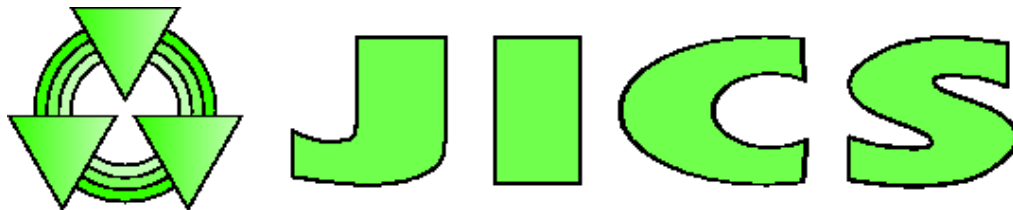


The Wayback Machine - <https://web.archive.org/web/20030619070712/http://www-jics.cs.utk.edu:80/SP2/SP...>



JOINT INSTITUTE FOR COMPUTATIONAL SCIENCE

Stephanie Wolf, Lisa Manne, and Todd Bryan

[Joint Institute for Computational Science](#)

[University of Tennessee](#)

Knoxville, TN 37996-1508, USA

Last Revision: June 23, 2000

Author's Email Address: jics@cs.utk.edu

A Beginner's Guide to the IBM SP

Copyright (C) 2000, Joint Institute for Computational Science

Retrieve a PostScript file with encapsulated color figures using [ghostview](#) **Acknowledgments**

Thanks to Maui High Performance Computing Center and Cornell High Performance Center for their on-line information and permission to use the documents and labs to promote education on the IBM SP2.

Thanks to Christian Halloy and Travis Kerr for critiquing and editing drafts of this guide.

Any mention of specific products, trademarks, or brand names within this document is for identification purposes only. All trademarks belong to their respective owners.

Contents

- [Foreword](#)
- [Introduction](#)
- [Accessing the IBM SP2 at UT](#)
- [IBM SP Hardware](#)
- [IBM SP Software](#)
- [Parallel Operating Environment \(POE\)](#)
 - [Important Environment Variables](#)
 - [.rhosts file](#)
 - [host.list file](#)
 - [Compiler Options](#)
- [Communication Methods](#)
- [Message Passing Interface \(MPI\)](#)
- [System Status Array](#)
- [Visualization Tool \(VT\)](#)
 - [Tracefiles in VT](#)
 - [Performance Monitoring with VT](#)
- [LoadLeveler](#)

- [LoadLeveler Overview](#)
- [A Couple of LoadLeveler Commands](#)
- [Backfilling on the LoadLeveler Queue](#)
- [XLoadLeveler Overview](#)
- [Additional Information on the IBM SP](#)
 - [InfoExplorer](#)
 - [Information Available on the World Wide Web](#)
 - [Typewritten Manuals and man Pages](#)
- [More on MPI](#)
- [PVM Hints](#)
- [About this document ...](#)

Foreword

This document provides an overview of the IBM RS/6000 SP for beginning users. We summarize the important features and capabilities of SP machines, and include concise information on hardware, software tools, and compilers. This information is essential for beginners who want to program and run effectively on a SP.

Whenever possible, our descriptions will cover the general class of SP machines; some sections, though, are specific to the SP2 installed at University of Tennessee, Knoxville.

Introduction

The IBM SP is a "Scalable POWERParallel system". Distributed memory parallel machines like the SP are also known as massively parallel processors (MPP). They provide a reasonably scalable architecture for parallel computing.

The SP systems are variants of IBM's RS/6000 line, which is built around the PowerPC, POWER2, and POWER3 microprocessors. The PowerPC and its derivative chips have a RISC (Reduced Instruction Set Computer) architecture and have increased from an original 60 MHz clock rate to current rates of 375 MHz or more. Older SP systems using the POWER2 chip were referred to as "SP2" machines. Newer installations using the POWER3 chips are sometimes called "SP3", but more often the names "SP2" or "SP" are used to refer to these systems.

An SP can consist of up to several hundred compute nodes all interconnected by a high performance switch system. The compute nodes may be uniprocessors (one CPU per node) or multiprocessors (multiple CPUs per node). The PowerPC chips used in the compute nodes are floating-point superscalar - they can perform more than one floating-point operation per clock cycle.

Parallel programs on the SP must be written using some form of message passing, since a process running on one compute node cannot access the memory of another compute node. Instead, data must be moved via messages by the programmer. The SP architecture and environment allows both SIMD (single instruction, multiple data) and MIMD (multiple instruction, multiple data) programming styles.

Accessing the IBM SP2 at UT

First, you must obtain an account on an SP machine. For UTK's SP2 you should fill in the account request form found at <http://www-jics.cs.utk.edu/account.html>. Once you have an account and password, you can begin to use the machine. Most sites prefer the use of Secure Shell (SSH) for remote access to an SP. If you are unsure if SSH is required or installed, ask the site administrator.

At UTK, one should use SSH to log into the SP2:

```
% ssh -l username hal.cas.utk.edu
```

If you are working on an X-Window terminal and wish to create new windows or open X-Window based tools on your workstation, type **xhost +** in a window on your workstation. Then, once you are logged in to the SP machine, you will want to set your DISPLAY environment variable by typing

```
% setenv DISPLAY workstation_name:0.0
```

There are also other important environment variables which may need to be set. These will be discussed later in this document.

IBM SP Hardware

So, how can an SP be configured? There can be 4 to 16 processors per frame (or cabinet) and 1 to 16 frames per system, although custom systems have been as large as 162 frames. Frames can contain thin, wide and high nodes. High nodes are symmetric multiprocessors and usually serve as interactive nodes for code development and job submission. Thin nodes are the primary type of compute node and may have single or multiple CPUs per node. Wide nodes are similar to thins but have the capacity for larger memory and a larger cache. Standard SP systems can contain up to 256 nodes. 512 nodes or more can be ordered from IBM by special request. UT has a 34 node (48 processor) SP2 and ORNL has a 184 node SP. Other notable academic installations include the 1,152 processor "Blue Horizon" SP at San Diego Supercomputing Center, and the 360 processor SP at North Carolina Supercomputing Center.

An optional high performance switch (HPS) for inter-processor communication is included on most systems. This switch can interconnect up to 16 nodes with up to 16 other nodes. More than one HPS may be included when the number of nodes in the SP machine is greater than 32. The current high performance switch on UTK's SP2 is a TB3 with a 150 MB/sec peak bi-directional bandwidth.

The IBM SP is fairly robust in that it has **concurrent maintenance**, which means each processor node can be removed, repaired, and rebooted without interrupting operations on other nodes.

Node Type	SP2 Thin (old/new)	SP2 High
Processor Type	POWER2SC	POWERPC 604
Peak Performance	480/640 MFLOPS	224 MFLOPS
Clock Speed	120/135 MHz	112 MHz
Memory Bus	256 bits	256 bits
Data Cache	128 KB	16 KB
Instruction Cache	32 KB	16 KB
Memory	256 MB	1 GB
Disk (not including 50GB home area)	2.2/4.4 GB	20.2 GB

Table 1: Processor Nodes on UTK's SP2 - Thin vs. High

IBM SP Software

Each node of an SP runs the AIX Operating System, IBM's version of UNIX. The SP has a Parallel Operating Environment, or POE, which is used for handling parallel tasks running on multiple nodes. The POE also

provides an implementation of the Message Passing Interface (MPI), as well as some important tools for debugging, profiling, and system monitoring. The software available for monitoring, profiling and debugging includes the Visualization Tool (VT) for parallel MPI codes, the Program Visualizer (PV) for serial code, XPVM for pvm codes, the **prof**, **gprof**, **xprof**, and **tprof** profilers, the Hardware Performance Monitor (HPM), the **pdbx** and **xpdbx** parallel debuggers, and other profiling tools.

The Parallel Virtual Machine (**PVM**) is available on the IBM SP, as well as MPI (Message Passing Interface). Some systems may even have High Performance FORTRAN (HPF) available. Important math libraries are available, such as PETSc, IBM's Parallel Engineering and Scientific Subroutine Library (**PESSL**), and others. There may also be software available to assist you in parallelizing codes, like the FORGE90 package. More information about what is available on the SP can be found on the WWW pages for your particular machine. At UT, the top-level SP2 page is

<http://www-jics.cs.utk.edu/SP2/>

and the listing of software tools may be found at

<http://www-jics.cs.utk.edu/SP2/SP2tools/tools.html>

Parallel Operating Environment (POE)

The POE consists of parallel compiler scripts, POE environment variables, parallel debuggers and profilers, IBM MPI, and parallel visualization tools. These tools allow one to develop, execute, profile, debug, and fine-tune parallel code.

A few important terms you may wish to know are **Partition Manager**, **Resource Manager**, and **Processor Pools**. The **Partition Manager** controls the partition - a group of nodes on which to run your program. The **Partition Manager** requests the nodes for your parallel job, acquires the nodes necessary for that job (if the Resource Manager is not used), copies the executables from the initiating node to each node in the partition, loads executables on every node in the partition, and sets up standard I/O.

The **Resource Manager** keeps track of the nodes currently processing a parallel task, and, when nodes are requested from the Partition Manager, it allocates nodes for use. The Resource Manager attempts to enforce a "one parallel task per node" rule.

The **Processor Pools** are sets of nodes dedicated to a particular type of process (such as interactive, batch, I/O intensive) which have been grouped together by the system administrator.

For more information about the configuration of your system, type **jm_status -P** on the SP machine.

Important Environment Variables

There are many environment variables and command line flags that can be set to influence the operation of POE tools and the execution of parallel programs. A complete list of the POE environment variables can be found in the IBM manual "AIX Parallel Environment Operation and Use". Some environment variables which Affect program execution are listed below.

MP_PROCS - sets the number of processes to allocate for your partition. Often one process is assigned to each CPU, but you can also run multiple processes on one CPU.

MP_RES - specifies whether or not the Partition Manager should connect to the Resource Manager to allocate nodes. If the value is set to "no", the Partition Manager will allocate nodes without going through the Resource Manager. The allocations are instead controlled by a user-supplied host list file (Section 6.3).

MP_RMPOOL - sets the number of the pool that should be used by the Resource Manager for non-specific node allocation. This is only valid if you are using the Resource Manager for non-specific node allocation (from a single pool) without a host list file (Section 6.3).

Information about available pools on a machine may be obtained by typing the command **jm_status -P** at the Unix prompt.

MP_HOSTFILE - specifies the name of a host file for node allocation. This can be any file name, NULL or ``". The default host list file is **host.list** in the current directory.

MP_HOSTFILE does not need to be set if the host list file is the default, **host.list**.

A host list file (see also Section 6.3) must be present if either:

- 1) specific node allocation is required,
- 2) non-specific node allocation from a number of system pools is requested, or
- 3) a host list file named something other than the default **host.list** will be used.

MP_SAVEHOSTFILE - names the output host file to be generated by the partition manager. This designated file will contain the names of the nodes on which your parallel program actually ran.

MP_RETRY - specifies the period of time (in seconds) between allocation retries if not enough processors are available.

MP_RETRYCOUNT - specifies the number of times the partition manager should attempt to allocate processors before returning without having run your program.

MP_EUILIB - specifies which Communication SubSystem (CSS) library implementation to use for communication. Set to **ip** (for Internet Protocol CSS) or **us** (for User Space CSS, which allows one to drive the high-performance switch directly from parallel tasks without going through the kernel or operating system).

MP_EUIDEVICE - sets the adaptor used for message passing: **en0** (Ethernet), **fi0** (FDDI), **tr0** (token-ring), or **css0** (the high-performance switch adaptor). [Note: This variable is ignored if the **us** CSS library is used as above.]

MP_EUIDEVELOP - whether MPI should do more detailed checking during program execution. (This takes a value of yes or no).

MP_TRACEFILE - name of the trace file to be created during the execution of a program where tracing commands appear within the code.

MP_TRACELEVEL - level of VT tracing (0 = NONE, 9 = ALL trace records are on, 1, 2, & 3 are each a different combination of some trace records).

MP_INFOLEVEL - amount of diagnostic information your program displays as it runs (0 - 5 with 0 being the least amount of information given.)

You may wish to use a shell script to set the appropriate environment variables or you could set important environment variables in your **.login** file.

.rhosts file

Some IBM SP systems will already have a **.rhosts** file set up for you when you receive your account. However, on other systems, you may have to set up your own **.rhosts** file. This file should include the names of all the

nodes and switches you may ever want to use.

Make sure that the node you are logged onto is in your .rhosts file. If it is not, you will want to add that node into the .rhosts file to avoid problems.

A sample .rhosts file on UT's SP2 might look similar to this (but contain more lines):

```
f1n1.cas.utk.edu
f1n2.cas.utk.edu
f1n3.cas.utk.edu
f1n4.cas.utk.edu
```

Some systems require you to place your user_name on each line after the node or switch name. Codes running with PVM require that the .rhosts file include the names of the switches that access each node, for example **sw1n[1-16].cas.utk.edu** for frame 1 nodes at UTK. You will not need the dot extensions if you wish to access nodes of only that machine. Ask your system help about the naming of the nodes on the system you wish to use.

host.list file

The host file should contain a list of all the nodes (or pools of nodes, but not both) you wish to run your code on. The first task will be run on the first node or pool listed, the second task will run on the second node or pool listed, etc. If you are using pools in the host file and do not have enough pools listed for all the tasks, the last tasks will use additional nodes within the last pool listed. However, if you are listing nodes, you must have at least as many nodes listed in the host file as you wish to run. You are allowed to repeat a node name within a host file. Doing so will cause your program to run multiple tasks on one node.

The default host file is host.list, but you can change the MP_HOSTFILE environment variable to be some other file name. If you have decided to run your code on ONE pool and have set MP_HOSTFILE to Null and RM_POOL to the appropriate pool number, you do not need to have a host file.

A sample host file on UT's SP2 using nodes:

```
!This is a comment line
!Use an exclamation at the beginning of any comment
f1n1.cas.utk.edu shared multiple
f1n2.cas.utk.edu shared multiple
f1n3.cas.utk.edu shared multiple
f1n4.cas.utk.edu shared multiple
!
!Host nodes are named f1n1, f1n2, f1n3, and f1n4
!When using MPL, shared means you share with others.
!multiple means you allowing multiple MPL tasks on one node.
!
!dedicated in place of shared or multiple means you do not want
!to share with other people's MPL tasks, or you do not want
!to allow multiple MPL tasks of your own on one node.
```

A sample host file using pools:

```
!This line is a comment
@0 shared multiple
@1 shared multiple
@3 shared multiple
@0 shared multiple
!0, 1, and 3 are the pool numbers.
!Again, shared means you share with others.
!multiple means you allow multiple tasks on one node.
```

In this example, one node is chosen from pool 0 by the Resource daemon for the first task, one node from pool 1 is chosen for the next task, one node from pool 3 is chosen for the following task, and the nodes for any remaining task(s) are chosen from pool 0.

Compiler Options

The following compiler flags are available for both Fortran (**xlf**, **mpxlf**) or C (**xlc**, **mpcc**) compilers in addition to the usual flags available for these compilers.

-p or **-pg** provides information necessary for the use of the profilers **prof** or **gprof**, respectively.

-g makes the compiled program suitable for debugging with **pdbx** or **xpdbx**. This option is also necessary to use the Source Code view in the Visualization Tool, **vt**.

-O optimize the output code. (The **-O** can be followed by an optimization level: **-O2**, for example.) Remember, optimize only if NOT debugging. (One cannot use **-O** with **-p**, **-pg**, or **-g**.)

-ip causes the IP CSS library to be statically bound with the executable. Communication during execution will use the Internet Protocol.

-us causes the US CSS library to be statically bound with the executable. Communication will occur over the high-performance switch.

(If neither **-ip** nor **-us** is used at compile time, a CSS library will be dynamically linked with the executable at run time. This library is determined by the **MP_EUILIB** environment variable.)

-qarch=pwr2 (at UT and for FORTRAN only) allows the program to take advantage of the SP2's particular architecture, and could yield a speed-up of 50%. See the man page for **xlf** for more information.

Communication Methods

There are two types of communication methods available on the IBM SP2. these are the User Space protocol (**us**) and the Internet protocol (**ip**). The User Space protocol is much quicker, but does not allow one to share the communicating nodes with other IBM Message Passing Interface (MPI) processes. This User Space protocol always uses the high performance switch. The Internet Protocol (**ip**) is slower, but allows communicating nodes to be shared by other MPI processes. One can use the Internet Protocol over the ethernet or the high performance switch (which is quicker than using the ethernet).

Message Passing Interface (MPI)

The Message Passing Interface or MPI is a standard for message passing communication inside parallel programs. Since MPI is just a standard, it must be implemented by researchers or vendors. There are free MPI implementations available, like MPICH or LAM, but IBM provides their own MPI library for the SP systems. It is highly optimized for the SP and should outperform other implementations.

MPI allows both process-to-process communication and collective (or global) communication. Process-to-process communication is used when two processes communicate with one another using sends and receives. Collective communication is used when one process communicates with several other processes at one time using broadcast, scatter, gather, or reduce operations.

A summary of basic MPI commands and syntax is in an appendix to this manual. To compile a MPI program on the SP, one can use the special compile script ``mpcc''.

System Status Array

This is an X-Windows analysis tool, which allows a quick survey of the utilization of processor nodes. You can get into the System Status Array by typing **poestat&** Within this System Status Array, each node is represented by a colored square:

Pink squares

- nodes with low or no utilization.

Yellow squares

- nodes with higher utilization.

Green frames within the squares

- nodes with running parallel MPL processes.

Grey squares

- nodes not available for monitoring.

To the right appears a list of node names; nodes are listed in order from left to right and from top to bottom.

Remember, if you want the status of all active MPL jobs running on the nodes, the command is **jm_status -j**. This could be lengthy if many jobs are running on the SP2 at the time.

Visualization Tool (VT)

The Visualization Tool(VT) is a graphical user interface which enables you to perform both trace visualization and (real time) performance monitoring within IBM's Parallel Environment. Note that this tool is only useful in the monitoring and visualization of MPI jobs.

You can get into VT by typing
vt &

OR by typing **vt tracefile &**
where **tracefile** is the name of a previously created tracefile.

Many ``views'' are also available for looking into Computation and Communication, and System, Network, and Disk utilizations under trace visualization. All of these ``views'' except Communication are available under real-time performance monitoring. These ``views'' can be in the form of pie charts, bar charts, and grids. There is one 3-D representation of processor CPU utilization.

Tracefiles in VT

One can create a tracefile for future viewing through VT in the following manner. First add trace start and trace stop calls into the program to indicate where in the code to begin the trace, where to stop the trace and what to trace through the setting of the trace flag.(0 = no tracing, 1 = trace application markers from the Program Marker Array only, 2 = trace kernel statistics(such as cpu utilization, number of system calls, number of page faults, etc.) and application markers, 3 = trace the MPL/MPI message passing and collective communications only, 9 = trace everything listed above) One should always set the trace flag to include the least amount of data necessary. Then, run the program (using the -g option if one will want to view the source code as one plays the tracefile back again. (By default, VT will name the tracefile the same as the program name specified with the -o option when

the code was compiled followed by the .trc extension.) Then, call VT using **vt -tfile tracefile.trc**, where tracefile.trc is the name of the tracefile one has already created.

The calls required inside FORTRAN code for tracing:

```

INTEGER FUNCTION VT_TRC_START
INTEGER FUNCTION VT_TRC_STOP
INTEGER FLAG, VTERR

FLAG = 3
VTERR = VT_TRC_START(FLAG)

...

...

...

VTERR = VT_TRC_STOP

```

The calls required within C code for tracing:

```

int VT_trc_start( int );
int VT_trc_stop();
int flag, vterr;

flag = 3; /* This could also be 1, 2, or 9 as listed above */
vterr = VT_trc_start(flag);

...

...

...

vterr = VT_trc_stop();

```

Performance Monitoring with VT

Performance monitoring allows one to view kernel statistics and/or current MPI activity on SP nodes. (MPI communications between the nodes cannot be monitored in real time).

One begins performance monitoring by selecting "Performance Monitoring" under the "File" pull-down menu.

Two new windows will appear, the Performance Monitor and the Performance Monitor View Selector Windows.

One can select nodes for viewing from the Performance Monitor Window by clicking on job list, the node matrix, or the node list. The selected nodes will appear in the Selected Nodes area at the bottom of the window.

Then, one should select the views one wishes to monitor from the Performance Monitor View Selector Window. Then, one should select the "Toggle Monitoring" from the "File" pull-down menu in the Performance Monitor Window. Note: if one wishes to select different or additional nodes and/or views, one must first toggle the performance monitoring off, select the new nodes and/or views, and then, toggle monitoring on again.

More information about VT can be found using the InfoExplorer. Also, the Maui High Performance Computing Center has some very detailed information available on the World Wide Web. Their URL is listed in the "More Information" section of this document.

LoadLeveler

The LoadLeveler is a batch scheduling system available for the SP through IBM. It provides a facility for building, submitting and processing serial or parallel (PVM or MPI) batch jobs within a network of machines. LoadLeveler scheduling matches user-defined job requirements with the best available machine resources. A user is allowed to load level his or her own jobs only.

LoadLeveler Overview

The entire collection of machines available for LoadLeveler scheduling is called a ``pool". Note that this is NOT the same as the processor, or node pools discussed earlier. The LoadLeveler ``pool" is the group of nodes which the LoadLeveler manages. On Cornell's SP2 these nodes are the nodes in the sub-pools called ``batch". (A sub-pool is a smaller division of a larger node pool.) On some smaller machines, the LoadLeveler pool includes every node on the machine. Every machine in the pool has one or more LoadLeveler daemons running on it.

The LoadLeveler pool has one Central Manager (CM) machine, whose principal function is to coordinate LoadLeveler related activities on all machines in the pool. This CM maintains status information on all machines and jobs, making decisions about where jobs should run. If the Central Manager machine goes down, job information is not lost. Jobs executing on other machines will continue to run, while jobs waiting to run will start when CM is again restarted, and other jobs may continue to be submitted from other machines. (Such jobs will be dispatched when the Central Manager is restarted.) Normally, users do not need to know about the Central Manager.

Other machines in the pool may be used to submit jobs, execute jobs, and/or schedule submitted jobs (in cooperation with the Central Manager).

Every LoadLeveler job must be defined in a job command file whose filename is followed by **.cmd**. Only after defining a job command file may a user submit the job for scheduling and execution. Note that job command files are shell scripts and may contain shell commands after the LoadLeveler directives.

A job command file, such as *Sample1.cmd* in the following examples, can be submitted to the LoadLeveler by typing

lsubmit Sample1.cmd

Lines in a **.cmd** file that begin with a # not followed by a @ are considered comment lines, which the LoadLeveler ignores. Lines that begin with a # followed by a @ (even if these two symbols are separated by several spaces) are considered to be command lines for the LoadLeveler.

Listed below are five sample **.cmd** files.

Sample1.cmd is a simple job command file which submits the serial job **pi** in the `~jsmith/labs/poe/C` subdirectory once. *Sample2.cmd* submits that same serial job (in the same directory) four different times, most likely on four different SP2 nodes. *Sample3.cmd* is a script **.cmd** file which submits a parallel job.

Sample1.cmd:

```
#!/bin/sh
#
#The executable is ~/labs/poe/C/pi in user jsmith's home directory
#
#The serial job is submitted just one time
#
# @ executable = /user/user14/jsmith/labs/poe/C/pi
#   No input is required
```

```
# @ input = /dev/null
# @ output = sample1.out
# @ error = sample1.err
# @ notification = complete
# @ checkpoint = no
# @ restart = no
# @ requirements = (Arch == "R6000") && (OpSys == "AIX42")
# @ queue
```

Sample2.cmd:

```
#!/bin/sh
#
#The executable is ~/labs/poe/C/pi in user jsmith's home directory
#
#This submits the serial pi job four times by listing "queue" four times.
# Starting on August 18, 2001 at 4:35 PM
#
#@ executable = /user/user14/jsmith/labs/poe/C/pi
# If the input should be read from sample2.in
#@ input = sample2.in
#@ output = sample2.out
#@ error = sample2.err
#@ startdate = 16:35 08/18/01
#@queue
#@queue
#@queue
#@queue
```

Sample3.cmd:

```
#!/bin/sh
#
#The executable is ~/labs/poe/C/pi in user jsmith's home directory
#
# This submits the code three times with a unique input file,
# output file, and error file for each run
#
#@ executable = /user/user14/jsmith/labs/poe/C/pi
#@ input = sample3.in1
#@ output = sample3.out1
#@ error = sample3.err1
#@queue
#@ input = sample3.in2
#@ output = sample3.out2
#@ error = sample3.err2
#@queue
#@ input = sample3.in3
#@ output = sample3.out3
#@ error = sample3.err3
#@queue
```

Sample4.cmd:

```
#!/bin/sh
#
#The executable is ~/labs/poe/C/pi in user jsmith's home directory
#
# Six serial jobs with unique input, output, and error files are
# submitted.
#
#@ executable = /user/user14/jsmith/labs/poe/C/pi
# If the input should be read from sample2.in
#@ input = sample2.in.$(Process)
#@ output = sample2.out.$(Cluster).$(Process)
```

```

#@ error = sample2.err.$(Cluster).$(Process)
#@queue
#@queue
#@queue
#@queue
#@queue
#@queue
#@queue

```

Sample5.cmd:

```

#!/bin/csh
#The executable is ~/labs/poe/C/pi_reduce in jsmith's home directory
#
#This time, a script command file is used to submit a parallel job
#
#@ job_name      = pi_reduce
#@ output       = sample3.out
#@ error        = sample3.err
#@ job_type     = parallel
#@ requirements  = (Adapter == "hps_user")
#@ min_processors = 4
#@ max_processors = 4
#@ environment  = MP_INFOLEVEL=1;MP_LABELIO=yes
#@ notification = complete
#@ notify_user  = jsmith@cs.utk.edu
# This sends e-mail to jsmith@cs.utk.edu once the job has been submitted
#@ queue
echo $LOADL_PROCESSOR_LIST >! sample3.hosts
/usr/lpp/poe/bin/poe /user/user14/jsmith/labs/poe/C/pi_reduce

```

Script **.cmd** files similar to the *Sample5.cmd* file shown here are necessary for parallel jobs. After the LoadLeveler processes its command lines, the rest of the script file is run as the executable.

A Couple of LoadLeveler Commands

To submit a **.cmd** file, type **llsubmit filename.cmd**, where *filename.cmd* is the name of the command file one has created. (**.cmd** is generally the extension recommended for command files).

To view the jobs in the LoadLeveler queue, type **llq**. This will show the jobs, their job class, and who submitted the job.

For more information on LoadLeveler and XLoadLeveler commands and **.cmd** file command lines, try browsing through the information available on the InfoExplorer. (The InfoExplorer is discussed in the Additional Information section of this SP2 guide.)

For more information on PVM LoadLeveler scripts, view the web pages for the particular IBM SP2 site where the scripts will be run.

Backfilling on the LoadLeveler Queue

Backfilling occurs when a process - one which requires many nodes to run - must wait for enough free nodes. Instead of blocking smaller processes from running in the interim, LoadLeveler ``backfills" the queue by running smaller jobs while waiting for enough free nodes to run the large job. This allows a LoadLeveler to run more jobs in an allotted amount of time. You may be able to take advantage of backfilling by asking for a short time limit on your job. Estimate your run time carefully, and don't ask for more time than you need. Doing this will help minimize the time your job spends waiting in the queue.

XLoadLeveler Overview

One can get into the XLoadLeveler, the X-Windows version of the LoadLeveler, while on the SP2 by typing **xloadl&**

A large window sectioned into three parts will appear on the screen. The three sub-windows are the **Jobs** window, the **Machines** window, and the **Messages** window.

The **Jobs** Window will list the current jobs that have been submitted to LoadLeveler, whether they are running or not. This window also allows one to build jobs, prioritize them, cancel or hold them, and to display each job along with its status. A newly built job can be saved into a **.cmd** file for further use, such as job submission.

The **Machines** Window lists the machines, or nodes, available to the LoadLeveler CM. From this window, you can also create jobs, prioritize them, stop, or hold them. Jobs can be displayed by the machines they run on.

The **Messages** Window gives information on LoadLeveler activities. (Each activity is time-stamped.)

The best way to learn more about this X-Windows tool is to actually get into the XLoadLeveler and try it. The more you practice using a tool such as the XLoadLeveler, the better you will understand what the tool is capable of doing.

Additional Information on the IBM SP

InfoExplorer

The InfoExplorer is a powerful Window-based information tool which allows the user to access some useful reference manuals for the SP2.

To get information specific to the *parallel environment*, type **info -l pe &**

The **-l** option allows one to look into a particular library available on the InfoExplorer. Other libraries may also be interesting to view. Two of these are the **mpxlf** and **mpcc** compiler libraries.

To get general information, type **info &**.

Again, the best way to learn this, or any other, X-based tool is to actually get into the tool and use it.

Information Available on the World Wide Web

Here are some URLs that may also prove useful:

<http://www-jics.cs.utk.edu/sp2info.html>
 * <http://www.mhpcc.edu/>
<http://www.tc.cornell.edu>
<http://www.mcs.anl.gov/computing/machines/quad>
<http://www.qpsf.edu.au/sites/sites.html>
<http://math.nist.gov/~KRemington/Primer/tutorial.html>
<http://ibm.tc.cornell.edu/ibm/pps/doc/primer/>
<http://lscftp.kgn.ibm.com/pps/aboutsp2/sp2sys.html>
<http://www.netlib.org/utk/icl/xpvm/xpvm.html>
<http://www.ptools.org>

Typewritten Manuals and man Pages

``IBM AIX Parallel Environment - Parallel Programming Subroutine Reference"

``IBM AIX Parallel Environment - Operations and Use"

``Parallel Programming with MPI on Clusters of Workstations and on the IBM SP2 (postscript version)."

Presentation materials from the Joint Institute for Computational Science, UTK.

Man pages for the SP2. (Use the C language name for the routine when looking for the FORTRAN version.)

More on MPI

Every MPI program must make a call to initialize the MPI communication layer, as well as a call to deallocate the layer when done.

In FORTRAN:

```
CALL MPI_INIT(IERROR)
(** program **)
CALL MPI_FINALIZE(IERROR)
```

In C:

```
MPI_Init(int *argc, char ***argv);
/* program */
MPI_Finalize();
```

The basic blocking send and receive operations are:

If you are using FORTRAN:

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS,
         IERROR)
```

Or, if you are using C:

```
MPI_Send(void *buf, int count, MPI_Datatype type, int dest,
         int tag, MPI_Comm comm);
MPI_Recv(void *buf, int count, MPI_Datatype type, int source,
         int tag, MPI_Comm comm, MPI_Status *status);
```

where

buf - memory buffer containing the data to be sent or received. (in FORTRAN, this is simply the name of the buffer; in C, it is the buffer's corresponding address.)

count - number of data items to send/receive

type - datatype of items to be sent/received

dest - id number (rank) of the process, or task, to which the message is being sent

source - id number of the source from which you want to receive a message

tag - user-defined non-negative integer used to identify the messages transferred (perhaps you want one process to receive only messages tagged with 12 at some point in the program execution).

comm - group (communicator) within which this message is valid

status - MPI status structure for error reporting

Sends and receives can be blocking or non-blocking. Those that are blocking wait until the send or receive has finished before continuing with other instructions. The non-blocking ones continue with other instructions even when the send or receive has not yet finished.

The non-blocking send and receives in FORTRAN are **MPI_ISEND** and **MPI_IRECV**. Other MPI commands (in FORTRAN) which you may find useful are:

MPI_COMM_SIZE returns the total number of tasks.

MPI_COMM_RANK returns the rank of the calling process within a group

For example, in Fortran, use **CALL MPI_COMM_SIZE(COMM,SIZE,IERROR)**
CALL MPI_COMM_RANK(COMM,RANK,IERROR)

In C, use **MPI_Comm_size(comm, &size)**

MPI_Comm_rank(comm, &rank)

MPI_PROBE checks whether a message has arrived yet.

MPI_TEST provides a non-blocking check of the status of a specified non-blocking send or receive, and

MPI_WAIT which blocks to "wait" for a non-blocking send or receive to finish.

Many other MPI operations are available. You may find more information on those, as well as the appropriate syntax for the above-listed commands in IBM's "AIX Parallel Environment - Parallel Programming Subroutine Reference", the online man pages, or the InfoExplorer (discussed in the Additional Information section of this document).

PVM Hints

First of all, one will need to look into the web pages for the site where the SP which one will be working on is located. There should be more information on PVM and who to contact whenever you have a problem.

PVM is often used for its portability of application. However, PVM is quite interesting to learn. Often there may be problems with printing or memory. Sometimes the error messages may be quite cryptic. A set of codes could work fine for some cases and crash and/or give strange error messages for other cases.

Hints:

1. Children Task Output Printing-

If one is having trouble getting the output from the children tasks to print to the `/tmp/pvml.userid`, where `userid` is one's id number on the machine, (this number can usually be obtained by typing `id` on the machine), one must first be certain the parent process does not exit before the children are finished printing. If the parent has not exited and the information from the children tasks is still not showing up in the correct file, one can get into the PVM console by typing `pvm` and then type `reset` at the `pvm>` prompt. The messages which may not have been written into the `/tmp/pvml.userid` before PVM had been reset should be there after the reset.

Another option is setting the catchout to 1, or turning on the catchout of PVM on. Catchout doesn't allow the parent to exit until after all children have printed any output.

2. Starting the Group Server -

The group server is necessary for working with groups to broadcast, scatter, and gather information, etc., from a set or sets of tasks. One has to start the group server manually by typing `pvmgs &` before executing

the codes if the group server is required. Otherwise, the code will crash with the first group call.

3. Dimensions of Arrays in C and FORTRAN -

One should remember how two-dimensional arrays are stored in the programming language of C or FORTRAN and make sure that messages get sent correctly. Also, one must remember that sending the starting location and number of filled positions in a two-dimensional array MAY not effectively send the information. Assume the array is dimensioned as $M \times M$, but only $N \times N$ spaces have been filled. (N is less than M). Sending the starting position and the number $N \times N$ will send M positions in the first row or column (depending on whether one is using C or FORTRAN), M positions in the second row or column, and so on until the number $N \times N$ has been reached. In actuality, this was not the intended message. One could either send all $M \times M$ positions in the matrix or pack and send just the information in the $N \times N$ portion of the matrix.

4. Message Size and Number of Messages Sent -

Remember that smaller amounts of large messages tends to be better than massive amounts of small message communications in PVM. However, the daemon must save a message in its buffer until a process can receive the message. Once the daemon has expanded to hold a message of a particular size, the daemon's buffer remains at least that large. Therefore, if memory becomes a problem, one should try smaller messages. This can be a process of trial and error.

5. Running PVM on the SP2 When running PVM over a homogeneous network or an SP2, use PVMDataRaw when sending messages. Otherwise, use PvmDataDefault when sending messages.

`pvm_psend` (`pvmfpsend`) and `pvm_prekv` (`pvmfprekv`) are quicker forms of communication than packing and sending and receiving and unpacking.

6. Site Specifics -

One is well advised to know the amount of real and CPU time one is allowed to use on a machine, how often the processes are erased from machines, around what time during the day or night such a "clean up" may occur, and what procedures are required for reserving time on certain machines.

About this document ...

A Beginner's Guide to the IBM SP

This document was generated using the [LaTeX2HTML](#) translator Version 96.1 (Feb 5, 1996) Copyright © 1993, 1994, 1995, 1996, [Nikos Drakos](#), Computer Based Learning Unit, University of Leeds.

The command line arguments were:
`latex2html -split 0 sp2guide.tex.`

The translation was initiated by JICS on Fri Jun 23 14:33:24 EDT 2000

JICS

Fri Jun 23 14:33:24 EDT 2000