

WORDT
NIET UITGELEEND

NIET
UITLEEN-
BAAR

NATIONAAL LUCHT- EN RUIMTEVAARTLABORATORIUM
NATIONAL AEROSPACE LABORATORY NLR THE NETHERLANDS



NEC

MEMORANDUM IR-94-030

FINAL REPORT
PARALLELISING NaS/RVS VISUALISER

Klaas Jan Wierenga
for National Aerospace Laboratory NLR
and *University of Groningen*
The Netherlands

Supervisors:
Ir. G. Westenberg (NLR)
Dr. J.B.T.M. Roerdink (RuG)
Prof.Dr. N. Petkov (RuG)

March 30, 1995

Rijksuniversiteit Groningen
Bibliotheca Informatica / Rekencentrum
Landelven 5
Postbus 800
9700 AV Groningen

Contents

List of abbreviations	iv
Foreword	v
Project description	vi
I Parallel Environment	1
1 The Cenju-3 Parallel Computer	3
1.1 Hardware Configuration	3
1.1.1 Cenju-3 Processing Elements	3
1.1.2 Interprocessor connection network	4
1.1.3 Host workstation	5
1.1.4 NLR configuration	6
1.2 Software Configuration	7
1.2.1 Message-passing programming	7
1.2.2 Data-parallel programming	11
1.2.3 Parallelisation tools	12
2 Cenju-3 Communication Bandwidth	15
2.1 Interconnection network bandwidth	15
2.2 Cenju-3/Workstation connection bandwidth	16
2.3 Interference between parallel programs	18
2.4 Conclusions	20
3 Programming system	21
3.1 Automatic parallelisation tools	21
3.2 Data-parallel programming	22
3.3 Message-passing programming	23
3.4 Why choose MPI?	25
II Parallel Visualiser	27
4 The NaS/RVS System	29
4.1 The flow-solver	31
4.2 The visualiser	31
4.2.1 Object renderer	32
4.2.2 Contour plot	33
4.2.3 Particle tracer	35

4.2.4	Volume rendering	36
5	Parallelisation of a visualisation tool	39
5.1	Design considerations	39
5.1.1	Performance metrics	39
5.1.2	Scalability	41
5.1.3	Parallel overhead	41
5.2	Data partitioning	41
5.2.1	Partitioning of object space	42
5.2.2	Partitioning of screen space	42
5.3	Parallel algorithms	43
5.3.1	Parallel main loop	43
5.3.2	Parallel visualiser	45
5.3.3	Parallel contour plot	48
5.4	Discussion of the design	51
5.4.1	Scalability	51
5.4.2	Parallel overhead	52
5.5	Performance	52
5.5.1	Measurement setup	52
5.5.2	Rendering times	53
5.5.3	Speedup	54
5.5.4	Load balance	57
6	Parallel Image Composition	59
6.1	Image composition as a reduction operation	59
6.2	Efficient parallel reduction	61
6.3	Performance	64
6.4	Implementation	66
III	Message Passing Interface	69
7	MPI Standard: An Overview	71
7.1	Point-to-point communication	72
7.1.1	Communication modes	72
7.1.2	Blocking and Non-blocking communications	73
7.1.3	Persistent communications	73
7.1.4	Derived data types	73
7.1.5	Pack and unpack	74
7.2	Collective Communication	74
7.2.1	Barrier synchronisation	74
7.2.2	Broadcast	75
7.2.3	Gather, scatter, allgather, and alltoall	75
7.2.4	Reduction operations	75
7.2.5	Reduce, allreduce, and reduce-scatter	75
7.2.6	Scans	76
7.2.7	User defined operations	76
7.3	Support for libraries	76
7.3.1	Communicators	77
7.3.2	Intra-communicators	78
7.3.3	Inter-communicators	78
7.4	Environmental Management	79

7.4.1	Implementation information	79
7.4.2	Error handling	79
7.4.3	Error codes and classes	80
7.4.4	Timers	80
7.4.5	Starting and terminating parallel programs	80
7.5	Profiling interface	80
8	Literature study: MPI	81
8.1	Introductory	81
8.2	Writing libraries	81
8.3	Extending MPI	82
8.4	Language binding	82
8.5	MPI implementations	83
8.5.1	MPICH	83
8.5.2	CHIMP	83
8.5.3	LAM	83
8.5.4	Unify	84
8.6	Applications	84
8.7	Profiling and debugging MPI programs	84
8.8	Performance of MPI	85
9	Porting full MPI to Cenju-3	87
9.1	Porting approach	87
9.2	The Abstract Device Interface	88
9.3	Example of operation	89
IV	Parallel Direct Volume Rendering	93
10	Parallel Direct Volume Rendering	95
10.1	Volume Rendering Model	95
10.2	Sequential Rendering Methods	98
10.2.1	Ray Casting	99
10.2.2	Splatting	100
10.2.3	Volume Shearing	102
10.3	Parallel Algorithms	102
10.3.1	Object Partition, Object-Order Rendering	102
10.3.2	Object Partition, Image-Order Rendering	103
10.3.3	Image Partition, Object-Order Rendering	104
10.3.4	Image Partition, Image-Order Rendering	104
10.4	Parallel Rendering on the Cenju-3	105
11	Conclusions	107
11.1	Parallel Environment	107
11.2	Parallel Visualiser	108
11.3	Message Passing Interface	109
11.4	Parallel Direct Volume Rendering	109
A	Full MPI on the Cenju-3	111
A.1	Resources on the internet	111
A.2	The Cenju-3 device	111
A.3	Installation on the Cenju-3	112
A.4	Using the MSU MPI implementation	113

List of abbreviations

ADAPTOR	Automatic Data Parallelism Translator
ANL	Argonne National Laboratory
API	Application Program Interface
CFD	Computational Fluid Dynamics
CHIMP	Common High-level Interface to Message Passing
CJP	Cenju process
CPU	Central Processing Unit
CT	Computed Tomography
DALIB	Distributed Array Library
DNS	Direct Navier-stokes Simulation
EPCC	Edinburgh Parallel Computing Centre
FEM	Finite Element Methods
FLOPS	Floating Point Operations per Second
GUI	Graphical User Interface
GRAVICS	GUI and Visualisation system for computational steering
HOS	Host Server
HPF	High Performance Fortran
HPFF	High Performance Fortran Forum
LAM	Local Area Multi-computer
MIPS	Million Instructions per Second
MPE	Multi-Processing Environment
MPI	Message Passing Interface
MPIF	Message Passing Interface Forum
MPXI	MPI eXtension library
MPMD	Multiple Program Multiple Data mode of parallel programming
MRI	Magnetic Resonance Imaging
MSU	Mississippi State University
NaS/RVS	Navier Stokes Real-time Visualisation System
NEC	Nippon Electric Company, Limited (founded 1899). Renamed NEC Corporation, effective April 1, 1983, both referred to as NEC
NLR	National Aerospace Laboratory NLR, The Netherlands
PARC	Polygon Assisted Ray Casting
PE	Processing Element; CPU in a multi-processor computer
PUL	Parallel Utilities Library
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
SPMD	Single Program Multiple Data mode of parallel programming
SU	Switching Unit
2D	2-dimensional
3D	3-dimensional

Foreword

This report is the result of my work during the last year of my studies in computer science at the 'Rijksuniversiteit Groningen', the Netherlands. The work was done at the 'National Aerospace Laboratory NLR the Netherlands' to whom I am very grateful for providing me with the resources necessary to perform this work.

During the last months of the year 1993 I started looking for an institution external to the university to do my final project. My goal was to do a project on parallelisation, if possible combined with some visualisation work. I applied for a position at the 'National Aerospace Laboratory (NLR) The Netherlands' and was successful. I started working at the NLR in the 'Noordoostpolder' on March 1st 1994. The project was concerned with the parallelisation of visualisation software; a combination of both my fields of interest. After an initial period of changing the setup of the software to fit the NLR environment I had to decide how to parallelise the 7500 lines of Fortran 77 code. The options were to use an automatic paralleliser, transform the fortran source into a data-parallel program, or use explicit message-passing. The requirement that the code should eventually be portable to the NEC Cenju-3 limited the possibilities. Due to the fact that the machine was not available until June 1994 and that without the machine automatic paralleliser output and data-parallel programs could not be tested, I decided to use the brand new message-passing standard MPI (Message Passing Interface). A big advantage of using MPI was that public domain implementations for clusters of workstations were available at that time so the parallelisation and testing of the visualiser could start right away. When the Cenju-3 became available I moved the parallel visualiser to the Cenju-3 and continued development. This report describes the development of part of the parallel visualiser and evaluates the first experiences with the Cenju-3 and MPI. A short introduction to MPI as well as literature studies of MPI and 'parallel direct volume rendering' are also contained in this report.

Project description

The project is concerned with the parallelisation of existing visualisation code on the Cenju-3 computer. There is a twofold purpose for this project.

1. Performance improvement of the visualisation code.
2. Investigating the effectiveness of the Cenju-3 architecture for visualisation applications.

The NLR does contract work for NEC in the area of Real-time Visualization for CFD flow-solvers. The project takes as a starting point the NEC software consisting of:

- A flow-solver, currently running on the NEC SX-3 vector-supercomputer. This flow-solver will be parallelised for the Cenju-3 and can also be replaced by NLR solvers.
- A user-interface (X/Motif), currently running on a NEC EWS4800 workstation. This user-interface will be changed intensively to integrate with the GRAVICS system for interaction with applications.
- Visualisation-code, currently running on the SX-3 and using an UltraNet frame buffer for presentation of graphics.

The visualisation-code will be reviewed to remove dependencies on a certain hardware infrastructure. This will result in a version that can be used in the NLR environment. Part of the low-level graphics code will probably move to a graphics workstation and it is the high-level visualisation-code that should be parallelised for the Cenju-3. The parallelisation should be performed in such a way that the code optimally uses the MPP architecture independent of the number of available processing elements. Essential part of the project is the evaluation of the Cenju-3 hardware and software for suitability for this kind of application. This might result in suggestions for improvement of hardware or software to NEC.

Introduction

This report is the result of my work at NLR during the last year of my studies in computer science at the 'Rijksuniversiteit Groningen'.

The report consists of four parts. Part one is concerned with the parallel environment of the NEC Cenju-3 parallel computer and its programming systems. Chapter one gives a description of the processing elements and the interconnection network of the Cenju-3, and an overview of how the Cenju-3 can be programmed. In chapter two, the communication bandwidth between the processing elements of the Cenju-3 as well as the bandwidth between the Cenju-3 and the host workstation are measured and evaluated. This chapter also analyses the effect on the bandwidth between the processing elements when multiple parallel programs are run on separate partitions of the Cenju-3. The final chapter of the first part, chapter three, motivates the choice of the programming system that has been used for the parallel implementation of the NaS/RVS visualiser.

The second part of the report is about this parallel implementation of the visualiser. Chapter four gives a description of the NaS/RVS system and introduces the tools of the visualiser. In chapter five, the design and implementation of the parallel visualiser and one of the tools is presented. Two approaches have been taken to parallelise a visualisation tool and performance results are included for both approaches. Finally, chapter six shows that parallel image composition, which is used in one of the parallelisation approaches, can be viewed as a reduction operation and implemented efficiently on a parallel computer.

The parallel visualiser uses the new message-passing standard MPI and the work done related to MPI is presented in part three. Chapter seven provides an overview of the MPI standard followed by chapter eight which is concerned with literature on MPI. The last chapter of this part shows how a public domain implementation of MPI was ported to the Cenju-3.

The last part of the report contains a literature study on 'Parallel Direct Volume Rendering'. Chapter ten studies sequential rendering algorithms and their parallel counterparts, and concludes with a discussion on parallel rendering using the Cenju-3.

The conclusion of the report are contained in the eleventh and final chapter.

Parallel Environment
(cont.)

Copyright © 1984

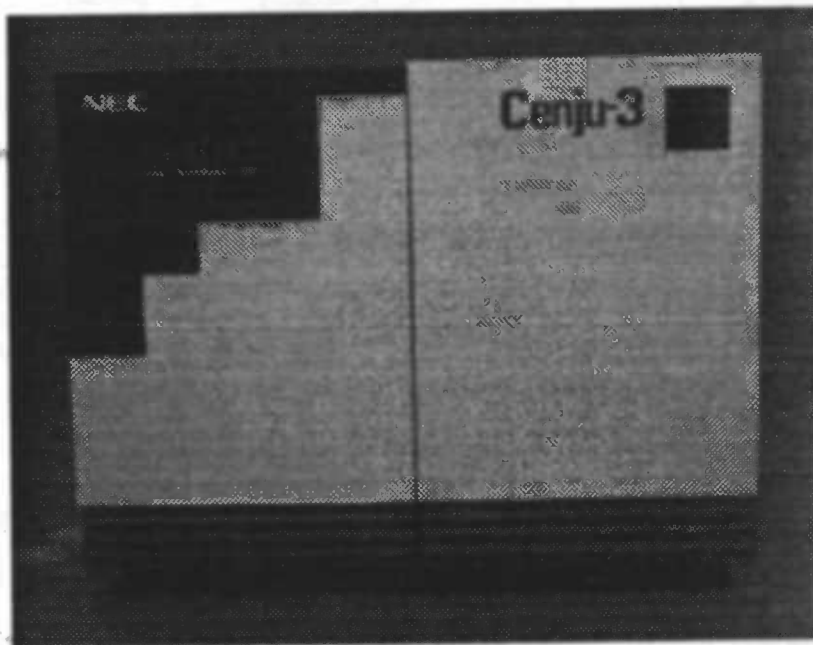
The Ceryu-3 Parallel Computer

Part I

Parallel Environment

Chapter 1

The Cenju-3 Parallel Computer



The Cenju-3 system is a new development from NEC. It is based on the Cenju and Cenju-2 parallel computers¹, developed by NEC's research and development group. This chapter will introduce the Cenju-3 hardware and software configuration.

1.1 Hardware Configuration

The Cenju-3 system is a distributed memory parallel computer system. Each processing element (node) has its own private memory and the nodes communicate by passing messages to each other. A workstation is used as the host computer for the Cenju-3 and provides services for compiling and loading programs on the parallel computer.

1.1.1 Cenju-3 Processing Elements

Each node consists of a RISC processor with a primary on-chip cache and a secondary off-chip cache, a memory controller for the local memory and interprocessor connection

¹Cenju and Cenju-2 are not commercially available.

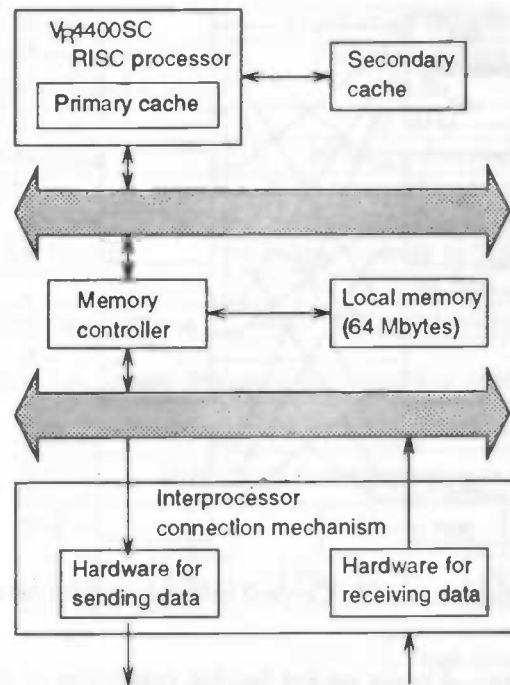


Figure 1.1: Block Diagram of a processing element

hardware for sending and receiving messages (Figures 1.1). It also has an on-chip floating point unit (FPU) which runs at a clock speed of 75 MHz. The FPU consists of three independent operation units: an adder, a multiplier, and a divider. The multiplier and divider can overlap with the adder. Some restrictions are imposed on multiplication and division, since they both use the adder in their last cycle. The adder starts the next operation one cycle before the current instruction has been completed. Therefore, execution can be started every three cycles, addition/subtraction requiring four cycles to be executed. On the other hand, the multiplier can start the next double-precision multiplication every four cycles, and the next single-precision multiplication every three cycles. (Single-precision multiplication requires seven cycles. Double-precision multiplication requires eight cycles.) So for single-precision operations 2 instruction, namely multiplication/division and addition, can be executed every 3 cycles. With a clock speed of 75MHz this leads to a theoretical single-precision peak performance of 50 MFlops. For double-precision operations 2 instruction can be executed every 4 cycles, leading to a theoretical peak performance of 37.5 MFlops.

The Cenju-3 is available in configurations ranging from 8 nodes to the largest model with 256 nodes.

1.1.2 Interprocessor connection network

The nodes can communicate with each other over a multi-stage switching network. Each node can be directly connected to every other node in this *fully interconnected* network. When a message is sent from one node to another there is no need to pass messages through other nodes on the way. The network uses packet switching. Every message has a destination and passes through the network by acquiring a connection path at each switching unit (SU) until it reaches its destination. A 4-input \times 4-output switching unit is used as a building block. The routing algorithm is static and determ-

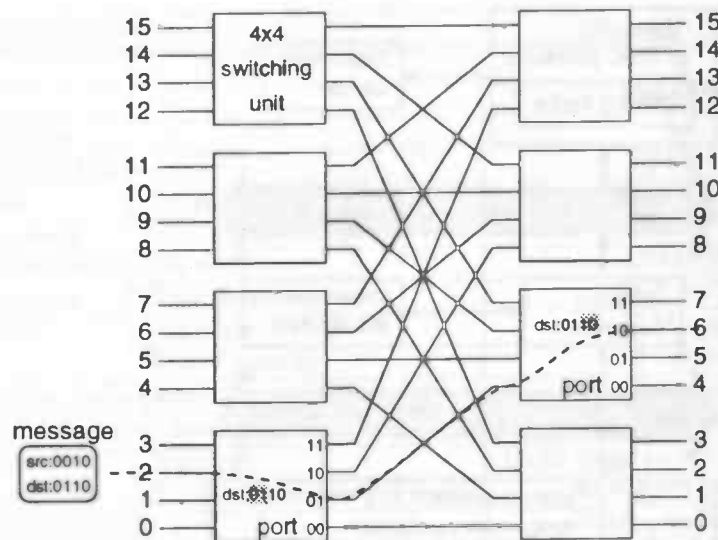


Figure 1.2: Routing algorithm used in Cenju-3 interconnection network.

inistic. Each packet on the Cenju-3 has a packet header, consisting of up to 10 bits. The upper two bits are used to specify the host id, the remaining 8 bits specify the destination. Since each SU is a 4×4 cross bar switch, the switch at each stage decodes the two bits corresponding to the stage and determines which of the 4 ports the message should be sent to. If the port is currently not being used, the packet is immediately forwarded. If the port is currently used, then the packet is not forwarded until the port is free. Figure 1.2 illustrates the routing algorithm. Assume we have a sixteen node machine, and 4 bits are used for the destination. A message is being sent from node 2 to node 6, so the destination address is 0110. At each switch, two bits of the destination are inspected, starting with the most significant bits, to determine the port on which to forward the message. In our example the message is forwarded on port 01 because the two most significant bits in the destination 0110 are 01. At the next switch, the next two bits are inspected and the message is forwarded on port 10 which is connected to node 6.

Besides the normal packet communication function the SUs have a packet copying function to perform effective one-to-many communication. Eight SUs are interconnected to form a single $16\text{-input} \times 16\text{-output}$ switch as shown in figure 1.2. A single network board contains such a 16×16 switch and can subsequently fully interconnect 16 nodes. The interprocessor distances are small and equal in length between all nodes when this type of network is used, e.g. messages are passing 2 SUs when 16 nodes are interconnected and 4 SUs when 64 nodes are used.

The specifications of the switching network are shown in table 1.1.

1.1.3 Host workstation

The EWS4800 Series workstation, which acts as the Cenju-3 host computer, is responsible for:

- Compilation and loading of parallel programs.
- File access from the nodes
- Network access

Topology	Multi-stage (baseline)
Switching method	Packet switching
Transfer word width	16 bits
Clock rate	20 MHz
Throughput	40 Mbyte/second
Delay	5 clk/stage (Single-cast) 6 clk/stage (Multi-cast)
Packet length	Variable from 18 to 518 bytes
SU	4 x 4 cross bars
Built-in function	Multi-cast, synchronization

Table 1.1: Specifications of the switching network

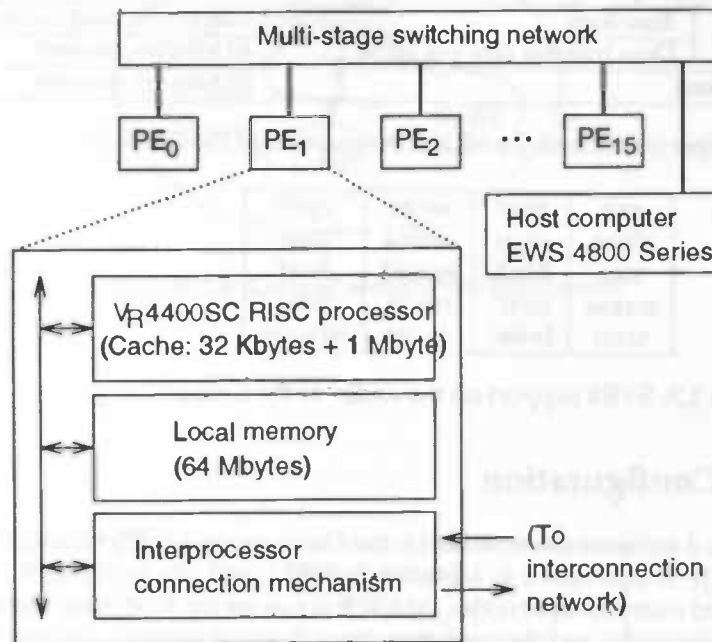


Figure 1.3: Configuration of Cenju-3 at NLR.

It is connected to the Cenju-3 by a 20 Mbytes/second link for data communication and a RS232 diagnosis link for detecting hardware errors in the switching network. The host workstation is used to compile and link parallel programs. The resulting executables can then be run on the Cenju-3.

1.1.4 NLR configuration

The Cenju-3 system installed at NLR is configured with 16 nodes. Since each node can perform 50 Mflops the theoretical peak performance of a 16 node system is 800 MFlops. Figure 1.3 shows the configuration of the Cenju-3 system at NLR. Table 1.2 shows the specifications of the system.

Cenju-3 NLR Configuration		
Model		16
CPU		VR4400SC
Number of nodes		16
CPU external clock rate		75 MHz
Floating point peak performance	Per node	50 MFLOPS
	Total performance	0.8 GFLOPS
Instruction execution peak performance	Per node	150 MIPS
	Total performance	2400 MIPS
Local memory capacity	Per node	64 MBytes
	Total Capacity	1 GByte
Primary on-chip cache capacity		32 Kbytes
Secondary of-chip cache capacity		1 Mbyte
Interprocessor connection network	Topology	Multi-stage switching network
	Data transfer rate per node	40 Mbytes/second
EWS Cenju-3 link speed		20 Mbytes/second

Table 1.2: Specification of the NLR configuration of the Cenju-3.

exit	read	write	open
close	creat	unlink	time
stat	lseek	getpid	fstat
access	ioctl ²	rmdir	lstat
xstat	lxstat	fxstat	rename

Table 1.3: SVR4 support on the nodes of the Cenju-3.

1.2 Software Configuration

The basis of the Cenju-3 software environment is the Cenju process (CJP) management system, MASER. A CJP is equivalent to a process in UNIX and can be thought of as one process distributed over multiple nodes. MASER is run on the host computer and manages the usage of the nodes and the operating states (loading, running, finishing) of the CJP's. A CJP consists of one or more Cenju workers (CJWs). Each worker executes its code on a single node and most likely communicates with other workers via the communication network. Each CJW is connected to the host server (HOS) which in turn is connected to the MASER. The HOS is used to process requests, such as file access requests, received from the CJWs (figure 1.4).

The CJWs are controlled by a kernel that implements a few SVR4 compatible system calls. They are listed in table 1.3. IO-requests, such as a file system access, are handled by the host workstation via the HOS. The relationship between software and hardware components is shown in figure 1.5.

1.2.1 Message-passing programming

There are several ways to program the Cenju-3. One can choose between using explicit message-passing, data-parallel programming or (interactive) automatic parallelisation tools.

When writing a parallel program using the message-passing paradigm, the programmer has to determine explicitly how to distribute the computations over the

²Partially supported. Only ioctl for terminal-IO is supported.

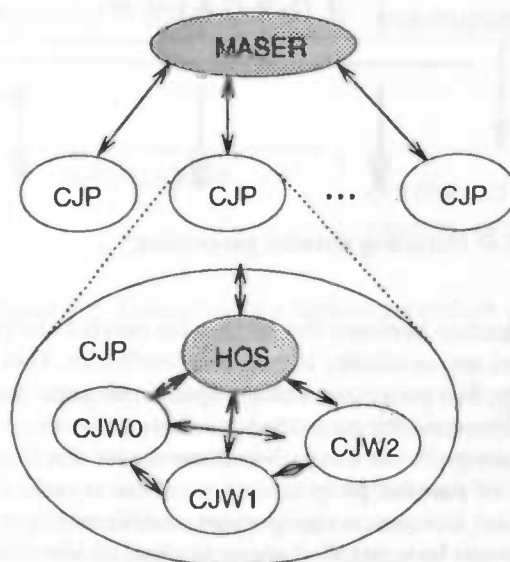


Figure 1.4: Cenju-3 software configuration.

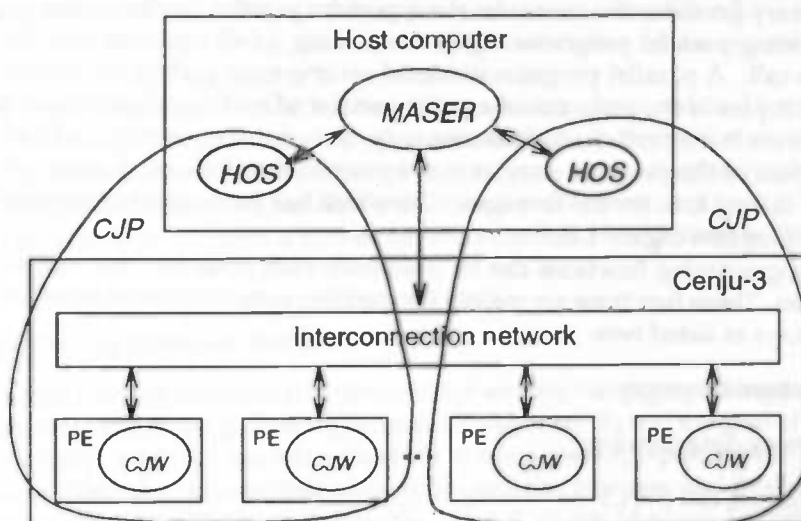


Figure 1.5: Relationship between Cenju-3 software and hardware components.

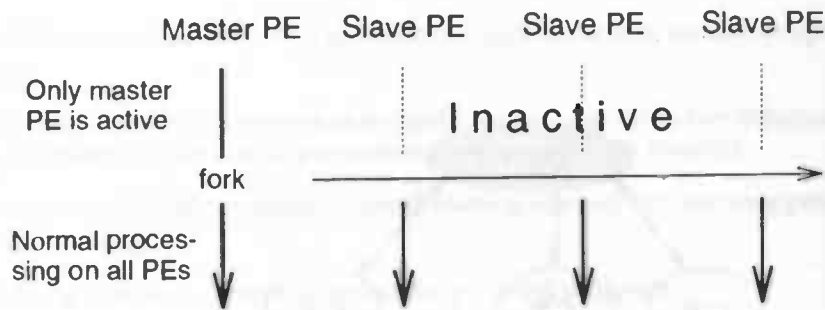


Figure 1.6: Initiating parallel processing.

available nodes. The communication between the nodes also needs to be programmed explicitly. It is the programmers responsibility to prevent deadlocks. This complicates message-passing programming, but programs using explicit message-passing can be optimised much better than automatically generated parallel programs.

Message-passing programming on the Cenju-3 is done under the Single Program Multiple Data (SPMD) model of parallel programming. There is only one program (executable) that runs on all nodes, but each node operates on different data. Differences in flow of control on each node can be controlled using the logical identification of the node. For example, when a parallel program uses 4 nodes these nodes are logically numbered 0 up to and including 3, and this identification number is available to the programmer. Each node can execute different statements based on this logical identification.

The Cenju-3 has two Application Program Interfaces (APIs) for message-passing. There is the low-level Paralib/CJ library and the Message Passing Interface (MPI).

Paralib/CJ

The Paralib/CJ library provides the lowest level support for parallel programming on the Cenju-3. Initiating parallel programming is done using a call equivalent to the UNIX fork system call. A parallel program is started on one node (called the master node) which forks copies of the program on a given number of nodes (called the slave nodes). The master node may call the fork routine only once, it is not permitted to fork more copies of the program once, it is not permitted to fork more copies of the program after calling fork for the first time. Once fork has been called each node executes the same program (figure 1.6).

The parallel programming functions can be used once each node has received its copy of the program. These functions are mainly for enabling communication between nodes in several ways as listed here.

- Distributed shared memory
- Remote memory data copying
- Remote procedure call (RPC)
- Barrier synchronization

Distributed shared memory can be described as follows. A node can map a memory area of a remote node into its own address space. Whenever a node accesses a mapped memory location in the local memory area the operating system automatically performs communication for reading or writing the remote memory location. The programmer

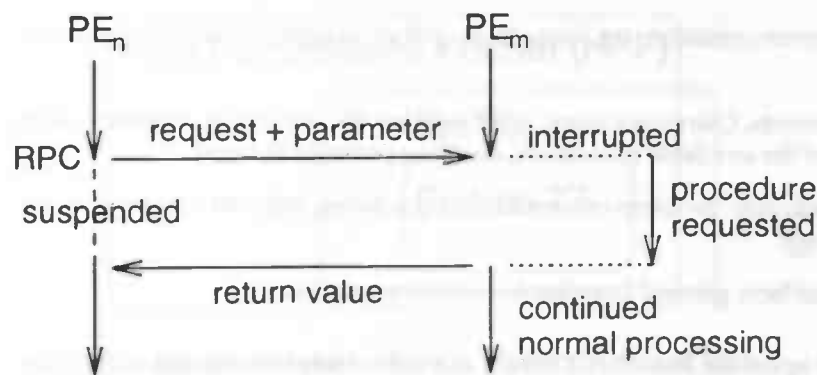


Figure 1.7: Execution of a remote procedure call.

should prevent the following situations: Reading or writing the mapped memory when the remote node or some other node is writing to the mapped memory, and writing the mapped memory when the remote node or some other node is reading from the mapped memory. If these situations are avoided data consistency is guaranteed. Note that communication is performed for every single access into the mapped memory area. This may lead to performance problems when distributed shared memory is accessed frequently.

Remote memory data copying is used to transfer the contents of memory between nodes. Data can be transferred to a remote node (write) or from a remote node (read). No cooperation is needed from the remote node for the transfer itself but again the programmer should ensure data consistency.

A *remote procedure call (RPC)* is used to execute a procedure (routine) on a remote node. A program requests a RPC, the remote node interrupts its normal processing to execute the procedure and continues normal processing after the call has finished and the result is sent to the node that issued the RPC (figure 1.7). If two node's simultaneously request a RPC to a third node, the execution of the RPC for the first node that is served will not be interrupted by the execution of the second RPC. This facility enables critical sections to be controlled easily.

Barrier synchronization is used to ensure that all nodes have reached a certain point in their processing. It can be used to guarantee data consistency. For example before copying an area of a remote node's memory, a node can issue a barrier synchronization to ensure that the remote node has finished writing into that memory area. It is assumed that the remote node requests a barrier synchronization after writing to that memory area.

Message Passing Interface (MPI)

The Message Passing Interface is a standard for writing message-passing programs. It was defined by the Message Passing Interface Forum (MPIF) and the goal of this forum was to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message-passing. The final report, Version 1.0, of the Message Passing Interface Forum [22] has been the basis for various vendor and public domain implementations of the standard. The MPI standard is extensively discussed in chapter 7. Some features of the standard are.

- Point-to-point communication: e.g. (non-)blocking point-to-point communications, packing, buffering, complex data types.

- **Collective communication:** e.g. broadcast, gather/scatter, reduction operations, scans.
- **Groups, Contexts, Communicators, and Caching:** e.g. collective communication in a subset of the available processors, enabling portable libraries.
- **Process Topologies:** problem oriented naming scheme, efficient mapping to machine topology.
- **Profiling Interface:** general interface for profiling purposes.

Mini-MPI Built upon the Paralib/CJ library is a subset implementation of the Message Passing Interface (MPI) called the mini-MPI library. Some of the features of MPI that have been implemented in the mini-MPI library are:

- Non-blocking point-to-point communications
- Broadcasting
- Gather/Scatter operations
- Reduction operations

A summary of the mini-MPI implementation can be found in [47].

Full-MPI The lack of an MPI library that implements the full standard has been the motivation for porting a public domain implementation of MPI to the Cenju-3. The implementation that resulted from a joint-effort project between Argonne National Laboratory and Mississippi State University has been ported to the Cenju-3. A description of the port can be found in chapter 9. We will refer to this implementation of MPI as the ANL/MSU implementation.

This MPI library has the full standard MPI functionality. Users can thus write standard MPI programs and these programs will be portable to a variety of parallel computers or clusters of workstations.

1.2.2 Data-parallel programming

The Cenju-3 can also be programmed using the data-parallel programming paradigm. This is implemented in the High Performance Fortran (HPF) pre-compiler. HPF is a standard defined by the High Performance Fortran Forum (HPFF). It is based on Fortran 90 which in turn is based on Fortran 77. The relationship is shown in figure 1.8. The standard is defined in the High Performance Fortran Language Specification [21]. The goal of the HPF Forum was to define language extensions for Fortran supporting:

- Data parallel programming.
- Top performance on MIMD and SIMD computers with non-uniform memory access costs.
- Code tuning for various architectures.

HPF is based on Fortran 90 but due to the incompatibility of some features of the Fortran 90 language with the distribution of data in HPF restrictions were necessary on the Fortran 90 standard.

The HPF Forum also defined a subset HPF ([21] Section 8) to provide a portable interim HPF capability. The HPF pre-compiler on the Cenju-3 is an implementation of

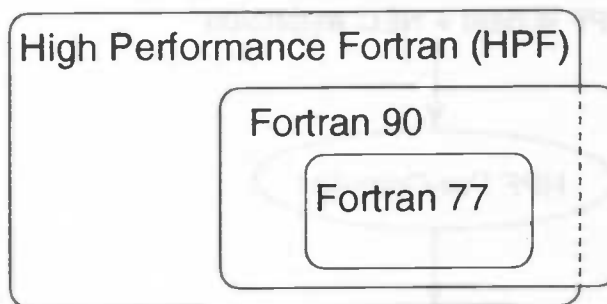


Figure 1.8: Relationship of Fortran 77, Fortran 90 and HPF.

Fortran 90	HPF Subset
DO WHILE statement	multi-dimensional distribution and processor array declaration
arithmetic and logical array features	cyclic distribution
intrinsic procedures	all HPF intrinsic functions
declarations	
optional procedure arguments	
keyword argument passing	

Table 1.4: Unsupported features of the NEC subset HPF implementation.

this subset and its configuration is shown in figure 1.9. Unsupported features of the NEC subset HPF implementation are listed in table 1.4. Features supported outside the subset HPF specification are listed in 1.5.

1.2.3 Parallelisation tools

The last method of programming the Cenju-3 is to use PCASE, an automatic interactive parallelisation tool. PCASE allows users to interactively parallelise their sequential Fortran 77 programs with minimal effort. The PCASE programming environment has the following functionality:

- Automatic data transfer insertion to transfer data between shared/distributed memory and local memory.
- Dependency analysis of do-loops which tests loop optimisation possibilities such as vectorisation, parallelisation, and loop restructuring.
- Interactive parallelisation to utilize the programmer's high level knowledge of the application. Users can change the decisions made by the parallelisation software in order to improve performance.

- REALIGN, REDISTRIBUTE, and DYNAMIC directives
- INHERIT directive used with a dist-format-clause or dist-target
- EXTRINSIC function attribute
- NEC directives

Table 1.5: Features supported by the NEC HPF pre-compiler that are outside the subset HPF specification.

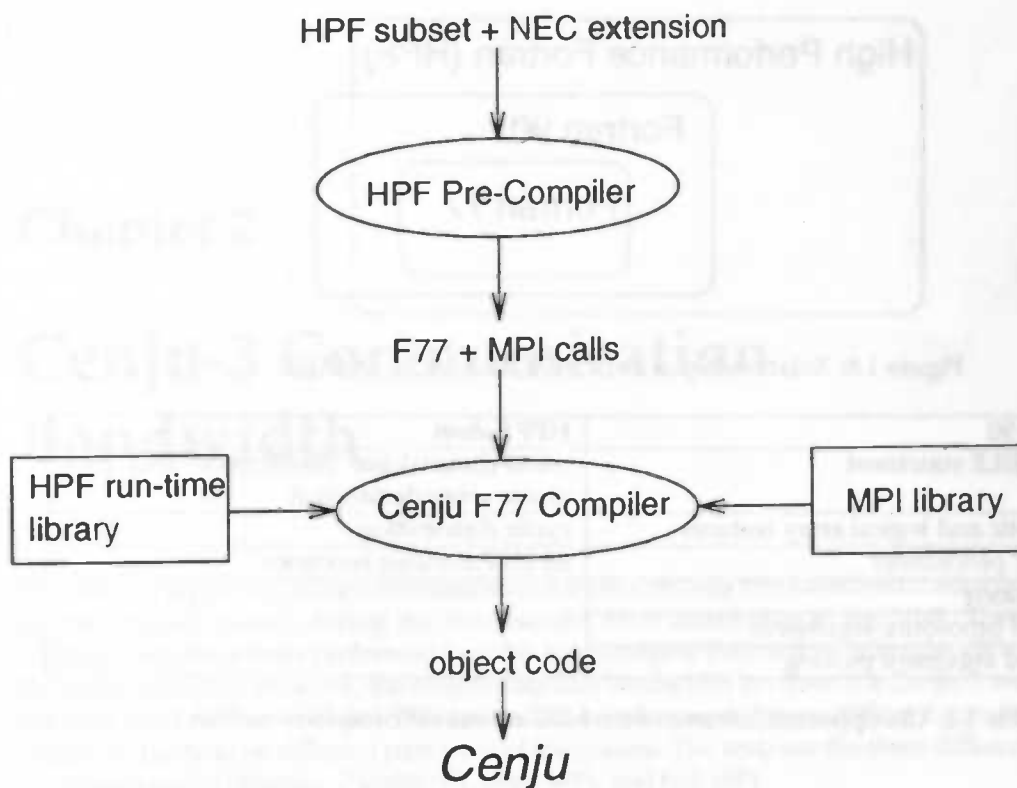


Figure 1.9: HPF Compiler configuration on Cenju-3.

- Performance prediction. The effect of changes in the parallelisation process, made by the user, on the performance of the application can be predicted.

Further information regarding PCASE can be found in [35].

	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030										
Population	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150

Chapter 3

Chapter 3: Communication

The purpose of this chapter is to provide a comprehensive overview of the communication process, from the sender to the receiver, and the various factors that can influence the effectiveness of communication. This chapter will explore the different types of communication, the barriers to communication, and the strategies for improving communication. The chapter will also discuss the importance of communication in the workplace and in society.

1.1 Introduction to Communication

Communication is the process of exchanging information between two or more people. It is a fundamental part of human life and is essential for the functioning of society. Communication can take many forms, including verbal, written, and non-verbal. The process of communication involves several steps, including encoding, transmission, and decoding. The effectiveness of communication depends on many factors, including the clarity of the message, the channel of communication, and the relationship between the sender and the receiver.

There are many barriers to communication, including physical barriers, psychological barriers, and cultural barriers. These barriers can prevent the message from being received or understood correctly. It is important to be aware of these barriers and to take steps to overcome them.

Effective communication is essential for success in the workplace and in society. It is important to use clear and concise language, to listen actively, and to be aware of the needs and expectations of the other person.

Communication is a complex process that involves many factors. It is important to understand the process of communication and to be aware of the barriers to communication. By using effective communication strategies, we can improve our communication skills and achieve our goals.

This chapter will provide a comprehensive overview of the communication process, from the sender to the receiver, and the various factors that can influence the effectiveness of communication. This chapter will explore the different types of communication, the barriers to communication, and the strategies for improving communication. The chapter will also discuss the importance of communication in the workplace and in society.

Chapter 2

Cenju-3 Communication Bandwidth

This chapter reports on some communication bandwidth tests that have been conducted on the Cenju-3 system during the first months after installation at the NLR. Three different tests have been performed in order to investigate the maximum bandwidth of the interconnection network, the communication bandwidth between the Cenju-3 and the host workstation, and the communication interference between different parallel programs running on different partitions of the system. The tests use the three different message-passing libraries: Paralib/CJ, mini-MPI, and full-MPI.

2.1 Interconnection network bandwidth

To measure the maximum bandwidth of the interconnection network two tests have been conducted for each of the three available message-passing libraries. The amount of data sent in the tests is 8 MBytes since this has been shown to be the smallest amount of data resulting in the maximum measured bandwidth. Using larger amounts of data does not improve the bandwidth calculated in these tests. The first test sends data from one processor to another and calculates the bandwidth from the obtained timing information. The second test broadcasts data to respectively 2, 4, 8 and 16 processors. Again the measured time is used to calculate the bandwidth. Two test sessions have been performed, one for the initial 1.3c release of the Cenju-3 software environment (table 2.1) and one for the next release (1.4c) of the Cenju-3 environment (table 2.2).

Note that in table 2.1 no timing for the broadcast using the mini-MPI library is listed. This is because the broadcast routine in mini-MPI erroneously does not handle buffers with a size in excess of 256 kBytes.

	Paralib/CJ	mini-MPI	full-MPI
pt2pt	14.2	8.17	8.48
bcast 2	14.2	—	8.47
bcast 4	4.72	—	4.08
bcast 8	2.03	—	2.72
bcast 16	0.95	—	2.04

Table 2.1: Bandwidth in MBytes/sec of interconnection network for point-to-point and broadcast communication for 1.3c release of Cenju-3 environment.

	Paralib/CJ	mini-MPI	full-MPI
pt2pt	13.9	8.18	—
bcast 2	12.4	10.8	—
bcast 4	4.22	4.99	—
bcast 8	1.84	3.45	—
bcast 16	0.88	3.37	—

Table 2.2: Bandwidth in MBytes/sec of interconnection network for 1.4c release of Cenju-3 environment.

Note also that in table 2.2 no figures are listed for the full-MPI library. This is because the full-MPI implementation did not work with the 1.4c release of the Cenju-3 environment at the time of the tests. In this table there are figures for the mini-MPI library, but they were obtained by repeating the broadcast communication 32 times for a buffer size of 256 kBytes and the resulting bandwidth will thus be smaller than it would be for one broadcast of 8 MBytes.

Point-to-point communication For point-to-point communication the bandwidth of the low-level Paralib/CJ library is significantly larger than the bandwidth for the mini-MPI library that is built on top of Paralib/CJ. This is because for these tests no synchronisation was needed for the Paralib/CJ program where the mini-MPI program suffers synchronisation protocol overhead. The full-MPI implementation however outperforms the mini-MPI library for point-to-point communication. No cause has been found for this small difference.

Broadcast communication For broadcast communication using Paralib/CJ a special routine (`CJrmwrite()`) was used that is supposed to use the packet copying functionality of the interconnection network to perform broadcasting. One would expect this to outperform a software solution for broadcast in which data is communicated explicitly between processors. The results however show that the bandwidth for Paralib/CJ broadcast is significantly smaller than the bandwidth for mini-MPI and full-MPI. No explanation was found for this disappointing performance since the source code of the Paralib/CJ library is not available, but it is expected that this behaviour is due to a software error in the Paralib/CJ library.

2.2 Cenju-3/Workstation connection bandwidth

Since the primary goal of the work presented in this report is to apply the Cenju-3 to visualisation tasks, one is interested in the bandwidth between the parallel machine and the host workstation, since all graphical output will use this connection. This bandwidth determines the rate at which the framebuffer can be updated and should thus be large enough. In order to measure the bandwidth of the connection between the Cenju-3 processing elements and the host workstation, a test was conducted in which different amounts of data were read from and written to the host workstation. To eliminate diskcaching artifacts the test program reads data from `/dev/zero`¹, and writes data to `/dev/null`². In this way, the I/O is constrained to memory-accesses only and the calculated bandwidth will only be limited by the speed of the connection between the Cenju-3 and the host workstation. All tests have been conducted on

¹A file that when read produces an unlimited amount of zero bytes.

²A file that discards all bytes that are written to it.

kBytes sent	read kBytes/sec	write kBytes/sec
64	133.6	463.8
128	177.8	533.3
256	178.0	572.7
512	183.2	610.3
1024	185.2	621.4
2048	186.9	632.5
4096	187.5	636.3
8192	186.6	637.1

Table 2.3: Bandwidth in kBytes/sec for file read and file write between Cenju-3 and host workstation for the 1.3c environment.

kBytes sent	read kBytes/sec	write kBytes/sec
64	755.2	969.7
128	782.3	979.0
256	782.0	987.5
512	894.0	999.3
1024	832.8	1006.6
2048	829.8	983.9
4096	813.1	994.4
8192	837.3	947.3

Table 2.4: Bandwidth in kBytes/sec for file read and file write between Cenju-3 and host workstation for 1.4c environment.

a system with only one user running programs. Tests have also been conducted to measure the bandwidth while more than one parallel program performed I/O via the host workstation.

The documentation of the Cenju-3 does not specify the bandwidth of the connection between the Cenju-3 and the host workstation. However, NEC technicians stated that this bandwidth could be 40 MBytes/sec, but that it is limited to 20 MBytes/sec due to lower clockspeed of the hardware at the host workstation. They expect that in practice the bandwidth degrades further to approximately 14 MBytes/sec due to software overhead.

Table 2.3 shows the bandwidth measured for the 1.3c release of the Cenju-3 environment. It shows that the maximum bandwidth for both read and write is achieved for transfers larger than approximately 2 MBytes. The bandwidth for file read is very disappointing and is shown to be caused by a software error that was fixed in release 1.4c of the Cenju-3 environment.

Table 2.4 shows the bandwidth measured when using the 1.4c release of the Cenju-3 environment. A large improvement can be seen for the file read, with a maximum bandwidth of 894 kBytes/sec. The bandwidth of the file write is also improved and reaches a maximum of 1006 kBytes/sec which is almost one Mbyte/sec but this figure is nowhere near the expected 14 MBytes/sec and improvement in this area is necessary. The difference between file read and file write bandwidth is now of normal proportions.

In table 2.5 the bandwidth is listed when more than one parallel program performs I/O. The amount of data transferred in this test is 4 MBytes. The bandwidth listed is the average of the measured bandwidths for all programs performing I/O. It is obvious from the table that the bandwidth decreases linearly with the number of programs performing I/O.

No. parallel progs.	read kBytes/sec	write kBytes/sec
1	775.5	931.3
2	427.1	631.1
4	236.1	324.1
8	111.5	135.2
16	59.5	76.6

Table 2.5: Bandwidth in kBytes/sec for file read and file write when more than one parallel program is performing file I/O. Results are for the 1.4c release of the Cenju-3 environment.

2.3 Interference between parallel programs

The Cenju-3 allows multiple users to run parallel programs on non-overlapping partitions of the parallel system. It is interesting to know if the communication of one parallel program affects the bandwidth for communication in another parallel program. If two different parallel programs use the same switch in the interconnection network it is apparent that these two programs will interfere with each other and the bandwidth for communications within the programs will degrade. On the other hand if the programs do not use the same switches, they should not interfere with each other's communication and the bandwidth for communication within the programs should not be affected.

Figure 2.1 shows two parallel programs that interfere with each others communication. Program 1 uses processing elements 0 and 1, and program 2 uses processing elements 2,3,4 and 5. In this example, program one possibly uses the bottom two switches for communication, and program two possibly uses all four marked switches. At any point in their execution, these two programs may want to use any of the two switches that are necessary for communication in both programs, leading to interference.

Figure 2.2 shows two parallel programs that do not interfere with each other because they do not use the same switches. Program 1 uses the bottom two switches while program 2 uses the upper two switches, so they will not use the same switch at any point in their execution.

To measure the interference of parallel programs running on non-overlapping partitions, a test was performed in which the programs used to measure the broadcast bandwidth of the interconnection network were loaded multiple times on non-overlapping partitions in such a way that they were running (and communicating) at the same time. Again the bandwidth was measured and compared to the bandwidth for the tests where only one program was running. The tests were performed for both the Paralib/CJ and the mini-MPI library and the results are shown in table 2.6. The first two entries in the table are for one and two programs broadcasting data over eight nodes each. If both programs would use the same switches it would be expected that the bandwidth be half the bandwidth for one program. However the bandwidth degrades but is not halved. This shows that there is some interference, but it is not due to the fact that both programs use the same switches because in that case the bandwidth is expected to be half the bandwidth for a single program. 2.2. However, the bandwidth for the last one of the next three entries (4x4 procs) does not follow our expectations. If each of the four programs uses nodes connected to a single switch, then the bandwidth for 4x4 procs should be similar to the results for 1 and 2 times 4 procs. The results for programs running on two nodes are also unexpected. In the 2x2 procs configuration we expect the programs to use the same switches and therefore the bandwidth should be half the bandwidth for 1x2 procs. This however is not the case, the bandwidth is almost the

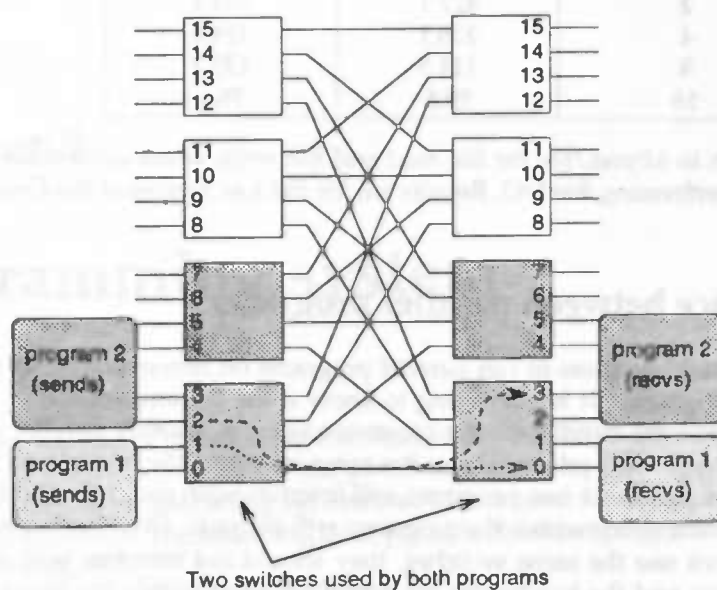


Figure 2.1: The communication of two parallel programs that use the same switch will interfere.

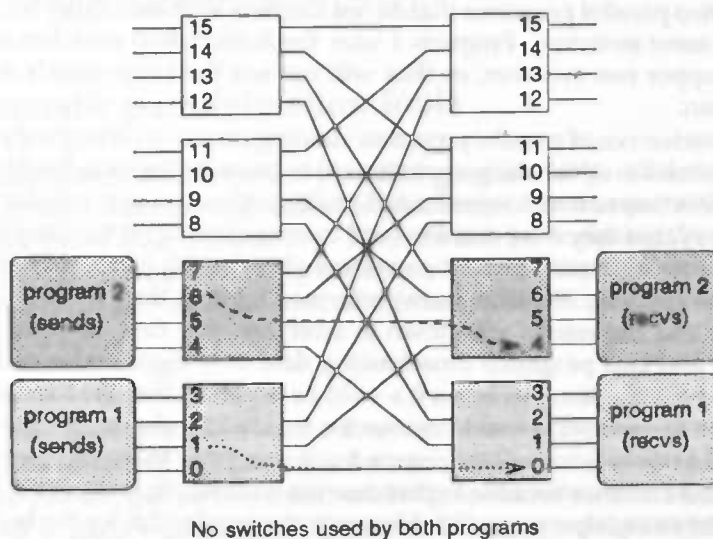


Figure 2.2: The communication of two parallel programs that do not use the same switch should not interfere.

	Paralib/CJ	mini-MPI
1x8 procs	2.02	2.44
2x8 procs	1.92	1.59
1x4 procs	4.72	4.94
2x4 procs	4.48	4.06
4x4 procs	2.90	2.70
1x2 procs	14.2	10.7
2x2 procs	13.4	10.6
4x2 procs	8.95	7.58
8x2 procs	4.51	6.82

Table 2.6: Bandwidth in MBytes/sec for broadcast communication for various partitions running the same program.

same. For 4x2 procs the bandwidth drops to 8.95 which is roughly what we would also expect for 2x2 procs. The 8x2 procs should give a similar bandwidth, because at any point in time only two parallel programs will use the same switch.

In the above the assumption is made that the system program that loads the parallel programs onto the machine allocates the nodes in sequence, starting from node zero up to node sixteen. The output of the program that lists the state of the processors confirms this assumption. If the allocation would be random we can nothing about which program uses which switch, but in general the switches will be used by more parallel programs.

2.4 Conclusions

In this chapter we have investigated the bandwidth provided to the user on the Cenju-3. It shows that the Paralib/CJ library provided the largest bandwidth which is to be expected since full-MPI is implemented ontop of mini-MPI, which is built ontop of Paralib/CJ. However, for some reason, that we have not been able to reveal, in some cases the full-MPI implementation performs better than the mini-MPI library on which it is implemented. The Paralib/CJ library does have a problem with broadcasting data using the function that was specifically designed for this purpose. At this moment it is better to use explicit communication between nodes when broadcasting is needed than to use the Paralib/CJ routine.

The bandwidth between the Cenju-3 and the host workstation is disappointingly small compared to what should be theoretically possible. Even using an improved release of the software environment the maximum transfer rate between the Cenju-3 and the host is less than 1 MByte/sec. This is a major disadvantage for I/O bound applications such as visualisation tasks.

Multiple parallel programs degrade the bandwidth of the interconnection network even when they do not use the same switches. No explanation can be given for this unless more detailed information about the hardware and software of the interconnection network is available.

The bandwidth provided on the interconnection network is relatively high. This makes the Cenju-3 suitable for programs written in a data-parallel language although explicit message-passing programs will most likely perform better.

Chapter 3

Programming system

The project description of parallelising the visualiser of NaS/RVS does not specify the programming system to be used for the parallelisation task. This chapter discusses the choice of parallel programming system that will be used for the parallelisation of the NaS/RVS visualiser. The following three possible classes of programming systems have been investigated:

- Automatic parallelisation tools.
- Data-parallel programming systems.
- Message-passing programming systems.

Several programming systems from these three classes will be introduced. They will be evaluated for the task of parallelising NaS/RVS visualiser.

3.1 Automatic parallelisation tools

Automatic parallelisation tools generate parallel programs or parallel object code, for a particular parallel machine and software platform, from a sequential program in some language (most often Fortran 77). They do this by analysing dependencies in loops and where possible parallelising the loops. Parallelising compilers fall into this category. On the Cenju-3 an automatic paralleliser called PCASE is available which will be discussed next.

PCASE PCASE is an interactive source-to-source translator that automatically parallelises Fortran 77 programs and generates code for the Cenju-3 (section 1.2.3, [35]). The parallelisation process can be controlled with a graphical user interface (GUI) which allows the user to make explicit parallelisation decisions in order to improve performance. In addition to this PCASE enables the user to investigate the consequences of changes in the parallelisation by predicting the performance of the application. When a user makes an explicit parallelisation decision he can evaluate the effect of that decision by predicting the performance and comparing this to the situation before the change. The output of PCASE is either a Fortran 77 program with mini-MPI message-passing calls or a Fortran 77 program with parallelisation directives for a parallelising compiler.

Evaluation The PCASE parallelisation tool is easy to use and it has nice features. It enables the user to create an initial parallel version of his Fortran 77 program. Once this version runs fine on the parallel machine the programmer can use his high-level knowledge of the application to improve the performance. All this is done in an interactive manner. The user can for example select a loop and remove dependencies that were found by PCASE and remove them because they are not necessary, so PCASE can parallelise the loop. PCASE performs interprocedural analyses which may result in a better initial parallelisation thus reducing the need for extensive optimisation. Source code can be generated for both HPF and Fortran 77 with MPI message-passing calls.

3.2 Data-parallel programming

Data-parallel programming is a paradigm in which there is one global name space (as with 'normal' sequential programs) but the data is distributed over a number of processors. The compiler generates code for communication in case data is needed that is located on a different processor. Compiler directives can be used to control the distribution of data in order to minimize the necessary communication for a given computation. No explicit communication code is written by the programmer; it is the compiler that is responsible for correct deadlock-free communication.

High Performance Fortran (HPF) HPF is the new Fortran standard which can be used for writing data-parallel programs [21]. It is designed as a set of extensions and modifications to Fortran 90 to support data-parallel programming, top performance on MIMD and SIMD computers with non-uniform memory access costs, and code tuning for various architectures. An important goal of HPF is to achieve portability across a variety of parallel machines. This requires not only that HPF programs compile on all target machines, but also that an efficient HPF program on one parallel machine be able to achieve reasonably high efficiency on another parallel machine with a comparable number of processors, otherwise the effort in optimising the program on one machine would be wasted when the HPF code is ported to another machine. HPF has various features to express both the parallelism as well as the communication inherent in a computation. In this way the programmer has control over the efficiency of the parallel application.

Evaluation HPF is a data-parallel language which would require the transformation of a Fortran 77 program to data-parallel form. Most Fortran 77 programs use large arrays extensively which suits data-parallel programming well. This is also the case for NaS/RVS visualiser thus a data-parallel implementation could be implemented naturally. The performance of such an implementation is dependent on the underlying HPF implementation; in our case the HPF pre-compiler delivered with the Cenju-3.

ADAPTOR Adaptor (Automatic DATA Parallelism TranslatOR) [3] is a tool for transforming data-parallel programs written in Fortran with array extensions, parallel loops, and layout directives to parallel programs with explicit message-passing. The input language is a subset of CM Fortran and HPF although not all features of these languages are supported. The generated message-passing programs will run on different multiprocessor systems with distributed memory, but also on shared or virtual shared memory architectures. The Adaptor tool is a source-to-source translator and uses a basic set of routines for message passing and operations on sequential and distributed

arrays. To parallelise NaS/RVS using the Adaptor system the Fortran 77 program would have to be converted to data-parallel form in either subset CM Fortran or subset HPF or a mixture of these two.

Evaluation ADAPTOR can be used in both batch and interactive mode. In batch mode it acts as a source-to-source compiler generating Fortran 77 programs with message-passing calls without intervention of the user. In interactive mode the result of the transformation, such as the chosen distribution for a specific array, can be investigated but the transformation can not be influenced in the way one can influence the parallelisation process in PCASE.

3.3 Message-passing programming

Message-passing is the parallel programming paradigm in which communication necessary in the parallel program is explicitly programmed. It is a much more complex task to write a message-passing program than it is to write a data-parallel program, but message-passing programs can yield better performance. Using message-passing programming the programmer has much lower level control over the communications performed in a parallel program than with data-parallel programming.

Parallel Virtual Machine (PVM) PVM is one of the most well known message-passing systems. It has become the defacto standard for message-passing and many parallel computer vendors support the PVM application interface for message-passing. PVM is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. Using PVM one can write Multiple Program Multiple Data (MPMD, a super-set of SPMD in which each node can run a different program) parallel programs that communicate with each other by passing messages. PVM provides the following functionality.

- Process Control: functions to start and stop parallel tasks.
- Information on the possibly changing hosts configuration.
- Dynamic configuration: adding and deleting hosts from the parallel virtual machine.
- Dynamic process groups: leaving and joining groups and getting information on process groups.
- Message buffers: creating and destroying message buffers, retrieving information on message buffers.
- Signalling processes in the parallel virtual machine.
- Sending, receiving, packing and unpacking of messages.
- Error handling: error messages for the various functions.

Evaluation PVM is a much used message-passing library which is stable and robust. It is available on a variety of platforms and can be used in a heterogeneous environment. There are various tools that can be used with PVM such as tools for monitoring the state of processes and tools for performance analysis of applications. One of the nice features of PVM is that it allows dynamic addition of hosts and processes during the execution of a parallel program which can be very useful in some applications.

Paralib/CJ The Paralib/CJ library is the native parallel programming library for the Cenju-3. It features remote memory copying, remote procedure calls, and distributed shared memory as well as various utility routines to support parallel programming. The message-passing paradigm can be implemented in Paralib/CJ by using remote memory copies and synchronisation to ensure the consistency of data. It has been discussed in the section on the Cenju-3 software configuration (section 1.2.1).

Evaluation The low-level control one has over the Cenju-3 with the Paralib/CJ library enables the development of high-performance applications. But as it is with assembler programming, the interface poses problems when writing large applications. On the other hand the proposed method of using synchronisation to implement message-passing with Paralib/CJ leads to inefficient applications because the synchronisation is machine-wide which possibly involves processors that are not participating in the communication. One could implement more elaborate mechanisms for message-passing but this is thought to be the responsibility of the parallel computer vendor. Portability of Paralib/CJ programs is also a problem since this library is only supported on the Cenju-3.

Message Passing Interface (MPI) MPI is the emerging new standard for writing portable message passing programs. One of the main features of MPI is that one can write portable parallel libraries without the need for synchronisation on library entry and exit. The concept behind this ability is the notion of disjunct communication universes. The communication in one universe does not interfere with the communication in another. Another feature of MPI is the rich ensemble of collective communication functions. Below is a short summary of MPI features:

- Point-to-point communication.
- Complex data type support.
- Collective communication.
- Virtual topologies.
- Support for libraries.
- Environmental management.
- Profiling interface.

MPI has been discussed in section 1.2.1 and an overview is given in appendix 7.

Evaluation MPI is an intuitive interface to message-passing. The concepts of intra- and inter-communicators take time to fully understand but once the concept is clear it serves as a consistent basis on which the rest of the interface builds. When one does not need process groups one can easily use MPI without knowing about communicators. The virtual topology functions enable a programmer to express the necessary communication in terms of the structure of the problem thus making the resulting program much more readable. The library support functions of MPI can lead to safe, efficient and portable parallel libraries which is an important feature to simplify the use of parallel computers in general. Once programming systems based on MPI become available applications will be naturally portable to a variety of parallel architectures.

3.4 Why choose MPI ?

Based on the information in the previous sections we will now discuss why we chose MPI as the programming system to be used for the parallelisation of the NaS/RVS visualiser. Unfortunately the freedom of choice is mainly constrained by the fact that the Cenju-3 was not available during the first four months of the project. The choice for MPI is based on the following arguments.

Due to the unavailability of the Cenju-3 system during the first four months of the project all tools that generate code for the Cenju-3 exclusively or use libraries for the Cenju-3 could not be used since the resulting code could not be tested in any way. This means that HPF, PCASE and the Paralib/CJ library could not be used to parallelise NaS/RVS visualiser.

The ADAPTOR tool will not be used because the input language is a mixture of subset HPF and subset CM Fortran so the portability of the resulting program will become a problem. Would we have chosen to use ADAPTOR, only the HPF subset should have been used to be able to port the resulting code to the Cenju. We were unable to compare the ADAPTOR subset HPF to the Cenju subset HPF due to the unavailability of the specifications of the Cenju HPF pre-compiler and have thus chosen not to use ADAPTOR.

PVM has not been chosen to parallelise NaS/RVS visualiser although it was the defacto standard for writing message-passing programs at the beginning of the project. PVM runs on a variety of workstation clusters which would have made development possible as long as the Cenju-3 was unavailable, but it was known at the time that PVM would not be supported on the Cenju-3. If NaS/RVS were written in PVM, a conversion from PVM to a message-passing system that is supported by the Cenju-3 would be necessary. Due to the time constraints of the project this conversion should be avoided.

Then why did we choose MPI as the programming system for the parallelisation of NaS/RVS visualiser? First of all, MPI is going to be the new standard for writing message-passing programs. It is already supported on a variety of parallel architectures through the public domain implementation of Argonne National Laboratory and parallel computer vendors will develop their own optimised native versions in the near future, and it has been shown that MPI can be implemented efficiently on various architectures. MPI provides more functionality than PVM especially with regard to the collective communications and the support for writing parallel libraries. The arguments that apply to the specific case of the parallelisation task at hand are: at the time of the start of the project various public domain MPI implementations were available for workstation clusters (MPICH, LAM, CHIMP). Secondly, it was clear at that time that the Cenju-3 system would be delivered with an MPI interface to message-passing. Although this interface would be a subset of the full MPI standard the possibility of porting an application written in MPI to the Cenju-3 was apparent. Having considered this we were able to start developing the parallel version of the visualiser on a cluster of IBM RS6000 workstations, using both the LAM and MPICH public domain implementations of MPI.

Since the NaS/RVS code is written in Fortran 77 and has been optimised for execution on the NEC SX/3, perhaps a better choice would have been to use HPF for parallelisation. One would then have to transform the program into a data-parallel program. The compiler directives for loop optimisation on a vector machine like the SX/3 map very well to equivalent directives in HPF. This way one could put up an initial implementation in a relatively small amount of time, and then use the directives already present in the source to include HPF directive to improve performance of the initial implementation without detailed knowledge of the algorithms. This approach

makes advantage of the knowledge that has already been put into the optimisation of the application for the vector supercomputer and probably enables parallelisation of all visualisation algorithms in a fraction of the time necessary when explicit message-passing is used. Unfortunately this approach was not feasible within the time frame of the project, therefor we decided to use MPI.

Parallel Visualiser

Chapter 4

The NaS/RVS System

The Navier Stokes Real-time Visualisation System (NaS/RVS) is a system for 3-dimensional (3D) Computational Fluid Dynamics (CFD) simulations. It could be used to investigate the flow of fluids or gasses around objects such as cars and airplanes. The *flow-solver* calculates values for the pressure, temperature, and velocity of the flow on a 3D grid by solving the Navier Stokes equations which is a system of second order differential equations. The output of the flow-solver is sent to a visualiser which displays the data as a 2-dimensional (2D) image on a display.

The grid used in NaS/RVS is a so called *structured* or *curvilinear* grid which is commonly used in CFD [44]. This grid consists of *hexahedral* (warped) cells whose connectivity is defined in CFD as follows: a cell with address (i, j, k) is defined by the eight vertices (i, j, k) , $(i + 1, j, k)$, $(i, j + 1, k)$, $(i + 1, j + 1, k)$, $(i, j, k + 1)$, $(i + 1, j, k + 1)$, $(i, j + 1, k + 1)$, $(i + 1, j + 1, k + 1)$. These eight vertices define a cell in the grid.

The flow-solution is defined on the vertices of the cells where other software sometimes defines the solution on the central point of the faces of the cells or in the center of the grid cells. An example (extract) of a curvilinear grid is shown in figure 4.1.

There are two notions of space applicable in CFD applications. First there is the notion of physical space in which the three directions are most often called x , y and z . This is the space in which an object is placed for subsequent visualisation. Secondly there is the notion of grid space in which the directions are most often called i , j and k . A hexahedral cell is defined by eight points in this curvilinear grid space.

NaS/RVS is an integrated system in which the flow-solver sends its intermediate solutions directly to the visualiser. This way the intermediate solutions from the flow-solver can be visually verified right away. This is called *tracking* of the simulation. NaS/RVS also allows the user to change parameters for both the flow-solver and the visualiser during simulation. This is useful for adjusting parameters for the flow-solver for example to achieve the desired result. The tracking feedback is necessary in this case to evaluate the result of changing parameters to both the visualiser and the flow-solver. This changing of parameters during the simulation is called *steering*.

In the NEC setup of NaS/RVS both the flow-solver and the visualiser execute on a NEC-SX3/22 supercomputer and the output is sent to an UltraNet frame buffer (figure 4.2). This setup has been changed as suggested in the project description to adapt the system to the NLR environment. The flow-solver now executes serially on one of the nodes of the parallel system and the visualiser is partially parallelised (figure 4.3). In the future the flow-solver will be parallelised as well or be replaced by a different parallel flow-solver. Another change in the setup was that the output is now sent to a window on an X-terminal instead of sending it to an UltraNet frame buffer. For this an additional module was developed (the display module) during the project.

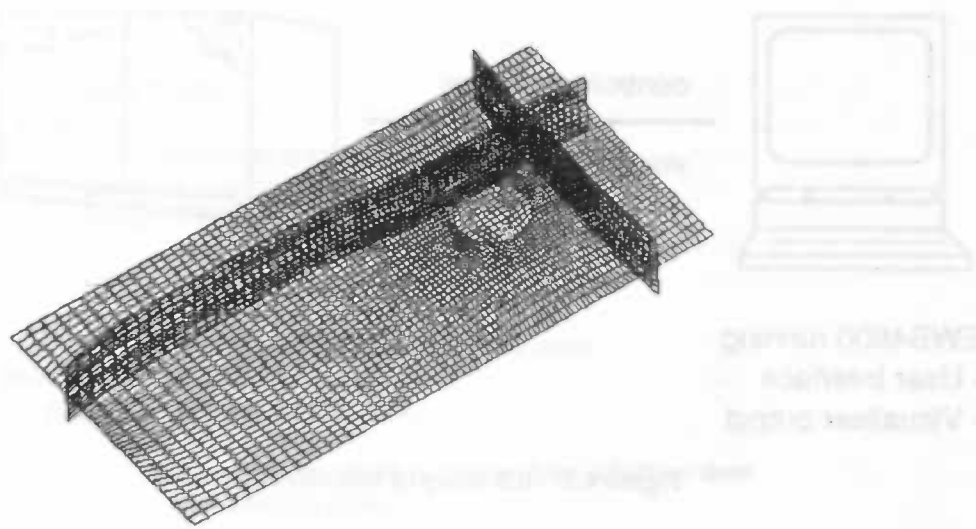


Figure 4.1: Extract of a curvilinear grid with warped grid cells in the area of a cylinder.

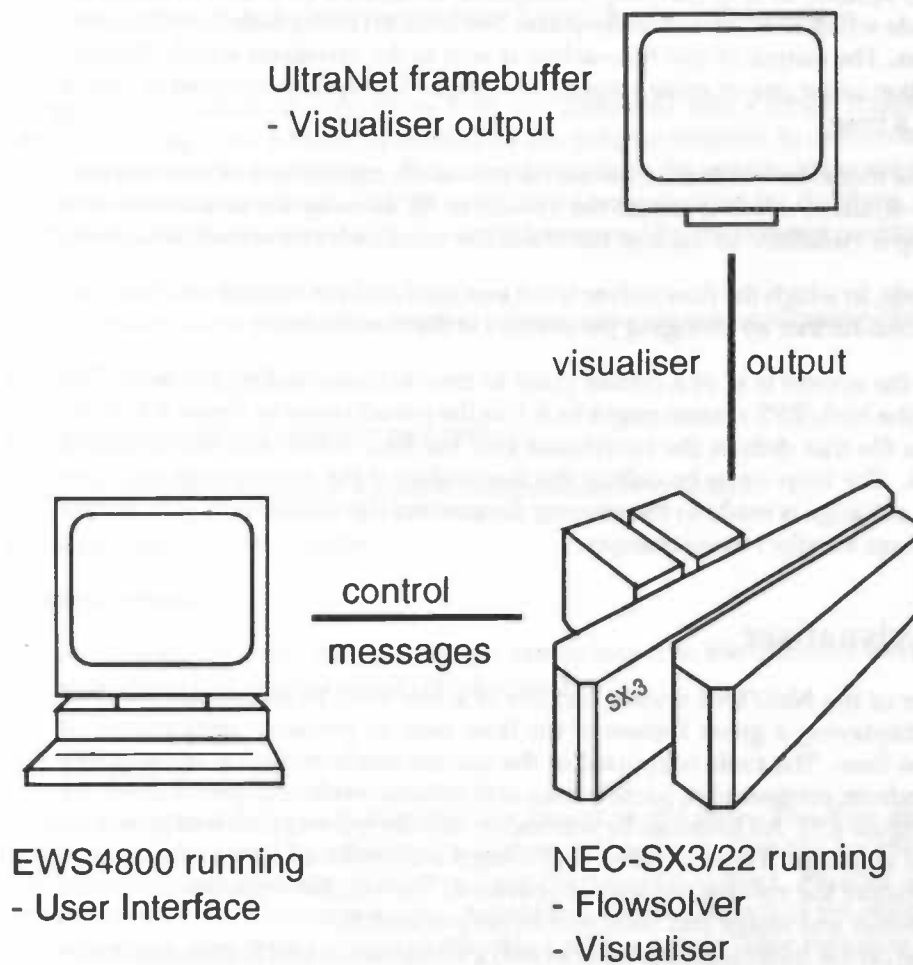


Figure 4.2: NEC setup of NaS/RVS.

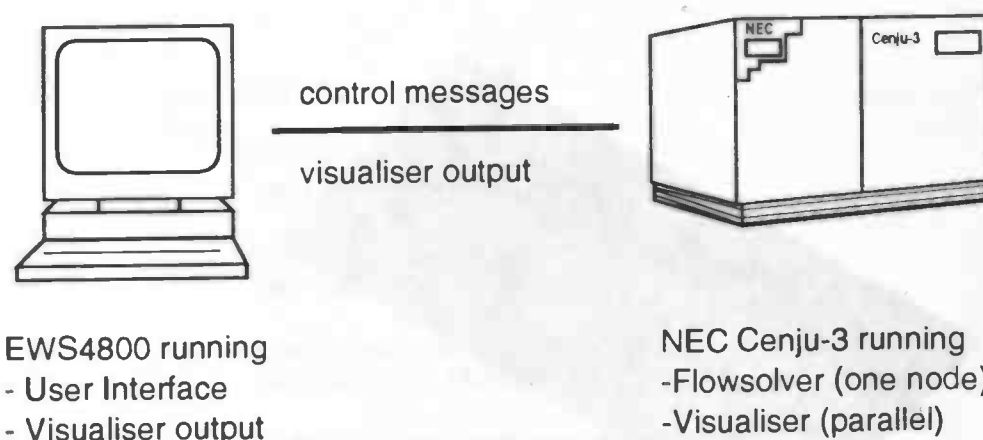


Figure 4.3: NLR setup of NaS/RVS.

4.1 The flow-solver

The NaS/RVS system, as it is currently configured, contains a flow-solver based on the Akiba¹ code which is a Direct Navier-stokes Simulation (DNS) code for 3D incompressible flows. The output of this flow-solver is sent to the visualiser which displays the flow solution using one or more visualisation tools. The system operates in one of two modes at a time:

- Real-time mode, in which the flow-solver constantly calculates a new intermediate flow-solution which is sent to the visualiser. In this way the solution of each time-step is visualised so the user can check the solution for its visual correctness.
- Halt mode, in which the flow-solver is not executed and the current solution may be explored further by changing parameters to the visualiser.

Which mode the system is in at a certain point in time is controlled by the user. The main loop of the NaS/RVS system might look like the pseudo code in figure 4.4. After reading in the file that defines the curvilinear grid the flow-solver and the visualiser are initialised. The loop starts by calling the flow-solver if the system is in real-time mode. When a change is made to the viewing parameters the visualiser is activated to update the image to reflect these changes.

4.2 The visualiser

The visualiser of the NaS/RVS system consists of a few tools to investigate the flow solution by displaying a given feature of the flow such as pressure, temperature or velocity of the flow. The tools supported in the current implementation of NaS/RVS are: object renderer, contour plot, particle trace and volume rendering (pseudo code for visualiser in figure 4.5). All tools can be selected or de-selected and each tool generates a z-buffer and an image. These z-buffers and images are combined into a single image and z-buffer before the volume renderer is activated. The volume visualiser takes this combined z-buffer and image and combines its output with it.

Depending on the state (selected/de-selected) a visualisation tool is activated or not activated. After the execution of the object renderer, contour plot, and particle tracer,

¹name of NEC software engineer.


```

procedure MAIN
begin
    read curvilinear grid coordinates;
    initialise flow-solver;
    initialise visualiser;

    while (not end of simulation) do
        begin
            check for parameter changes by user;
            if(real-time mode) then
                call FLOW_SOLVER;

            if (changed parameters or new flow-solution) then
                call VISUALISER;
        end
    end

```

Figure 4.4: Pseudo code for the main loop of the NaS/RVS system.

the images and z-buffers of these three tools are combined into a single image and z-buffer. This image and z-buffer is needed by the volume renderer to determine the color of each pixel in the final image. This color depends on the opacity of the sampling points for volume visualisation and the color that was generated by the three other tools. The resulting final image on the screen is determined by the viewing parameters. These are:

- View vector/view point, view reference point, view-up vector, clipping planes.
- Viewport or image size.
- Type of projection (parallel or perspective).
- Data to be visualised (flow pressure, temperature or velocity).
- Information on light sources.
- Colour tables.

How these parameters affect the final images can be found in the standard works on computer graphics [20] chapter 6 and [48] chapter 3.

4.2.1 Object renderer

The object renderer displays object(s) that are placed in the flow. The flow around an object can be visualised by coloring the surface of the object. One can for example visualise the pressure or velocity of the flow on the surface of the object. Another option is to give the object its own constant color. Apart from the total surface of the object one can also display a wire frame model of the object with a certain color mapping. The rest of the object will be transparent. The objects are uniquely defined by two vertices in the curvilinear grid which define a (possibly warped) cube consisting of one or more hexahedral cells (figure 4.1).

```

procedure VISUALISER
begin
    if(object renderer selected) then
        call OBJECT_DISPLAY;

    if(contour plot selected) then
        call CONTOUR_PLOT;

    if(particle tracer selected) then
        call PARTICLE_TRACER;

    if(at least two tools activated) then
        combine images of object renderer ,contour plot
        and particle tracer by comparing z-buffer values;

    if(volume renderer selected) then
        call VOLUME_RENDERER;

    send resulting image to screen;
end

```

Figure 4.5: Pseudo code for the visualiser.

4.2.2 Contour plot

The contour plot displays lines or areas of similar data value (pseudo code in figure 4.9) by interpolation the data defined on the vertices of the grid (figure 4.6). For example one can make a contour plot for the temperature in a certain plane in grid space. This plane need not coincide with the cell vertices so an interpolation between the vertices of the nearest cells might be necessary to calculate the data values on the selected plane (figure 4.7). The contour plot allows a detailed inspection of the interior of the grid. At the moment only cross-sections perpendicular to an axis (i , j or k) of the grid can be visualised so arbitrary cross-sections are not supported. The contour plot takes the following parameters:

- The plane (e.g. ij -plane) and the value of the free variable (in this case k) that defines the plane (figure 4.8).
- Minimum and maximum color index. These two define the maximum possible number of contour lines in the output image.
- Minimum and maximum data value to be visualised. This along with the previous parameter defines the size of the data value interval that is mapped to the same color.

The contour plot also has a capability of filling in the space between contour lines with the color of the nearest contour line. The resulting image is called a *contour fringe*.

The contour plot in NaS/RVS is adapted to meet the need for real-time visualisation. Under the assumption that the viewing parameters as described in section 4.2 do not change frequently certain information that is dependent only on these viewing

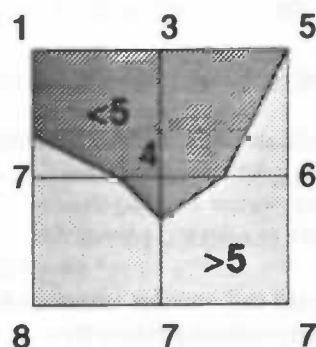


Figure 4.6: Contour line for data value 5. The data values are interpolated along the edges of the four grid cells.

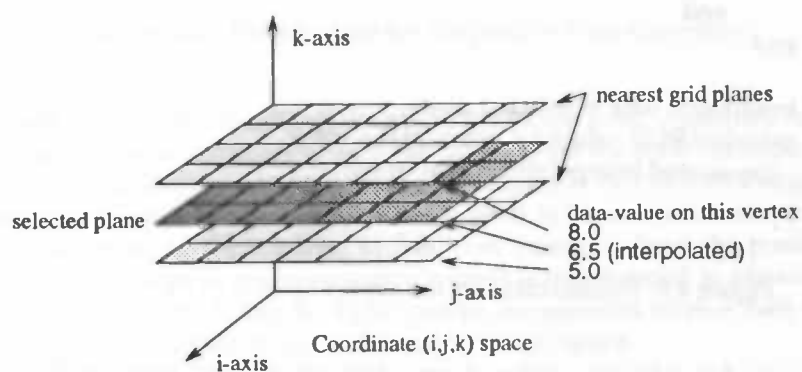


Figure 4.7: Interpolation of the position of the contour plane from the grid points

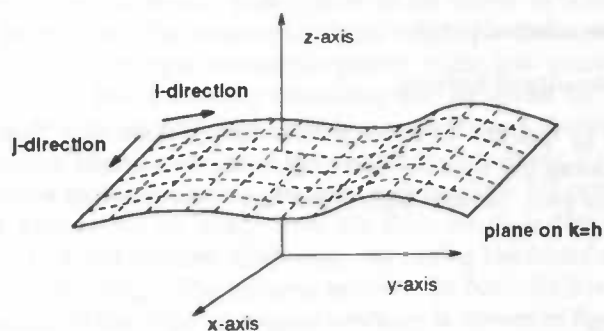


Figure 4.8: ij -plane on k value h

```

procedure CONTOUR_PLOT
begin
  if (change in viewing parameters) then
    begin
      calculate coordinates of selected plane by interpolating
        neighbouring grid coordinates;

      forall (grid cells in selected plane) do
        begin
          project grid cell vertices to screen space;
          store interpolation parameters;
          set pixels of projection of grid cell;
        end

      forall (active pixels) do
        begin
          calculate unit normal;
          calculate reflection coefficient;
        end
      end

      interpolate data to selected plane;
      calculate RGB values for active pixels using
        pre-stored interpolation parameters;
    end
  end

```

Figure 4.9: Pseudo code for the contour plot.

parameters is computed in advance. When a new data-set emerges from the flow-solver and the viewing parameters are unchanged, the new image is computed rapidly using the pre-stored information. The pre-stored information consists of:

- For each pixel the corresponding cell in the grid.
- Parameters for the interpolation along the edges of cells in the grid.
- Normal vectors to the selected plane.
- For each pixel the intensity of lighting.

This information is used to quickly generate a new image once new flow data is available. This greatly reduces the time needed to generate this image improving the performance of the contour plot. Similar techniques have been applied to other tools aswell.

4.2.3 Particle tracer

Using this tool one can inject particles into the flow to investigate certain aspects of the flow. The initial position of the particles is given in grid coordinate space. The particle positions change due to the flow and every time-step the new particle position is displayed along with the previous positions. The particles thus leave a trace in

```

procedure PARTICLE-TRACER
begin
    if (real-time mode) then
        begin
            inject new particles;
            foreach(particle) do
                move particle to new position;
            end
        else /* halt mode */
            foreach (particle) do
                begin
                    transform particle to physical coordinates;
                    calculate color for particle;
                    transform particle to screen space;
                    set RGB- and Z-buffer values;
                end
            end
        end

```

Figure 4.10: Pseudo code for the particle trace algorithm.

the medium which may clarify certain properties of the flow. Again the color of the particles can be chosen constant or they can be assigned the color corresponding to a certain data value. The pseudo code for the particle tracer can be seen in figure 4.10.

When the visualiser is in real-time mode the particle tracer injects new particles into the flow and moves all the particles to their new position. Next the position of the particles (which is kept in grid space coordinates) is transformed to physical (world, x, y, z space) coordinates. Using the light sources, the particles receive their color after which they are transformed to screen (image/z-buffer) space.

4.2.4 Volume rendering

This tool gives an impression of the full 3-dimensional dataset by representing the data in the volume by clouds of varying color and opacity. Depending on the data value at each point in the grid and the gradient of these data values a color and opacity is assigned to each sample point. This results in an image in which certain aspects of the flow will be visible. The volume renderer might for example be configured to show high velocity flow as opaque sample points while low velocity flow is shown as transparent points. The following taxonomy can be given for volume renderers [50]: surface fitting methods and direct volume visualisation. Surface fitting methods extract a surface from the flow-solution and render the resulting 3D primitives. Direct volume visualisation methods do not generate intermediate 3D primitives but render the volume as is, generating an image from the flow-solution directly. There are two major approaches in direct volume rendering: *ray casting (backward mapping)* and *plane compositing (forward mapping)*. The volume renderer in NaS/RVS is of the latter type. The general structure of this type of volume renderer is shown in figure 4.11 and figure 4.12 (adapted from [48]).

```

foreach(plane in the volume data) do
  foreach(cell in the plane) do
    begin
      find the pixels -the 'footprint'- that the cell projects onto;
      calculate new color for those pixels
        based on color resulting from previous planes;
    end
  end
end

```

Figure 4.11: General structure of a plane compositing (forward mapping) volume renderer.

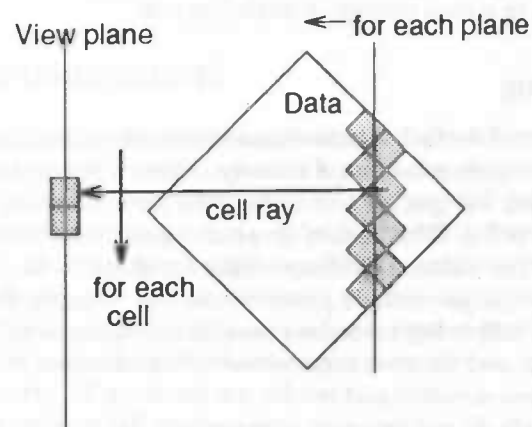


Figure 4.12: Plane compositing or forward mapping.

Chapter 5

Parallelisation of a visualisation tool

This chapter will present the design and implementation of the parallel contour plotting algorithm. This tool was chosen to be implemented because it is used quite often for the visualisation of CFD data. Secondly, because there was no English¹ documentation, not on paper nor as comments in the source code, on the algorithms being used in any of the visualisers contained in NaS/RVS, the contour plot was chosen to be implemented first since it is a relatively simple algorithm. It still took me a few weeks to unravel the mysteries of this specific implementation of the contour plot. Due to the time constraints of the project this is the only tool that has been parallelised.

First a few design considerations [29] will be introduced after which the sequential and the parallel contour plot algorithms will be presented. Two parallel algorithms using two approaches for data distribution have been implemented and the performance of the algorithms resulting from these two approaches will be discussed in the last section of this chapter.

5.1 Design considerations

When designing a parallel algorithm one has to consider a few things. Parallelisation is mostly aimed at improving the performance of a particular piece of software so one of the questions is how to measure the performance of the parallel implementation. There are a number of performance metrics that will be discussed in the next subsection. Another consideration is whether the parallel algorithm is scalable, meaning that adding more processors will not lead to loss of efficiency. Finally one should keep in mind that the parallelisation of an algorithm will most probably lead to parallel overhead. These considerations will be discussed in the following subsections.

5.1.1 Performance metrics

This section will introduce some metrics that are commonly used to measure the performance of parallel algorithms.

Run time The run time of a sequential algorithm is the time elapsed between the beginning and the end of the program. Parallel run time is the time elapsed between

¹There is some documentation in Japanese.

the moment that the parallel program begins and the time the last processor finishes execution. Serial run time will be denoted by T_s and parallel run time by T_p .

Speedup When parallelising an algorithm we are most often interested in how much we gain in speed by running the program on p processors, say, as opposed to running it on a single processor. A metric for this is the *speedup* which is defined as the ratio between the run time of the algorithm on a single processor and the run time of the parallel algorithm on multiple processors.

$$S = \frac{T_s}{T_p}$$

For the value of T_s we have a choice of taking the run time of the parallel algorithm on a single processor or the run time of the sequential algorithm that was parallelised. The best choice is to use the run time of the sequential algorithm because the parallel algorithm running on a single processor will experience overhead from the communication routines although they may not perform any data transfer. Another possible choice for T_s is to take the run time of the fastest known sequential algorithm with the same functionality. This might be an algorithm that is hard to parallelise, but has good performance on sequential machines. This choice of T_s however is impractical because most likely the fastest sequential algorithm is not available to the developer or can not be compared to the sequential that is used as a basis for parallelisation because of extra or missing functionality.

The value of the speedup metric will usually be greater than zero and less than or equal to the number of processors p used. A speedup greater than p is called *super-linear speedup* and is most often caused by the reduced resource needs per processor of the parallel algorithm. For example the sequential algorithm might need the swapping capability of a single processor machine to meet its memory needs while the distribution of data onto the processors of the parallel machine causes the parallel algorithm to be less consumptive with memory per processor which consequently does not need the performance degrading swapping mechanism.

Efficiency This metric is defined as the fraction of time the processor is usefully employed. It is defined as the ratio of the speedup S to the number of processors p .

$$E = \frac{S}{p}$$

An optimal parallel algorithm will have speedup p and thus efficiency 1.0, but for super-linear speedup the efficiency will be larger than 1.0.

Cost The last commonly used metric is cost. It is defined as the product of parallel run time T_p and the number of processors p .

$$C = pT_p = \frac{pT_s}{S} = \frac{T_s}{E}$$

A parallel algorithm is said to be *cost-optimal* if the cost of solving the problem on a parallel computer is proportional to the run time of the sequential algorithm on a single processor. This is equivalent to the statement that a parallel algorithm is cost-optimal if the efficiency is independent of the number of processors.

5.1.2 Scalability

When parallelising an algorithm for a certain parallel computer one wants to be able to increase the number of processors and the problem size while maintaining the efficiency of the parallel algorithm. This property is called the scalability of a parallel algorithm. If for example, a scalable algorithm running on a certain number of processes yields a certain efficiency, the same efficiency can be achieved if we multiply the number of processors by two and the problem size by a number of the same order of magnitude. Thus if the increase in the number of processors and the increase in the problem size are of the same order of magnitude, while maintaining the same efficiency, the algorithm is scalable. This property is valuable because the tendency in parallel computing is to add more and more processors to parallel systems to be able to handle more complex problems (increase in problem size).

5.1.3 Parallel overhead

The parallelisation of an algorithm often introduces *parallel overhead*. Three types of parallel overhead will be presented here.

Interprocessor communication. Any nontrivial parallel program requires communication among the processors to solve a problem. The time spent communicating is usually the most significant source of parallel overhead. In the ideal situation where one can make communications and computation overlap using non-blocking communications this type of overhead will be reduced.

Load imbalance. Parallel programs usually divide the problem into subtasks. In many cases the size of the subtask can not be predicted so the problem can not be subdivided statically among the processors, resulting in a nonuniform load balance. This leads to less efficient parallel programs because the end time of parallel processing is the time the last processor ends processing. In the case of a load imbalance this time is larger than the optimum case in which all processors spend an equal amount of time processing.

Extra computation. The last source of parallel overhead are the extra computations that might be necessary if for example certain results, that would be reused in a sequential algorithm, can not be reused in the parallel algorithm and need to be recomputed. Often the boundaries between the subtasks are replicated in neighbouring processors to eliminate the need to communicate these boundaries but the consequence is that the computations for the boundaries are replicated as well.

5.2 Data partitioning

In the parallelisation of rendering applications two approaches are commonly used. One can partition the *object space* meaning that multiple objects, in our case the grid cells, are distributed over the available processors. The second approach would be to partition *screen space* meaning that each processor is responsible for a collection of pixels that contribute to the final image. These two different approaches are illustrated in figure 5.1. This section will evaluate the parallelisation of the contour plot algorithm using these two approaches.

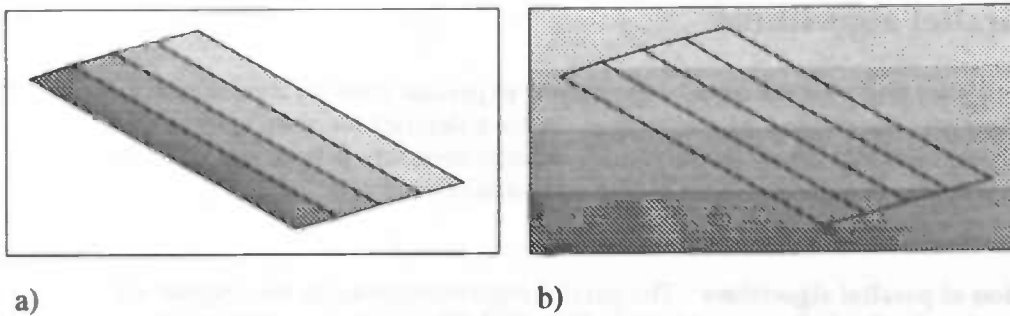


Figure 5.1: Assignment to processors for parallelisation in object space (a) and screen space (b).

The NaS/RVS system has two large data structures that are candidate for partitioning, the *grid coordinates* and the *flow solution*². In general both the grid coordinates and the flow solution may change from one time-step to another but we will only consider the case in which grid coordinates are stable and the flow solution is unstable, since this is the case for the NaS/RVS system.

Because the grid coordinates are stable a good choice is to replicate the grid coordinates in each processor, if there is enough memory, so each processor has its own private copy of the grid coordinates. If we replicate the grid coordinates in each processor no communication during the computation is necessary to get geometry data from other processors. Broadcasting the grid coordinates to each processor will only affect the setup time for the algorithm while increasing the sustained performance of the application because of the elimination of some of the communication during the computations.

The flow solution however is not stable. Each time-step the flow-solver calculates a new intermediate result that needs to be visualised. Parts of the intermediate result should be available to the processors that need it. Which part of the result is needed depends on the partitioning chosen, that is object space partitioning or screen space partitioning.

5.2.1 Partitioning of object space

Each visualisation tool has the same screen space but may have a different object space. The object space for the object renderer for example is the geometry of the object that is placed in the flow, while the object space for the contour plot is the plane that intersects the volume and on which the contour should be displayed. The object space for the particle tracer might be the particles that are moving through the flow or it might be the 3D grid space as a whole. The object space for the volume renderer is the 3D grid space. When referring to the object space we mean the object space that corresponds to the visualisation tool in question. The partition of object space as shown in figure 5.1 would correspond to the contour plot.

5.2.2 Partitioning of screen space

The screen space is uniquely defined to be the 2D array of pixel values and z-buffer values produced by the various visualisation tools. Each tool operates on the same screen space and a particular partitioning of the screen space affects all tools.

²velocity in u , v and w direction, pressure and temperature.

5.3 Parallel algorithms

In this section we will give the parallel algorithms in pseudo code for the main loop, the visualiser and the contour plot algorithm. After a description of each algorithm there is a paragraph that describes the communication necessary in each algorithm in order to evaluate each algorithm's scalability and parallel overhead.

Formulation of parallel algorithms The parallel algorithms listed in this chapter are programs using the Single Program Multiple Data (SPMD) model of parallel programming. This means that a copy of the same program runs on each node of the parallel machine. The parallel functions, like for example the broadcast operation, are executed by all nodes. Depending on the logical rank of the node in the parallel machine the function performs the necessary actions to implement the intended functionality. In the case of the broadcast operation one node would send the data to all the others which in turn would receive the data from the sending node. So depending on the logical rank the root node performs a different action than the rest of the nodes. This is also the model of parallel message passing programming that is intended with the MPI standard and this is the reason why it is used.

5.3.1 Parallel main loop

The parallel main loop presented in this section is equal for both the object and screen space partitionings (Figure 5.2). The scalar parameters for the various tools are broadcasted during initialisation and broadcasted again if they change. These scalar parameters control the behaviour of the various tools. All interaction with the application is through the root node. The root node receives all user input and broadcasts this information to the other nodes when necessary. For example, if the user changes the viewpoint, a boolean will be set on the root node to indicate that the viewpoint has been changed and the viewpoint parameter will consequently be broadcasted to inform each node of the change so it can use this parameter in the various tools in the next call of the visualiser.

The grid coordinates are broadcasted before the initialisation of the parallel visualiser. The barrier synchronisation before the call to the parallel visualiser is necessary to prevent that the nodes end up in a busy wait loop, waiting for the flowsolver to complete when the parameters to the visualisation tools are unchanged. It thus ensures that the root node and the other nodes execute in lock step, meaning that the other nodes wait for the flowsolver on the root node to complete before calling the parallel visualiser.

Communication The communication involved in the main loop of NaS/RVS is mainly to initialise NaS/RVS such as broadcasting the default scalar parameters and broadcasting the grid coordinates. The time for this communication is added to the setup-time and does not affect the sustained performance of NaS/RVS. The broadcast of the scalar parameters after a change has been made to these parameters, occurs in the main loop and may affect performance, but since the amount of data transferred is small this will not affect the overall performance of the loop. The barrier synchronisation also implies communication, but this again involves only very small messages and will not affect the performance of the main loop.

```

procedure PARALLEL_MAIN
{ For both object and screen partitioned visualisers }
begin
    broadcast scalar parameters;
    if (root node) then
        begin
            read curvilinear grid coordinates;
            initialise flow-solver;
        end
        broadcast grid coordinates;
        initialise parallel visualiser;

        while (not end of simulation) do
            begin
                if(root node) then
                    begin
                        check for parameter changes by user;
                        if(real-time mode) then
                            call FLOW_SOLVER;
                        end

                        if(changed parameters on root node) then
                            broadcast changed parameters;
                            barrier synchronisation;
                        if (changed parameters or new flow-solution) then
                            call PARALLEL_VISUALISER;
                        end
                    end
            end
        end

```

Figure 5.2: Pseudo code for the parallel main loop.

```

procedure PARALLEL_VISUALISER
{ For partitioning of object space }
begin
    if(object renderer selected) then
        call PARALLEL_OBJECT_RENDERER;

    if(contour plot selected) then
        call PARALLEL_CONTOUR_PLOT;

    if(particle tracer selected) then
        call PARALLEL_PARTICLE_TRACER;

    if (at least two tools activated) then
        begin
            combine local images of object renderer, contour plot
            and particle tracer, by comparing z-buffers values;
            image composition to one globally valid image
            using z-buffer values;
        end

    if(volume renderer selected) then
        begin
            call PARALLEL_VOLUME_RENDERER;
            image composition to one globally valid image using
            z-buffer and  $\alpha$ -values generated by volume renderer;
        end
    end

```

Figure 5.3: Pseudo code for the parallel visualiser using partitioning of object space.

5.3.2 Parallel visualiser

For the parallel visualiser we will only consider the case in which all tools use the same partitioning of data, either object space or screen space partitioning. This is to keep only two essential variants of the parallel visualiser, namely the case in which all tools use partitioning of object space and the case in which all tools use partitioning of screen space.

Partitioning of object space

In the case of object space partitioning of the data, each node produces its own full sized framebuffer by combining the results from the activated visualisation tools into one local full sized framebuffer. The contents of each of these framebuffers is not valid independently because the images of different processors may overlap. Only by combining these framebuffers one can obtain the final image as shown in figure 5.4. Before calling the volume renderer, the framebuffers of the other tools should be combined based on the z-buffer values. A pixel that has a smaller z-value (closer to the viewer) will obscure a pixel with a larger z-value. In this way the framebuffers from all processors will be depth-sorted in order to obtain a valid intermediate image that can be used by the volume renderer to combine its output with it.

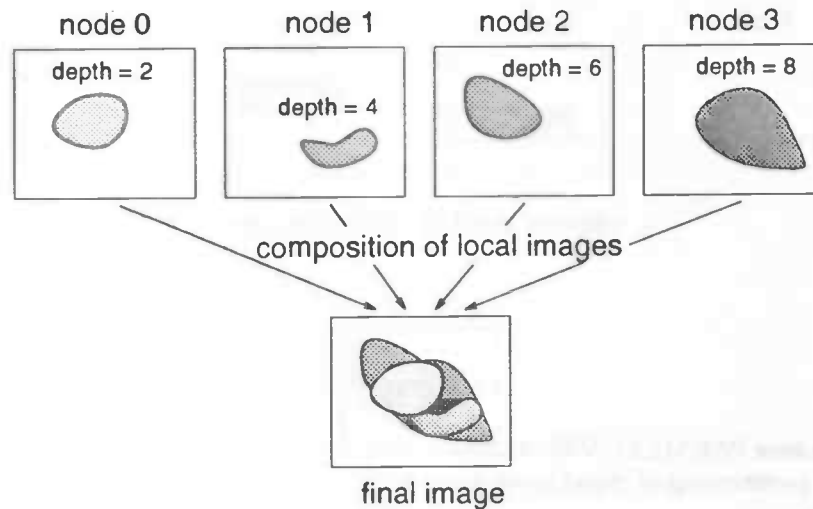


Figure 5.4: Image composition of all images to one image.

The volume renderer needs the full intermediate framebuffer because the partitioning of the object space for the volume renderer may differ from the partitioning of the object space of all other tools. Once each processor has combined the output of the volume renderer, which is a convex part of the volume, with the intermediate image, another image composition step is necessary to combine the outputs of the volume renderer into a final image. This image composition step is different from the previous image composition step in that it uses not only the z -value, but also the α -value, of each pixel to determine the color. Since the combination of the α -values is not commutative the order in which the results are retrieved from the processors to be combined with the intermediate result is important; the images should be combined in depth order.

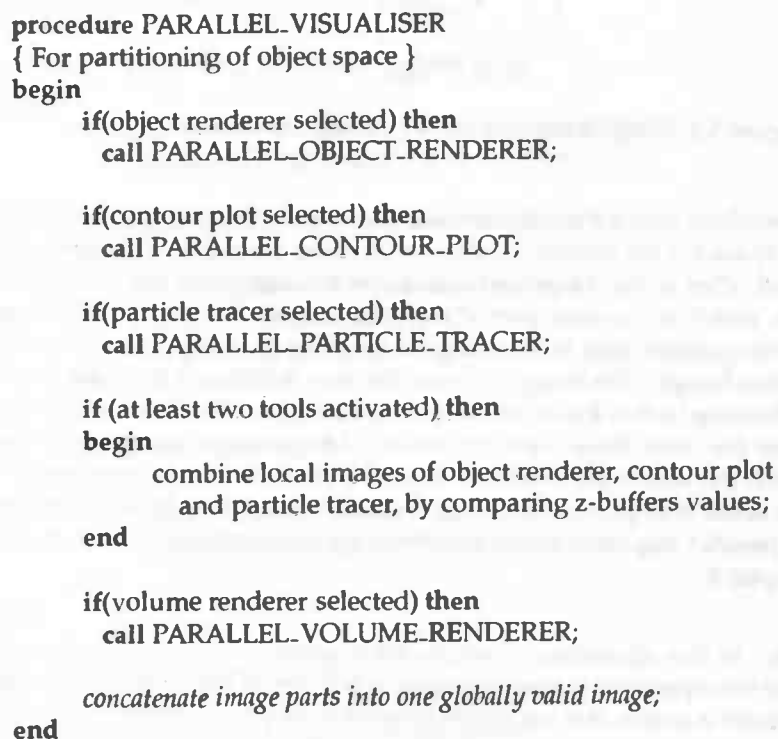
An efficient parallel implementation of the image composition operation will be presented in chapter 6.

Communication In this algorithm communication arises in the image composition operations. Since this operation is executed twice when any of object renderer, contour plot or particle tracer is active, it is important to achieve an efficient implementation of this operation.

Two image composition operations are necessary because the partitioning of object space in each of the object renderer, contour plot and particle tracer may be different. The first image composition might be eliminated, if the partitioning of object space would be uniform over these tools. The consequence of this is that if the object to render falls entirely within a single processor's part of the volume only one processor would be computing for the object renderer, resulting in poor load balance.

Partitioning of screen space

In the case of screen space partitioning of the data, each node produces part of the final image. The part of the image that is produced by a node is a valid part of the final image. The first combine statement, combines the resulting framebuffers from the activated visualisation tools. This results in a valid part of the final image on each node. Because we assume that the parallel volume renderer also uses the screen space partitioning, no reduction operation is needed to combine (parts of) framebuffers from



```

procedure PARALLEL-VISUALISER
{ For partitioning of object space }
begin
  if(object renderer selected) then
    call PARALLEL-OBJECT-RENDERER;

  if(contour plot selected) then
    call PARALLEL-CONTOUR-PLOT;

  if(particle tracer selected) then
    call PARALLEL-PARTICLE-TRACER;

  if (at least two tools activated) then
    begin
      combine local images of object renderer, contour plot
      and particle tracer, by comparing z-buffers values;
    end

  if(volume renderer selected) then
    call PARALLEL-VOLUME-RENDERER;

  concatenate image parts into one globally valid image;
end

```

Figure 5.5: Pseudo code for the parallel visualiser using partitioning of screen space.

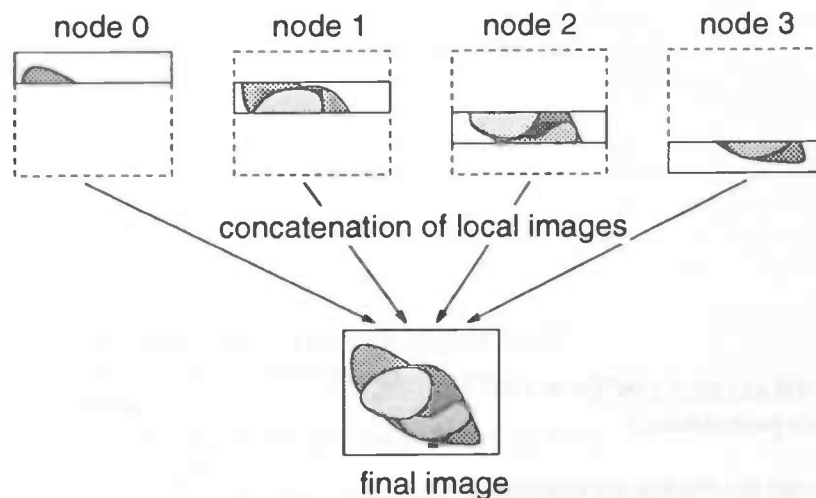


Figure 5.6: Concatenating the parts of the final images from the nodes into one globally valid image.

all nodes. The parallel volume renderer takes the local part of the final image and adds its result for that part to it. The last statement in the code concatenates the results from all the nodes into one valid final image on the root node (Figure 5.6).

Communication The only communication in this algorithm is in the last statement in which the partial images on all processors are concatenated into the final image on the root node.

5.3.3 Parallel contour plot

The design of the parallel contour plot algorithm for both the partitioning of object and partitioning of screen space will now be presented.

Partitioning of object space

The algorithm for the contour plot algorithm that assumes partitioning of object space is listed in Figure 5.7. This routine is invoked by the parallel visualiser and at that time the grid coordinates have already been broadcasted to all processors by the main loop. The only communication necessary is to get the part of the new flow solution that is assigned to this processor. That node will then perform the necessary computations to obtain the contour plot for that part of the selected plane. This object space algorithm does not deviate much from the non-parallel contour plot algorithm. Instead of operating on the whole plane it operates on the part of the plane for which it is responsible.

Communication The communication in the object partitioned contour plot algorithm is to get that part of the flow-solution for which the processor is responsible. Apart from this there is no communication in this algorithm.

Partitioning of screen space

The algorithm for the contour plot that assumes partitioning of screen space is listed in Figure 5.8. Since we do not know to which part of the screen the grid cells of the

```

procedure PARALLEL_CONTOUR_PLOT
{ Object space partitioning }
begin
  if (change in viewing parameters) then
    begin
      calculate coordinates for my part of selected plane by
        interpolating neighbouring grid coordinates;

      forall(grid cells in my part of selected plane) do
        begin
          project grid cell vertices to screen space;
          store interpolation parameters;
          set pixels of projection of grid cell;
        end

      forall (active pixels) do
        begin
          calculate unit normal;
          calculate reflection coefficient;
        end
      end

      get my part of new flow solution;
      interpolate data to my part of selected plane;
      calculate RGB-values for active pixels using
        pre-stored interpolation parameters;
    end
  end

```

Figure 5.7: Pseudo code for the parallel object space partitioned contour plot.

```

procedure PARALLEL-CONTOUR-PLOT
{ Screen space partitioning}
begin
  if (change in viewing parameters) then
    begin
      calculate coordinates of selected plane by
        interpolating neighbouring grid coordinates.

      forall(grid cells in selected plane) do
        begin
          project grid cell vertices to screen space;
          if (a vertex of grid cell projects to my part of screen) then
            begin
              store interpolation parameters;
              set pixels of projection of grid cell;
            end
          end
        end

      forall (active pixels) do
        begin
          calculate unit normal;
          calculate reflection coefficient;
        end
      end

      broadcast new flow solution;
      interpolate data for cells whose projection falls into the
        region assigned to this processor to selected plane;
      calculate RGB-values for active pixels using
        pre-stored interpolation parameters;
    end
  end

```

Figure 5.8: Pseudo code for the parallel screen space partitioned contour plot.

selected plane will project we need all the grid cells on all processors. This is why the complete new flow solution is broadcasted to all processors after the if-statement. The main difference between the non-parallel and parallel algorithm is that the grid cells that the procedure handles are constrained to the cells of which the projection onto the screen falls (at least partially) within the region of the screen that was assigned to a processor. This can be seen in the first forall loop. Only those pixels that fall within this region are set and processed by the next forall loop. The last two statements of the procedure only process the grid cells that map into the region assigned to a processor.

Communication Broadcasting the new flow-solution involves communication in the screen space partitioned contour plot algorithm. A different approach would be to send flow-solutions for grid cells when they are needed. Because of the many small communications, which are probably inefficient, and the additional program complexity this design was not chosen.

5.4 Discussion of the design

The algorithms listed in the previous section will be discussed in this section keeping the design considerations stated in section 5.1 in mind. We can only discuss the design by reasoning about the scalability and the parallel overhead for each of the algorithms. For this purpose we have summarized the communication characteristics of each of the algorithms. The discussion of the performance of the actual implementation of the algorithms will be postponed to section 5.5.

5.4.1 Scalability

The scalability of the main loop of NaS/RVS can only be affected by the communication of changed scalar parameters. Since in most cases only a small number of scalar parameters is changed at the same time, the amount of data to be communicated is small and this will not affect scalability of the main loop.

The object space partitioned visualiser's scalability is dependent on the scalability of the two image composition steps in the algorithm, and on the scalability of the visualisation tools that are called by the visualiser. If all visualisation tools are scalable, and the image composition steps are scalable, then the parallel visualiser is scalable. Chapter 6 presents such a scalable image composition algorithm. The scalability of the visualisation tools depends on the parallel algorithm used in their implementation.

Scalability of the parallel visualiser that uses partitioning of screen space for parallelisation is apparent. Only the final step where the partial image are concatenated into one final image requires communication. Since the image size will probably not change in order of magnitude, this stage is scalable. The total amount of data communicated will remain of the same order of magnitude as the size of the framebuffer. When more processors are used, each processor will be responsible for a smaller part of the framebuffer thus concatenating these parts into one final image will involve concatenating more smaller parts into one image. This concatenation operation is scalable.

The object space partitioned contour plot algorithm is scalable. The only communication necessary is to get the part of the new flow solution for which the processor is responsible. Using more processors will reduce the amount of data that needs to be communicated.

The screen space partitioned contour plot needs the complete flow solution each time a new solution is available. If the broadcast operation is implemented as a scalable algorithm, then this operation is scalable.

5.4.2 Parallel overhead

Parallel overhead in the object space partitioned visualiser is caused by necessary communication in the parallel algorithm. The amount of communication involved in image composition is equal to the size of the framebuffer times the number of processors. Since not all communication uses the same communication channel, the communication is performed in parallel. Another source of parallel overhead may be bad load balance in any of the steps. The load balance of the visualisers depends on the parallel algorithm used therein, and the load balance of the image composition step will be discussed in chapter 6.

The amount of communication overhead in the screen space partitioned visualiser is in the order of magnitude of the size of the framebuffer. With a constant framebuffer size, this amount of communication remains constant for a larger number of processors. Load balance depends on the load balance of the tools called in the visualiser. When screen space is partitioned, load balancing tends to become a problem for the tools since one does not know in advance on what part of the screen an object will be projected. By adapting the size and shape of the parts for which a processor is responsible based on the images from previous calls to the visualiser, one can achieve better load balance. Extra computation in this design of the parallel visualiser might be caused by the tools called in the visualiser. If certain objects project into more than one part of the screen, computations will have to be performed for such an object on each processors that is responsible for a part of the screen into which the object projects. Each processor will only keep the part of the result that affects his part of the screen. This is especially true for large numbers of processors.

Communication overhead in the object space partitioned contour plot algorithm is caused by the need to get part of the new flow solution. Load balancing should be good when the object space is partitioned. Each processor is responsible for an equally sized part of the object space, and this fair partitioning of the object space is easy compared to a fair partitioning of the screen space.

The parallel overhead that is present in the screen space partitioned contour plot algorithm is due to the need to check whether a cell projects into the part of the screen for which a processor is responsible. This overhead becomes larger when more processors are used because the size of the region assigned to a processor becomes small compared to the total size of the object's projection. This extra computation however is only necessary when the viewing parameters have changed.

5.5 Performance

As said before the contour plot algorithm has been parallelised for both an object and screen space partitioning. In this section the performance of both implementations will be presented and compared in order to determine the optimal parallelisation strategy for NaS/RVS visualisers. It is believed that the parallelisation method for the contour plot algorithm with the best performance will also yield good performance for the other visualisers since they all use a similar program structure.

5.5.1 Measurement setup

In order to measure and compare the performance of the two parallel implementations of the contour plot algorithm two setups were used. The choice of the setup was constrained by the fact that only one of the sample datasets delivered with the original NaS/RVS system could be used on the Cenju-3 due to insufficient memory for the larger sample dataset. This is caused by the disproportionate large memory requirements of

no. procs	partitioning			
	object space		screen space	
	min time	max time	min time	max time
sequential	2.89			
2	1.91	1.94	1.05	1.06
4	1.40	1.46	0.65	0.83
8	1.31	1.35	0.19	0.58
16	1.22	1.46	0.09	0.50

Table 5.1: Rendering times for small projection and stable viewing parameters.

no. procs	partitioning			
	object space		screen space	
	min time	max time	min time	max time
sequential	7.33			
2	4.51	4.67	3.62	3.63
4	3.19	3.43	2.71	3.00
8	2.71	2.89	0.41	1.80
16	2.27	3.43	0.19	1.13

Table 5.2: Rendering times for small projection and unstable viewing parameters.

the NaS/RVS system which pose no problem on a large supercomputer like the SX-3 but which do limit the maximum problem size on the Cenju-3. The sample dataset used for the performance measurements is the so called 'Karman' dataset. It consists of a 81x41x11 curvilinear grid containing a cilinder.

Small projection setup The first setup renders a contour plot on two planes through the volume and the setting of the viewing parameters results in an image of size 512x512 of which only 26% of the total number of pixels is occupied by the rendered object. Therefore this setup is called the *small projection setup*.

Large projection setup The second setup also renders a 512x512 image but in this case the viewing parameters are such that the rendered object occupies 98% of the image. This setup is therefore called the *large projection setup*.

Stable vs. unstable viewing parameters The NaS/RVS system is optimised to render consecutive frames of a scene when the viewing parameters do not change between frames. This is the case when the flow solver calculates a new solution in each timestep and these consecutive solutions are all visualised using the same viewing parameters. The performance of the renderer is analysed both in the case of stable (unchanging) viewing parameters and in the case of unstable (changing) viewing parameters.

5.5.2 Rendering times

Tables 5.1 through 5.4 contain the rendering times measured on the Cenju-3 for the small and large projection setup and for stable and unstable viewing parameters. On the left the tables contain the rendering times for the object space partitioned contour plot algorithm and on the right the rendering times for the screen space partitioned algorithm. These times do not include sending the final image that has been collected on a single processor of the Cenju-3 to the front end. In each of the tables the rendering

no. procs	partitioning			
	object space		screen space	
	min time	max time	min time	max time
sequential	6.06			
2	3.25	3.67	2.00	2.01
4	1.97	2.50	1.00	1.19
8	1.56	2.07	0.48	0.79
16	1.30	2.02	0.24	0.61

Table 5.3: Rendering times for large projection and stable viewing parameters.

no. procs	partitioning			
	object space		screen space	
	min time	max time	min time	max time
sequential	21.3			
2	14.6	14.8	9.17	9.18
4	5.98	8.40	4.50	4.68
8	3.94	6.17	2.24	2.57
16	2.68	6.20	1.10	1.54

Table 5.4: Rendering times for large projection and unstable viewing parameters.

time for the sequential algorithm is obtained not by running the parallel algorithm on a single processor, but by running the original NaS/RVS system on the front end of the Cenju-3 which uses the same processor architecture as the nodes of the Cenju-3. The time was measured while the front end was idle and only one user was logged onto the system. Rendering times have been measured on all processors and the minimum and maximum of these times are shown in the two columns for each algorithm in the tables. These times will be used to estimate the load balance of the algorithms in a later section.

5.5.3 Speedup

The performance of the parallel contour plot algorithms will be illustrated by the speedup graphs for the small projection setup and the large projection setup. A comparison will also be made between the stable and unstable viewing parameter cases of the large projection setup for the screen space partitioned algorithm. The speedup graphs have been created using the maximum rendering times since this is the actual time the parallel algorithm takes to complete its task.

Small projection, unstable viewing parameters First the speedup graph for the small projection setup with unstable viewing parameters will be discussed. The speedup graph is shown in figure 5.9. The graph shows that for the small projection setup, the screen space partitioned algorithm performs better than the object space partitioned algorithm, but the speedup of 5.78 on 16 processors is not very good. The disappointing speedup of the screen space partitioned algorithm is due to load imbalance as we will see in the next section. The bad performance of the object space partitioned algorithm is due to the nature of the contour plot algorithm. The contour plot algorithm is, just like most of the visualisers in NaS/RVS, a image order algorithm. This means that in its main loop the algorithm iterates over the image pixels. Because the object space partitioned algorithm renders a full sized image at each processor the main loop

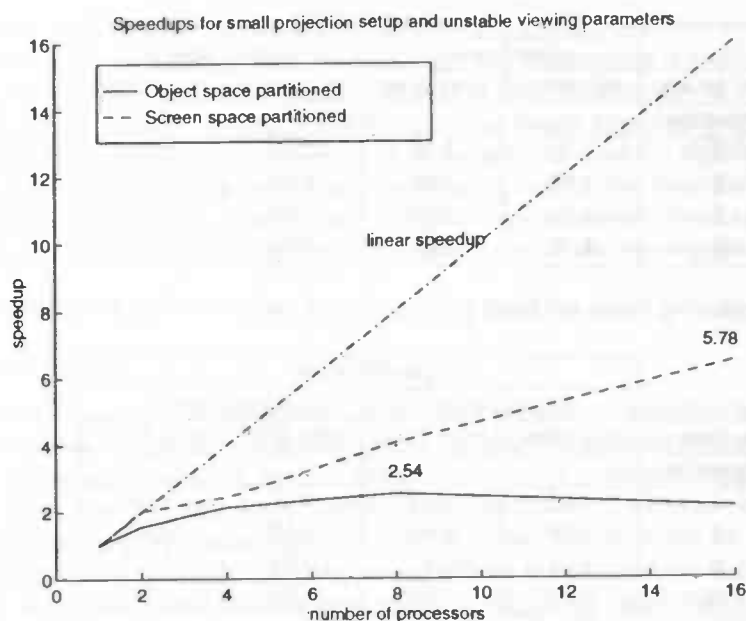


Figure 5.9: Speedup for object and screen space partitioned contour plot algorithm for unstable viewing parameters and small projection setup.

algorithm still iterates over all image pixels but for a smaller dataset. This results in a maximum speedup of 2.54 for 8 processors.

Large projection, unstable viewing parameters The speedup graph for the large projection setup again with unstable viewing parameters is shown in figure 5.10. In this graph the speedup for the screen space partitioned algorithm is good: 13.8 on 16 processors. Because 98% of the rendered image is occupied by the object each partition of the screen has an equal amount of work associated with it, thus resulting in good load balance. The super-linear speedup for 2, 4 and 8 processors can be explained by operating system overhead on the host workstation on which the sequential time was measured. The operating system on the host is much more complex than the simple kernel that runs on each of the nodes of the parallel system. Although the sequential program was run on the host workstation while it was idle, a number of other operating system related processes were using CPU time and memory. The nodes of the Cenju-3 run only one process, namely the NaS/RVS program. Another explanation could be the improved cache hit of the algorithm because it acts on a smaller part of the image.

Again the performance of the object space partitioned algorithm is poor with a maximum value of 3.45 for 8 processors. The image order nature of the algorithm accounts for the very low speedup for a partitioning in object space.

Stable vs. unstable viewing parameters Until now, only the case of the unstable viewing parameters has been discussed. This is the most common case since in order to explore a flow solution, the user wants to navigate through the 3D volume and thus constantly changes the viewing parameters. It is interesting however to compare the speedup of the screen space partitioned algorithm in the large projection setup for stable and unstable viewing parameters. Figure 5.11 shows the speedup for both these cases. This shows that the screen space partitioned algorithm has better speedup for the

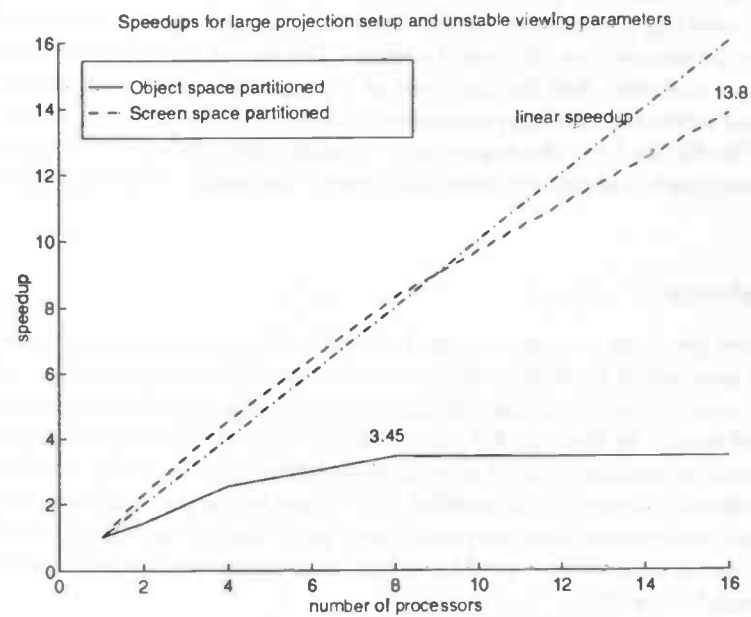


Figure 5.10: Speedup for object and screen space partitioned algorithm for unstable viewing parameters and large projection setup.

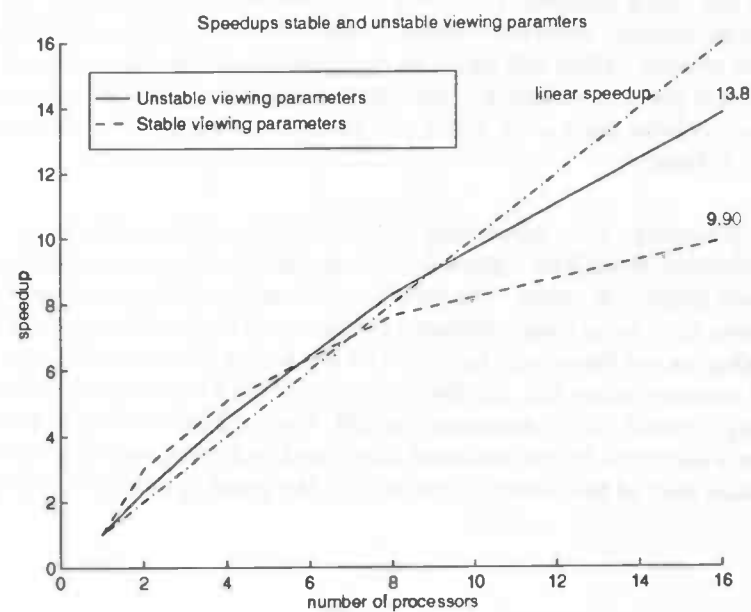


Figure 5.11: Speedup for screen space partitioned algorithm for stable and unstable viewing parameters in the large projection setup.

unstable viewing parameters than for the stable viewing parameters, however tables 5.3 and 5.4 show that the rendering time for stable viewing parameters is much smaller than for unstable viewing parameters, which is due to the optimisation of NaS/RVS for stable viewing parameters as discussed earlier. The larger speedup for unstable viewing parameters indicates that the first part of the parallel contour algorithm, the part that is executed when the viewing parameters change, scale better than the last part of the algorithm (See figure 5.8). The super-linear speedup has already been explained in the previous paragraph and this applies to the case of the stable viewing parameters as well.

5.5.4 Load balance

When all processors perform an equal amount of work an optimal situation is created in which the total amount of work is performed in the smallest amount of time. When there is great variance in the work load of the processors the parallel execution time will be larger than would be the case for an equally distributed work load. Improper load balance affects the performance of a parallel algorithm significantly. In the table containing the rendering times for the parallel algorithms in the previous sections, the minimum and maximum times that the processors spent during rendering have been listed. These two times will now be used to define an estimate for the load imbalance of each of the parallel algorithms.

Estimate Using the minimum and maximum time that the processors spent rendering an image we define the following estimate for the load imbalance:

$$imbalance = 2 \frac{t_{max} - t_{min}}{t_{max} + t_{min}}$$

Thus *imbalance* is the value obtained by taking the difference between t_{max} and t_{min} and dividing it by the mean of these two values. Large values of *imbalance* indicate poor load balance while smaller values will result for a parallel algorithm that has good load balance. This value is just an estimate for the load balance because only the minimum and maximum values have been used, but it will indicate poor load balance when the value of *imbalance* is large.

Screen space partitioning It is interesting to take a look at the load balance of the screen space partitioned algorithm. Figure 5.12 shows the load imbalance for both the large and the small projection setup. The simple partitioning of the screen in equally sized parts accounts for a large load imbalance for the small projection setup and thus for the low speedup as we have seen before. The rendering times for this case vary from 0.19 to 1.13 seconds (table 5.2). On the contrary, the load imbalance for the large projection speedup is small, with a maximum of 0.34. The load balance is good because 98% of the screen is occupied by the rendered object and each processor is responsible for an equally sized part of the screen. This reflects the good speedup of 13.8 for 16 processors.

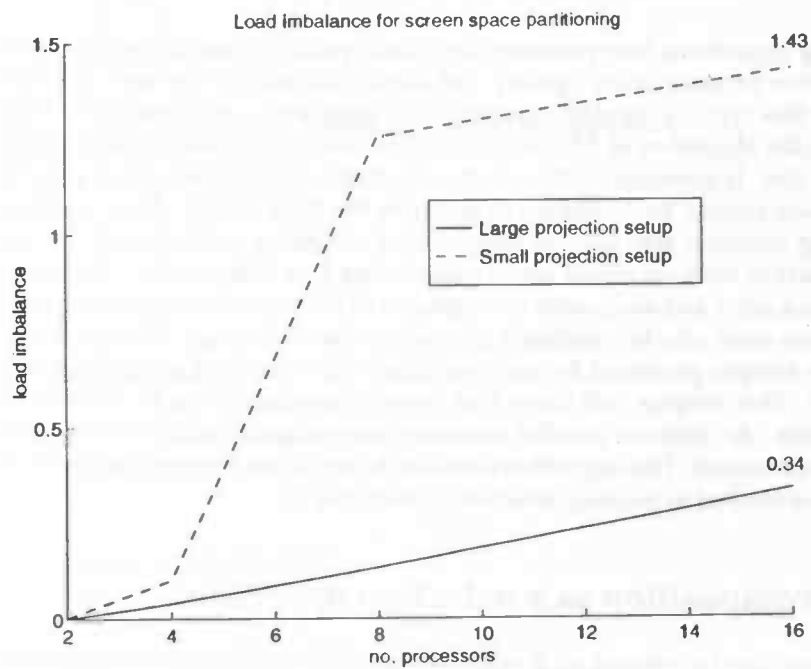


Figure 5.12: Load imbalance for the screen space partitioned algorithm for both small and large projection setup.

Chapter 6

Parallel Image Composition

Parallel rendering algorithms that partition the object space and use an image-order rendering algorithm produce color, opacity and depth information for each pixel at each processor. This type of parallel visualisation algorithm is very common. The parallel contour plot algorithm of NaS/RVS in which the object space is partitioned is an example of this. It generates a color and a z-buffer value for each pixel at each processor and these should be combined to generate the final image. Most parallel volume rendering software also uses an image-order rendering method such as ray casting in conjunction with an object space partitioning (see chapter 10). These algorithms generate a color and an opacity for segments of the ray for each pixel on each processor and these must also be combined to produce the final image. The operation of combining the images produced by each processor into one final image is called *image composition*. This chapter will show that image composition can be viewed as a *reduction operation*. An efficient parallel reduction algorithm called the *binary-swap* algorithm will be presented. This algorithm was used to implement image composition in [19], and will be applied to parallel reduction in this chapter.

6.1 Image composition as a reduction operation

Image composition can be viewed as a *reduction operation* in general. A reduction operation combines two elements of a certain type into one element of the same type using a *reduction function*. For example, addition of integers can be regarded as a reduction operation where the reduction function is addition (Figure 6.1). Integer addition (a) *reduces* two integers into one by adding their values. The reduction function can be extended to act on sequences of elements. In our example this would be the addition of two vectors (b) in which the addition is performed elementwise. This can of course be extended to more than two dimensions. The reduction function now works on two vectors but it can just as well be defined on more than two vectors (c). We must however keep in mind that the order in which the reduction operation is applied is important when the reduction operator is not commutative. In the case of a commutative operator the order in which the reduction operation is applied is unimportant.

So how does image composition fit into this framework? Suppose we have color and depth information for each pixel of an image (sample declarations are in 6.2). Combining images of this type involves comparing the depth values. Suppose the depth values are larger when a point is further away from the viewer, then a pixel at (x, y) will obscure a pixel at the same position with a larger depth value. The reduction

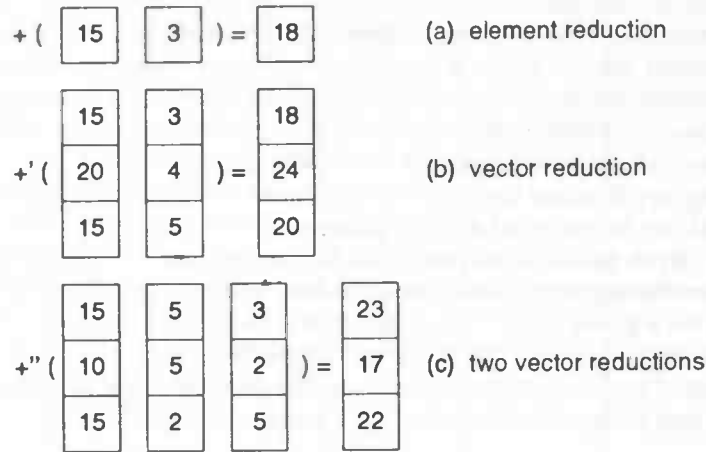


Figure 6.1: Addition as a reduction operation.

```
typedef struct {
    int    color;
    double depth;
} PIXEL;

#define IM_SIZE_X 512;
#define IM_SIZE_Y 512;

PIXEL image1[IM_SIZE_X][IM_SIZE_Y];
PIXEL image2[IM_SIZE_X][IM_SIZE_Y];

void depth_combine(x, y)
int x, y;
{
    if ( image1[x][y].depth > image2[x][y].depth ) {
        image1[x][y].color = image2[x][y].color;
        image1[x][y].depth = image2[x][y].depth;
    }
}
```

Figure 6.2: Sample declarations and routine for image composition.

function, that combines the two images pixelwise (*image1* and *image2*) and stores the result in *image1*, can thus be defined as shown in figure 6.2.

So combining images of color and depth is a reduction operation that acts on two-dimensional arrays of pixels and combines these elementwise using the *depth_combine* reduction function declared above. When a parallel computer contains an image at each processor of the system these can be combined on one processor by fetching an image from a different processor and combining this image with the local image. When the images of all processors have been combined, the final image is available.

In volume rendering applications that use ray-casting as the rendering method, a segment of the casted ray is produced at each processor. This ray has a color, an opacity and boundary depth values associated with it. To combine these segments from different processors the segments should be combined in depth order to produce the correct color value for a given pixel. This is due to the fact that the operator that combines color and opacity tuples is not commutative. To produce the correct result the reduction operation should fetch the images in depth-sorted order from the processors. When this condition is met the result of the reduction operation will be correct.

6.2 Efficient parallel reduction

We have shown that image composition can be described as a reduction operation. This section will present an efficient parallel reduction algorithm that can be used to implement image composition.

Simple parallel algorithm The simple parallel reduction algorithm outlined in the previous section fetches an image from each processor and combines these images with the local image. When all images have been combined, the final image is available at the combining processor. This scheme is shown in figure 6.3. To analyse the differences between the algorithms shown in this section we will assume the following: the images that need to be combined are square images with d elements (pixels). The processor time needed to combine d elements of one image with d elements of another image is assumed to be linear in the number of elements and equal to $r(d)$. The time for communicating d elements over the interconnection network between the processors of the parallel computer is assumed to be linear in the number of elements and equal to $c(d)$. When an algorithm is executed in parallel the execution time of that algorithm is defined to be the maximum of the execution times of all individual processors. The number of processors in the parallel system is denoted by p . Example: each step of the algorithm in figure 6.3 takes $c(d) + r(d)$ time. With four processors there are three steps so the total execution time of the algorithm is $3(c(d) + r(d))$. For p processors the total execution time T_{simple} , which depends on p and d , is:

$$T_{simple}(p, d) = (p - 1)(c(d) + r(d))$$

The conclusion is that the time necessary to combine p images is linear in p . The problem of this algorithm is that only one processor combines all the elements and that the communication network is not efficiently used because in each step there is communication between only two processors.

Tree reduction algorithm An improvement on the simple algorithm is targeted at better load balance and better utilisation of the interconnection network. This is the algorithm that is used in the NEC mini-MPI library on the Cenju-3. The tree algorithm (figure 6.4) makes better use of the processing power of the nodes in a parallel system

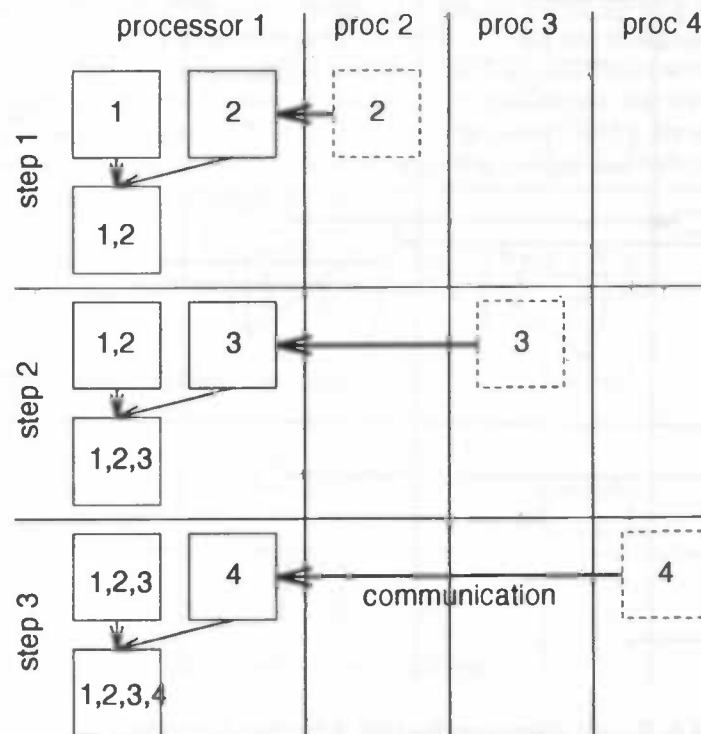


Figure 6.3: Simple reduction algorithm.

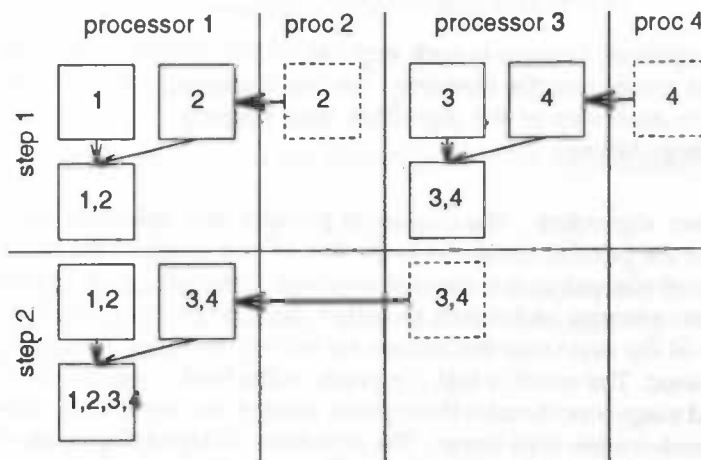


Figure 6.4: Tree parallel reduction algorithm.

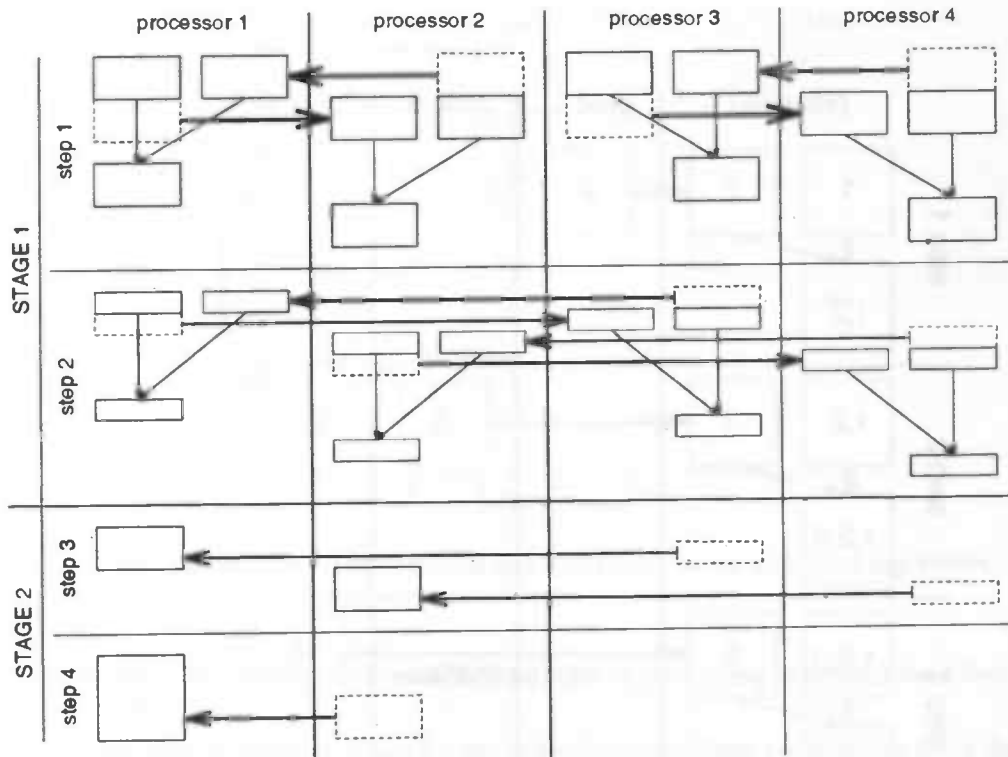


Figure 6.5: Binary-swap parallel tree reduction algorithm.

and utilises the network better than the simple algorithm. The execution time of this algorithm is not linear in the number of processors p , but logarithmic in p . Assuming that the number of processors p is a power of 2, then the number of steps in this algorithm is $^2\log(p)$. Each step takes $c(d) + r(d)$ time so the total execution time $T_{tree}(p, d)$ is:

$$T_{tree}(p, d) = ^2\log(p)(c(d) + r(d))$$

This algorithm is not optimal, because in each step half of the processors become idle, while the other half is combining the elements. We can improve on this situation by splitting the images in each step of the algorithm thus keeping all processors busy during most of the computation.

Binary-swap reduction algorithm The improved parallel tree reduction algorithm utilises *all* processes of the parallel computer in the first of two stages of the algorithm. The algorithm will be explained as it is applied to image composition. It is based on a 'divide-and-conquer' strategy and it will be called the *binary-swap* algorithm from now on. In each step of the first stage the images are halved, and then sent to another processor to be combined. The result is that d/p pixels of the final image reside at each processor. The second stage concatenates these parts, using a tree algorithm. After this stage one processor contains the final image. The algorithm is depicted graphically for four processors in figure 6.5. In the first step processor one and two exchange half of the image. Processor one gets the upper half, and processor two the lower half. The same is done for processors three and four. All processors then combine the received half of the image with their own half of the image. In the next step they again exchange a part of the image with a processor that has the same part of the image. After $^2\log(p)$ steps,

each processor holds d/p elements of the final image. The second stage concatenates the parts of the individual processors. These parts are sent to processor one (or another processor chosen to collect the final image) in a specific order using a tree algorithm which again needs $2\log(p)$ steps. So both the first and the second stage have $2\log(p)$ steps. In the first stage communication and reduction is performed while the second stage only involves communication. Looking at processor one we see that the time spent in the first step is equal to $2c(d/2) + r(d/2)$. Because c and r are assumed to be linear this is equal to $c(d) + r(d)/2$. In each step the images are halved so the total execution time $T_{stage1}(p, d)$ of stage one is:

$$T_{stage1}(p, d) = \left(\sum_{i=1}^{2\log(p)} \frac{1}{2^{i-1}} \right) c(d) + \left(\sum_{i=1}^{2\log(p)} \frac{1}{2^i} \right) r(d)$$

The summations have the following closed forms:

$$\sum_{i=1}^{2\log(p)} \frac{1}{2^{i-1}} = \frac{2(p-1)}{p}$$

$$\sum_{i=1}^{2\log(p)} \frac{1}{2^i} = \frac{p-1}{p}$$

Replacing the sums with their closed forms we obtain:

$$T_{stage1}(p, d) = \frac{p-1}{p} (2c(d) + r(d))$$

For the second stage of the algorithm we find the following execution time:

$$T_{stage2}(p, d) = \frac{p-1}{p} c(d)$$

The total execution time $T_{binary_swap}(p, d)$ of the algorithm is thus equal to the execution time of stage one added to the execution time of stage two:

$$T_{binary_swap}(p, d) = \frac{p-1}{p} (3c(d) + r(d))$$

By taking the limit for $p \rightarrow \infty$ the execution time for $T_{binary_swap}(p, d)$ becomes constant:

$$\lim_{p \rightarrow \infty} \frac{p-1}{p} (3c(d) + r(d)) = 3c(d) + r(d)$$

The execution time of the tree algorithm does not have this property because the limit for $p \rightarrow \infty$ does not exist. The time needed in the tree reduction algorithm will grow as more processors are used.

6.3 Performance

The theoretical performance of the tree and the binary-swap algorithm have been plotted in figure 6.6. For $c(d)$ and $r(d)$ the following values have been computed for combining two 512^2 images: $c(d) = 0.1215$, $r(d) = 0.4450$. These figures show that the interconnection network of the Cenju-3 outperforms the processing elements for

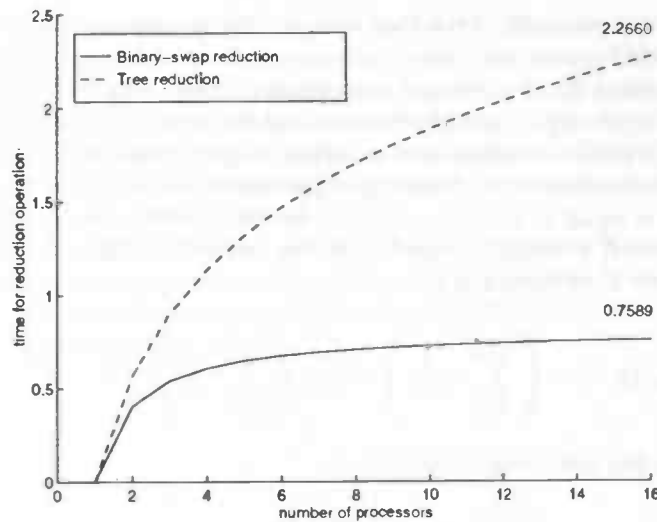


Figure 6.6: Theoretical performance of tree and binary-swap reduction algorithm.

this operator. The maximum execution times (for 16 processing elements) have been written in the figure.

The measured execution times for the reduction algorithms are different from the theoretical execution times and in fact will not converge to a maximum value for the binary-swap algorithm because of the assumptions made about the linearity of $c(d)$ and $r(d)$. In practice these functions are not linear. Especially the communication time will not be linear in the number of elements communicated. For a small number of elements the relative overhead in the communication will be larger than for a large number of elements. As the number of processors increases while the value of d is constant, the algorithm communicates smaller amounts of data towards the end of stage one and in the beginning of stage two. The communication overhead in these parts will limit the performance of the algorithm.

The measured performance of the tree and the binary-swap reduction algorithms is shown in figure 6.7. For each algorithm there are two curves. One for the reduction of two 512^2 images, and one for the same image size but a reduction operator that performs the reduction twice. This is used to indicate that for a more complex reduction operator the binary-swap algorithm will scale better than the tree algorithm. By comparing figure 6.7 and 6.6 we see that the predicted performance of the tree reduction algorithm is worse than the measured performance. This is due to the computation of $c(d)$, which was computed using the mini-MPI library, while the mini-MPI reduction function uses the low-level Paralib/CJ point-to-point communication routines which perform better than the mini-MPI point-to-point routines.

The problem of the number of elements to be communicated becoming too small to maintain efficient communication occurs when the total number of elements to be reduced is small compared to the number of processors. This is the case for image composition where the number of elements will most certainly not exceed 1024^2 . The reduction algorithm will be efficient if the number of elements d divided by $^2\log(p)$ is sufficiently large to achieve efficient communication.

A problem that arises when using a reduction operation for image composition is the sparsity of the images. When using reduction, all pixels in an image are combined even

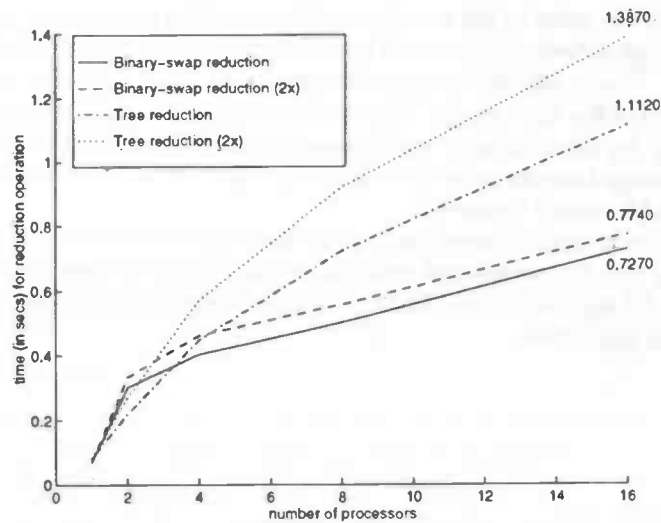


Figure 6.7: Measured performance of tree and the binary-swap reduction algorithm.

when maybe only a few pixels are active¹. When each of many processors is rendering only a small part of the primitives in a scene the number of active pixels tends to be small when compared to the image size. This means that the reduction operation will involve a lot of unnecessary element reductions. Michael Cox and Pat Hanrahan [15] identified this problem and proposed the 'distributed snooping algorithm' for image composition. This algorithm requires the parallel machine to have a globally shared bus on which active pixels can be sent to the framebuffer. When a pixel is sent to the framebuffer over the bus, each processor 'snoops' the pixel from the bus and compares it to the depth of the same pixel on the local processor. When this depth is larger than the depth of the pixel on the bus, that pixel will not be sent to the framebuffer. In this way the bandwidth required to send pixels to the framebuffer is limited resulting in better compositing performance.

6.4 Implementation

The key step in implementing the binary-swap algorithm is determining with which processor to exchange elements. Determining the 'partner' for exchange is based on the binary representation of the rank of the processors. Assume we have a parallel computer with four processors. Each processor has a rank from zero to three. The binary representation of the ranks is 00, 01, 10 and 11. In the first step of the algorithm the processor with the least-significant bit (right-most bit) set to zero, exchanges half of its elements with the processors which has that bit set to one. So to determine a processors partner in communication, we can OR the rank of the processor with a mask. In the first step the mask would be 01. So processor 0 communicates with processor 1 in the first step, and processor 2 with processor 3. In the second step the mask is shifted left one position and becomes 10. So in this step processor 0 communicates with processor 2, and processor 1 with processor 3. The net result of stage one is that each processor holds d/p scanlines of the final image. Stage two concatenates the parts of the final image on processor zero. Instead of simply concatenating the parts of the final image in rank order we must calculate the order in which the images should be concatenated

¹Inactive pixels have the background color and depth.

since stage one shuffles the order of the result in such a way that processor i holds the j th part of the final image where j is obtained by taking the binary representation of i and mirroring the bits. Thus with eight processors, processor 6 (=110 binary) holds the third (= 011 binary) part of the final image. The second stage of the algorithm uses this mirroring of the binary representation of each processors rank to determine where to get parts of the final image from in order to concatenate them to processor zero using a tree algorithm as can be seen in figure 6.5.

The stripped down code for the binary-swap reduction algorithm is shown in figure 6.8. This code assumes that the number of processors p is a power of two and that the number of elements to be reduced is a multiple of $2^{\log(p)}$. This eliminates special cases and leaves the essential algorithm.

```

memcpy(recvbuf, sendbuf, count*extent);
mask = 0x1;
blen = 0;

/* Stage one:
 * This is the actual reduction operation. It results in
 * each processor having count/nr_procs elements of the final buffer.
 * The loop is executed 2log(nr_procs) times.
 */
while ( mask < nr_procs ) {
    count /= 2;
    if ( (mask & rank) == 0 ) { /* bit set to 0 => receive */
        partner = ((rank | mask) + root) % nr_procs;
        MPI_Recv(buffer, count, datatype, partner,
                 SWAP_TAG, coll_comm, &status);
        MPI_Send((char*)recvbuf+(count*extent), count, datatype,
                 partner, SWAP_TAG, coll_comm);
    } else { /* bit set to 1 => send */
        partner = ((rank & (~mask)) + root) % nr_procs;
        MPI_Send(recvbuf, count, datatype, partner,
                 SWAP_TAG, coll_comm);
        MPI_Recv(buffer, count, datatype, partner,
                 SWAP_TAG, coll_comm, &status);
        memcpy(recvbuf, (char*)recvbuf+(count*extent), count*extent);
    }
    (*reduction_func)(buffer, recvbuf, &count);
    mask <= 1;
    blen++;
}

/* Stage two:
 * Now each processor holds count elements. It is now time to
 * transfer those partial results to the root processor. This is just a
 * gather operation which uses a tree algorithm.
 * The loop is executed 2log(nr_procs) times.
 */
mask = 0x1;
rank = bitmirror(blen, rank);
while ( (mask & rank) == 0 && mask < nr_procs ) {
    partner = (rank | mask);
    if ( partner < nr_procs ) {
        partner = (partner + root) % nr_procs;
        MPI_Recv((char*)recvbuf+(count*mask*extent),
                 count*mask, datatype, bitmirror(blen,partner),
                 CONCAT_TAG, coll_comm, &status);
    }
    mask <= 1;
}
if ( mask < nr_procs ) {
    partner = ((rank & (~mask)) + root) % nr_procs;
    MPI_Send(recvbuf, count*mask, datatype,
             bitmirror(blen,partner), CONCAT_TAG, coll_comm);
}
}

```

Figure 6.8: Stripped down version of the binary-swap reduction algorithm.

Part III

Message Passing Interface

THE UNIVERSITY OF CHICAGO LIBRARY

1000 S. EAST ASIAN LIBRARY

CHICAGO, ILL. 60607

TEL: 773-936-5000

FAX: 773-936-5000

WWW.CHICAGO.EDU

LIBRARY

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

CHICAGO, ILL. 60607

Chapter 7

MPI Standard: An Overview

During the last years more and more distributed memory parallel computers have become available. The processors and the interconnection networks for these machines have become faster and faster. The distributed memory parallel computer is slowly catching up on the plain vector super computers¹. To keep this development going, stable and standardised software is needed to program these computers. Most of the parallel systems are (on the lowest level) programmed using the message-passing paradigm of parallel programming and this is the level where standardisation should be accomplished to be able to increase the portability of parallel software. The Message Passing Interface (MPI) is an attempt to standardise a library for message-passing programming. It has been designed by the Message Passing Interface Forum (MPIF) building on the experience gained in the design and implementation of many different message-passing libraries (NX, Express, Vertex, PARMACS, Zipcode, Chimp, PVM, Chameleon and PICL).

The main advantage of establishing a message-passing standard is portability and ease-of-use. Once all vendors of concurrent computers support the standard, writers of parallel software can easily port their software to new machines and program a new machine without going through the pain of learning yet another message-passing library with its own specific features and pitfalls.

The goal of the Message Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing. The standard will be introduced below and consists of the following 7 sections:

- Point-to-point communication
- Collective communication
- Groups, contexts, communicators, and caching
- Process topologies
- Environmental management
- Profiling interface

The following is an introduction to the MPI sections containing an introductory text on the features of a specific group of functions in MPI. Each section will end with a discussion of the applications for the functionality presented in the section. This will show the importance of the provided functionality

¹The current trend is a parallel computer with vector processor nodes, which combines the best of both worlds.

7.1 Point-to-point communication

All message-passing libraries contain the *send* and *receive* mechanism, so does MPI. For these simple message-passing operations a *message envelope* is defined consisting of

source, destination, tag, communicator

Source and destination specify from where to where a message travels. Tags can be used on the receiver side to select a message from others with different tags, this might be used to distinguish between different types of messages. A communicator specifies the *communication context* for a communication operation. Each communication context provides a separate *communication universe*. Messages are always received in the context they were sent in. Thus messages sent in different contexts do not interfere with each other. A *send* executed in a specific context will not be matched by a *receive* in a different context. The communicator also specifies the *process group* that shares this communication context. The process group is an ordered collection of processes that are identified by their rank in the group. A predefined communicator has a process group associated with it that contains all processes available after MPI initialisation. All processes can thus be identified by their rank in this predefined group.

7.1.1 Communication modes

Before we continue the following definitions are necessary: A MPI function is called *local* if the completion of the call does not depend on communication with another process. A MPI function is called *non-local* if its completion depends on communication with one or more other processes.

Each send operation can be performed in four different modes.

standard, buffered, synchronous, and ready mode.

In *standard* mode it is up to MPI to decide whether or not to copy the message to a buffer to be sent later or to block the process waiting for a matching receive and the completion of the communication. A standard mode send is a non-local function, if the message is unbuffered communication with the receiver is necessary to check if a matching receive has been posted.

In *buffered* mode the message is unconditionally buffered and sent when a matching receive is posted. When using this mode the send function call may complete before the message is actually sent. This mode of sending messages is local. There is no need to communicate with the receiver to complete the send operation. In MPI the user can allocate a buffer of a specific size for the buffering of the messages sent in this mode.

A send that uses the *synchronous* mode can be started whether or not a matching receive has been posted. The send will not complete until the message is being received and the send buffer can safely be reused. The synchronous send is obviously a non-local operation.

A send using the *ready* mode may be started only if the matching receive is already posted. When using this mode, the programmer provides MPI with the information that the matching receive has already been posted. This might reduce overhead. The ready mode send is a local operation.

These modes only apply to the send operation in MPI. There are no corresponding semantics for the receive operation.

7.1.2 Blocking and Non-blocking communications

Communications can be of two types: *blocking* or *non-blocking*. A blocking communication does not complete until the contents of the message buffer can be reused in the case of a send or is available in the case of a receive. The initiation and the completion of a communication are united in one function.

A non-blocking communication separates the initiation and the completion of a communication. A non-blocking send for example initiates the send, meaning that the message buffer contains the data to be sent. After this initiation the MPI system may send the contents of the buffer to the receiver. Separate functions can wait for the communication to complete or check if the communication has been completed. Until a program has the confirmation that the communication has actually been completed the contents of the message buffer may not be changed. The use of non-blocking communications can greatly improve performance of parallel programs by allowing communications and computations to overlap. This is done by initiating a send as soon as possible and wait for the completion not earlier than the completion is required by the semantics of the application.

7.1.3 Persistent communications

Often, a communication operation with the same message envelope is executed within a loop. In this situation one can use a *persistent* communication in MPI. After a setup call that initialised the message envelope, a new message can be sent and completed using this message envelope. In this way the communication operations within a loop may be optimised by reducing overhead.

7.1.4 Derived data types

In most message-passing systems only contiguous blocks of basic typed elements (character, integer, real) can be used in communication. This scheme is too restricted if one for example wants to send a section of a 2D or 3D array. To enable these kind of messages MPI defined *derived data types* which enable the communication of non-contiguous data or mixed type data in a single message. From the basic data types such as integer, character and double one can construct derived data types using four different constructors.

contiguous, vector, indexed, and struct.

The *contiguous* data type constructor takes a (basic or derived) data type and defines a new data type that consists of a number of copies of the old data type concatenated together. For example: to send 100 elements of an array of integers one can call the send operation with data type integer and number of elements equal to 100, or one can construct a new contiguous data type consisting of 100 integers and use this new data type in a send with number of elements equal to 1.

The *vector* constructor allows the construction of a data type that consists of equally spaced blocks of a number of copies of the old data type. A section of a 2D array can be modeled using this constructor.

The *indexed* constructor allows replication of an old data type into a sequence of blocks, where each block can contain a different number of copies of the old data type and have a different displacement relative to the first element. Using this constructor one can for example define a data type that contains the upper or lower triangular part of a matrix.

The *struct* constructor is the most general constructor. In addition to the blocks with different sizes and displacements each block can consist of replications of different data

to\ from	one	all
one	send/receive	gather
all	bcast/scatter	allgather/alltoall/allreduce

Table 7.1: Relation between the different (collective) communication routines.

types. If one wants to communicate a record from a database this constructor can be used to model the contents of this record.

Using derived data types expressive communication statements can be formulated that simplify programming when complex data types are involved.

7.1.5 Pack and unpack

The derived data types introduced in the previous section can be used to efficiently communicate non-contiguous messages. The data is directly taken from the non-contiguous buffer by the message passing system. This way no explicit packing or unpacking is necessary. The pack and unpack functions in MPI are provided for compatibility with other libraries and provide some additional functionality that is not otherwise available in MPI. The pack and unpack functions can be used to receive a message in parts in case the first part of the message determines the way in which the rest of the message is handled. Another use for the pack and unpack functions in MPI is for efficient explicit buffering of messages thus bypassing the system buffering.

7.2 Collective Communication

Collective communications are communications that involve a group of processes. All collective communication functions in MPI need to be called by all processes in a group and with matching arguments. When we refer to 'all processes' in the following we mean all processes belonging to the group in which the collective communication is performed. The group in which a collective communication is executed is taken from the communicator context in which the collective communication is performed. Collective communications whose semantics ask for a single process with a special task (such as a gather operation in which one process receives and the rest sends) take the logical rank of that process as an argument. Such a process is called the *root process* and has *root rank*. Depending on the value of this rank a process decides which actions it has to perform (send or receive). The root process need not be the same process every time. A broadcast for example can be initiated by any process. The root process is the initiator of the broadcast.

The collective communications supported in MPI are: barrier synchronisation, broadcast, gather, scatter, global reduction, combined reduction and scatter, and scans (also called prefix operations). All collective communication functions allow derived data types to be used as type for the items in the send/receive buffers. The relation between the collective communication functions is shown in table 7.1.

7.2.1 Barrier synchronisation

The *barrier* synchronisation function is called by all processes in a particular group and blocks the caller until all processes in that group have called the function. It can be used to ensure that all processes in a group have reached a certain point in their computation.

7.2.2 Broadcast

The *broadcast* function sends a message from the process with the specified rank to all members of a group including itself. It can be used for replication of data to all processes in a certain group.

7.2.3 Gather, scatter, allgather, and alltoall

This group of functions is used for communications concerning distributed data. There are three categories: all-to-one, one-to-all, and all-to-all.

The *gather* function is an all-to-one operation in which each process (including the root process) sends an equally sized block of data to the root process which stores these blocks in rank order for the group in which the operation was executed.

Scatter is the inverse of *gather* in which the root process sends an equally sized block of data to each process including itself. Using this function one can distribute data that is located in the local memory of the root process.

The *allgather* function is equivalent to the *gather* function with the difference that all processes receive the result. This is logically equivalent to a *gather* operation to a certain process followed by a *broadcast* from the same process to all other processes.

The *alltoall* function is an extension to the *allgather* function where each process sends distinct data to each other process including itself. This is logically equivalent to a *scatter* or *gather* operation executed by every process.

All functions in this section have a 'vector' counterpart in which the length of the buffer can vary from process to process as well as the displacements on the sending and receiving side. This allows non-contiguous data to be send using the collective communication functions.

7.2.4 Reduction operations

A reduction operation is an operation that combines data located at different processes into one result using a certain combine operation. One can for example combine a number of integers, one at each process, using summation as the operator. This will result in the sum of the integers at the root process. The data type of the elements to be combined determines which operations are valid. One can define a new reduction operation which is useful to combine elements of a derived data type for which no predefined operations exist, or for defining a new operation on a basic (non derived) data type.

7.2.5 Reduce, allreduce, and reduce-scatter

The *reduce* operation is the basic reduction function. It can combine data consisting of a single element at each process or combine data consisting of multiple elements at each process in which case the combine operation is performed element-wise. For example, an array of integers located at each process can be combined element-wise resulting in a single array of the same length at the root process in which each position contains the result of the combination of the elements of that position at each process. The result is stored at the root process. The following operations are predefined:

- Maximum, minimum, sum, and product.
- And, or, and xor, all logical and bit-wise.
- Maximum or minimum value and location.

process:	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
values:	1	8	2	3	4	8	2	9
result:	1	9	11	14	18	26	28	37

Figure 7.1: Prefix scan using summation as the operator.

Not all operations are defined for all basic data types. The valid combinations are specified on page 114 of [22].

Allreduce is equivalent to reduce but the result is received by all processes. This is logically equivalent to a reduce operation followed by a broadcast from the same root process.

The *reduce-scatter* operation is a reduce operation where the result is scattered to all processes. This function performs an element-wise reduction operation on the data elements after which segments of the result can be sent to each process by performing a vector scatter.

7.2.6 Scans

The *scan* operation is used to perform prefix reduction operations on the data at the processes in a group. As an example we will compute the scan of an array of integers using sum as operator. Figure 7.1 shows that the prefix reduction on process i includes the value on process i , this is a so called *inclusive* scan. An alternative is that the result on process i only includes the values from processes up to and including process $i - 1$ in which case the operation is called a *exclusive* scan. In MPI only the *inclusive* scan operation is provided (see rationale on page 125 of [22]). The scan operation can be used with all the operators listed in the previous subsection. The user-defined operators can be used as well for the scan operation.

7.2.7 User defined operations

Apart from the predefined reduction operations such as maximum and minimum the user can define his or her own reduction operation. This is done with the *operation create* function which binds a user-defined operation to a reduction operation handle that can be used in calls to the reduction functions. When creating a new operation it is assumed that the operation is associative and the user specifies whether the operation is commutative. If an operation is not commutative the order of operations is fixed and defined to be in ascending process rank order beginning with process zero.

7.3 Support for libraries

One of the main aspects when defining a standard for writing message-passing programs is to enable the development of *parallel libraries*. These libraries would encapsulate implementation details of key algorithms such as matrix-vector operations for example. To enable parallel libraries the message-passing system should provide features such as a safe communication space in which communications within the library will not interfere with communications in the user program. This is called a *context* of communication. Another feature would be the notion of *process groups* for collective operations that would eliminate the need for synchronisation with unrelated processes. An abstract naming scheme or *virtual topology* would be beneficial in describing communications for a program in terms of their data-structures and algorithms. Finally features to extend the message-passing system with user-defined collective operations

would be needed to enable user-defined collective communications. The mechanism used for this in MPI is called *attribute caching*. In addition a unified mechanism is needed to encapsulate the mentioned features. This mechanism is called *communicator* in MPI. The following subsections will describe MPI's support for writing parallel libraries.

7.3.1 Communicators

The *communicator* encapsulates all support for writing parallel libraries. Amongst others it contains pointers to the group, communication contexts, cached attributes and virtual topologies. There are two types of communicators namely *intra*-communicators for operations within a single process group and *inter*-communicators for point-to-point communication between two process groups. All MPI routines require a communicator to be passed to it. This communicator provides the communication context and group for the MPI communication routine. At initialisation a predefined communicator exists which contains a group of all available processes. The following information is contained in a communicator:

Process groups. A group is an ordered set of process ranks. Each process in a group is associated with an integer rank. These ranks start from zero and are contiguous. A group is part of a communicator and to create a new group, a new communicator should be created with which this group is associated. MPI provides functionality to query group information, construct new groups from previous ones by taking the union, intersection or difference of two groups, or by explicitly listing the ranks of the processes that should belong to the new group.

Communication contexts. A context is a property of a communicator that divides the communication space in separate disjunct subspaces. A communication performed in one context will not interfere with a communication in a different context. More specifically, a receive in one context will not match a send executed in a different context. Furthermore, collective communications will never interfere with point-to-point communications, they are executed in different contexts. Contexts are not explicit objects in MPI, they are implicitly defined in communicators. Communicators enable the development of efficient parallel libraries by eliminating the need for synchronisation of parallel processes on library entry. Traditionally, one would synchronise all processes before entering a library call to prevent user-program communication to interfere with library communication. By creating a new context when entering a library the need for synchronisation is eliminated because the communications in the new context (the library context) will not interfere with user-program communication.

Attribute caching. The ability to attach arbitrary pieces of data to a communicator in MPI is called *attribute caching*. This can be used to pass information between calls by associating it with a communicator. There are functions to attach the information to the communicator and subsequently retrieve that information from the communicator. It is guaranteed that out-of-date information is never retrieved even if the communicator is freed. Attribute caching can be used for example to store information that is computed the first time a user-defined collective communication routine is called. In subsequent calls to that routine this information can be retrieved without the need to recompute. This allows for a more efficient implementation for that user-defined function. MPI provides functionality to create attributes, assign a value to an attribute, retrieve the

information stored for an attribute and delete an attribute that has been associated with a communicator.

Virtual process topologies. The virtual process topology defines a special mapping of the ranks in a group to and from a user defined topology. It will enable the programmer to define the communication in an application in terms of the data-structures and algorithms as well as possibly aid in the efficient mapping of the processes to processors in the parallel computer.

One can for example define a virtual topology in which the processes are arranged in a grid with each process having a two dimensional rank. Sending a message from one process to the other can now be done by using the two coordinates of the rank instead of the linear rank that would have been used otherwise. This information might be used by the run-time system of the MPI implementation to efficiently map the processes onto a grid of processors.

Virtual process topologies can only be used with intra-communicators. MPI provides functionality to create cartesian and graph topologies. A cartesian topology is defined by the number of dimensions and the number of processes in each dimension. The resulting grid of processes may or may not be periodic in any dimension. The graph topologies are specified by the number of processes in the graph and the edges between the nodes. Various inquiry functions are defined for the cartesian and graph topologies such as retrieving adjacency information to determine the neighbours of a process.

7.3.2 Intra-communicators

Intra-communicators are used for communication within a single process group. They contain an instance of that group, contexts for point-to-point and collective communications and the ability to contain virtual topology and cached attributes. MPI provides functionality to construct new intra-communicators, compare communicators, retrieve information from a communicator, duplicate communicators and split communicators in two communicators with associated groups.

Process groups. The group of an intra-communicator defines the participants in the communication that uses the communicator.

Communication contexts. The context differentiates messages belonging to different communication universes. This could be implemented using tagged messages in which the tag defines in which communication context the message was sent. Point-to-point communications are executed in a different context than collective communication which guarantees that these two types of communication will not interfere.

Attribute caching. Attribute caching defines the local information the user has attached to a communicator for later reference.

Virtual process topologies. Virtual topologies can only be defined on intra-communicators.

7.3.3 Inter-communicators

Communication within a process group is handled using intra-communicators. The communication *between* members of two non-overlapping groups is handled using

inter-communicators. The inter-communicator binds two groups together with communication contexts shared between the two groups. Using inter-communicators it is possible to address a process in a different group using the local rank of that process in that group. MPI provides functionality to create, query, and merge inter-communicators. Only point-to-point communication can be used on an inter-communicator. Collective communications are undefined on inter-communicators.

Process groups. An inter-communicator contains a local and a remote group between which only point-to-point communication is possible.

Communication contexts, attribute caching and virtual process topologies. Communication contexts and attribute caching are defined for inter-communicators as well as for intra-communicators. The virtual process topologies as described in the previous subsection can not be used with inter-communicators.

7.4 Environmental Management

To properly communicate with the environment (hardware and software) the MPI standard defines various environmental variables and functions to communicate with the environment. This section will describe that functionality.

7.4.1 Implementation information

A few attributes are defined on the predefined communicator that exists upon startup of a MPI program. The upper bound for tag values used in communication routines is defined as well as the rank of the process that acts as a host in a host-node parallel computer. This last attribute is undefined if no such architecture exists. The last predefined attribute gives the rank of the process that has regular I/O facilities. By regular I/O we mean standard input/output operations defined by the language in use. There is also a routine that returns the name of the processor the process runs on. In a cluster of workstations this could be the name of the workstation concatenated with the process number of the process on that workstation. In a parallel machine this could be a string containing the physical rank or id of the processor.

7.4.2 Error handling

An MPI implementation may or may not handle some errors. Which errors are handled by MPI is implementation dependent. Errors occurring in MPI routines are handled by an error handler. Error handles in MPI are associated with the communicator that is passed to all MPI routines. Two predefined error handlers are defined, one aborts the program on all executing processes and another predefined error handler returns without any action. A user may define its own error handler and associate it with a communicator. New communicators inherit the error handler from their parent. So if an error handler is associated with the only predefined communicator this error handler is propagated to all children of that communicator. Thus the error handler will be used throughout the application until a different error is associated with one of the child communicators. MPI provides functionality to create an error handler, attach an error handler to a communicator, free an error handler and to retrieve the pointer to the error handler from a communicator.

Another function in MPI can be used to retrieve the error string corresponding to an error return code from an MPI routine.

7.4.3 Error codes and classes

The error codes returned by a specific MPI implementation are left entirely to the implementors except for the return code in case a routine is successful. To reduce the number of different error situations to be handled by an application, MPI introduces *error classes*. A routine is provided to map an error return code to one of a few error classes. This error class can be used by the application to generate an appropriate error message.

7.4.4 Timers

MPI contains two routines that enable timing of (parallel) programs. The first function retrieves the value of the timer in seconds as a real value, which is convenient to measure the performance of a parallel application. The second routine returns the resolution of the first function as the number of seconds between successive clock ticks. If the clock is incremented every millisecond the value returned by the last routine would be 10^{-3} . These routines enable timing of programs and determining the accuracy of the timing.

7.4.5 Starting and terminating parallel programs

One goal of the MPI standard is to enable source code portability. This means that details on how to load parallel programs onto the parallel machine or cluster of workstations or how to stop a parallel program should not be handled by the application program. An implementation may include platform specific initialisation in an initialisation routine. This routine must be called before any other MPI routines can be called. There is one routine that may be called before this initialisation however and this routine checks whether the initialisation routine has already been called. To terminate a parallel program there are two functions defined in MPI: the first one terminates the program 'normally' in the case no error occurred. The last one terminates the program 'abnormally' in the case of a fatal error. If one of these programs is called in an MPI program, no MPI routines may be called afterwards including the initialisation routine.

7.5 Profiling interface

Being able to profile a parallel program is an important feature of any message-passing library since performance measurement and debugging of parallel programs are always issues in parallel processing. An MPI implementation should provide two versions of each routine in the library. One with the name defined in the standard and one prefixed with the letter "P". The normal standard versions can be replaced by wrappers that generate profiling information and call the "P" variants to perform the corresponding MPI routine. The profiling interface of MPI is just the definition of an interface, it says nothing about what information is profiled. This is all determined by the writer of the profiling library.

A special routine is used to provide control over a profiling MPI library. This routine has the level of profiling as an obligatory argument. The meaning of this level argument is profiling library specific except for the three values for which predefined behaviour should be implemented. There is a level for disabling profiling, one for enabling profiling at a normal level of detail and a level that tells the profiling library to flush the profiling buffers. All other levels of profiling are implementation specific.

Chapter 8

Literature study: MPI

Since the MPI1 (initial specification) proposal document [16] various articles on the MPI standard and its implementation have emerged. This section tries to summarise the literature on MPI in various categories. These categories are: introductory literature, literature on writing parallel libraries, literature on extending the MPI standard, literature on additional language bindings, literature on MPI implementations, literature on early application written using MPI and literature on performance of MPI implementations, and profiling and debugging of MPI programs.

8.1 Introductory

An introductory paper on MPI can be found in [11]. This paper introduces the MPI standard and gives an overview of MPI. Groups, contexts and communicators are discussed as well as point-to-point, collective communications and process topologies. Attribute caching and inter-communication are not discussed.

8.2 Writing libraries

An important feature of the MPI standard is that it enables the development of parallel libraries. The importance of this feature has already been identified in the Common High-level Interface to Message Passing (CHIMP) [4] and the Zipcode message-passing system [39]. In [12] the authors of CHIMP state that a common message passing interface is needed but that the portability of such a system should not compromise the performance. To enable parallel libraries secure communication contexts are necessary (MPI, CHIMP and Zipcode). The Parallel Utilities Library (PUL) project of the Edinburgh Parallel Computing Centre (EPCC) has proven that development of portable parallel libraries is possible if the message-passing system contains the necessary support.

Another article on writing libraries in MPI can be found in [40]. This article motivates the need for parallel libraries and identifies the features missing in common message-passing libraries that have been included in the MPI standard. The support for parallel libraries in MPI is then summarised along with guidelines on how to write libraries in MPI. These guidelines are then applied in a case study of a 2D-grid vector class library.

8.3 Extending MPI

The MPI standard contains support for extending the message-passing functionality with for example new collective communications. An example of this can be found in [41]. The authors describes extensions to the inter-communicators in MPI to enable collective communications and virtual process topologies on inter-communicators; in MPI collective communications and virtual topologies can only be used with intra-communicators. This functionality is part of the MPI extension library (MPIX) developed by the authors. They conclude that the collective communication and virtual topologies should be defined on inter-communicators as well to be consistent with the functionality provided for intra-communicators. Final remark is that threads and parallel I/O are the next logical extension to MPI to be investigated.

The next paper of the same authors identifies important extensions to MPI [42]. It gives a summary of concepts that were left from the MPI standard such as active messages, threads, virtual shared memory, parallel I/O, dynamic load balancing, etc. They propose possible implementations of inter-communicator extensions for collective communications, spawning of processes during execution of a parallel program, thread extensions to MPI, interrupt receive calls, remote memory access extensions and an active messages extension.

A thread is a single flow of control with a process. Multi threaded processes have independent threads of control that share the same address space. Since the context of a thread is small, context switches by the scheduler are cheap. Threads can provide better performance in many situations for example in the case of I/O where one thread performs I/O while the other thread continues the computation. The issue of making MPI thread-safe is addressed by [10]. This article discusses the implementation of MPI atop the P4 parallel programming system [5]. To make this implementation thread-safe the P4 system is made thread-safe first. Then the potential thread-unsafe features of MPI such as explicit buffering are discussed.

The MPI standard specification does not address the issues of parallel file I/O. People at IBM T.J. Watson Research Center and NASA Ames Research Center have drafted a proposal parallel I/O interface for MPI called MPI-IO [34]. The goal of the MPI-IO interface is to provide a widely used standard for describing parallel I/O operations within an MPI message-passing application. The interface should establish a flexible, portable, and efficient standard for describing independent and collective I/O operations by processes in a parallel application. The MPI-IO interface has the following features: independent (no coordination between tasks) and collective (each task must participate to the collective access) I/O requests, blocking and non-blocking I/O calls and system maintained individual and shared file pointers. MPI-IO is designed to be as MPI-friendly as possible. When opening a file, a communicator is specified to determine which group of tasks can get access to the file in subsequent I/O operations. MPI derived datatypes are used for expressing the data layout in the file as well as the partitioning of the file data among the communicator tasks. The data access can be performed using explicit absolute or relative offsets or by using the system maintained file pointers. The MPI-IO interface is intended to be submitted as a proposal for an extension of the MPI standard in support of parallel file I/O.

8.4 Language binding

The MPI standard was developed using an object-oriented approach. However interfaces have been defined only for the non-object-oriented C and Fortran 77 languages. In [1] examples of a C++ interface to MPI are being discussed. This interface can be used

to build object-oriented libraries in a language that directly supports object-oriented programming. The article also proposes extensions of the MPI data types and topology functionality by incorporating features of the Zipcode [39] message-passing system. This would simplify creation of data types and make the data types re-sizable whereas they are of constant size in the current MPI standard. The topology functionality could be improved by providing predefined topologies and simplifying the addressing in a topology by eliminating the need to map from topology coordinates to process ranks.

8.5 MPI implementations

Since the completion of the MPI standard in May of 1994 several implementation projects have begun. Four public domain implementations of the MPI standard will be discussed in this section with appropriate references. A workshop on implementing MPI was held at the Argonne National Laboratory of which a report is available in [24]. This workshop was attended by representatives from universities and industries from all over the world including NEC, Intel, IBM, Cray Research, Meiko, and Convex.

8.5.1 MPICH

The MPI Chameleon (MPICH) implementation of MPI is authored by the Argonne National Laboratory and the Mississippi State University [17]. The MPICH implementation is a portable implementation that is based on an Abstract Device Interface (ADI) which may be used to efficiently implement MPI [23]. MPICH is based on this ADI and to port MPICH to a new architecture a core of message-passing routines need to be implemented. A set of extension routines that provide the rest of the necessary functionality are implemented using the core routines. These extension routines can be rewritten for use with a specific hardware and software platform to provide extra performance. This way the initial effort of porting MPICH to a new architecture is small and one has the ability to increase performance in the future. Interesting is that the authors claim the following: "If you find that this implementation of MPI is not at least 95% as fast as what you are currently using on any machine or network, please let us know and we will try to fix it".

More information on the ADI can be found in chapter 9 which describes the abstract device interface for the Cenju-3.

8.5.2 CHIMP

The Common High-level Interface to Message Passing (CHIMP) [4] is authored by the Edinburgh Parallel Computing Centre (EPCC) at the University of Edinburgh in Scotland. Since the inception of CHIMP in 1991 it has evolved from prototype through Version 1 to the current Version 2 interface specification. In recognition of the importance of the MPI effort the current version of CHIMP supports the MPI interface functionality in an MPI compatibility library built directly on top of CHIMP. Apart from the message-passing functionality CHIMP provides a rich environment for executing parallel programs on workstations to automatic placement of processes across a combination of parallel and distributed computing resources.

8.5.3 LAM

Local Area Multicomputer (LAM) [6] is a programming environment and development system for a message-passing multicomputer constituted with a UNIX network that

was developed at the Ohio Supercomputer Center. It is a subset of a greater Trollius [18] system which extends the software to dedicated parallel computer ensembles with no other native operating system. LAM runs only on a network of UNIX machines and presents the topology of the multicomputer as a fully connected graph thus allowing nodes to communicate in 1-hop distances. The latest version of LAM includes APIs for both MPI and PVM. The MPI point-to-point functionality is built directly upon LAM and the collective communication functions, virtual topology mapping, and the group, contexts and caching functionality is implemented on the point-to-point functionality. It is thus independent of LAM.

8.5.4 Unify

Unify [9] is a subset of MPI that has been built on top of PVM. It is based on a computer science master's project [8] of Mississippi State University. Unify is dual-API in which programs can use both MPI and PVM calls but it is also possible to use MPI calls solely. The resulting executable will run in the PVM environment. This subset of MPI has been built to show the relative ease of implementation of MPI, and also to ease migration of code from PVM to MPI. This is important because PVM still is the defacto standard or at least has been the defacto standard for many years.

8.6 Applications

In [43] the authors describe a number of efforts to make use of MPI in real applications. The paper is not a definite statement of MPI development work but describes the initial successes, progress, and impressions of application developers during porting of their applications to MPI. A number of application and application-enabling libraries are discussed by various aspects such as: goals of the application project, parallel formulation/conversion, helpful feature of MPI, most understandable/most frustrating features of MPI, implementation used, and early indications of performance results. The paper concludes that there were only positive comments on porting applications to MPI. The standard document needs to be more understandable and include much more example programs. The variety of machines running a portable version of MPI was extremely interesting. Researchers use machines like the IBM SP-1, CM-5, and nCUBE/2, but also newer machines like the Cray T3D, and less well known machines like the AP-1000.

8.7 Profiling and debugging MPI programs

In [28] the authors discuss the MPI profiling interface and three profiling libraries that make use of it. These libraries are distributed with MPICH. The idea behind the MPI profiling library is to use the linker to substitute "wrapper" functions for the MPI functions called by the application. In order to do this the profiling version of the function must have the same name. The *real* function should then be available by some other name so the profiling version can call it to do the real work. This means that every MPI function should be available by two names, one in case the profiling version is not defined and one that is called by the profiling version. Three profiling tools for MPI are described in the paper. These tools are supplied with MPICH. Two of them rely on the Multi-Processing Environment (MPE) which is also part of MPICH. MPE supplies useful functionality such as event logging and simple graphics. The first profiling tool just accumulates time spent in MPI routines. The second tool generates logfiles

containing time-stamped events that can be used by a variety of profile visualisation tools. The third library does a simple form of real-time program animation. The tool uses MPE to display communication patterns and program output graphically.

The Edinburgh Parallel Computing Centre (EPCC) authored two projects, in the framework of their Summer Scholarship Programme, to develop a visualisation tool for performance analysis and debugging (VISPAD) of parallel programs [45] [38]. The initial version of VISPAD supported the CHIMP message-passing system. The followup project migrated VISPAD from CHIMP to MPI and incorporated additional functionality. VISPAD uses an instrumentation library (INS) for event-logging. The INS library provides structured logging of events in phases, buffered or unbuffered instrumentation, and other functionality to log events from any program and output these events to a so called *tracefile*. An instrumented MPI library, implemented using the MPI profiling interface, is built upon the INS library. Linking an MPI program against this profiling library and running the program will result in tracefiles of the execution being created which can then be examined using the visualisation tool VISPAD. VISPAD contains a navigation display for moving through the (usually large) tracefiles and it controls selection, filtering, and expansion of the structured events. It also controls animation of the execution trace. The membership matrix display visualises which communicators each process belongs to. The communication display visualises the communication events in an MPI program using a visual encoding for each type (blocking, non-blocking) of communication as well as a textual description of the communication. The statistics display provides a list of communication specific metrics for each process. Finally, the profile display presents quantitative information about the various events in a parallel program and uses numeric values as well as a bar graph to display that information.

8.8 Performance of MPI

Already early performance results are available on different implementations on different architectures. This section will present literature on performance results on the IBM-SP1, Intel Delta, Paragon, CM-5, and on workstations clusters.

Performance on IBM-SP1 In an early draft of [25] the authors compare the performance of IBM's proprietary library *euhi* with IBM's experimental MPI-F implementation on the SP1, and the MPICH implementation which are both built on top of the *euhi* library. The performance is measured using a simple round-trip bandwidth test between two nodes of the SP1. The results show that a vendor implementation of MPI (MPI-F) can be as fast as the vendor's proprietary system (*euhi*), and that a public domain implementation can be nearly as fast. The difference in performance between the public domain and the vendor implementation of MPI is due to copying of messages between system and user buffers in the public domain implementation which is absent in the vendor's implementation.

In [2] the performance results for different implementations of the broadcast operation are analysed and compared on the Intel Delta, Paragon, IBM SP1, and the CM-5. The authors recognize that MPI allows software portability, and that it should lead to efficient code since each vendor will implement it for its own hardware. The paper evaluates the efficiency of MPICH by comparing the MPI broadcast collective operation to optimised versions written using both MPI and the native NX message-passing system on the Intel Delta, the Paragon, the IBM-SP1 and the CM-5 where appropriate. The paper concludes that MPI is performing well on many systems, and that MPICH

runs correctly on many systems. However the MPICH implementation still has room for performance improvement.

Performance on workstation clusters In [37] the authors provide a guideline on how to select an appropriate public domain MPI implementation for a workstation cluster, which in their case consists of DEC Alpha workstations connected by a DEC GIGAswitch. They investigate several aspects of MPI, including its functionality and performance, for CHIMP, LAM, MPICH, and Unify. The paper points out the strength and weakness of each implementation. Their results indicated that software overhead in communication is very high and should be reduced. From the four implementations they choose LAM as the best implementation for their environment although it incurs relatively high software overhead for short messages. Unify has good performance, but is not a full MPI implementation. MPICH and CHIMP have memory management problems and have therefore not been chosen.

Chapter 9

Porting full MPI to Cenju-3

Since the completion of the MPI standard a few implementations have emerged. One of these has resulted from a joint-effort project between Argonne National Laboratory (Gropp and Lusk) and Mississippi State University (Skjellum and Doss). We will refer to this implementation as the MSU implementation of MPI. This implementation conforms to the MPI standard and is a full implementation of all the routines specified in the standard. It is also a portable implementation in the sense that it is reasonably easy to put up an implementation for a new machine such as the Cenju-3. This is done by implementing a small machine dependent piece of software called the *Abstract Device Interface* (ADI). This chapter gives a short description of the way in which the MSU implementation of MPI was ported to the Cenju-3 and shortly describes the interface itself, as well as an example of its operation.

9.1 Porting approach

A paper by Gropp and Lusk [23] describes the *Abstract Device Interface* that was used to provide an implementation of MPI. Essentially the interface between the ADI and the *Application Program Interface* (API), in our case the MPI library, is defined and implemented. This interface consists of a number of routines provided by the device, such as simple message-passing routines, and a number of routines provided to the device by the API, such as routines to transfer data between the device and the API. The operations provided by the device are very simple routines such as routines for sending and receiving messages, and routines for message queue management. These simple operations can be used to implement more complex operations such as defined in the MPI standard. However, if the device supports some of these complex operations, they can be used directly instead of implementing them in terms of simple operations. This allows for performance improvement once a rudimentary implementation that does not use the extended support of the device has been completed. One can think of using collective communication support provided by the device as a candidate for this kind of improvement. In this way one can use as much of the features provided by the parallel device to increase the performance of the ADI and thus the performance of the message-passing system (in this case MPI) that is implemented on top of it.

The ADI for the Cenju-3 is based on the NEC mini-MPI library. The mini-MPI library is a subset implementation of MPI that does not conform to the standard completely. To provide a full MPI implementation and minimise the implementation effort, the implementation of the ADI is based on the mini-MPI library. The layers of the full MPI implementation are shown in figure 9.1.

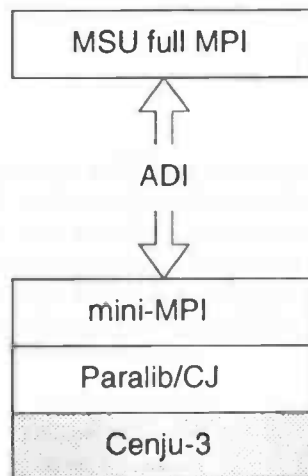


Figure 9.1: Layers of full MPI implementation using ADI.

To support the Cenju-3 device all that was done was to copy the implementation of the p4 (message-passing system) device and modify it to use the mini-MPI library routines. The ADI for p4 provides message queuing and relies on the message buffering of the message-passing system that is used for message-passing. Since mini-MPI supplies adequate buffering it can be used as a replacement for p4. All calls to p4 message-passing routines were replaced by equivalent calls to the mini-MPI routines. All other routines not related to message-passing were replaced by their mini-MPI counterparts. In this way a full MPI implementation that conforms to the standard was realized on the Cenju-3.

9.2 The Abstract Device Interface

Each side of the interface between the ADI and the API provides services to the other side. This section lists prototype data structures and routines that each side of the interface should provide to the other.

Services provided by the device The routines and data structures provided by the device are the definitions on which the MPI implementation (API layer) relies. The definitions, and its implementation, are provided by the device. Two data structures one for `send_handle` and one for `recv_handle` are provided. These data structures contain information about the messages

Data structures provided by the device:

```
typedef ... A_send_handle
typedef ... A_recv_handle
```

```
A_alloc_send_handle(D_send_handle)
A_alloc_recv_handle(D_recv_handle)
A_free_send_handle(D_send_handle)
A_free_recv_handle(D_recv_handle)
```

```
A_post_send(D_send_handle)
```

```
A_post_recv(D_recv_handle)
A_complete_send(D_recv_handle)
A_check_device(blocking)
```

Services provided to the device The routines and data structures provided to the device are the definitions on which the device (ADI layer) relies. The ADI will invoke these functions and access these data structures to interact with the MPI implementation. This interaction is necessary for example to transfer data from the API to the ADI when the API runs in user address space and the ADI runs in kernel space. The definitions, and its implementation, are provided by the MPI implementation (API layer).

Data structures provided to the device:

```
typedef ... D_send_handle
typedef ... D_recv_handle

D_mark_send_completed(D_handle)
D_message_arrived(src, tag, context_id, D_recv_handle, status)
D_get_contig(D_send_handle, address, maxlen, actual_len)
D_put_contig(D_recv_handle, address, maxlen, actual_len)
D_mark_send_completed(D_send_handle)
D_mark_recv_completed(D_recv_handle)
D_check_mpi
```

9.3 Example of operation

How these routines are used to implement high-level message-passing will be illustrated in this section by explaining the actions necessary on the User Program, the API and the ADI layer, to perform a message-passing operation.

Table 9.1 shows the actions that are performed by each of the User Program layer, the API layer and the ADI layer for the MPI_Isend operation, which is a non-blocking send. The MPI function MPI_Isend initiates a send operation. The user program can then perform some useful other work and once it has finished this it waits for the send operation to complete. When the user program calls MPI_Isend the first thing the API does is to convert the user request into the correct request for the ADI. This is done by calling A_alloc_send which sets up a D_send_handle that can be used by the ADI layer. Once the data structures are initialized A_post_send is called to start the send operation in the ADI layer. The ADI is responsible for actually transmitting the message. The ADI and API work together to actually transfer the data between these two layers (D_get_totallen, D_get_into_contig); this allows the API to provide a richer set of data layouts that are not supported by the device. When the API calls the A_post_send, the ADI initiates the send possibly using D_get_totallen and D_get_into_contig to transfer the data from the API to the ADI layer. When the data has been transferred to the ADI layer, the ADI and the API layer return. The user program can now perform some useful work until at a certain moment the processor is interrupted because the message has been sent. The interrupt is handled in the ADI layer which marks the message to indicate that it has been sent by calling D_mark_send_completed which in turn update the API data structures and frees the device send handle by calling A_free_send_handle. After the interrupt has been handled the user program runs again possibly calling MPI_Status which will indicate

User Program	API	ADI
MPI_Isend	<p>A_alloc_send allocates D_send_handle A_post_send calls device layer to start send operation.</p>	<p>Initiates send operation possibly calling D_get_totallen and D_get_contig to transfer data (Or may just notify destination that message is available)</p>
(User code runs)	<p>return ...</p>	<p>... Interrupt; message sent; calls D_mark_send_completed</p>
(User code runs)	<p>Posts send completed in MPI data structures A_free_send_handle frees device's data structures</p>	
MPI_Status	<p>MPI data structures show send completed</p>	
MPI_Wait	<p>MPI data structures indicate send completed free MPI data structures return</p>	

Table 9.1: Nonblocking send followed by wait

that the message has been sent. Finally the user program will call `MPIWait` to find out that the message has been sent. It will also free the API data structures associated with the send operation.



Part IV

Parallel Direct Volume Rendering

The teacher is the central figure in the classroom. It is the teacher who is responsible for the learning and development of the students. The teacher must be able to create a positive learning environment and to provide the students with the necessary support and guidance.

Chapter 10

Teacher's Role

Introduction

The teacher is the central figure in the classroom. It is the teacher who is responsible for the learning and development of the students. The teacher must be able to create a positive learning environment and to provide the students with the necessary support and guidance.

The teacher is the central figure in the classroom. It is the teacher who is responsible for the learning and development of the students. The teacher must be able to create a positive learning environment and to provide the students with the necessary support and guidance.

Conclusion

The teacher is the central figure in the classroom. It is the teacher who is responsible for the learning and development of the students. The teacher must be able to create a positive learning environment and to provide the students with the necessary support and guidance.

Chapter 10

Parallel Direct Volume Rendering

This chapter presents recent literature on parallel direct volume rendering algorithms. Research on direct volume rendering as opposed to surface fitting methods will be presented. Direct volume rendering views all data in the volume at once while *surface fitting* represents interesting structures in volumes by geometric primitives such as polygons. These primitives are then rendered into an image by a geometry renderer. *Direct volume rendering* does not use an intermediate geometric representation of the data but renders the data directly by viewing the volume data as material of variable transparency and color.

In section 10.1 we will present a model of volume rendering [36] followed by a description of sequential (as opposed to parallel) volume rendering methods in section 10.2. Section 10.3 presents the parallel direct volume rendering algorithms described in the literature.

Volume rendering is a technique that can be applied for visualisation and analysis of data produced in various simulation applications such as Computational Fluid Dynamics (CFD) and Finite Element Modeling (FEM) or in medical imaging applications such as Computed Tomography (CT) and Magnetic Resonance Imaging (MRI). These applications generate data defined on different computational grids. CFD and FEM usually produce data on curvilinear and unstructured grids, where CT and MRI produce data mostly on rectilinear grids. Certain volume rendering algorithms are not applicable to curvilinear and unstructured grids whereas others are applicable to all three grid types. Another problem of some of the algorithms is that they are not capable of handling the full range of viewing parameters. They may support parallel projection but not perspective projection or they support only a limited number of different view angles instead of providing arbitrary view rotation. These limitations are present in both sequential and parallel direct volume rendering algorithms.

10.1 Volume Rendering Model

An analytical model of volume rendering will be given in this section to provide a framework for discussing volume rendering algorithms. The volume rendering model is based on the attenuation of light as it passes through a medium of varying transparency and color. The volume data to be visualised is a collection of samples f obtained from sampling some real-world continuous function G . The samples f are used to construct a continuous function F which may be re-sampled to produce images

from different view points. Reconstruction is performed by convolving the samples with a filter K in the spatial domain. Note that in general the reconstructed function F is not equal to the function G in the sampling points. The use of different filters in volume rendering algorithms accounts for the different images generated.

Volume rendering can be modelled in the following three steps:

1. Reconstruction of F .
2. Classification and shading.
3. Integration of intensity and transparency along viewing-rays through the volume.

Reconstruction By convolving the sample points f with a reconstruction filter K we obtain the reconstructed continuous 3D scalar function F .

$$F_{i,j,k} = \sum_{i',j',k'} f_{i',j',k'} * K_{i-i',j-j',k-k'}$$

Classification and Shading Based on the values of F assign an opacity and color to each sample point. The opacity of a data point of low importance will be small, meaning that one will 'see through' these points and be able to look at the more interesting points. The color can be used to highlight certain features in the data or provide a mapping between the type of data and the color that represents this type of data. This classification is performed by applying two functions O (opacity) and S (shading) to the continuous scalar field F .

$$\Omega = O(F) \quad E = S(F)$$

Integration Now that each sample point has an opacity and a color these values are integrated along a viewing-ray through the volume. The integral may be taken toward or away from the viewer. When taken towards the viewer the accumulated opacity T and intensity I along the ray over $[0, p]$ (0 is starting point at back of volume, p end point that is closest to the viewer) is

$$T(p) = e^{-\int_0^p \Omega(\nu) d\nu}$$

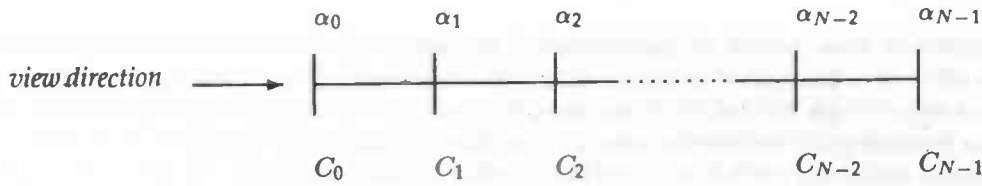
$$I(p) = T(p) \int_0^p \frac{E(\nu)}{T(\nu)} d\nu$$

The intensity function can be solved analytically only if E and Ω are constant or multiples of each other over the interval. This is not generally the case so numeric methods are used to approximate the whole integral. This is done by breaking the path into small segments on which E and Ω are approximated by constants. For segment $[0, q]$ the approximate T and I become

$$T(q) = e^{-\Omega q}$$

$$I(q) = \frac{E}{\Omega} (1 - e^{-\Omega q})$$

These segments are composed using an operation which is consistent with the equations for transparency and intensity. Each segment i has a color $C_i = I$ and an opacity $\alpha_i = 1 - T$, as shown in the picture below, which are to be composed in a back-to-front or front-to-back order.



We can derive recurrences for both front-to-back and back-to-front composing. They are shown below with the corresponding programs next to them. For the front-to-back recurrences the final color is given by I_{N-1} . For the back-to-front recurrence the final color is given by I_0 .

Front-to-back recurrences

$$I_0 = C_0$$

$$\begin{aligned} I_{i+1} &= \sum_{j=0}^{i+1} C_j \prod_{k=0}^{j-1} (1 - \alpha_k) \\ &= C_{i+1} \prod_{k=0}^i (1 - \alpha_k) + \sum_{j=0}^i C_j \prod_{k=0}^{j-1} (1 - \alpha_k) \\ &= C_{i+1} O_{i+1} + I_i \end{aligned}$$

$$O_0 = 1$$

$$\begin{aligned} O_{i+1} &= \prod_{k=0}^i (1 - \alpha_k) \\ &= (1 - \alpha_i) \prod_{k=0}^{i-1} (1 - \alpha_k) \\ &= (1 - \alpha_i) O_i \end{aligned}$$

The corresponding program

```
I = C[0]; O = 1;
for (i=0; i<N-1; i++) {
    O = (1-alpha[i])*O;
    I = C[i+1]*O + I;
}
```

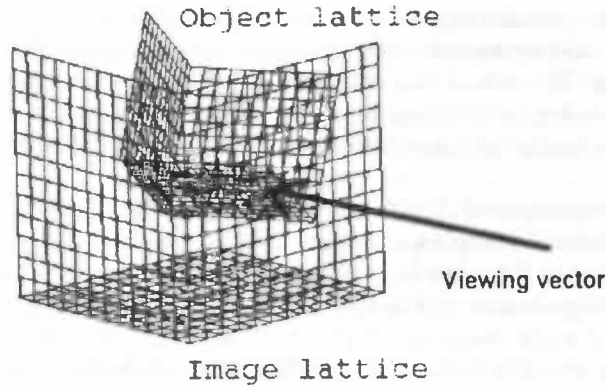


Figure 10.1: Image versus object lattice.

Back-to-front recurrence

$$I_{N-1} = C_{N-1}$$

$$\begin{aligned} I_{i-1} &= \sum_{j=i-1}^{N-1} C_j \prod_{k=i-1}^{j-1} 1 - \alpha_k \\ &= C_{i-1} \prod_{k=i-1}^{i-2} 1 - \alpha_k + \sum_{j=i}^{N-1} C_j \prod_{k=i-1}^{j-1} 1 - \alpha_k \\ &= C_{i-1} + (1 - \alpha_{i-1}) \sum_{j=i}^{N-1} C_j \prod_{k=i}^{j-1} 1 - \alpha_k \\ &= C_{i-1} + (1 - \alpha_{i-1}) I_i \end{aligned}$$

The corresponding program

```
I = C[N-1];
for (i=N-1; i>0; i--)
    I = C[i-1] +
        (1-alpha[i-1])*I;
```

10.2 Sequential Rendering Methods

This section will present three representative sequential algorithms for direct volume rendering of which parallel implementations will be explored in section 10.3. To simplify discussion of the sequential and parallel algorithms we first introduce terminology defined in [36]. In his dissertation Neumann uses the term *object lattice* for the sample points of the volume to be visualised and the term *image lattice* for the volume that lies behind the image plane in the direction of the viewing vector (Figure 10.1). Not only the object lattice is considered a 3D space, but the image lattice is also recognized as such. This means that we do not restrict ourselves to a 2D image plane, but we view the image lattice as a stack of planes behind the image plane in the view direction. In this way the image lattice can be considered a volume as well and this makes reasoning about distributions of the image lattice easier when the parallel algorithms are discussed.

The direct volume rendering process can be described by three steps as we have seen in the previous section: reconstruction, classification and shading, and integration.

The first step in the volume rendering process is to calculate the field values on the image lattice. For this we must reconstruct the 3D function and re-sample this function on the image lattice points. The second step is to classify and shade these points. This step assigns an opacity (α value) and a color intensity to each image lattice point. The last step is to compose the image lattice points along the viewing direction into a 2D image.

Essentially there are two classes of direct volume rendering algorithms. The **backward mapping** and the **forward mapping** algorithms [48]. These two classes differ in the data set that is traversed in the inner loop of an algorithm. A backward mapping algorithm traverses the image lattice and determines the contribution of the volume to the color and opacity of each image lattice point. A forward mapping algorithm traverses the object lattice and determines the contribution of each object lattice point to the final image.

There are three important direct volume rendering algorithms to consider: ray casting, splatting, and volume shearing. Ray casting is a backward mapping algorithm. Splatting and volume shearing are both forward mapping algorithms. Splatting and volume shearing are both considered here because they are quite different, even though they are two instances of algorithms from the same class of forward mapping algorithms. These three algorithms differ most in the reconstruction step of the volume rendering process. This step is considered the computationally most expensive step in the algorithm, and it is the parallelisation of this step that should improve the performance of parallel algorithms over the performance of the sequential algorithms. The algorithms will be explained briefly below and the known optimisations (speed-ups) will be discussed. It is necessary to discuss the optimisations because some of the optimisations are eliminated when a certain distribution of object or image lattice is chosen. This might result in a parallel algorithm with a disappointing speedup over the sequential algorithm. This knowledge is necessary to choose the algorithm most suitable for optimal parallelisation.

10.2.1 Ray Casting

The ray casting algorithm re-samples the volume along view rays calculating a color and opacity for each image lattice point along the ray. The interpolation used mostly is trilinear interpolation of the eight neighbouring object lattice values. The Ω (opacity) and E (color) functions are applied to the new sample points. Successive samples along the ray are combined to produce the pixels of the final image. Perspective projection is supported by casting rays from one point, the eye point. In the case of parallel projection all rays are parallel to each other. The pseudo code for the ray casting algorithm is:

```
void ray_casting ()
{   for (each pixel) {
        for (each z-step along ray) {
            reconstruct field  $F$  at new sample point by trilinear interpolation;
            shade and classify new sample to obtain  $\Omega$  and  $E$ ;
            compute new segment  $I$  and  $T$  from averaged  $\Omega$  and  $E$  of new
                and last sample;
            combine new ray segment  $I$  and  $T$  behind accumulated ray color;
        }
    }
}
```

Optimisations Ray casting is suitable for a number of optimisations. The first optimisation is to compute Ω and E on the object lattice and is called **shade-before-**

reconstruction. Instead of reconstructing the field F and shading and classifying the new sample [46], one reconstructs Ω and E from the object lattice by trilinear interpolation from the Ω and E values on the eight neighbouring object lattice points [32]. The last approach gains efficiency by using the precomputed data gradient when shading the object lattice. This optimisation is a trade-off between speed and image quality.

Adaptive ray termination [46] [33] stops the computation for a ray whose accumulated opacity exceeds a certain threshold. Samples behind this point will not contribute to the color of the pixel anyway. This technique can only be used when steps in the image lattice are taken from the image plane away from the viewer (front-to-back). This technique is especially useful if the classification and shading steps produce high opacities.

Adaptive sampling [51] casts a subset of rays into the volume. If the variance of neighbouring samples in the object lattice is above some threshold more rays are cast in this area of the volume. This is repeated until the variance is below the threshold or sampling has progressed to the level of the pixels. The color of pixels through which no ray was traced is interpolated from the color of neighbouring pixels.

Progressive refinement [30] is the technique of under-sampling the volume to achieve high frame rates when the user is modifying viewing parameters and use more samples once the viewing parameters are stable. This will result in high frame rates when navigating the volume and high quality images when the users wants to inspect the image in detail. Under-sampling can be achieved in several ways. One can subsample the volume in all three directions or raise the threshold of variance in adaptive sampling. Lowering the threshold on accumulated opacity in adaptive ray termination will result in under-sampling of the image lattice. Image pixels that are not sampled will compute the color by interpolation of the neighbouring pixels. A low-pass filter should be used on the resulting sub-sampled images to eliminate aliasing artifacts.

Hierarchical data-structures [33] may also be used to speedup ray casting by eliminating the processing of uninteresting regions of the volume. Octrees may be used to decompose the volume recursively. Each node contains eight children representing the eight octants into which a volume is decomposed. A node contains a summary of the data contained in the sub-volumes. Empty sub-volumes can be skipped since they do not contribute to the image.

Finally, ray casting may be optimised by using a **distance function** on the volume [27]. This technique calculates a volume that contains the radial distance from each point in the object lattice to the closest point with an interesting value, that is a value above some threshold. This distance volume allows sample rays in empty regions to be skipped in much the same way an octree does.

10.2.2 Splatting

The splatting method [50] of direct volume rendering is a forward mapping volume rendering algorithm. The name 'splatting' comes from the similarity with throwing snowballs at a wall. The snowballs are the object lattice points which are 'thrown' towards the image plane and hit this plane, thereby they are splatted and distribute their contents over the neighbourhood of the point where they hit. This depicts the splatting method.

Splatting convolves each object lattice point with a 3D reconstruction filter and accumulates the contribution of the filtered points on the image lattice. The slices in the image lattice are then composed to produce the final image. Because of the computational expense of 3D convolution, implementations have used the projection of the 3D filter kernel on the image lattice and perform convolution in the 2D domain.

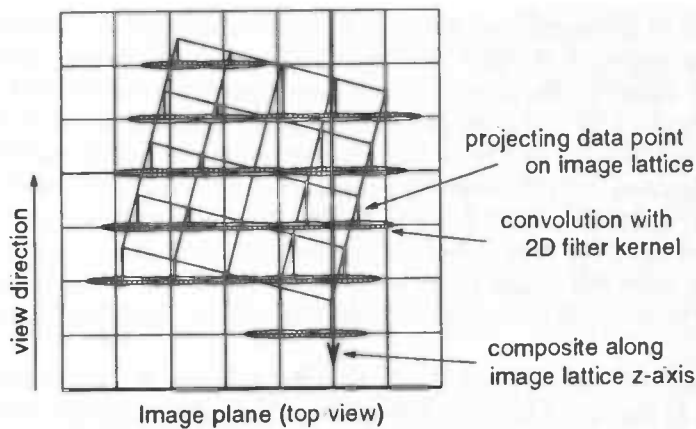


Figure 10.2: Splatting process looked at from the top.

A picture of how the algorithm works is shown in figure 10.2. The pseudo code for the splatting algorithm is:

```
void splatting ()
{
  for (each slice of the object lattice most perpendicular to the view direction) {
    for (each point in a slice) {
      project a point onto image lattice and filter ( $f * K$ );
      accumulate energy at image lattice points to produce  $F$ ;
    }
    shade and classify image lattice points to obtain  $\Omega$  and  $E$ ;
  }
  compute segment  $I$  and  $T$  from averaged  $\Omega$  and  $E$  of two adjacent slices;
  combine all segments  $I$  and  $T$  values to produce final image;
}
```

Optimisations Since the filters used for splatting are invariant under affine transformations, software lookup tables can be used to approximate the filter resulting in faster convolution [49]. However to support perspective projection a 3D filter kernel has to be computed for each ray. Polygon rendering hardware can be used to project 3D filter kernels [31].

Splatting is also suitable for shade-before-re-sample optimisation. In this case object lattice points are shaded and classified to obtain Ω and E . These functions are used in the convolution with the filter and projected onto an image lattice plane. Energy is accumulated to incrementally produce Ω and E .

Threshold of interest is a technique that can be applied to any forward mapping volume rendering method. Since these algorithms traverse the object lattice points they can ignore data values below a threshold of interest. For many data sets many of the data points are uninteresting; therefore this technique may increase the performance significantly.

Hierarchical data structures can also be used to speedup splatting [31]. An octree representation of the volume is created where each node contains I and T values that approximate all that node's children and the error associated with the approximation. At each node the error is compared with an error threshold. If the error is greater than

the threshold, the node's children are visited, otherwise the node is splatted and none of the children are visited.

10.2.3 Volume Shearing

Volume shearing [26] is also a forward mapping technique. It applies a sequence of 1D transformations (shears) to the object lattice transforming it one dimension at a time into the image lattice. Once re-sampled, the volume is aligned with the image lattice and ready for shading, classification and composing. Since the transformations are one-dimensional, only 1D filters must be used for reconstruction. The sequence of shears is determined by the view direction such that minimal signal degradation occurs. The pseudo code for the volume shearing algorithm is:

```
void volume_shearing ()
{
    shear volume  $f$  along first-axis;
    shear volume  $f$  along second-axis;
    shear volume  $f$  along third-axis;
    shade and classify re-samples volume to obtain  $\Omega$  and  $E$ ;
    compute  $I$  and  $T$  as the average of successive points along the image
        lattice z-axis;
    combine  $I$  and  $T$  of all segments along the image z-axis;
}
```

Optimisations Like splatting, volume shearing may be optimised by defining a threshold of interest and ignoring all data points with a value below this threshold. Neumann [36] has shown that two volume shears are sufficient to reconstruct the object lattice on an image lattice. This saves one third of the transformation time without additional costs elsewhere in the algorithm. The volume shears are suitable for fast implementation on vector computers.

10.3 Parallel Algorithms

The literature presented in this section aims at implementing parallel direct volume rendering on general purpose parallel (MIMD) architectures. We will divide the literature into four groups that result from the combination of two issues: whether an image or object-partition is used and whether an image or object-order method is used. In an image partition nodes are assigned volumes of image lattice points to compute. Communication occurs when volume data moves between nodes. In an object partition, each node renders a color and opacity image of its subset of the object lattice. Communication occurs when the local images are composed into a complete image. The difference between image and object-order volume rendering methods should be clear from the previous sections. Where possible performance figures are given for a 64 processor architecture, rendering a data-set of 128^3 elements in a 256^2 image. If performance results are not available for these specifications, the closest figures available will be taken.

10.3.1 Object Partition, Object-Order Rendering

The algorithms in this group of parallel volume rendering methods use an object-order rendering algorithm such as splatting where the object lattice is distributed among the processors of the parallel machine.

In a comparative study of parallel volume rendering algorithms for non-rectilinear computational grids, Challenger compares the performance of a cell projection rendering method to a ray casting rendering method [7]. The architecture used is a BBN TC2000 multicomputer which is a distributed-memory architecture which provides virtual shared memory. Since the cell projection rendering method is an object-order rendering method the results for that method are presented here. The cell projection method integrates across the depth of the cell to give a contribution at each pixel the cell covers. Reconstruction of samples on the cell's faces or inside the cell is accomplished by interpolation between the nodes of the cell. Each cell may be projected to the screen independently, however composing operations from cells which project to the same pixels must be ordered. To ensure the correct rendering order the parallel algorithm maintains a *ready list* of cells that are ready for rendering. Since the cells are rendered in back-to-front direction a cell is ready for rendering when all cells which it obscures have been rendered. The *visibility graph* indicates when a given cell can be transferred to the ready list. For each cell a count is kept of the number of cells that are directly obscured by the given cell. When this count reaches zero the cell can be transferred to the ready list. Both the ready list and the visibility graph are maintained in shared memory and accessed using atomic operations to maintain consistency. In summary, the parallel projection algorithm generates tasks for each cell in the volume. Each task locks the ready list and obtains a cell for processing. The cell is rendered into the frame buffer which resides in shared memory and the visibility graph is updated. The cells ready for processing are transferred to the ready list.

The parallel implementation of projection methods is not very promising. Large amounts of memory are used to maintain the ready list and the visibility graph and the synchronisation needed to access these lists accounts for the fact that the algorithm is not scalable. The main problem is the size of the tasks. One cell per task is much too small. Increasing the task size would improve performance, but introduces problems with task generation due to the constraints on the order in which the cells should be rendered. A 100x120x16 dataset rendered to a 512^2 image on 100 processors takes 26 seconds.

10.3.2 Object Partition, Image-Order Rendering

In this group of parallel algorithms the method used is an image-order method such as ray casting and the object lattice is distributed among the processors of the parallel machine. Most implementations of parallel direct volume rendering use the ray casting algorithm in combination with an object partitioning. Each node renders an image of the same size as the final image and these images are composed using the color and opacity values to produce the final image.

Neumann analyses the communication costs associated with parallel volume rendering algorithms. Two main classes of parallel algorithms are identified: *image* and *object* partitions. The intrinsic communication costs for algorithms in each class are analysed independent of an implementation. Any nontrivial parallel volume rendering algorithm needs communication of object or image data between the nodes. The communication costs for three classes of parallel algorithms are analysed: image partition with static data distribution, image partition with dynamic data distribution and object partition. It is shown that the redistribution cost in the image partition algorithms is strongly dependent on the viewing angle. But for an object partition with block data distribution the redistribution cost decreases when the number of processors increases. An object partition volume rendering algorithm was implemented on the Touchstone Delta and on the Pixel-Planes 5. Local images are rendered by ray casting. The algorithm uses a dynamic block data distribution which achieves load balancing

by storing larger blocks of data at each node than is strictly required by the partition and adjusting the partition boundaries between the nodes. This is the first reported implementation that uses such a load balancing method. A 64 node Touchstone Delta using this algorithm renders a 128^3 data-set in a 256^2 image at 2.5 frames/sec. The same algorithm on a Pixel-Planes 5 with 32 nodes, renders the image at 2.7 frames/sec.

In [13] Silva and Kaufman propose a space-efficient, fast load balanced parallel algorithm. The implementation uses a static object partition and an image-order algorithm called *polygon assisted ray casting* (PARC). The idea behind PARC is simple. One needs to integrate along a ray. By finding tighter bounds for the integral the amount of work is reduced, substantially lowering rendering time. PARC does this by enclosing the volume with a rough polygonal approximation, which is transformed and scan converted into front and back Z-buffers. For each ray the front one gives us an estimate for the lower limit of the integral, and the back one an upper limit estimate. Load balancing is driven by the cubes generated by the PARC algorithm. As the cubes are a very close approximation to the amount of work one has to perform during ray casting, the number of cubes a processor holds is used as the measure of how much work is performed by that particular processor. For ray casting to be efficient one needs contiguous blocks of data thus making composing easier and reducing the number of intersection calculations needed. For this reason slabs, which are consecutive slices of the data sets aligned on two major axes, are used as the basic blocks for the load balancing scheme. The number of cubes in a slab is used as a measure for the amount of work necessary to render the slab. This leads to much better load balance compared to a static distribution where each processor renders an equal part of the volume. The implementation has the following structure: in the processors, a set of working requests are queued and serviced on demand. There are two different kinds of requests, *rendering requests* and *compose requests*. The first type of request is received directly from the host where the user is waiting for images to show up. The second request comes from neighbouring processors. The processor keeps servicing requests by picking a message from each queue. After servicing a rendering request the resulting image is buffered until a compose request for that image is received. The image is composed and sent to the neighbouring processor. The last processor sends the complete image to the host. In this way computation and communication are overlapped resulting in a pipe-lined generation of images at interactive frame-rates. For an Intel Paragon with 32 processors a certain data-set (size unknown) is rendered in a 256^2 image at 1.5 frames/sec.

10.3.3 Image Partition, Object-Order Rendering

The algorithms in this group of parallel volume rendering methods use an object-order rendering algorithm such as splatting, where each processor is responsible for rendering a part of the image lattice.

Westover proposed a parallel splatting algorithm [50]. Each processor traverses a block of the volume and acts as a splat-server for a subset of scan-lines. This makes the algorithm an image-partition, object-order rendering method. A 4 processors Sun-VX/MVX visualisation accelerator renders a $128 \times 128 \times 93$ data-set in a 512^2 image in 41.2 seconds.

10.3.4 Image Partition, Image-Order Rendering

In this group of parallel algorithms the volume rendering method used is an image-order method such as ray casting and the image lattice is distributed among the processors of the parallel machine.

Corrie and Mackeras implemented a parallel ray casting volume renderer [14] on the Fujitsu AP1000 which is a distributed memory MIMD parallel computer. The system uses a software distributed virtual memory system for volume data. The volume renderer makes use of the data coherence that is inherent in rendering volume data by means of a caching scheme. The work involved in rendering an image of a volume data set is allocated to the processors by subdividing the image-lattice into square pixel blocks and each block is rendered using a ray casting algorithm. This makes the algorithm an image partition, image-order algorithm. The square pixel blocks are shown to achieve better performance than other work item shapes because of the potential for exploiting data coherence. Load balancing is performed by using the worker-farm paradigm in which work items are farmed out to the available processors on demand. Image coherence¹ however, can result in poor load balance towards the end of the computation when idle processors are waiting for processors working on a few complex work items to complete. To improve the load balance a work item timeout is set. When a processor is rendering a work item and the timeout expires, it informs the master about this. If there are no more work items to be rendered and there are idle processors, the remainder of the work item is distributed among the idle processors. If there are no idle processors, the processor on which the timeout occurred continues the rendering of the work item. It is shown that work items of moderate size (800-1000 pixels) yield the best results. Data is distributed among the processors and each processor has a least recently used (LRU) cache that contains volume cells from other processors. When a processor requires a volume cell which it does not have, it requests the cell from its neighbourhood. On the Fujitsu AP1000 using 128 processors the algorithm renders a 256x256x109 data-set in a 512² image in 55 seconds.

10.4 Parallel Rendering on the Cenju-3

This section presents a rationale for implementing parallel rendering software on the Cenju-3. The Cenju-3 system has been described in chapter 1. The main features are summarized here. The Cenju-3 is a distributed memory MIMD parallel computer in which up to 256 processors can communicate with each other via a multi-stage interconnection network. The Cenju-3 does not have parallel I/O facilities and does not have support for a parallel frame-buffer. Since all I/O from and to the Cenju-3 is through the host computer, and the bandwidth of this connection is approximately 1MB/sec, the suitability of the Cenju-3 for real-time (>20 frames/sec) visualisation software in general is limited because these applications are all I/O bound. A bandwidth of 1 MB/sec for an image size of 512² limits the frame rate to approximately 1 frame/sec. Images of size 256² can be updated at approximately 4 frames/sec. These frame-rates can be increased by using compression for sending the frames from the Cenju-3 to the host workstation but still for interactive rendering applications the bandwidth should be increased.

An object-partition, image-order (ray-casting) algorithm is the most suitable algorithm to be implemented on MIMD architectures as is apparent from the literature. Ray-casting is a simple, general and efficient method of volume rendering the local object space on each processor and it can be used to render embedded geometry using parallel or perspective projection. For efficient image composition one can use the binary-swap algorithm or employ a pipelined image composition approach as was done in [13].

To date the parallel volume rendering algorithm presented in [13] is the fastest direct volume rendering implementation on a MIMD parallel machine with a relatively

¹A group of pixels requiring similar processing is said to be image coherent.

small number of processors. The software runs on the Intel Paragon using 32 iPSC860 compute nodes each capable of performing 75MFlop/s. This algorithm seems suitable for implementation on the Cenju-3 as well, the only problem being the frame update limits of the Cenju-3. A major improvement of the Cenju-3 parallel computer would be to incorporate parallel I/O in the architecture. This way updating a framebuffer as well as reading volume data from disk could be performed in parallel.

Chapter 11

Conclusions

This final chapter will summarise the conclusions that can be drawn from the work presented in this report. Each part of the report has a separate section in this chapter.

11.1 Parallel Environment

Hardware During the time after the installation of the Cenju-3 at NLR, no real problems have occurred with the parallel hardware. The processing elements and the interconnection hardware have functioned properly. A few software problems with the host workstation of the Cenju-3 have been reported to NEC but they were mainly unrelated to parallel processing. Some minor problems with the parallel programming system have been fixed by NEC.

Software The SVR4 operating system support on the nodes of the Cenju-3 is limited. The number of SVR4 system calls supported at the moment is 20 out of 204. Most of these system calls are not needed for parallel processing, but it would at least be beneficial if socket communication were possible from the nodes to ease communication with the host and other workstations; in our case to communicate graphical information and control messages with the host workstation.

The standard parallel programming environment on the Cenju-3 (Paralib/CJ, mini-MPI) still has a few problems: The `CJrmwrite()` routine in Paralib/CJ has poor performance although it is supposed to use the packet copying functionality of the interconnection network and thus should perform better than any software implementation. Secondly, the broadcast routine of the mini-MPI library is not capable of sending buffers with a size in excess of 256 Kbytes which limits its use unnecessarily. The documentation reports this limitation for the `MPI_Send` routine and suggests to use `MPI_Ssend` instead, a routine that can send buffers of any size. However such an alternative is not available for `MPI_Bcast`. To be able to broadcast buffers larger than 256 kBytes, the broadcast routine should use `MPI_Ssend`'s instead of `MPI_Send`'s internally. At the moment it is the user's responsibility to write a new broadcast routine that uses `MPI_Ssend`.

The Cenju-3 lacks a debugger for parallel programs. It is impossible at the moment to use a debugger to execute a parallel program step by step and inspect its variables. Inserting 'print' statements in the source and recompiling the program is the only way to debug a program. Furthermore, the run-time error messages generated by programs running on the Cenju-3 are obscure. Essentially only the contents of processor registers is printed. This information can be helpful to determine the subroutine in which

a program produced the run-time error by looking up the address of the program counter, at the time of error, in the symbol table of the executable.

Choice of programming system Although the choice of the programming system to be used for the parallelisation of the NaS/RVS visualiser was mainly constrained by the unavailability of the NEC Cenju-3 during the first few months of the project, and although we would rather have used HPF for the parallelisation task because of its close relation to the compiler directives for the NEC SX-3 that were present in the original source code, the decision to use MPI has been advantageous. The portability of the MPI standard has proven its effect. Initially a cluster of four IBM RS6000 workstations was used for the development of the parallel version of the NaS/RVS visualiser using the LAM and MPICH public domain implementations of MPI. Once the Cenju-3 system was available it took approximately one week to port the code to the Cenju-3. Most of the problems of porting the application to the Cenju-3 were unrelated to the use of MPI. The program mapped well onto the subset of MPI that is available on the Cenju, because only the most basic features of MPI were used in the parallel version of NaS/RVS.

11.2 Parallel Visualiser

The main objective of the project was to parallelise the visualisation algorithms of NaS/RVS. The first step in achieving this was to modify the system to adapt it to the NLR environment. For this purpose a new module, capable of displaying the output of NaS/RVS in an X window instead of using UltraNet framebuffer hardware, was developed. Parts of this module have been used in the project that is concerned with the development of a new user-interface to NaS/RVS. The use of hardware for which NaS/RVS was not intended initially¹ revealed a number of non-portable sections in the code that have been rewritten. When this phase of the project was finished, development of the parallel contour plot algorithm started on a cluster of IBM workstations. Development continued on the Cenju-3 once it was available.

The work involved in adapting NaS/RVS to the NLR environment (SGI, NEC SX-3) and porting the code to other platforms (IBM RS6000, NEC Cenju-3) have been time consuming. The lack of proper documentation for NaS/RVS and its visualisation algorithms also contribute to this. Therefore only a relatively small part of the visualisation code (contour plot algorithm) has been parallelised in the remaining time. The resulting parallel contour plot algorithm does show good performance when the screen space partitioning approach to parallelisation is applied. Speedups on 16 processors of 13.8 for unstable viewing parameters and 9.90 for stable parameters have been reported. Unfortunately, the parallel implementation using object space partitioning has very poor performance; 3.45 maximum on 16 processors. The main reason for this big difference is the fact that the implementation of the visualisation algorithms in NaS/RVS is aimed at efficient execution on a vector-supercomputer using a vectorising compiler. Therefore any future efforts to parallelise the visualisation algorithms in NaS/RVS should aim at screen space partitioning algorithms. In this case additional research is necessary to achieve good load balance for these algorithms when the rendered object occupies a small area of the final image in which case a simple partitioning algorithm such as the one used in this report results in a large load imbalance.

It is believed that visualisation algorithms using partitioning of the object space will scale better on large parallel systems (say more than processors) than the screen

¹NaS/RVS was developed for a NEC EWS workstation in combination with NEC SX-3 supercomputer, NLR has been using a SGI workstation with NEC SX-3 supercomputer, IBM RS6000's and NEC Cenju-3

space partitioned algorithm. When partitioning of the object space is used an efficient algorithm for combining the images from all processors is needed. Such an algorithm, with constant run-time in theory, has been presented in chapter 6.

11.3 Message Passing Interface

The MPI standard has proven to be a message-passing system well suited to message-passing programming on the Cenju-3. During the project only the basic features of MPI have been used to enable easy porting from the cluster of workstations to the Cenju-3. The public domain MPICH implementation of MPI was ported to the Cenju-3 after the development of the parallel NaS/RVS visualiser was finished. The NaS/RVS visualiser still uses the mini-MPI library provided by NEC. The public domain MPI implementation will be replaced by a native full MPI implementation from NEC once that becomes available. It is expected that this implementation will perform equally well as the Paralib/CJ library while on the same time providing much more functionality and a standard for message-passing which improves portability of applications to a variety of parallel platforms.

Current literature on MPI can be divided into two groups: literature on how to use the MPI standard, and literature on the possible extension of MPI. The MPI Forum is starting an effort, called MPI 2, to extend MPI. MPI 2 considers extensions to the MPI message-passing standard for parallel programming. This effort has broad participation from vendors, users, and software developers. MPI 2 is not changing MPI; it is considering extending MPI. This means that programs written in MPI will be portable to a MPI 2 environment.

11.4 Parallel Direct Volume Rendering

Most parallel direct volume rendering software uses partitioning of the object space in combination with an image-order rendering algorithm such as ray-casting. This class of algorithms shows better performance than the algorithms from any of the other three classes presented in chapter 10.

One can use the Cenju-3 for parallel direct volume rendering although some limitations exist. The total processing power and speed of the interconnection network is sufficient to achieve interactive frame rates (> 1 frames/sec), but the bandwidth of 1 MByte/sec between the Cenju-3 and the host workstation is insufficient to achieve real-time rendering (> 20 frames/sec) on the Cenju-3. By compressing the images before sending them over to the host workstation possibly increases the maximum frame-rate, but the gain in speed will not be sufficient to achieve a frame-rate of more than 20 frames/sec. The operating system support for communication between the processing elements of the Cenju-3 and the host workstation is insufficient. The fact that all communication should use files is too constraining.

Because the Cenju-3 does not provide parallel I/O, all I/O uses the same interconnection between the Cenju-3 and the host workstation. Rendering applications would benefit from parallel I/O hardware and software in order to efficiently fetch large datasets from disk and output images to disk or a parallel display device.

1. The first part of the document is a letter from the President of the United States to the Congress, dated January 1, 1861.

2. The second part is a report from the Secretary of the Treasury, dated January 1, 1861.

3. The third part is a report from the Secretary of the Interior, dated January 1, 1861.

4. The fourth part is a report from the Secretary of the Navy, dated January 1, 1861.

5. The fifth part is a report from the Secretary of the War, dated January 1, 1861.

6. The sixth part is a report from the Secretary of the State, dated January 1, 1861.

Appendix A

Full MPI on the Cenju-3

This appendix describes how to port the MSU portable implementation of MPI to the Cenju-3. It first describes where to get the portable MSU implementation of MPI and provides pointers to other interesting resources on the internet. Furthermore, all new files and files that need changes are listed with a description of the necessary changes.

A.1 Resources on the internet

The sources of the MSU implementation of MPI can be found on the following URL: `ftp://info.mcs.anl.gov/pub/mpi` in the file `mpich.tar.Z` which contains the latest version of the MSU MPI implementation. Other interesting URL's are shown below, one for the MPI Home Page, and one for the MPI newsgroup. The MPI Home Page contains pointers to more information concerning MPI.

MPI Home Page: `http://www.mcs.anl.gov/mpi/index.html`

MPI new group: `news://comp.parallel.mpi`

A.2 The Cenju-3 device

The ADI for the Cenju-3 system is based on the ADI for the p4 message-passing system. This p4 message-passing system is very similar to the mini MPI library and it is thus reasonably simple to edit this device to support the Cenju-3 device. The sources of the ADI's for the various devices are in `mpich/mpid/ch_*`. The sources for the p4 device for example is in `mpich/mpid/ch_p4`. We have copied the files in this directory into the `mpich/mpid/ch_cj3` directory. All files have been renamed and the calls to the p4 message-passing library have been replaced by calls to the mini-MPI library. This is basically what needs to be done to port the MSU implementation to the Cenju-3.

Next is a list of new or changed files:

- `mpich/configure.in` CHANGED
Configuration input file. This file is used as input for GNU autoconf which produces the `configure` script. This script can then be run to configure makefiles for installation. This file contains the necessary changes for the Cenju-3 system but due to the unavailability of GNU autoconf and GNU m4 is not used at the moment. Instead the generated script `configure` has been edited by hand to apply the necessary changes.

- `mpich/configure` CHANGED
File generated by `autoconf` from `configure.in`. This file has been edited by hand as said before.
- `mpich/mpid/ch-cj3` NEW
Directory containing the ADI for the Cenju-3 parallel computer.
- `mpich/mpid/ch-cj3/Makefile.in` NEW
Makefile used by `configure` script to generate Makefile.
Copy of `mpich/mpid/ch-p4/Makefile.in` to which changes have been applied for the Cenju-3.
- `mpich/mpid/ch-cj3/cj3*.c` NEW
These files are copies of the `mpich/mpid/ch-p4/p4*.c` files. All names containing "P4" have been renamed with "CJ" substituted for "P4". All calls to p4 message-passing library (`p4_sendx`, `p4_recv`) have been replaced by calls to mini-MPI library (`mpi_send`, `mpi_recv`).
- `mpich/mpid/ch-cj3/dmcj3.h` NEW
Copied from `mpich/mpid/ch-p4/dmp4.h`. Renaming of identifiers containing "P4" to identifiers containing "CJ".
- `mpich/mpid/ch-cj3/mpid.h` NEW
Copied from `mpich/mpid/ch-p4/mpid.h`. Renaming of identifiers containing "P4" to identifiers containing "CJ".
- `mpich/include/mpi-cj3.h` NEW
Edited copy of `$CENJU3HOME/include/mpi.h`. All macro's and prototypes for the unused mini-MPI routines have been removed. Various names were changed to prevent name conflicts with the MSU implementation. This file is included by most of the `mpich/mpid/ch-cj3/*.ch` files.

A.3 Installation on the Cenju-3

Instruction on how to install the MSU source and the changes for the Cenju-3 are in this section. There are two archives needed for this:

`mpich-Jul22.tar.Z`

which is available by anonymous ftp from `info.mcs.anl.gov` in `/pub/mpi` and

`cj3device.tar.Z`

which is available by anonymous ftp from `ftp.nlr.nl` in `/transit`¹.

The next steps should install MSU MPI implementation with the Cenju-3 device. Lets assume that we want to install the sources in `/usr/local/src/MPI`, the libraries in `/usr/local/lib` and the include files in `/usr/local/include`.

- Get the `mpich-Jul22.tar.Z` from `info.mcs.anl.gov /pub/mpi` and place it in `/usr/local/src/MPI`.
- Get the `cj3device.tar.Z` file from `ftp.nlr.nl /transit` and place it in `/usr/local/src/MPI`.

¹You will not be able to `ls` the files in this directory

- Uncompress and untar `mpich-Jul22.tar.Z`. This will create the `mpich` directory and all its subdirectories.
- Rename `mpich/configure.in` to `mpich/configure.in.ORIG` and `mpich/configure` to `mpich/configure.ORIG` so untarring in the next step will not overwrite these files.
- Uncompress and untar `cj3device.tar.Z`. This will create the `mpich/mpid/ch-cj3` directory and all the files in it. It will also create the following files: `mpich/configure.in`, `mpich/configure` and `mpich/include/mpi_cj3.h`.
- Run `./configure -arch=IRIX -device=ch-cj3`. The settings for the IRIX architecture work fine for the Cenju-3.
- Now run `make profile` in the `/usr/local/src/MPI` directory.
- Next run `make mpilib` in the `/usr/local/src/MPI` directory.
- After the compilation has finished two files `libmpi.a` and `libpmpi.a` have been created in the `/usr/local/src/MPI/mpich/lib/IRIX` directory. Create (symbolic) links in the `/usr/local/lib` directory which points to these files. The links should be named `libMPI.a` and `libPMPI.a` in `/usr/local/lib`. This is to distinguish them from the file `$CENJU3HOME/lib/libmpi.a`.
- Create a symbolic link named `/usr/local/man` and which points to the `/usr/local/src/MPI/mpich/man` directory or copy all manpages to the appropriate directory under `/usr/local/man`.
- Create symbolic links in `/usr/local/include` for the files `mpi.h`, `mpif.h`, `binding.h` and `mpi_errno.h` which are in `/usr/local/src/MPI/mpich/include`.

The directory structure should now be as follows:

```
/usr/local/lib
/usr/local/lib/libMPI.a -> /usr/local/MPI/src/mpich/lib/IRIX/libmpi.a
/usr/local/lib/libPMPI.a -> /usr/local/MPI/src/mpich/lib/IRIX/libpmpi.a

/usr/local/include
/usr/local/include/binding.h -> /usr/local/src/MPI/mpich/include/binding.h
/usr/local/include/mpi.h -> /usr/local/src/MPI/mpich/include/mpi.h
/usr/local/include/mpi_errno.h -> /usr/local/src/MPI/mpich/include/mpi_errno.h
/usr/local/include/mpif.h -> /usr/local/src/MPI/mpich/include/mpif.h

/usr/local/src/MPI
/usr/local/man -> /usr/local/MPI/src/mpich/man
```

The system is now installed and ready to use.

A.4 Using the MSU MPI implementation

To use the MSU MPI implementation and the accompanying documentation follow the next steps.

- Add, if necessary, `/usr/local/man` to your `MANPATH`.
- Use the following variables in your makefiles:

```
MPI_FLAGS      = -dn \  
                -I/usr/local/include \  
                -L/usr/local/lib \  
                -L$(CENJU3HOME)/lib  
  
CFLAGS         = $(MPI_FLAGS) # other flags  
FFLAGS        = $(MPI_FLAGS) # other flags  
  
LDFLAGS        = -lMPI -lmpi -lparacj  
#              or -lPMPI for profiling library  
  
CC             = cc  
FC             = f77
```

Programs using MSU MPI can be launched on the Cenju-3 using the normal `cjsh` or `cjbr` commands.

Bibliography

- [1] Purushotham V. Bangalore, Nathan E. Doss, and Anthony Skjellum. MPI++; Issues and features. In *OON-SKI '94*, pages 323–338, April 1994.
- [2] Christian Bischof, Steven Huss-Lederman, Xiaobai Sun, Anna Tsao, and Thoman Turnbull. A case study of MPI: Portable and efficient libraries. Technical report, Argonne National Laboratory and Supercomputing Research Center, 1994.
- [3] T. Brandes. *ADAPTOR Users Guide*. German National Research Center for Computer Science, version 2.0 edition, March 1994.
- [4] R. Alasdair A. Bruce, James G. Mills, and A. Gordon Smith. Chimp version 2.0 interface. Technical Report EPCC-KTP-CHIMP-V2-IFACE 1.7, Edinburgh Parallel Computing Centre, University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom, February 1994.
- [5] Ralph Bulter and Ewing Lusk. A users' guide to the p4 parallel programming system. Technical report, Argonne National Laboratory, October 1992.
- [6] Gregory D. Burns. The local area multicomputer. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*. ACM Press, March 1989.
- [7] Judy Challinger. Scalable Parallel Direct Volume Rendering for Nonrectilinear Computational Grids. Ph.D. dissertation, Department of Computer and Information Science, University of California, Santa Cruz, 1993.
- [8] Fei-Chen Cheng. Unifying the MPI and PVM 3 systems. Technical report, Mississippi State University, Dept. of Computer Science, May 1994.
- [9] Fei-Chen Cheng, Paula Vaughan, Donna Reese, and Anthony Skjellum. *The Unify System*. NSF Engineering Research Center, Mississippi State University, September 1994.
- [10] Aswini K. Chowdappa, Anthony Skjellum, and Nathan E. Doss. Thread-safe message passing with P4 and MPI. Technical report, Computer Science Department & NSF Engineering Research Center, Mississippi State University, 1994.
- [11] Lyndon Clark, Ian Glendinning, and Rolf Hempel. The MPI message passing interface standard. Technical report, University of Southampton, Southampton, SO9 5NH, United Kingdom, 1994.
- [12] Lyndon J. Clarke, Robert A. Fletcher, M. Trewin, R. Alasdair A. Bruce, A. Gordon Smith, and Simon R. Chapple. Reuse, portability and parallel libraries. Technical Report TR9413, Edinburgh Parallel Computing Centre, The University of

- Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom, 1994.
- [13] Arie E. Kaufman Cláudio T. Silva. *Parallel Performance Measures for Volume Ray Casting*. *IEEE Visualization*, pages 196–203, 1994.
 - [14] Brian Corrie and Paul Mackerras. *Parallel Volume Rendering and Data Coherence*. In *Parallel Rendering Symposium*, pages 23–26, October 1993.
 - [15] Michael Cox and Pat Hanrahan. *Pixel Merging for Object-Parallel Rendering: A Distributed Snooping Algorithm*. In *Parallel Rendering Symposium*, pages 49–56, October 1993.
 - [16] Jack Dongara, Rolf Hempel, Anthony J.G. Hey, and David W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report ORNL/TM-12231, Engineering Physics and Mathematics Division, Mathematical Sciences Section - Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, June 1993.
 - [17] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A model implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1993. in preparation.
 - [18] Gregory D. Burns et al. All about trollius. In *Occam Users Group Newsletter*, August 1990.
 - [19] Kwan-Liu Ma et al. *A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering*. In *Parallel Rendering Symposium*, pages 15–22, 1993.
 - [20] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, second edition, 1990.
 - [21] High Performance Fortran Forum. High performance fortran language specification. Technical report, Rice University, Houston Texas, May 3 1994. Appeared in *Scientific Programming*, vol. 2, no. 1, June 1993.
 - [22] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. To appear in the *International Journal of Supercomputing Applications*, Volume 8, Number 3/4, 1994.
 - [23] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Technical Report MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1994. in preparation.
 - [24] William Gropp and Ewing Lusk. Implementing MPI: the 1994 MPI implementors' workshop. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.
 - [25] William Gropp and Ewing Lusk. Some early performance results with MPI on the IBM-SP1. Early Draft, May 1994.
 - [26] Pat Hanrahan. *Three-Pass Affine Transforms for Volume Rendering*. *Computer Graphics*, 24(5):71–78, November 1990.

- [27] Max Viergever Karel Zuiderveld, Anton Koning. *Acceleration of Ray Casting Using 3D Distance Transforms. Visualization in Biomedical Computing, SPIE*, 1808:324-335, October 1992.
- [28] Ed Karrels and Ewing Lusk. *Performance analysis of MPI programs*, 1994.
- [29] Vidin Kuma, Ananth Grama Anshul, and Gupta George Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.
- [30] e.a. Larry Bergman, Henry Fuchs. *Image Rendering by Adaptive Refinement. Computer Graphics*, 20(4):29-37, November 1986.
- [31] David Laur and Pat Hanrahan. *Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. Computer Graphics*, 25(4):285-288, July 1991.
- [32] Mark Levoy. *Display of Surfaces from Volume Data. IEEE Computer Graphics and Applications*, 8(3):29-37, March 1988.
- [33] Mark Levoy. *Efficient Ray Tracing of Volume Data. ACM Transactions on Graphics*, 9(3):245-261, July 1990.
- [34] mpi io@nas.nasa.gov. *MPI-IO: A parallel file i/o interface for mpi. Technical Report Research Report 19841 (87784), IBM T.J. Watson Research Center, NAS Systems Division, NASA Ames Research Center*, November 18 1994.
- [35] NEC Corporation. *Parallel Programming Environment PCASE, User's Guide*, version 1 edition, October 20 1993.
- [36] Ulrich Neumann. *Volume Reconstruction and Parallel Rendering Algorithms, A Comparative Analysis. Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill*, 1993.
- [37] Natawut Nupairoj and Lionel M. Ni. *Performance evaluation of some MPI implementations on workstation clusters. Technical report, Departement of Computer Science, Michigan State University, East Lansing, MI 48824-1027*, 1994.
- [38] Kesavan Shanmugam and Konstantinos Toulas. *Application engineering tools for MPI and PUL. Technical Report EPCC-SS94-01, Edinburgh Parallel Computing Centre, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom*, 1994.
- [39] A. Skjellum, S. Smith, C. Still, A. Leung, and M. Morari. *The zipcode message-passing system. Technical report, Lawrence Livermore National Laboratory*, September 1992.
- [40] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. *Writing libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, Proceedings of the Scalable Parallel Libraries Conference*, pages 166-173. IEEE Computer Society Press, October 1993.
- [41] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. *Inter-communicator extensions to MPI in the MPIX (MPI extension) library. Technical report, Mississippi State University*, August 1994.
- [42] Anthony Skjellum, Nathan E. Doss, Kishore Viswanathan, Aswini Chowdappa, and Purushotham V. Bangalore. *Extending the message passing interface (MPI). Technical report, Computer Science Department & NSF Engineering Research Center, Mississippi State University*, 1994.

- [43] Anthony Skjellum and Ewing Lusk. Early applications in the message-passing interface (MPI), June 1994. Submitted to Special Issue of International Journal of Supercomputing Applications.
- [44] Don Speray and Steve Kennon. *Volume Probes: Interactive Data Exploration on Arbitrary Grids*. *Computer Graphics*, 24(5):5-12, November 1990.
- [45] Neven Tomon and Klaas Jan Wierenga. Application engineering tools for CHIMP and PUL. Technical Report EPCC-SS93-15, Edinburgh Parallel Computing Centre, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, United Kingdom, 1993.
- [46] Craig Upson and Michael Keeler. *V-BUFFER: Visisble Volume Rendering*. *Computer Graphics*, 22(4):59-64, August 1988.
- [47] M.E.S. Vogels. Memorandum IW-94-025, report of the Cenju-3 course at NLR on july, 12 and july 15, 1994. Technical report, National Aerospace Laboratory NLR The Netherlands, July 25 1994.
- [48] Alan Watt. *3D Computer Graphics*. Addison-Wesley Publishing Company, second edition, 1993.
- [49] Lee Westover. *Interactive Volume Rendering*. *Chapel Hill Workshop on Volume Visualization*, pages 9-16, May 1989.
- [50] Lee Alan Westover. *SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm*. Ph.D. dissertation, Department of Computer Science, The University of North Carolina at Chapel Hill, 1991.
- [51] Turner Whitted. *An Improved Illumination Model for Shaded Display*. *ACM Communications*, 23(6):343-349, June 1980.