A Self-Starting Guide for the IBM SP2

Roger B. Sidje
(rbs@maths.uq.oz.au)
High Performance Computing Unit
University of Queensland
Australia

January 1995

Contents

1 INTRODUCTION 2

2 GETTING STARTED 2

2.1 Coding/Tuning 4
2.2 Compilation/Execution 4
2.3 Interpretation of Results 6
2.4 Further Reading 6

3 ARCHITECTURE OF THE SP2 8

3.1 Overview 8
3.2 Processors 12
3.3 Interconnection Network 14
3.4 Interconnection Protocols 15

4 SOFTWARE AVAILABLE ON THE SP2 16

4.1 Application Development 17
4.2 Scientific Libraries 18
4.3 Commercial Applications 19

5 PROGRAMMING EXAMPLES 19

5.1 Reduction Operation 19
5.2 Polynomial Evaluation 22
5.3 Recursive Doubling 24
5.4 Sieve of Eratosthenes 26

6 USING LOADLEVELER 29

7 TROUBLESHOOTING 31

8 CONCLUSION 31

9 APPENDIX – OFFICIAL MANUALS 32

1 INTRODUCTION

The purpose of this report is to supply those starting using the IBM Scalable POWERparallel supercomputer with an introductory self-study note. This document is neither intended to replace the official manuals accompanying the supercomputer nor the wide assortment of documents on the Web with their famous hypertext links; on the contrary, it is hoped that it will be used as a complementary beginning guide. For the ease of the presentation, we focus on one programming language only, namely Fortran. But clearly, since we are dealing mainly with high level concepts rather than subtle algorithmic details, C or Pascal users may easily find their way by using the appropriate syntax and/or command call.

This guide was written in the aim of supplying you with enough materials that will enable you to achieve the following goals:

- You are able to use the SP2
- You have understood how it works
- You are well equipped to start reading the official manuals.

If you have constructive remarks that can improve the document, please don't hesitate to send them to `rbs@maths.uq.oz.au` so that they may be incorporated in later editions.

2 GETTING STARTED

Let us consider the typical case of a user (possibly familiar with other parallel systems) who would like to execute in parallel the program shown on Page 3 on several processors of the SP2.

We would like to answer the main and engrossing question: What shall s/he do? We assume that the user knows the basic preliminary steps to log in (**telnet sp2.qpsf.edu.au**, etc). The SP2 AIX working shell is by default the Korn Shell (**ksh**) but the C Shell comes out when typing: **csh**. Most of IBM manual examples are related to ksh hence it worth staying with it. The main steps around the execution of the program are identical to those in use in any other environment either monoprocessor or multiprocessor as shown in Figure 1:

```
C------ Initiatory example...
        integer length, igot, ileft, iproc, iright, nproc, lastproc
        integer msgTYPE, msgINFO
        double precision tok

        integer ISIZE, RSIZE, DSIZE, CSIZE, ZSIZE
        parameter( ISIZE=4, RSIZE=4, DSIZE=8, CSIZE=8, ZSIZE=16 )
        parameter( NPROCMAX=20 )
        integer DONTCARE, ALLMSG, NULLTASK, ALLGRP
        integer nbuf(4), nelem, nqtype

C------ Header to pick-up wild card values...
        nqtype = 3
        nelem  = 4
        call mp_TASK_QUERY(nbuf, nelem, nqtype)

        DONTCARE = nbuf(1)
        ALLMSG   = nbuf(2)
        NULLTASK = nbuf(3)
        ALLGRP   = nbuf(4)

C------ Program body starts here...
        call mp_ENVIRON( nproc,iproc )
        lastproc = nproc-1
        iright = iproc + 1
        ileft  = iproc - 1
        if ( iproc.eq.lastproc ) iright = 0
        if ( iproc.eq.0 ) ileft  = lastproc

        msgTYPE = 1
        do igot = 0,lastproc
          if ( igot.eq.iproc ) then
            print*,"I am node",iproc,", I got the token."
            if ( iproc.ne.lastproc ) then
              print*,"My right neighbour is waiting for it. Press <Enter>"
              read*
              call mp_BSEND( tok,length*DSIZE,iright,msgTYPE )
            endif
          else
            if ( igot.ne.lastproc ) read*
            if ( igot.eq.ileft .and. iproc.ne.0 ) then
              call mp_BRECV( tok,length*DSIZE,ileft,msgTYPE,msgINFO )
            endif
          endif
          call mp_SYNC( ALLGRP )
        enddo
        END
```

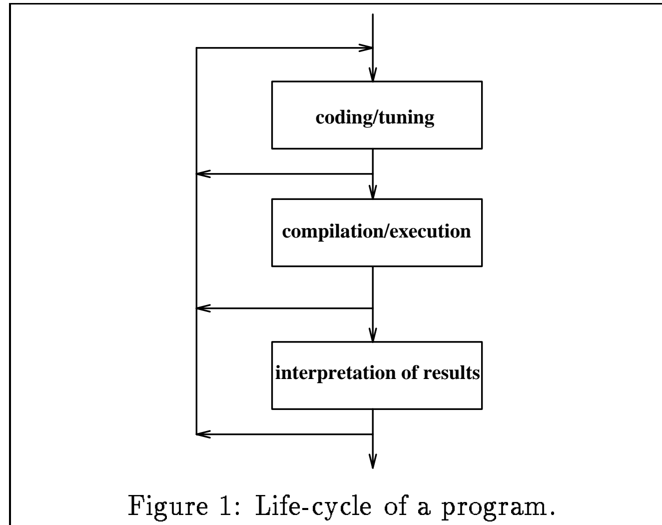

Figure 1: Life-cycle of a program.

2.1 Coding/Tuning

In our actual case, the code (as listed on Page 3) being already devised, it suffices to type it in. If you are familiar with display editing with **vi** or **emacs** (/opt/emacs/bin/emacs) then you may use them directly on the SP2 otherwise, you can edit the program using your favourite editor on your workstation and transfer it on the SP2 by using **ftp sp2.qpsf.edu.au**. We assume from now that it is saved as a file named `prog.f`. A pack containing all the source listings appearing in this brochure can be retrieved by anonymous ftp under the URL: **ftp://dingo.cc.uq.oz.au/pub/sp2guide.tar.Z** and it may also be provided upon request to the author.

2.2 Compilation/Execution

This is the main stage of interest to us. Compiling is done straightforwardly by entering the command:

```
mpxlf prog.f -o prog
```

Then, prior to executing the program, a set of preliminary steps are required mainly to setup the execution environment. A simple way to proceed for the first time is to be satisfied with default values preset to the environment variables. In this case you need only to:

(a) supply the number of nodes desired

(b) supply the list of nodes on which the parallel program will run.

Point (a) can be done by using an appropriate command line option when invoking the program. Point (b) is done by creating a file whose name is `host.list` and whose content is for example:

```
! host.list : available nodes in the environment
s1n03
s1n04
s1n05
s1n06
s1n07
s1n08
s1n09
s1n10
s1n11
s1n12
s1n13
s1n14
s1n15
s1n16
s1n17
s1n18
s1n19
s1n20
s1n21
s1n22
```

The `host.list` can contain more nodes than you plan to use. In this way you can keep the same `host.list` for subsequent programs. The nodes effectively used will be those appearing at first in `host.list`.

REMARK This technique of allocation is known as *specific node allocation*. The dual technique to this one is referred to as *non-specific node allocation* where instead of specifying explicitly the set of nodes desired, you just indicate *pools* in which the amount of nodes required will be automatically picked by the parallel operating system.

The execution is done by entering the command:

```
prog -procs 6
```

or

```
poe prog -procs 6
```

POE stands for Parallel Operating Environment, it is an interface with the parallel system. Further details regarding command line options applicable when invoking a parallel program are available on **poe** man pages (there are some thirty options!). The whole Parallel Environment (PE) includes other X Windows oriented tools such as the Program Maker Array (**pmarray**), Parallel Debugger (**pdbx, xpdbx**), Vizualization Tool (**vt**), InfoExplorer (**info**).

2.3 Interpretation of Results

For completeness we will just say what the above program does: it simulates a revolution of a token around a ring.

2.4 Further Reading

Official manuals are available in postscript files across several directories starting from: **/opt/doc**.

You can transfer them to your local site and consult them using a postscript browser such as **ghostview** (not available on the SP2). Of course due to their required space, the transfer is the last solution to be adopted. Please check if someone else in your group has done it previously.

Hardcopy can also be obtained on request to Wilfred Brimblecombe whose e-mail is: `wilfred@cc.uq.oz.au`.

A listserver for QPSF IBM SP2 users has been set up for discussions related to the supercomputer. It represents an open forum where QPSF SP2 community users share ideas (and problems!). If you have not subscribed as yet, please send a mail to Wilfred Brimblecombe for more details. The QPSF Home Page on World Wide Web (URL: http://www.qpsf.edu.au) leads to a great deal of useful information and links, as well as to official tutorials set up by IBM.

A critical problem with self-studying results from the myriad of official documents which when accumulated together yield thousands of pages. The figure displayed hereafter suggests a reading path. We have deliberately restricted our choice according to our prespecified objectives. A more detailed list of manuals is supplied in the appendix of the present note.

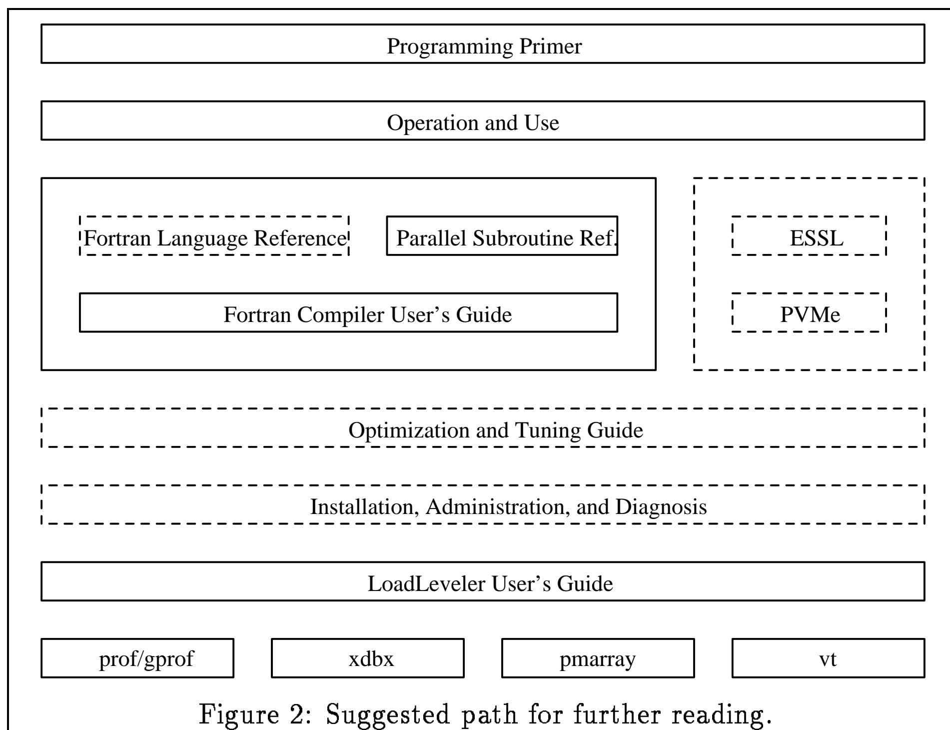

Figure 2: Suggested path for further reading.

Major documents represent hundreds of pages and obviously, it is unwise to attempt reading them with one stroke. Instead, cross-consultation is suitable and efficient. Plain boxes indicate books containing chapters necessary or strongly recommended while dash boxes enclose books whose reading is not a prerequisite for further continuation. However this is just a matter of the amount of time you would like to invest since some of these books deal with relevant topics (see below). The vertical stacking shows a preferred precedence whereas the horizontal juxtaposition exhibits books which can be read in any order.

AIX Parallel Environment Programming Primer 2.0
: This is a pleasant document. In any case, it should be read at first. It supplies an interesting initiation to message passing functions and illustrates how to parallelise programs.

AIX Parallel Environment Operation and Use
: The first chapter is the one of main interest at a start; it contains detailed description of parallel environment variables. Appendix A-pp.196-204 and B-pp.213-218 will be useful for cross-consultation.

AIX XL FORTRAN Compiler/6000 Language Reference
: The inevitable reminder book.

AIX XL FORTRAN Compiler/6000 User's Guide
: Describes how the compiler works and its various options. A good developer needs to know what the compiler does and find out how to use it efficiently.

AIX Parallel Environment Parallel Programming Reference
: Describes the calling syntax of Message Passing Library (MPL) functions.

AIX PVMe User's Guide and Subroutine Reference
: Describes how to work with PVMe.

AIX Optimization and Tuning Guide
: The material presented in this book is excellent.

AIX Parallel Environment Installation, Administration and Diagnosis
: This book contains literal explanations associated to error codes. Errors are unavoidable. Sometimes, you may find insights here. Sometimes, not!

AIX LoadLeveler User's Guide
: How to submit a batch job using a graphical interface. Also touched lightly in AIX Parallel Environment Operation and Use.

prof/gprof, xdbx, pmarray, vt
: The materials dealing with these tools are spread out across the other books. If you have walked along the above documents, you can pick out what you want without difficulty. The reliable technique is to "Do It Yourself".

3 ARCHITECTURE OF THE SP2

3.1 Overview

The IBM SP2 is a distributed-memory multicomputer, i.e., it consists of a collection of *nodes* which can execute distinct instruction streams in parallel; each node is a computer in its own right and thus possesses a private local memory and related caches, a private local disk space, a copy of the operating system, etc. *Nothing is shared* (except interconnection channels and cross-mounted files!). The operating system is AIX/6000 which respects the standard definition of UNIX and other open software environment standards (OSF, IEEE, ISO, POSIX, X/Open, etc).

Basically, a program to be executed is first copied to the local memory of each individual node and a firing-squad signal is set off to start simultaneously the computations which then affect the local variables within each node. This is know as SPMD (Single Program Multiple Data) model. It is however possible to operate under the MPMD (Multiple Program Multiple Data) model which means that different programs are loaded in each individual node at the beginning. Practically, this is achieved by just using an appropriate command line option:

```
poe -procs 4 -pgmmodel mpmd -cmdfile mpmd.list
```

where `mpmd.list` is a file containing the programs that will be loaded in each single node appearing in the `host.list` with respect to the same order.

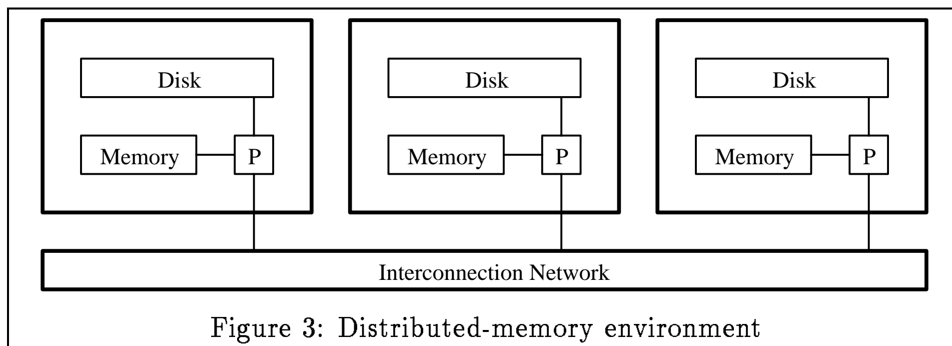

Figure 3: Distributed-memory environment

Parallel processing involves co-operation of separate nodes to complete a common global task. Therefore a mechanism to exchange data between processors must be embedded in the parallel environment. This is known as *interprocessor communication*. On the IBM SP2, the model of interprocessor communication used is *message-passing* which means that data are transferred *explicitly* thanks to function calls within the application. The art of distributed parallel programming resides in a proper manipulation of interprocessor communication. A variety of functions are supplied for this purpose and they are linked together into a high-level library referred to as MPL (Message Passing Library) which is automatically available when designing a parallel application. Examples of such functions include `mp_BSEND`, `mp_BRECV` used earlier. MPL functions can be either *synchronous* or *asynchronous*.

A communication is said to be synchronous or *blocking* if the process issuing the call is blocked until completion of its request. However this is just a *local* point of view. Indeed, saying that a blocking send (`mp_BSEND`) is completed in the sender node simply means that all the data to send have departed and it certainly doesn't imply that they have arrived at their destination. It doesn't ensure that they will arrive (especially if the user has specified a bad destination node or if a subsequent hardware failure emerges).

A communication is said to be asynchronous or *non-blocking* if the process issuing the call returns immediately while the communication mechanism takes place in the background. Asynchronous functions require more careful attention but a proper use of them can sometimes offer significant improvements to an application.

To relieve computational processors of message handling, *message co-processors* are usually attached. On the IBM SP2, the extra work generated by communication is entrusted to a *High Performance Switch* (HPS) which in fact is a complex fabric playing several roles (routing, temporary buffering, error detection/correction, etc).

REMARK It is more reliable and comprehensive to relate synchronous and asynchronous functions to the *state* of the message buffer (i.e., the variable in your code which is actually the argument of the communication function). In a program, the next statement following a call to a synchronous function can interact immediately on the message buffer (without risk of reading/writing corrupted data) whereas with an asynchronous call, it cannot; additional inspections must be carried out to check whether the message buffer is safely manipulable or not.

Roughly speaking, one can classify MPL functions into the following categories:

- message-passing: two processes are implied (e.g., `mp_BSEND`, `mp_BRECV`)
- collective communication: more than two processes are implied (e.g., `mp_SYNC`, `mp_BCAST`)
- process management: functions to examine/manipulate a group of processes (e.g., `mp_TASK_QUERY`, `mp_ENVIRON`)
- utilities: functions to format messages (e.g., `mp_PACK`, `mp_UNPACK`)

REMARK Other communication libraries are also available. For instance, PVMe, the IBM SP2 version of the well-known package PVM (Parallel Virtual Machine). The distinctive feature is that the underlying implementation relies upon the optimised message-passing functions exploiting the High Performance Switch – discussed later. Hence PVMe can be used as an efficient and reliable platform for the development of a standard PVM program which will be installed later on a cluster of possible mixed type machines.

The physical microprocessor chips present in each node are not necessarily identical; in fact they are actually heterogeneous and can be of any type within the following POWER (Performance Optimized With Enhanced Risc) *superscalar* family: thin POWER processors, 62 MHz; thin POWER2 processors, 66 MHz; POWER2 processors, 66 MHz. The thin/wide designation, so to say, comes mainly from their external appearance; the required space for a wide processor is twice the one of a thin node. Nevertheless, wide nodes achieve better performances than thin nodes. Their technical attributes are depicted below.

	Thin 62 MHz	Thin 66 MHz	Wide 66 MHz
Microprocessor	POWER	POWER2	POWER2
Frequency	62.5	66.7	66.7
Peak performance	125 Mflops	266 Mflops	266 Mflops
Instruction cache	32 Kb	32 Kb	32 Kb
Data cache	32 Kb	64 Kb	128 or 256 Kb
Memory	64 – 512 Mb	64 – 512 Mb	64 Mb – 2 Gb
Memory bus	64 bits	64 bits	128 or 256 bits
Memory bandwidth		533 Mb/s	2.2 Gb/s
Internal disk	1 – 4 Gb	1 – 9 Gb	1 – 18 Gb

Table 1: Technical features of existing processors.

	Thin 66 MHz (8 nodes)	Wide 66 MHz (12 nodes)	Wide 66 MHz (2 login nodes)
Microprocessor	POWER2	POWER2	–
Data cache	64 Kb	256 Kb	–
Memory	128 Mb	128 Mb	–
Memory bus	64 bits	256 bits	–
Internal disk	2 Gb	2 Gb	6 Gb

Table 2: Configuration for the QPSF SP2 installed at Griffith University.

The 22-node of the QPSF IBM SP2 are subdivided into two clusters:

(i) the thin cluster (8 nodes) and

(ii) the wide cluster (12 nodes).

The remaining 2 wide nodes are used by the Operating System as entry-points to the supercomputer, i.e., when you log in, you are welcomed by one between these two nodes! The actual configuration causes the service node to be allocated on a weekly (or monthly? or randomly?) rotation.

It is possible to run more concurrent processes than the actual number of physical processors, this is referred to as *task overlapping* and it is completely transparent to the user. Practically, to accomplish overlap on a given node, it suffices to duplicate this node in the `host.list` file. Even with task overlapping, the distributed-memory paradigm is not violated. Whether they run on the same physical node or not, parallel processes exchange their data through message-passing.

3.2 Processors

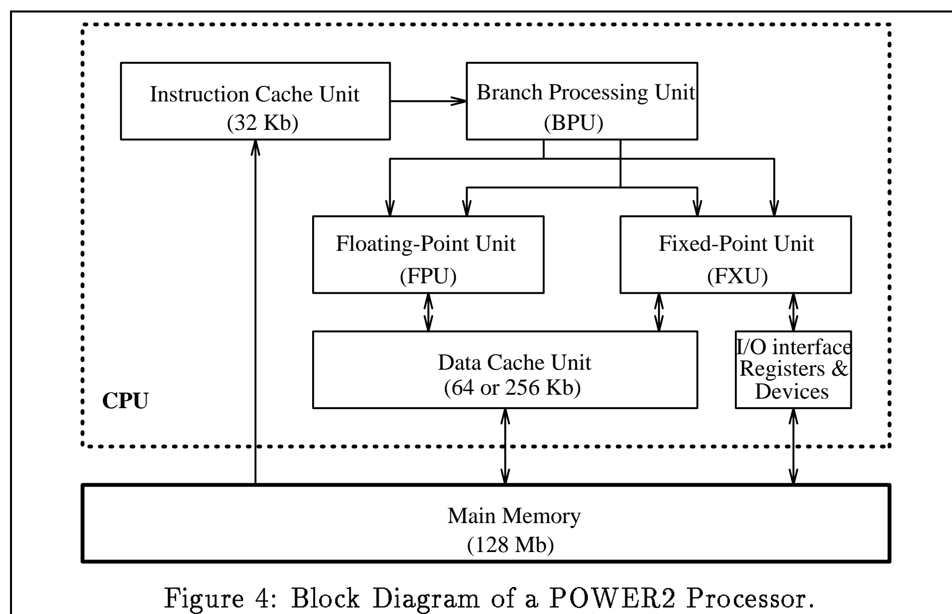

Figure 4: Block Diagram of a POWER2 Processor.

The understanding of the structure of processors can help in organising and tuning computations to use them efficiently. The design and the technology of the POWER2 processor yield a valuable processing element for numerically intensive applications arising in scientific computing and engineering as well as in multiuser applications coming from commercial contexts. It consists of three functional units: the *branch processing unit* (BPU), the *fixed-point unit* (FXU) and the *floating-point unit* (FPU). The BPU acts as a scheduler; it scans the instruction stream from the instruction cache; then depending on the nature of the actual instruction, it is dispatched to the appropriate unit. For instance nonbranching instructions are forwarded to the FXU or the FPU whereas branching instructions are performed by the BPU itself. The FPU handles floating-point operations

involving data loaded in floating-point registers. The FXU handles integer arithmetic, string manipulations, I/O, and functions that involves data loaded in fixed-point registers. Besides, it performs register load/store for both the FPU and itself. Along with its task of dispatcher, the BPU executes all operations that generate jumps to other locations within the code and those dealing with condition logic.

The superscalar feature of the POWER2 processor makes the BPU capable of scheduling the simultaneous execution of up to six instructions per clock cycle: one branch instruction, one condition register logic instruction, two fixed-point instructions, and two floating-point instructions. In addition, the FPU can perform in one cycle the dual operations consisting of a *floating-point multiply-add* which merely stands for two operations. These functionalities are fully exploited if

(i) the compiler is well designed (to organise instructions adequately);

(ii) both the instruction stream and the data stream flow out in a continuous way (to keep the functional units busy).

The point (i) is usually took for granted but unfortunately for the later point, difficulties are frequent. One has to suffer the so-called (data or instruction) cache-misses and their related cascading interferences. *Memory hierarchy* and *set associative caches* are hardware techniques that have been implemented to reduce their damaging effects.

Regarding the programmer, to minimise the number of interruptions/delays in the execution stream, it is of primary importance to adopt programming practices that help the compiler generating the most suitable code. These include, among others, the use of vendor supplied libraries, careful manipulation of arrays, careful organisation of loops. Software tools referred to as *profilers* are supplied to help the developer detecting hot spots.

When the code have been design with proper guidelines in mind, the use of the optimising preprocessor VAST (or KAP) and the correct use of appropriate compiler optimising flags may contribute significantly in achieving better performances. The way of doing so is usually preferred over hand tuning because it is safe, updating and maintenance are easy, and beside all it renders a performance/effort ratio superior to fastidious and error prone hand tuning.

Let us end this section by mentioning that, among many other features of the POWER2 processor, the square root instruction is performed in hardware.

	POWER PC601 IBM	Pentium Intel	Super Sparc Sun	PA 7150 HP	Alpha DEC	POWER IBM	POWER2 IBM
MHz	66	100	60	125	200	62.5	67.7
MFLOPS	66		120	250	200	125	266
int92	62.6	100	88.9	136	106.5	70.3	121.6
fp92	72.2	80.6	102.8	201	200.4	121.1	259.7

Table 3: Comparisons with a few other processors.

3.3 Interconnection Network

The key element to the interconnection network is the High Performance Switch (HPS) which is in charge of handling message passing among processors during runtime of parallel applications and which can also be used for fast data file transfers. The HPS includes the hardware boards (switch modules, adapters, DMA support) as well as the software control mechanisms. It provides an all-to-all internode connection (each node is connected to all the others). It contains redundant physical switchboard elements (to tolerate hardware failures) and it supplies multiple paths between nodes (to tolerate congestion and faulty components, this is also known as adaptive routing).

The HPS design and functionalities yield an interconnection network which is *multi-stage, omega, packet-switch, buffered-wormhole.*

The multi-stage feature is intended to keep the amount of bandwidth available per processor constant as the system expands. The actual design of the HPS guarantees that the all-to-all interconnection still remains in case of any further extension. The SP2 is technically *scalable* because its *bisectional bandwidth* scales linearly with the number of nodes in the system. The bisectional bandwidth is a common measure to characterise the scalability of a network topology, it represents the (average) total possible bandwidth between two equal halves of the system. The bisectional bandwidth of the HPS scales linearly as most multi-stage networks, crossbars and hypercubes do. On the other hand for a grid, the bisectional bandwidth scales with the square root of the number of nodes whereas for a ring, it remains constant.

The SP2 network topology is classified as an omega network because its interstage connection is regular, i.e., the lattice of wires is identical across switch modules. An omega network allows a broadcast using one pass through the switch.

The packet-switch is a data flow transfer mechanism opposite to circuit-switch. The circuit-switch method requires to reserve a path in the network

which is used solely by one message for the duration of its transfer. The path is unreserved at the end of the transmission by a control code attached to the tail of the message. It follows that long messages can monopolise the network to the detriment of short and possibly crucial messages. In contrast, the packet-switch method split the message into self-routing packets, each packet competing for its own transfer. Packets of different messages can interlace on a path.

The wormhole routing strategy is opposite to store-and-forward. It consists in breaking up each packet into *flits* (flow control digits) which are hardware-routed in a pipelined fashion. This yields a latency almost independent of the distance between the sender and the receiver. The store-and-forward strategy requires to assemble the packet hop-by-hop completely. Thus its latency is proportional to the number of hops along the path. Its was mainly used in the earlier ages of multicomputers. Now saying that a wormhole routing is buffered means that the component flits of a packet can be temporary buffered in switch elements depending on the actual network traffic-load. It is worth noting that the interlacement of packets resulting from the packet-switch transfer method entangles *entire* packets *not* flits.

3.4 Interconnection Protocols

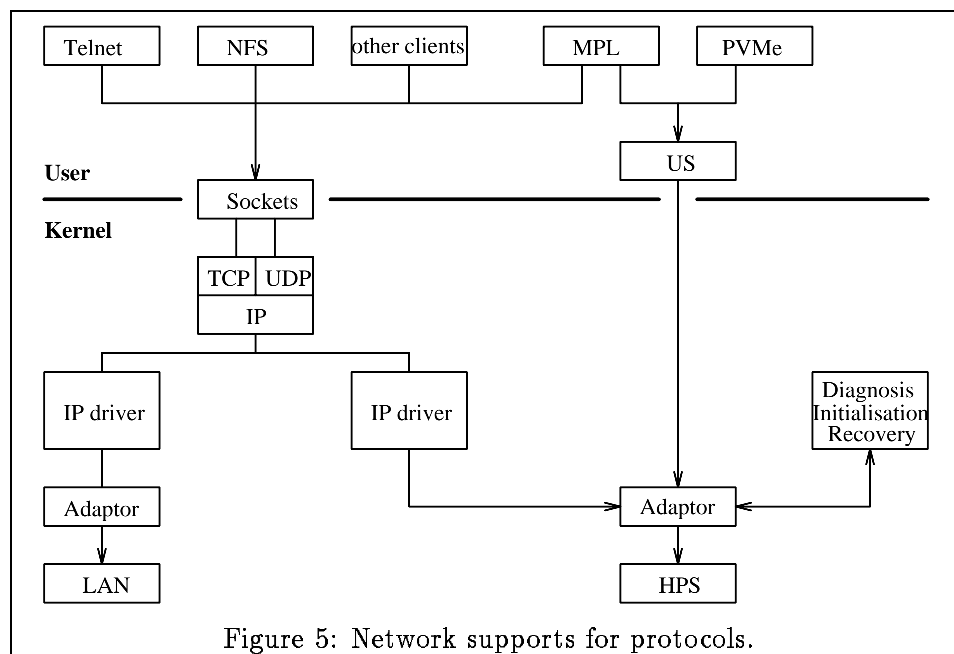

Figure 5: Network supports for protocols.

There exists two main types of communication protocol that can be used by a parallel application: IP (Internet Protocol) or US (User Space Protocol). The message passing library (MPL) exists in two implementations, each corresponding to a protocol. Their interfaces (synopsis of function calls) are identical and as a result, to work with a particular protocol, it suffices to specify the effective implementation to be bound with. The choice can be decided explicitly by the user with a **poe** command line option:

```
poe prog -eulib us
```

Also, the desired library can be statically linked to the program at the compilation stage by using the compiler flags **-ip** or **-us**. When the user ignores these specifications (while compiling and while invoking the program), the IP library is dynamically linked by the operating system to the program at its invocation.

The US library is a specialised library designed to exploit the features of the HPS intensely. It offers the lowest latency and the maximum bandwidth. Its use is strongly recommended for applications demanding extensive communications. However, it does not permit task overlapping.

The IP library is a general purpose library which may not interact with the HPS. It is the default library for SP2 systems not equipped with a native HPS. Besides, it can allows us to run a parallel application across two separate SP2 systems.

	hardware	US (measured)	IP (measured)
latency	125ns/switch stage	39.5μs	40μs
bidirectional bandwidth	40Mb/s	33.9Mb/s	10.3Mb/s

Table 4: Performance of protocols.

4 SOFTWARE AVAILABLE ON THE SP2

Becoming drawn into programming a multicomputer depends heavily on its parallel environment and vendors of multicomputers have devoted considerable efforts to provide user-friendly operating environments. There exists

several software packages on the IBM SP2 intended either for the system administrator, the application developer, or the last-end user. We will focus mainly on the laters but let us mention that for system administration, a package referred to as AIX PSSP (AIX Parallel System Support Programs) contains a suite of tools allowing the system administrator to pilot the environment in an unified way using a unique control point: a RS/6000 control workstation which is mandatory for any SP2 environment. System management includes among others: hardware maintenance, configuration, user accounts, system accountings, error detection and diagnosis, batch job management, etc.

4.1 Application Development

The application developer will usually log in, do some editing, compilation, debugging, execution. Editing can be done away from the QPSF SP2, the program being transferred later. Unfortunately the *cross-environment* development is not available. A cross-environment allows a developer to do all the preliminary steps away from the target machine and access the supercomputer *just* for the definitive execution stage.

Compilers are provided to support the following programming languages: Fortran (xlf/mpxlf), C (xlc/mpcc) and Pascal (xlp), extended with distributed-memory message-passing functions and some extra features of IBM. Also, a compiler for the object-oriented language C++ (xlC/mpCC) is supplied and the fortran compiler supports the entire specification of Fortran 90. For performance analysis, profilers are supplied (prof/gprof) and some X-Interface software packages have been installed:

Program Maker Array (**pmarray**)
: This is a tool to visualise thanks to a graphical array, the active evolution of a parallel program. Special "marker" should be add in the source code to activate/light entries of the graphical array.

LoadLeveler (**xloadl**)
: This is a job manager tool. It supplies a graphical interface over the commands necessary to submit and manage batch jobs. Its use is highly recommended for it will avoid you from suffering current unsteadiness in the system (see § 6 and § 7 hereafter).

Visualization Tool (**vt**)
: This tool is for postmortem performance analysis of parallel programs.

It supplies graphical displays (histograms, diagrams, etc) for quantifying and interpreting inter-communication traffic, I/O, CPU utilisation, etc.

An optimising preprocessor (VAST) and a parallel symbolic debugger (**pdbx/xpdbx**) are available. Additional information on software as well as on other subjects not discussed in this notice are retrievable online through standard Unix *man pages* or thanks to an ergonomic hypertext document browser known as InfoExplorer (see the file `/opt/doc/README`). To date, some documents may not be accessible since their corresponding InfoExplorer format have not been installed as yet. Provided that the `DISPLAY` environment variable is set adequately, the command:

```
info -l pe
```

pops out pretty graphical window menus with hypertext links to SP2 Parallel Environment documents.

4.2 Scientific Libraries

For the development of numerically intensive applications, IBM provides a large amount of routines that have been optimised for its RS/6000 architecture. These (non-parallel) routines are grouped together into high-performance fine-tuned scientific libraries. These include:

- The well-known Basic Linear Algebra Subprograms (BLAS, compiler flag: -lblas) which deal with vector-vector operations (BLAS-1), matrix-vector operations (BLAS-2) and matrix-matrix operations (BLAS-3). The BLAS set is automatically provided by IBM upon purchase of its supercomputer. But this is not the case for ESSL.

- The Engineering and Scientific Subroutine Library (ESSL, compiler flag: -lessl) supplies mathematical routines for linear algebra, eigensystem analysis, FFT, sorting and searching, interpolation, quadrature, etc. BLAS is implicitly embedded into ESSL. In addition, the optimising preprocessors can automatically recognise code sections that can be replaced by appropriate ESSL library calls. A subset of ESSL matches routines available in the public domain library LAPACK (Linear Algebra PACKage).

4.3 Commercial Applications

Other software packages exist or are about to be ported. These include software for data analysis, decision support, database management and other multiuser applications intended for commercial environments.

There are other software such as NAG Fortran, Linda, Forge, OSL, etc. These presentations are for the sake of the information. In addition not all the programs running on the SP2 have been purchased!

5 PROGRAMMING EXAMPLES

5.1 Reduction Operation

This example illustrates how to perform a binary reduction. More specifically, given a vector $v = [v_1, ..., v_n]$ and an associative operation $\circ$, we would like to compute $r = v_1 \circ v_2 \circ \cdots \circ v_n$ in $s = \lceil \log_2(n) \rceil$ steps. The serial algorithm $r = v_1;\ r = r \circ v_j,\ j = 2, 3, ..., n;$ is worthless in our parallel context. A more promising organisation of calculations is portrayed by the computational tree in Figure 6. The problem is to monitor processors interplay when n is an exact power of 2 or not. In the first proposed code, we require n to be an exact power of 2, but this constraint vanishes in the second implementation. We set $\circ$ to $+$ and v is a real n-vector whose components are randomly generated. This is merely a sketch of how an associative pair-by-pair manipulation may be carried out over a set of arbitrary objects in a logarithmic number of steps. It is also worth mentioning that if the actual number of processes is less than n, contiguous blocks can be evaluated prior to the reduction.

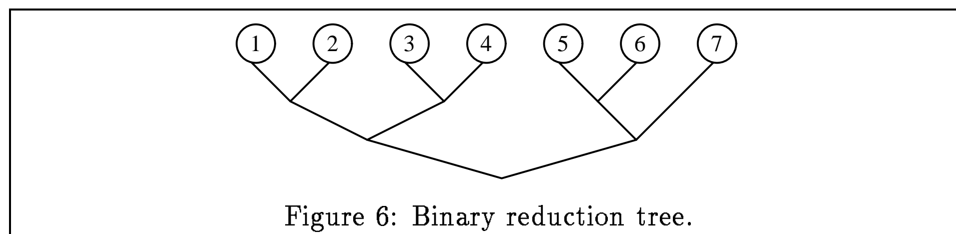

Figure 6: Binary reduction tree.

Remark Compiling with `-D` (debugging) enables the execution of D-statements. This will allow you to review that the number of send coincides with the number of receive and also that they can be grouped into pairs involving suitable nodes. The reduction operation is also called the *fan-in* or *tree operation* and it is performed by the built-in function `mp_REDUCE`.

Binary Reduction for n exact power of 2.

```
C---  This program implements the fan-in algorithm (tree operation) for the
C---  global sum of n values, n = number of processors, must be a power of 2.
C---  rbs@maths.uq.oz.au - High Performance Computing Unit - January, 1995
C---  University of Queensland - Australia.

      implicit double precision( a-h, o-z )
      integer iproc, nproc, msgTYPE, msgINFO
      integer ISIZE, RSIZE, DSIZE, CSIZE, ZSIZE
      parameter( ISIZE=4, RSIZE=4, DSIZE=8, CSIZE=8, ZSIZE=16 )
      parameter( NPROCMAX=20 )

C---  Program body starts here...
      call mp_ENVIRON( nproc,iproc )

C---  Check if the number of processors is an exact power of 2...
      nstep = int( dlog(dble(nproc))/dlog(2.0d0) )
      if ( 2**nstep.ne.nproc ) then
         print*,"Error: The number of nodes is not a power of 2."
         call mp_STOPALL( -1 )
      endif
D     if ( iproc.eq.0 ) print*,"Number of steps:",nstep

C---  Get a random value in each processor...
      seed = iproc+7
      call srand( seed )
      val = rand() ! or =iproc to check for correctness
      print*,"Proc:",iproc,"  value =",val

C---  Global sum in log2 steps...
      idle = 0
      igap = 1
      gsum = val
      msgTYPE = 1
      do istep = 1,nstep
         if ( mod(iproc,2*igap).ne.0 ) then
D           print*,"Proc:",iproc,"  send to:",iproc-igap
            call mp_BSEND( gsum,DSIZE, iproc-igap, msgTYPE )
            idle = 1
         else
D           print*,"Proc:",iproc,"  recv from:",iproc+igap
            call mp_BRECV( val,DSIZE, iproc+igap, msgTYPE,msgINFO )
            gsum = gsum + val
         endif
         if ( idle.ne.0 ) goto 100
         igap = 2*igap
      enddo

 100  continue
      if ( iproc.eq.0 ) print*,"Global sum:",gsum
D     print*,"Proc:",iproc," ended"
      END
```

Binary Reduction for n arbitrary.

```
C---  This program implements the fan-in algorithm (tree operation) for the
C---  global sum of n values, n = number of processors, is arbitrary.
C---  rbs@maths.uq.oz.au - High Performance Computing Unit - January, 1995
C---  University of Queensland - Australia.

      implicit double precision( a-h, o-z )
      integer iproc, nproc, lastproc, msgTYPE, msgINFO
      integer ISIZE, RSIZE, DSIZE, CSIZE, ZSIZE
      parameter( ISIZE=4, RSIZE=4, DSIZE=8, CSIZE=8, ZSIZE=16 )
      parameter( NPROCMAX=20 )

C---  Program body starts here...
      call mp_ENVIRON( nproc,iproc )
      lastproc = nproc-1

C---  Get the number of steps...
      nstep = int( dlog(dble(nproc))/dlog(2.0d0) )
      if ( 2**nstep.lt.nproc ) nstep = nstep + 1
D     if ( iproc.eq.0 ) print*,"Number of steps:",nstep

C---  Get a random value in each processor...
      seed = iproc+7
      call srand( seed )
      val = rand()  ! or =iproc to check for correctness
D     print*,"Proc:",iproc,"  value =",val

C---  Global sum in log2 steps...
      idle = 0
      igap = 1
      gsum = val
      msgTYPE = 1
      do istep = 1,nstep
         if ( mod(iproc,2*igap).ne.0 ) then
D           print*,"Proc:",iproc,"  send to:",iproc-igap
            call mp_BSEND( gsum,DSIZE, iproc-igap, msgTYPE )
            idle = 1
         else if ( iproc+igap.le.lastproc ) then
D           print*,"Proc:",iproc,"  recv from:",iproc+igap
            call mp_BRECV( val,DSIZE, iproc+igap, msgTYPE,msgINFO )
            gsum = gsum + val
         endif
         if ( idle.ne.0 ) exit
         igap = 2*igap
      enddo

      if ( iproc.eq.0 ) print*,"Global sum:",gsum
D     print*,"Proc:",iproc," ended"
      END
```

5.2 Polynomial Evaluation

Let a polynomial of degree n

$$s(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

where the degree is assumed for clarity to be a multiple of the number of processors p, i.e., $n = pq$. This example illustrates how to compute $s(x)$ in parallel by breaking up the above series into p partial components as follows

$$s(x) = a_0 + \sum_{k=0}^{p-1} x^{qk} s_k(x)$$

with

$$s_k(x) = \sum_{i=1}^{q} a_{qk+i} x^i.$$

A substantial parallelism is then introduced as indicated in the following steps:

1. compute in parallel
$$s_0(x) \;\Big|\; s_1(x) \;\Big|\; s_2(x) \;\Big|\; \cdots \;\Big|\; s_{p-1}(x)$$

2. compute in parallel
$$\Big|\; x s_1(x) \;\Big|\; x^2 s_2(x) \;\Big|\; \cdots \;\Big|\; x^{(p-1)q} s_{p-1}(x)$$

3. compute by reduction
$$a_0 + s_0(x) + x s_1(x) + x^2 s_2(x) + \cdots + x^{(p-1)q} s_{p-1}(x).$$

In the proposed code, the coefficients are set to $a_0 = a_1 = \cdots = a_n = 1$ so that

$$s(x) = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}.$$

This simple formula can be used to check for correctness.

```
C---  Program to evaluate a polynomial of degree n = pq, p = number of procs.
C---  rbs@maths.uq.oz.au - High Performance Computing Unit - January, 1995
C---  University of Queensland - Australia.

      integer iproc,nproc,nbuf(4),msgTYPE,msgINFO,ALLGRP,ISIZE,DSIZE
      parameter( ISIZE=4, DSIZE=8 )
      integer i, n, k, p, q, qmax
      parameter( qmax=100 )
      double precision a(0:qmax), x, sk, sx
      external D_VADD ! vector-addition, supplied function.

C---  Header to pick-up wild card values...
      call mp_TASK_QUERY( nbuf,4,3 )
      ALLGRP = nbuf(4)

C---  Program body starts here...
      call mp_ENVIRON( p,k )

C---  Get the degree of the polynomial and the value of the variable...
      if ( k.eq.0 ) then
         read*,n
         read*,x
         print '("degree:   n = ",I3)',n
         print '("variable: x = ",E10.4)',x
      endif
C---  Node 0 broadcasts now these values to all the others....
      call mp_BCAST( n,ISIZE, 0, ALLGRP )
      call mp_BCAST( x,DSIZE, 0, ALLGRP )

C---  Check input values...
      q = n / p
      if ( mod(n,p).ne.0 ) then
         print*,'Error: n is not a multiple of the number of nodes.'
         call mp_STOPALL( -1 )
      endif
      if ( q.gt.qmax ) then
         print*,'Error: not enough room for coefficients.'
         call mp_STOPALL( -1 )
      endif

C---  What are the coefficients of each partial polynomial...
      a(0) = 0.0d0
      do i = 1,q
         a(i) = 1.0d0
      enddo
      if ( k.eq.0 ) a(0) = 1.0d0

C---  Horner rule to evaluate each partial polynomial...
      sk = a(q)
      do i = q-1,0,-1
         sk = a(i) + x*sk
      enddo

C---  Global sum using the built-in reduction operation...
      sk = x**(q*k) * sk
      call mp_REDUCE( sk, sx, DSIZE, 0, D_VADD, ALLGRP )
      if ( k.eq.0 ) print '("value: s(x) = ",E10.4)', sx
      END
```

5.3 Recursive Doubling

The recursive doubling is an extension of the reduction operation, discussed previously, where in addition to obtaining a global result, one computes also partial contributions. More specifically, one is interested in

$$r_1 = v_1 \ ; \quad r_j = v_1 \circ \cdots \circ v_j = r_{j-1} \circ v_j \ , \ \text{ for } \ j = 2, ..., n.$$

This operation is also known as *parallel prefix* or *scan* and is available through the built-in function called `mp_PREFIX`. We would like to address its emulation. A quick look on Figure 6 reveals that a reduction tree-based method produces only some of the desired values. Therefore another organisation must be worked out. Let us define

$$r_{j,j} = v_j \ ; \quad r_{i,j} = v_i \circ \cdots \circ v_j \quad , \quad 1 \leq i < j \leq n,$$

and thus, the wanted values are

$$r_{1,j} \ , \quad j = 1, 2, ..., n.$$

A suitable computational plan is displayed in Table 5 below.

v_1	•	$r_{1,1}$	•	$r_{1,1}$	•	$r_{1,1}$	•	$r_{1,1}$
v_2	v_1	$r_{1,2}$	•	$r_{1,2}$	•	$r_{1,2}$	•	$r_{1,2}$
v_3	v_2	$r_{2,3}$	$r_{1,1}$	$r_{1,3}$	•	$r_{1,3}$	•	$r_{1,3}$
v_4	v_3	$r_{3,4}$	$r_{1,2}$	$r_{1,4}$	•	$r_{1,4}$	•	$r_{1,4}$
v_5	v_4	$r_{4,5}$	$r_{2,3}$	$r_{2,5}$	$r_{1,1}$	$r_{1,5}$	•	$r_{1,5}$
v_6	v_5	$r_{5,6}$	$r_{3,4}$	$r_{3,6}$	$r_{1,2}$	$r_{1,6}$	•	$r_{1,6}$
v_7	v_6	$r_{6,7}$	$r_{4,5}$	$r_{4,7}$	$r_{1,3}$	$r_{1,3}$	•	$r_{1,7}$
v_8	v_7	$r_{7,8}$	$r_{5,6}$	$r_{5,8}$	$r_{1,4}$	$r_{1,4}$	•	$r_{1,8}$
v_9	v_8	$r_{8,9}$	$r_{6,7}$	$r_{6,9}$	$r_{2,5}$	$r_{2,9}$	$r_{1,1}$	$r_{1,9}$

Table 5: Evolution of recursive doubling.

At the kth step, $k = 0, 1, ...$, there are 2^k results known. Therefrom a suitable combination involving known values as well as other intermediate values is effected; this *doubles* the number of results to $2.2^k = 2^{k+1}$ and generates other intermediate values useful for the continuation.

Recursive Doubling.

```
C---  This program emulates the recursive doubling over a length-n vector.
C---  rbs@maths.uq.oz.au - High Performance Computing Unit - January, 1995
C---  University of Queensland - Australia.

      implicit double precision( a-h, o-z )

      integer iproc, nproc, lastproc
      integer msgTYPE, msgINFO

      integer ISIZE, DSIZE, NPROCMAX
      parameter( ISIZE=4, DSIZE=8, NPROCMAX=20 )

C---  Program body starts here...
      call mp_ENVIRON( nproc,iproc )
      lastproc = nproc-1

C---  Get the number of steps...
      nstep = int( dlog(dble(nproc))/dlog(2.0d0) )
      if ( 2**nstep.lt.nproc ) nstep = nstep + 1
D     if ( iproc.eq.0 ) print*,"Number of steps:",nstep

C---  Get a random value in each processor...
      seed = iproc+7
      call srand( seed )
      val = rand() ! or =iproc to check for correctness
D     print*,"Proc:",iproc,"  value =",val

C---  Prefix in log2 steps...
      igap = 1
      prefix = val
      msgTYPE = 1
      do istep = 1,nstep
         if ( iproc+igap.le.lastproc ) then
D           print*,"Proc:",iproc,"  send to:",iproc+igap
            call mp_BSEND( prefix,DSIZE, iproc+igap, msgTYPE )
         endif
         if ( iproc-igap.ge.0 ) then
D           print*,"Proc:",iproc,"  recv from:",iproc-igap
            call mp_BRECV( val,DSIZE, iproc-igap, msgTYPE,msgINFO )
            prefix = prefix + val
         endif
         igap = 2*igap
      enddo

 100  continue
      print*,"Prefix:",iproc,": ",prefix
      END
```

5.4 Sieve of Eratosthenes

This last example addresses the parallel version of the sieve of Eratosthenes which is a very old algorithm for finding prime numbers (by the way, do you know another algorithm? I will be interested in hearing from you). This example display a glimmering of how the master/workers interaction can be monitored.

The serial version of the algorithm starts by laying down numbers within the range of interest: $1, 2, 3, 4, 5, 6, 7, ..., N$ and proceeds onward by striking all multiples of consecutive primes found: $2, 3, 5, 7, ...$ and so forth. The speed of the algorithm doubles when scanning only the restricted list of odd numbers: $1, 3, 5, 7, ...$ Also the striking course attached to a prime number, say a, might starts from a^2.

Denoting p the number of processors, a first approach to parallelise this algorithm is to divide the search list into p contiguous blocks and assign each block to an individual processor:

P_0 :	1	3	5	7	9	11	13	15
P_1 :	17	19	21	23	25	27	29	31
P_2 :	33	35	37	39	41	43	45	47
P_3 :	49	51	53	55	57	59	61	63

Therefrom, P_0 is elected to be the *master*; he scans his portion and once a prime is found, let us called this prime the *actor*, it is broadcast to the *workers*. Upon reception of the actor, the striking process is carried out independently in parallel by all the processors. When there is no more prime in the master, he *retired* and the next active neighbour becomes the new master. However although this strategy is simple to formulate, its distributed programming is cumbersome because it involves unsteady groups with changing leaders. Anyway, if you have some time to spare, go ahead... Hint: don't stick on the tatics that keep idle processors in the group. Make sure to free them.

P_0 :	1	3	5	7	57	59	61	63
P_1 :	9	11	13	15	49	51	53	55
P_2 :	17	19	21	23	41	43	45	47
P_3 :	25	27	29	31	33	35	37	39

The second approach will avoid adjusting the group by adopting another data distribution. Assume the search list is divided into $2p$ blocks; the first p blocks are assigned as previously whereas the last p ones are distributed in *reverse*. Therefore, the algorithm proceeds as before with the strike carried on the two sub-blocks, except that *actors* are retrieved *only* in the first sub-block and a previous master needs not retired because he might have to strike other composite numbers (i.e., non prime numbers) in his second sub-block. An implementation of this scheme follows.

Sieve of Eratosthenes.

```
C---  This program implements the sieve of Eratosthenes for a range N.
C---  Load balancing requires N not less than 4p, p = number of procs.
C---  rbs@maths.uq.oz.au - High Performance Computing Unit - January, 1995
C---  University of Queensland - Australia.

      integer k, p, lastp, nbuf(4)
      integer msgTYPE, msgINFO, ISIZE, ALLGRP, BBMAX
      parameter( ISIZE=4, BBMAX=2*1000 )
      integer N,N2,M,BB,B1,B2,L1,L2,master,actor,index,counter,istart,i
      integer prime(BBMAX)

      external I_VADD ! vector-addition, supplied function.

C---  Header to pick-up wild card values...
      call mp_TASK_QUERY( nbuf,4,3 )
      ALLGRP = nbuf(4)

C---  Program body starts here...
      call mp_ENVIRON( p,k )
      lastp = p-1
      master = 0
      if ( k.eq.0 ) then
         read*,N
         print '("range: N =",I5)',N
      endif
      call mp_BCAST( N,ISIZE, 0, ALLGRP )
      if ( N.lt.4*p ) then
         print*,'Too many processors. N needs not be less than 4*p.'
         call mp_STOPALL( -1 )
      endif
      M = int( sqrt(dble(N)) )

C---  Organisation of blocks...
      N2 = N/2
      if ( N2*2.ne.N ) N2 = N2 + 1
      BB = mod( N2,2*p )
      B1 = (N2-BB)/(2*p)
      B2 = B1
      if ( k.lt.BB ) then
         B1 = B1+1
         L1 = B1*k
      else
         L1 = B1*k + BB
      endif
      if ( 2*p-k-1.lt.BB ) then
         B2 = B2+1
         L2 = B2*(2*p-k-1)
      else
         L2 = B2*(2*p-k-1) + BB
      endif
      BB = B1 + B2
      if ( BB.gt.BBMAX ) then
         print*,'Insufficient space for prime numbers.'
         call mp_STOPALL( -1 )
      endif
      do i = 1,BB
         prime(i) = 1
      enddo
      if ( k.eq.0 ) prime(1) = 0    ! 1 is not a prime number
      index = 0    ! shall contain the relative index of actor in the master
```

```
C---  Countdown starts now...
      DO
         if ( k.eq.master ) then   ! look for the next prime, i.e., actor
            actor = 0
            index = index + 1
            do while ( index.le.B1 .and. 2*(L1+index)-1.le.M )
               if ( prime(index).ne.0 ) then
                  actor = 2*(L1+index)-1
                  print*,"actor =",actor
                  exit
               endif
               index = index + 1
            enddo
C---        If no more prime, change master if necessary...
            if ( actor.eq.0 .and. 2*(L1+index)-1.le.M ) then
               if ( master.ne.lastp ) actor = -(master+1)
            endif
         endif

C---     The current master broadcasts the current actor to all...
         call mp_BCAST( actor,ISIZE, master, ALLGRP )

C---     Check if the countdown is finished...
         if ( actor.eq.0 ) exit

C---     Check if it is time to change the master...
         if ( actor.lt.0 ) then
            master = -actor
         else
C---        Look for the first multiple beyond actor^2 in the first half...
            istart = (actor*actor+1)/2 - L1
            if ( istart.le.0 ) then   ! actor^2 is behind, adjust forward
               i = -istart/actor + 1
               istart = istart + i*actor
            endif
C---        Strike in the first half...
            do i = istart,B1,actor
               prime(i) = 0
            enddo
C---        Look for the first multiple beyond actor^2 in the second half...
            istart = (actor*actor+1)/2 - L2
            if ( istart.le.0 ) then   ! actor^2 is behind, adjust forward
               i = -istart/actor + 1
               istart = istart + i*actor
            endif
C---        Strike in the second half...
            do i = B1+istart,BB,actor
               prime(i) = 0
            enddo
         endif
      ENDDO

C---  Collect the existing primes...
      counter = 0
      do i = 1,BB
         if ( prime(i).ne.0 ) then
           counter = counter + 1
D          if ( i.le.B1 ) print*,2*(L1+i)-1
D          if ( i.gt.B1 ) print*,2*(L2+i-B1)-1
         endif
      enddo
      print*,"counter =",counter
      call mp_REDUCE( counter, i, ISIZE, 0, I_VADD, ALLGRP )
      if ( k.eq.0 ) print *,"Number of primes:", i+1   ! Do not forget 2
      END
```

6 USING LOADLEVELER

This utility is worth of consideration right from the beginning for the system suffers from instabilities when operating interactively whereas everything seems to go smoothly under LoadLeveler. So its use may avoid you to be in trouble. As it was announced before, it is a utility package enabling the submission of batch jobs. It can be used in raw text mode from a standard Unix shell window (see the next coming remark) or through a graphical X-based interface in which case, a simple click replaces a command that would have been typed in the shell window. Under **ksh** (Korn Shell), you may proceed as follows to invoke LoadLeveler:

```
export DISPLAY=algebra.maths.uq.oz.au:0
xloadl &
```

Obviously the `DISPLAY` variable should be yours not mine! And if you know your IP address, you should better use it. Subsequently to these commands, a window is popped out and it contains information related to the current status of the batch queue. You need to specify your particular job command file:

1. Press "File", this yields another window menu
2. Press "File_Submit_Job"
3. Press "Edit", this will allow you to make up your own command file with **vi**, see the template below. Once the file is saved,
4. Press "Filter" to update the list of job command files
5. Select your own job from the list available
6. Press "Submit".

For illustration, here follows the job command file `poly.cmd` that have been successfully used for the parallel polynomial evaluation example. It represents a template; for the other examples, you just have to modify the string `poly` convenably (four occurrences) as well as the e-mail address in `notify_user`. Also, set your number of processors where it deems necessary.

Upon completion of the submission, result files are generated. Once you will be familiar, feel free to customise the command file according to your preferences.

```
#!/bin/ksh
# MPL job for parallel polynomial evaluation

# @ job_type       = PARALLEL
# @ input          = poly.in
# @ output         = poly.$(Cluster).out
# @ error          = poly.$(Cluster).err
# @ initialdir     = .
# @ requirements   = (Arch == "R6000") && (OpSys == "AIX32")
# @ notification   = never
# @ notify_user    = your_email
# @ class          = half_hour_dedicated
# @ checkpoint     = no
# @ environment    = MP_PROCS=4;MP_LABELIO=yes;MP_STDOUTMODE=ordered;MP_INFOLEVEL=2;
# @ min_processors = 4
# @ max_processors = 4
# @ queue
  poe poly
```

Table 6: `poly.cmd`: job command file for the polynomial example.

The input data file `poly.in` consists, in this case, in two lines (the degree n and the variable x). When your job don't need input data simply set `input = /dev/null`.

```
16
2.0
```

Table 7: `poly.in`: input data file for the polynomial example.

REMARK If you have not succeeded in using **xloadl** or you wish to use LoadLeveler from the shell window, you may proceed as indicated here. Make up your job command file (according to the template presented above and using the same means – text editor, etc – that you utilised to build your programs) then type:

```
llsubmit poly.cmd
```

Your job will be sent to the queue and you can retrieve information on the job queue with the command **llq**. Other commands include **llstatus**, **llcancel**, **llprio**, **llhold**. Refer to their man pages for details.

7 TROUBLESHOOTING

The price paid when using a new generation computer is the emergence of a few itchings! You will certainly undergo some oddly behaviours while running your programs. Please report them to enrich this section. The followings are the principal ones that I have encountered:

read/write mishmash
: Although in your code, the write (or print) statement is *after* the read statement, it may come that their actions are *inverted* on the screen. If you are not aware of this fact, you may think that your code has a bug or even that the system is going wrong. Don't panic, it results from I/O buffering. Bear this behaviour in mind!

prompt is not gave back
: When running a program *interactively*, it may happen that the prompt is not gave back although, undoubtedly, the program is already terminated. These problems disappear when using LoadLeveler (i.e., in batch mode).

8 CONCLUSION

The IBM SP2 is one of the latest massively parallel computer in the market. As a result it was designed using latest sophistications in technology. But high technology evolves rapidly and in addition massively parallel architecture and their use are not yet mature.

As time passes, we are gaining more expertise. Computer architects will provide us with more powerful computers and the greatest challenge is to program them efficiently. This note was governed by this primary concern, i.e., enable you to start using the SP2. Along with this, our intent was also to display glimpses of its documentation and its programming environment. We have supplied an illustrative sample of practical instances mindful of how case studies can help in clarifying certain points. This was in fact a weighted way to present a few specimens of built-in message passing functions. Our expectation now is that you are well equipped for a further deep-ending.

Acknowledgements. The author would like to thank the members of Queensland Parallel Supercomputing Facility (QPSF) in Griffith University and University of Queensland for their support and for their valuable comments.

9 APPENDIX – OFFICIAL MANUALS

IBM AIX Parallel Environment Programming Primer Rel 2.0
IBM AIX Parallel Environment Operation and Use Rel 2.0
IBM AIX Parallel Environment: Parallel Programming Subroutine Reference Rel 2.0

IBM AIX Parallel Environment Installation, Administration, and Diagnosis Rel 2.0

IBM LoadLeveler User's Guide
IBM AIX PVMe User's Guide and Subroutine Reference Rel 3.0

AIX Version 3.2 System User's Guide: Operating System and Devices
AIX Version 3.2 System User's Guide: Communications and Network

AIX Version 3.2 for RISC System/6000: Optimization and Tuning Guide for Fortran, C and C++

AIX Version 3.2 Commands Reference. Vol 1: ac through dumpfs
AIX Version 3.2 Commands Reference. Vol 2: e through lvlstmajor
AIX Version 3.2 Commands Reference. Vol 3: sa through ypxfr
AIX Version 3.2 Commands Reference. Vol 4: m4 through rwhod

Optimization Subroutine Library: Guide and Reference Rel 2

Engineering and Scientific Subroutine Library: Guide and Reference: Vol 1 Ver 2.2
Engineering and Scientific Subroutine Library: Guide and Reference: Vol 2 Ver 2.2
Engineering and Scientific Subroutine Library: Guide and Reference: Vol 3 Ver 2.2

AIX XL Pascal/6000 User's Guide Ver 2.1
AIX XL Pascal/6000 Language Reference Ver 2.1

AIX XL Fortran Compiler: What's New Technical Newsletter Ver 3.1.1
XL Fortran Compiler/6000: Specifications Ver 3.1.1
AIX XL Fortran Compiler/6000: User's Guide Ver 3.1.1
AIX XL Fortran Compiler/6000: Language Reference Ver 3.1.1

C Set ++ for AIX/6000: User's Guide Ver 2.1
C Set ++ for AIX/6000: C++ Language Reference Ver 2.1
C Set ++ for AIX/6000: C Language Reference Ver 2.1
C Set ++ for AIX/6000: Reference Summary Ver 2.1
C Set ++ for AIX/6000: Application Support Class Library Reference
C Set ++ for AIX/6000: Source Code Browser User's Guide Ver 2.1
C Set ++ for AIX/6000: Collection Class Library Reference
C Set ++ for AIX/6000: HeapView Debuggers User's Guide Ver 2.1

IBM Visualization Data Explorer User's Guide 5th Ed
IBM Visualization Data Explorer Programmer's Reference 4th Ed

See /opt/doc/. Release and version may have changed.