers, a mechanism to fetch data in advance from the local memory of the T3E PEs. |
| Unix | The most widely used multi-user general-purpose operating system in the world. |

# Appendix C

# Metacomputer Environment



## E-mail

- **pine** (start the e-mail program)
- Reading: choose a message with arrow keys and press **return**
  - **i** (index of received messages)
  - **c** (send mail)
  - **r** (reply to mail)
  - **f** (forward mail)
  - **q** (quit)
  - **?** or **Ctrl-g** (help); notation Ctrl-g means "hold down the control key and press g"
  - **Ctrl-c** (interrupt the current operation)

Sending mail:
- **pine First.Surname@csc.fi** (send e-mail to the given e-mail address)
  Subject: **Hello!** (subject line)
  Cc: (other receivers)
  Write the message
  **Ctrl-x** (send the message)

## Usenet News

- **nn** (read the Usenet news)
  - **?** (get help)
  - **Q** (quit the program)

## Command shell tcsh

- **tcsh** is CSC's standard command shell with advanced command line editing
- Left and right arrow keys move the cursor in the command line
- Up and down arrow keys recall old command lines
- The **Tab** key tries to complete a file name or a command name
- **Ctrl-d** is the end-of-file character on Unix systems
- **Ctrl-d** lists possible choices while you write a file name or a command name
- Output of a command to a file:
  **ls -la** > *file*
- Chaining multiple commands:
  **ls -la | less**

## Compilers

| | Fortran 90 | Fortran 77 | C | C++ |
|---|---|---|---|---|
| Cray C94 | **f90** | **cf77** | **cc** | **CC** |
| Cypress | **f90,f90nag** | **f77** | **cc** | **g++, CC** |
| Cypress2 | **f90** | **f77** | **cc** | **g++, CC** |
| Caper | **f90,f95** | **f77** | **cc** | **g++, cxx** |
| Cray T3E | **f90** | **f90** | **cc** | **CC** |

## Compiling

Example of a compilation command (Cypress):
**f90 -o prog prog.f -lcomplib.sgimath**
Run the program: **./prog**

## Printing

Print a PostScript or text file:
**lpr -Pcsc_post** *file*
Check the status of the print job: **lpq -Pcsc_post**
Remove a print job: **lprm -Pcsc_post** *job_id*

## Computers

- **cray.csc.fi** (Cray C94 vector computer)
- **cypress.csc.fi** (compute server)
- **cypress2.csc.fi** (compute server)
- **caper.csc.fi** (compute server)
- **azalea.csc.fi**, **orchid.csc.fi** (graphics servers)
- **voxopm.csc.fi** (interactive front end machine)
- **fs.csc.fi** (file server)
- **t3e.csc.fi** (Cray T3E massively parallel computer)

## Contact information

- Address: Center for Scientific Computing, P.O. Box 405, FIN-02101 Espoo, Finland
- Street Address: Tietotie 6, Otaniemi, Espoo
- Exchange: (09) 457 2001, telefax (09) 457 2302
- CSC HelpDesk (09) 457 2821 or e-mail address **helpdesk@csc.fi**
- Accounts and passwords: (09) 457 2075 or e-mail address **usermgr@csc.fi**
- List of application software and specialists with the command **help ohjelmistolista**
- Operators' phone: 0400 465 293, e-mail **oper@csc.fi**

# Bibliography


[Craa] Cray Research, Inc. *CF90 Commands and Directives Reference Manual*. SR-3901. 2.6, 5.10

[Crab] Cray Research, Inc. *Cray C/C++ Reference Manual*. SR-2179 3.0.2. 2.6, 6.7

[Crac] Cray Research, Inc. *Cray T3E Fortran Optimization Guide*. SG-2518. 2.6, 5.5, 5.10

[Crad] Cray Research, Inc. *Introducing CrayLibs*. IN-2167 3.0. 4.6

[Crae] Cray Research, Inc. *Introducing the Cray TotalView Debugger*. IN-2502 3.0. 9.3.2

[Craf] Cray Research, Inc. *Introducing the MPP Apprentice Tool*. IN-2511 3.0. 9.4.1

[Crag] Cray Research, Inc. *Introducing the Program Browser*. IN-2140 3.0. 9.2

[Crah] Cray Research, Inc. *Message Passing Toolkit: PVM Programmer's Manual*. SR-2196 1.1. 7.3.3

[For95] Message-Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, 1995. 1.7, 7.2.6

[Fos95] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. Internet-osoite `http://www.mcs.anl.gov/dbpp`. 1.7, 10.4

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994. 1.7, 7.2.6

[HKR93] Juha Haataja, Juhani Käpyaho, and Jussi Rahola. *Numeeriset menetelmät*. CSC – Tieteellinen laskenta Oy, 1993. 1.7

[HM97] Juha Haataja and Kaj Mustikkamäki. *Rinnakkaisohjelmointi MPI:llä*. CSC – Tieteellinen laskenta Oy, 1997. 1.7, 7.2.6

[HRR96] Juha Haataja, Jussi Rahola, and Juha Ruokolainen. *Fortran 90*. CSC – Tieteellinen laskenta Oy, 1996. 1.7, 5.10

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing — Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., 1994. 1.7, 10.4

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994. 7.5

[KR97] Tiina Kupila-Rantala, editor. *CSC User's Guide*. CSC – Tieteellinen laskenta Oy, 1997. URL `http://www.csc.fi/oppaat/cscuser/`. 1.7

[Lou97] Kirsti Lounamaa, editor. *Metakoneen käyttöopas*. CSC – Tieteellinen laskenta Oy, 2nd edition, 1997. 1.7, 2.6, 5.10

[Pac97] Peter S. Pachero. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997. 1.7, 7.2.6

[Saa95] Sami Saarinen. *Rinnakkaislaskennan perusteet PVM-ympäristössä*. CSC – Tieteellinen laskenta Oy, 1995. 1.7, 7.3.3

[SOHL+96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996. 1.7, 7.2.6

# Index