Workshop on Teaching Cluster Computing (WTCC)

Tuesday, December 18th - Thursday, December 20th, 2001

Organized by

IEEE Computer Society, India
Gujarat Section

Department of Computer Science
Gujarat University
Ahmedabad, India
Instructor

Barry Wilkinson
Professor
Department of Computer Science
University of North Carolina at Charlotte
Email: abw@uncc.edu (preferred)
Tele: USA (704) 687 4879
Goals

• To help faculty teach cluster computing

• Provide teaching materials and experience

• Generate a dialog discussing all aspects of cluster computing with respect to undergraduate and graduate instruction.
Key problems

• Although quite reasonable to teach cluster computing at the undergraduate level, syllabus not well-defined.

• Very few practical textbooks - existing parallel programming/computing books often either theoretical or superficial.

• Message-passing software ok for teaching (MPI/PVM) but software for shared memory and distributed shared memory programming on clusters not well established for teaching.
# TIMETABLE

**Tuesday, December 18th, 2001**

**Basic Message-Passing Cluster Computing Techniques**

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
</table>
| 9:00 am - 10:30 am | Welcome  
Cluster Computing  
Message passing fundamentals |
| 10:30 am - 11:00 am | Break |
| 11:00 am - 12:30 pm | Overview of parallel techniques |
| 12:30 pm - 1:30 pm | Lunch |
| 1:30 pm - 2:45 pm | Parallel techniques continued |
| 2:45 pm - 3:00 pm | Break |
| 3:00 pm - 5:00 pm | Hands-on cluster computing  
- - Getting started |
<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 am - 10:30 am</td>
<td>Intro. to shared memory computing</td>
</tr>
<tr>
<td>10:30 am - 10:45 am</td>
<td>Break</td>
</tr>
<tr>
<td>10:45 am - 12:30 pm</td>
<td>DSM computing</td>
</tr>
<tr>
<td>12:30 pm - 1:30 pm</td>
<td>Lunch</td>
</tr>
<tr>
<td>1:30 pm - 2:45 pm</td>
<td>Algorithms and applications</td>
</tr>
<tr>
<td>2:45 pm - 3:00 pm</td>
<td>Break</td>
</tr>
<tr>
<td>3:00 pm - 5:00 pm</td>
<td>Hands-on cluster computing</td>
</tr>
</tbody>
</table>
Thursday December 20th, 2001
Creating an Undergraduate Cluster Computing Course

9:00 am - 10:30 am  Cluster computing courses at UNCC
10:30 am - 10:45 am  Break
10:45 am - 12:30 pm  Assignments and projects
12:30 pm - 1:30 pm  Lunch
1:30 pm - 3:00 pm  Forum/Discussion
3:00 pm - 3:15 pm  Break
3:15 pm - 4:00 pm  Wrap-up, questionnaire
  Follow-up support
Workshop Materials

Textbooks


• Instructors manual for above available to faculty by email.


On-line materials at:

http://ww.cs.uncc.edu/~abw/WTCC/

Includes:

• Workshop slides (about 500 slides)
• Link to extensive home page providing instructions to students to use software, course slides, papers, etc.
Acknowledgements

The instructor wishes to record his appreciation to Raj Buyya co-chairman of the IEEE task force on cluster computing for suggesting this workshop and to Professor Akshai Aggarwal for his work in organizing the workshop.

The instructor also wishes to thank Petra Recter and Greg Doench of Prentice Hall Inc. for providing textbooks without charge for workshop participants.

Finally, the instructor wishes to mention his appreciation to Dr. Andrew Bernat, Program Director of the National Science Foundation for suggesting and fully supporting the previous workshop from which is workshop is based.
Cluster Computing
Cluster Computing

Using a group of locally interconnected computers to solve a problem.

Motives

Usually faster computation

Other motives include: fault tolerance, larger amount of memory available, ...

Became interesting when cost of commodity computers became low and their performance increased in the early 1990’s.

Originally terms such as networks of workstations (NOWs), cluster of workstations (COWs) were used to describe cluster computing.
Background

Parallel computers - computers with more than one processor - and their programming - parallel programming - has been around for more than 40 years.
Gill writes in 1958:

“... There is therefore nothing new in the idea of parallel programming, but its application to computers. The author cannot believe that there will be any insuperable difficulty in extending it to computers. It is not to be expected that the necessary programming techniques will be worked out overnight. Much experimenting remains to be done. After all, the techniques that are commonly used in programming today were only won at the cost of considerable toil several years ago. In fact the advent of parallel programming may do something to revive the pioneering spirit in programming which seems at the present to be degenerating into a rather dull and routine occupation ...”

What is different now

Do not need access to supercomputers.

All institutions can establish their own parallel computing platform at little cost and the programming tools are readily available.

Can use commodity computers or computers in labs already there for general computing. Easy to upgrade over time.

Generally better to establish a dedicated cluster of computers.
At UNCC

Simply used computers displaced from labs during upgrades. *No cost whatsoever.*

We did purchase quad and dual Pentium systems this year and some gigabit ethernet connections to add to our teaching cluster of eight SUN’s that were taken from labs.
UNCC Department of Computer Science Teaching Cluster

Original cluster

SUN Ultra’s

100Mbs Ethernet switch

Newer systems being added

Dual Pentium Quad Pentium

College of Engineering network

Gigabit Ethernet

Gigabit Ethernet switch

100Mbs Ethernet up-links

× 6
Message-Passing Computing

A review of basic concepts
Message-Passing Multicomputer

Complete computers connected through an interconnection network:
Programming

Involves dividing problem into parts that are intended to be executed simultaneously to solve the problem.

Each part executed by separate computers.

Parts (processes) communicate by sending messages - the only way to distribute data and collect result.
Message Passing Parallel Programming
Software Tools for Clusters

Parallel Virtual Machine (PVM) - developed in late 1980’s.
Became very popular.

Message-Passing Interface (MPI) - standard defined in 1990s.

Both provide a set of user-level libraries for message passing. Use with regular programming languages (C, C++, ...).
Basics of Message-Passing Programming using user-level message passing libraries

Two primary mechanisms needed:

1. A method of creating separate processes for execution on different computers

2. A method of sending and receiving messages
Single Program Multiple Data (SPMD) model

Different processes merged into one program. Within program, control statements select different parts for each processor to execute. All executables started together - static process creation.

Basic MPI way

Compile to suit processor

Executables

Source file

Processor 0

Processor n – 1
Multiple Program Multiple Data (MPMD) Model

Separate programs for each processor. Master-slave approach usually taken. One processor executes master process. Other processes started from within master process - **dynamic process creation**.

![Diagram of process execution](image)
Basic “point-to-point” Send and Receive Routines

Passing a message between processes using send() and recv() library calls:

Generic syntax (actual formats later)
Synchronous Message Passing

Routines that actually return when message transfer completed.

Synchronous send routine

Waits until complete message can be accepted by the receiving process before sending the message.

Synchronous receive routine

Waits until the message it is expecting arrives.

Synchronous routines intrinsically perform two actions: They transfer data and they synchronize processes.
Synchronous `send()` and `recv()` library calls using 3-way protocol

(a) When `send()` occurs before `recv()`

- Process 1
  - `send();`
  - Suspend process
  - Both processes continue

- Process 2
  - `recv();`
  - Request to send
  - Acknowledgment
  - Message
  - Both processes continue

(b) When `recv()` occurs before `send()`

- Process 1
  - `send();`
  - Request to send
  - Message
  - Acknowledgment

- Process 2
  - `recv();`
  - Suspend process
Asynchronous Message Passing

Routines that do not wait for actions to complete before returning. Usually require local storage for messages.

More than one version depending upon the actual semantics for returning.

In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.
MPI Definitions of Blocking and Non-Blocking

**Blocking** - return after their local actions complete, though the message transfer may not have been completed.

**Non-blocking** - return immediately.

Assumes that data storage to be used for transfer not modified by subsequent statements prior to being used for transfer, and it is left to the programmer to ensure this.

Notices these terms may have different interpretations in other systems.)
How message-passing routines can return before message transfer completed

*Message buffer* needed between source and destination to hold message:

```plaintext
Process 1

```

```plaintext
send();
```

```
Message buffer
```

```plaintext
recv();
```

```
Process 2

```

Time

Continue process

Read message buffer

© Barry Wilkinson 2001. All rights reserved. This material is the property of Professor Barry Wilkinson and for sole and exclusive use of the participants of the workshop on teaching cluster computing (WTCC) held at Gujarat University, Ahmedabad, India, Tuesday, December 18th, 2001 - Thursday, December 20th, 2001. It is not to be sold, reproduced, or generally distributed.
Asynchronous (blocking) routines changing to synchronous routines

Once local actions completed and message is safely on its way, sending process can continue with subsequent work.

Buffers only of finite length and a point could be reached when send routine held up because all available buffer space exhausted.

Then, send routine will wait until storage becomes re-available - *i.e then routine behaves as a synchronous routine.*
Message Tag

Used to differentiate between different types of messages being sent.

Message tag is carried within message.

If special type matching is not required, a *wild card* message tag is used, so that the `recv()` will match with any `send()`.
Message Tag Example

To send a message, $x$, with message tag 5 from a source process, 1, to a destination process, 2, and assign to $y$:

```c
send(&x, 2, 5);
recv(&y, 1, 5);
```

**Movement of data**

Waits for a message from process 1 with a tag of 5
“Group” message passing routines

Apart from point-to-point message passing routines, have routines that send message(s) to a group of processes or receive message(s) from a group of processes - higher efficiency than separate point-to-point routines although not absolutely necessary.
Broadcast

Sending same message to all processes concerned with problem.

*Multicast* - sending same message to defined group of processes.

**Action**

**Code**

**MPI form**
Scatter

Sending each element of an array in root process to a separate process. Contents of \( i \)th location of array sent to \( i \)th process.

```
scatter();
```

```
data
```

```
buf
```

```
Process 0
```

```
scatter();
```

```
data
```

```
Process 1
```

```
scatter();
```

```
data
```

```
Process n – 1
```

```
scatter();
```
Gather

Having one process collect individual values from set of processes.

MPI form
Reduce

Gather operation combined with specified arithmetic/logical operation.

Example

Values could be gathered and then added together by root:

```
Action

Code

data
buf
reduce();
reduce();
reduce();

MPI form
```
PVM (Parallel Virtual Machine)

Perhaps first widely adopted attempt at using a workstation cluster as a multicomputer platform, developed by Oak Ridge National Laboratories. Available at no charge.

Programmer decomposes problem into separate programs (usually a master program and a group of identical slave programs).

Each program compiled to execute on specific types of computers.

Set of computers used on a problem first must be defined prior to executing the programs (in a hostfile).
Message routing between computers done by PVM daemon processes installed by PVM on computers that form the *virtual machine*.

MPI implementation we use is similar.
PVM Message-Passing Routines

All PVM send routines are nonblocking (or asynchronous in PVM terminology)

PVM receive routines can be either blocking (synchronous) or nonblocking.

Both message tag and source wild cards available.
Basic PVM Message-Passing Routines
**pvm_psend() and pvm_precv() system calls.**

Can be used if data being sent is a list of items of the same data type.
Full list of parameters for

\texttt{pvm\_psend()} \textbf{and} \texttt{pvm\_precv()}

\texttt{pvm\_psend(int dest\_tid, int msgtag, char *buf, int len, int datatype)}

\texttt{pvm\_precv(int source\_tid, int msgtag, char *buf, int len, int datatype)}
Sending Data Composed of Various Types

Data packed into send buffer prior to sending data.

Receiving process must unpack its receive buffer according to format in which it was packed.

Specific packing/unpacking routines for each datatype.
Sending Data Composed of Various Types

Example

```c
pvm_initsend();

pvm_pkint ( ... &x ...);
pvm_pkstr ( ... &s ...);
pvm_pkfloat ( ... &y ...);
pvm_send(process_2 ...);
```

```c
pvm_recv(process_1 ...);
```

Sending Data Composed of Various Types

Example

```c
pvm_upkint ( ... &x ...);
pvm_upkstr ( ... &s ...);
pvm_upkfloat ( ... &y ... );
```
Broadcast, Multicast, Scatter, Gather, and Reduce

pvm_bcast()
pvm_scatter()
pvm_gather()
pvm_reduce()

operate with defined group of processes.

Process joins named group by calling pvm_joingroup().

Multicast operation, pvm_mcast(), is not a group operation.
Sample PVM program.

Master

```c
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
#define SLAVE "spsum"
#define PROC 10
#define NELEM 1000
main() {
    int mytid,tids[PROC];
    int n = NELEM, nproc = PROC;
    int no, i, who, msgtype;
    int data[NELEM],result[PROC],tot=0;
    char fn[255];
    FILE *fp;
    mytid=pvm_mytid(); /*Enroll in PVM */

    /* Start Slave Tasks */
    no=
pvm_spawn(SLAVE,(char**)0,0,"",nproc,tids);
    if (no < nproc) {
        printf("Trouble spawning slaves \n");
        for (i=0; i<no; i++) pvm_kill(tids[i]);
        pvm_exit(); exit(1);
    }

    /* Open Input File and Initialize Data */
    strcpy(fn,getenv("HOME");
    strcat(fn, "/pvm3/src/rand_data.txt");
    if ((fp = fopen(fn,"r") == NULL) {
        printf("Can’t open input file %s\n",fn);
        exit(1);
    }
    for(i=0;i<n;i++) fscanf(fp,"%d",&data[i]);
    printf("%d from %d\n",result[who],who);
}
```

Slave

```c
#include <stdio.h>
#define PROC 10
#define NELEM 1000
main() {
    int mytid;
    int tids[PROC];
    int n, me, i, msgtype;
    int x, nproc, master;
    int data[NELEM], sum;
    ```
/* Open Input File and Initialize Data */
strcpy(fn, getenv("HOME"));
strcat(fn, "/pvm3/src/rand_data.txt");
if ((fp = fopen(fn,"r")) == NULL) {
    printf("Can’t open input file %s\n", fn);
    exit(1);
}
for (i=0; i<n; i++) fscanf(fp,"%d", &data[i]);

/* Broadcast data To slaves*/
pvm_initsend(PvmDataDefault);
msgtype = 0;
pvm_pkint(&nproc, 1, 1);
pvm_pkint(tids, nproc, 1);
pvm_pkint(&n, 1, 1);
pvm_pkint(data, n, 1);
pvm_mcast(tids, nproc, msgtag);

/* Get results from Slaves*/
msgtype = 5;
for (i=0; i<nproc; i++){
    pvm_recv(-1, msgtype);
    pvm_upkint(&who, 1, 1);
    pvm_upkint(&result[who], 1, 1);
    printf("%d from %d\n", result[who], who);
}

/* Compute global sum */
for (i=0; i<nproc; i++) tot += result[i];
printf ("The total is %d.\n\n", tot);
pvm_exit();/* Program finished. Exit PVM */
return(0);

mytid = pvm_mytid();
/* Receive data from master */
msgtype = 0;
pvm_recv(-1, msgtype);
pvm_upkint(&nproc, 1, 1);
pvm_upkint(tids, nproc, 1);
pvm_upkint(&n, 1, 1);
pvm_upkint(data, n, 1);

/* Determine my tid */
for (i=0; i<nproc; i++)
    if (mytid==tids[i])
        {me = i;break;}

/* Add my portion Of data */
x = n/nproc;
low = me * x;
high = low + x;
for (i = low; i < high; i++)
    sum += data[i];

/* Send result to master */
pvm_initsend(PvmDataDefault);
pvm_pkint(&me, 1, 1);
pvm_pkint(&sum, 1, 1);
msgtype = 5;
master = pvm_parent();
pvm_send(master, msgtype);

/* Exit PVM */
pvm_exit();
return(0);
MPI (Message Passing Interface)

Standard developed by group of academics and industrial partners to foster more widespread use and portability.

Defines routines, not implementation.

Several free implementations exist.
MPI

Process Creation and Execution

Purposely not defined and will depend upon the implementation.

Only static process creation is supported in MPI version 1. All processes must be defined prior to execution and started together.

*Originally SPMD model of computation.*

MPMD also possible with static creation - each program to be started together specified.
Communicators

Defines *scope* of a communication operation.

Processes have ranks associated with communicator.

Initially, all processes enrolled in a “universe” called *MPI_COMM_WORLD*, and each process is given a unique rank, a number from 0 to $n - 1$, where there are $n$ processes.

Other communicators can be established for groups of processes.
Using the SPMD Computational Model

main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */
    if (myrank == 0)
        master();
    else
        slave();
    
    MPI_Finalize();
}

where master() and slave() are procedures to be executed by master process and slave process, respectively.
“Unsafe” Message Passing

MPI specifically addresses unsafe message passing.
Unsafe message passing with libraries

(a) Intended behavior

(b) Possible behavior
MPI Solution

“Communicators”

A communication domain that defines a set of processes that are allowed to communicate between themselves.

The communication domain of the library can be separated from that of a user program.

Used in all point-to-point and collective MPI message-passing communications.
Default Communicator

`MPI_COMM_WORLD`, exists as the first communicator for all the processes existing in the application.

A set of MPI routines exists for forming communicators.

Processes have a “rank” in a communicator.
Point-to-Point Communication

PVM style packing and unpacking data is generally avoided by the use of an MPI datatype being defined.
Blocking Routines

Return when they are locally complete - when location used to hold message can be used again or altered without affecting message being sent.

A blocking send will send the message and return. This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message.
Parameters of the blocking send

```c
MPI_Send(buf, count, datatype, dest, tag, comm)
```

- Address of send buffer
- Datatype of each item
- Message tag
- Number of items to send
- Rank of destination process
- Communicator
Parameters of the blocking receive

MPI_Recv(buf, count, datatype, src, tag, comm, status)

Address of receive buffer  Datatype of each item  Message tag  Status after operation
Maximum number of items to receive  Rank of source process  Communicator
Example

To send an integer $x$ from process 0 to process 1,

```c
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);  /* find rank */
if (myrank == 0) {
   int x;
   MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
   int x;
   MPI_Recv(&x, 1, MPI_INT, 0,msgtag,MPI_COMM_WORLD,status);
}
```
Nonblocking Routines

**Nonblocking send** - MPI_Isend(), will return “immediately” even before source location is safe to be altered.

**Nonblocking receive** - MPI_Irecv(), will return even if there is no message to accept.
Nonblocking Routine Formats

\textbf{MPI\_Isend(buf, count, datatype, dest, tag, comm, request)}

\textbf{MPI\_Irecv(buf, count, datatype, source, tag, comm, request)}

Completion detected by \textbf{MPI\_Wait()} and \textbf{MPI\_Test(}).

\textbf{MPI\_Wait()} waits until operation completed and returns then.
\textbf{MPI\_Test()} returns with flag set indicating whether operation completed at that time.

Need to know whether particular operation completed.
Determined by accessing the \textit{request} parameter.
Example

To send an integer $x$ from process 0 to process 1 and allow process 0 to continue,

\begin{verbatim}
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD,
               req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD,
              status);
}
\end{verbatim}
Four Send Communication Modes

**Standard Mode Send**
Not assumed that corresponding receive routine has started. Amount of buffering not defined by MPI. If buffering provided, send could complete before receive reached.

**Buffered Mode**
Send may start and return before a matching receive. Necessary to specify buffer space via routine `MPI_Buffer_attach()`.

**Synchronous Mode**
Send and receive can start before each other but can only complete together.

**Ready Mode**
Send can only start if matching receive already reached, otherwise error. *Use with care.*
Each of the four modes can be applied to both blocking and nonblocking send routines.

Only the standard mode is available for the blocking and nonblocking receive routines.

Any type of send routine can be used with any type of receive routine.
Collective Communication

Involves set of processes, defined by an intra-communicator. Message tags not present.

Broadcast and Scatter Routines

The principal collective operations operating upon data are

- **MPI_Bcast()** - Broadcast from root to all other processes
- **MPI_Gather()** - Gather values for group of processes
- **MPI_Scatter()** - Scatters buffer in parts to group of processes
- **MPI_Alltoall()** - Sends data from all processes to all processes
- **MPI_Reduce()** - Combine values on all processes to single value
- **MPI_Reduce_scatter()** - Combine values and scatter results
- **MPI_Scan()** - Compute prefix reductions of data on processes
Example

To gather items from the group of processes into process 0, using dynamically allocated memory in the root process, we might use

```c
int data[10]; /*data to be gathered from processes*/

MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    MPI_Comm_size(MPI_COMM_WORLD, &grp_size); /*find group size*/
    buf = (int *)malloc(grp_size*10*sizeof(int)); /*allocate memory*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_COMM_WORLD);
```

Note that `MPI_Gather()` gathers from all processes, including root.
Barrier

As in all message-passing systems, MPI provides a means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.
Sample MPI program.

```c
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000

void main(int argc, char *argv)
{
    int myid, numprocs;
    int data[MAXSIZE], i, x, low, high, myresult, result;
    char fn[255];
    char *fp;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    if (myid == 0) { /* Open input file and initialize data */
        strcpy(fn,getenv("HOME"));
        strcat(fn,"/MPI/rand_data.txt");
        if ((fp = fopen(fn,"r")) == NULL) {
            printf("Can\’t open the input file: %s
", fn);
            exit(1);
        }
        for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
    }
    /* broadcast data */
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);
    /* Add my portion Of data */
    x = n/nproc;
    low = myid * x;
    high = low + x;
    for(i = low; i < high; i++)
        myresult += data[i];
    printf("I got %d from %d
", myresult, myid);
    /* Compute global sum */
    MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) printf("The sum is %d.
", result);
    MPI_Finalize();
}
```
Debugging and Evaluating Parallel Programs
Visualization Tools

Programs can be watched as they are executed in a space-time diagram (or process-time diagram):

- Process 1
- Process 2
- Process 3

Legend:
- Computing
- Waiting
- Message-passing system routine
- Message
PVM has a visualization tool called XPVM.

Implementations of visualization tools are available for MPI. An example is the Upshot program visualization system.
Evaluating Programs Empirically

Measuring Execution Time

To measure the execution time between point L1 and point L2 in the code, we might have a construction such as

```
L1:  time(&t1); /* start timer */
.
L2:  time(&t2); /* stop timer */

elapsed_time = difftime(t2, t1); /* elapsed_time = t2 - t1 */
printf(“Elapsed time = %5.2f seconds”, elapsed_time);
```

MPI provides the routine `MPI_Wtime()` for returning time (in seconds).
Home page - Original

http://www.cs.uncc.edu/par_prog
Parallel Processing Online Help With PVM
Parallel Virtual Machine

Table Of Contents

- Introduction to Parallel Programming
  - Introduction
  - Getting Started
  - Setting Up The Directories
  - Compiling PVM Programs
  - Running PVM Programs

- Using XPVM
  - Introduction
  - Adding Hosts
  - Spawning Tasks
  - Viewing Output
  - Exiting XPVM: Quit vs. Hal

- Using PVM
  - Introduction
  - Starting PVM
  - Adding Hosts

- Understanding Makefiles
  - The C Compiler
  - The Basics
  - Advanced Makefile
  - A PVM Makefile
  - More Information

- PVM Information
  - PVM Online Manual
  - PVM Man Pages

- PVM Sample Programs
- UNIX Information
- Parallel Processing Glossary

The picture above is a snapshot of a virtual parallel machine that has been set up using the PVM software package. To use this package for your own purposes, read the pages that are listed in the table of contents on the left.
New UNCC Home page

Better step-by-step instructions, being developed. Currently link from:

http://www.cs.unc.edu/~abw/
This website is provided as supporting material for the textbook

Parallel Programming:
Techniques and Applications Using Networked Workstations and Parallel Computers

Barry Wilkinson and Michael Allen

© Barry Wilkinson 2001. All rights reserved. This material is the property of Professor Barry Wilkinson and for sole and exclusive use of the participants of the workshop on teaching cluster computing (WTCC) held at Gujarat University, Ahmedabad, India, Tuesday, December 18th, 2001 - Thursday, December 20th, 2001. It is not to be sold, reproduced, or generally distributed.
Step 1 of 7: Definition

PVM (Parallel Virtual Machine) is a software system that allows you to combine a number of computers which are connected over a network into a parallel virtual machine. This machine can consist of computers with different architectures, running different flavors of the Unix operating systems and can still be treated as if it were a single parallel machine. As the software is public domain this means that many organizations which already have a cluster of workstations can get a parallel machine for free and solve larger problems using existing hardware resources.
Step 2 of 7: Getting Started

Insert the lines:

```bash
if (-r /afs/unc/usr6/jrobins/public/pvm.cshrc) then
    source /afs/unc/usr6/jrobins/public/pvm.cshrc
endif
```

into your `~/.login.s0126` file so that your `~/.login.s0126` file will contain:

```bash
... # Limit core dump size to 1megabyte
    limit coredumpsize 1
    if (-r /afs/unc/usr6/jrobins/public/pvm.cshrc) then
        source /afs/unc/usr6/jrobins/public/pvm.cshrc
    endif
    if ( !$[?DT] ) then
        ...
```

**Note:** The lines you need to insert are specific to the UNC Charlotte's Unix network. Please contact your network administrator for any other systems.
Step 3 of 7: Setting Up the Directories

The next step is to establish the appropriate directory tree inside your root directory. Your directories should look exactly like the diagram below. If they do not, the Makefile will not be able to store your executable files in the proper sub-directory.

```
% cd
% mkdir pvm3
% cd pvm3
% mkdir src
% mkdir bin
% cd src
% cd SUN4SOL2
% cd FREEBSD
% cd ...
```

The following steps are the commands you need to execute at the prompt line in order to obtain such a structure:

The `src` directory is where you will store the PVM source `.c` files, and the `Makefile`.aimk` file. Be sure that you are in the `src` directory each time you try to compile your code. When you follow this structure, `makefile.aimk` will compile your programs, and store the executable files in the proper directory under your `bin` sub-directory.
Step 1 of 5: Definition

MPI stands for Message Passing Interface. The goal of MPI, simply stated, is to develop a widely used standard for writing message-passing programs. As such the interface attempts to establish a practical, portable, efficient, and flexible standard for message passing. In designing MPI, the MPI Forum sought to make use of the most attractive features of a number of existing message-passing systems, rather than selecting one of them and adopting it as the standard. The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.
Step 2 of 5: Getting Started

The first step in using MPI is to create a MPI directory under your root directory. To do this, execute the 'mkdir' command in a command tool:

```
$ mkdir MPI
```

Second, you must have a file called .rhosts. This file will allow connections from the specified machines by the specified user names. This will allow MPI to start a process on the specified machine. An example of the .rhosts file is given below. This file should be stored in your home directory.

```
# .rhosts -- File allowing remote connections
# from specified machine and user
#
# The second argument must be your user name.
#
+jaalley
```

Once you have the .rhosts file, you need to execute the following commands at a command tool prompt from inside your home directory:

```
% chmod 600 .rhosts
% fa setacl . system:anynone r
% fa setacl -/MPI system:anynone r
```
Step 3 of 5: Understanding Makefiles

To compile MPI source code, we must make use of the Unix 'make' utility. Remember that you can only get the executable code for the type of architecture you compile your code on. Therefore, if you compile on a Sun workstation you will only get executable code for the Sun4SOL2 operating system.

Click [here](#) for a sample makefile that can be used in a MPI system.

Once you create your makefile, you must edit it so that it can compile the files you want to be compiled. The best way to do this is to open Makefile in a text editor, and copy the target and action lines for "example". Then paste these lines under the action lines for the "example", giving you duplicate target and action lines for "example". You must then change all the file names in one set of the rules to match the names of your program.
Compiling/executing PVM programs

Convenient to have two command line windows.

To start PVM:
At one command line:
   `pvm`
returning a pvm prompt (>)

To compile PVM programs
At another command line in pvm3/src/:
   `aimk file`

To execute PVM program
At same command line in pvm3/bin/?/ (where ? is name of OS)
   `file`

To terminate pvm
At 1st command line (>):
   `quit`
Basic instructions for compiling/executing MPI programs

Preliminaries

- Set up paths
- Create required directory structure
- Modify makefile to match your source file
- Create a file (hostfile) listing machines to be used (required)

Details described on home page.
Hostfile

Before starting MPI for the first time, need to create a hostfile

Sample hostfile

ws404
#is-sm1 //Currently not executing, commented
pvm1 //Active processors, UNCC sun cluster called pvm1 - pvm8
pvm2
pvm3
pvm4
pvm5
pvm6
pvm7
pvm8
Compiling/executing (SPMD) MPI program

For LAM MPI version 6.5.2. At a command line:

To start MPI:
First time: lamboot -v hostfile
Subsequently: lamboot

To compile MPI programs:
mpicc -o file file.c
or
mpiCC -o file file.cpp

To execute MPI program:
mpirun -v -np no_processors file

To remove processes for reboot
lamclean -v

Terminate LAM
lamhalt

If fails
wipe -v lamhost
Compiling/executing Multiple MPI programs

Create a file specifying programs:

Example

1 master and 2 slaves, “appfile” contains

n0 master
n0-1 slave

To execute:

mpirun -v appfile

Sample output

3292 master running on n0 (o)
3296 slave running on n0 (o)
412 slave running on n1
Usual First Student Assignment at UNCC

The objective of this assignment is to become familiar with the PVM or MPI environment. From the UNCC Computer Science Parallel Programming home page (http://www.cs.uncc.edu/par_prog), download into your home directory the sample program for adding numbers found under: Sample Programs in the Table of Contents.

1. Follow the procedures to compile and execute the program. Use a single computer.

2. Modify the program to find the maximum number of the set of numbers being added.

3. Submit a written report (hardcopy, NOT email). Include the modified code, the results of the program, and a short description of the work.
Alternative First Assignments

1. Write an “hello world” MPI or PVM program in which 4 processes each print out “hello world”.

2. Write a program to send a message around a ring of processes, printing it out at each stage.

(Note standard output may only be possible on system you are logion into, not the computers in the cluster.)
This is HelloWorld.c created on June 29, 2001 by K. Spry for CSCI3145-020. This program is a simple function used to familiarize myself with MPI calls. The program displays the rank of each processor executing the HelloWorld code.

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

int main (int argc, char *argv[]) {
    int myrank, num_procs, err;

    err = MPI_Init (&argc, &argv); //initialize MPI

    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); //get processor rank
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs); //get total number of processors

    printf("Processor %d of %d: Hello World! \n", myrank, num_procs);
}
```

This is HelloW_Again.c created on June 29, 2001 by K. Spry for CSCI3145-020. This program is a simple function used to familiarize myself with MPI calls. The program displays the rank of each processor executing the HelloWorld code. Uses master and slave.

```c
#include <stdio.h> //include files
#include <mpi.h>
void master(int, int); //function prototypes
void slave(int, int);

int main (int argc, char *argv[]) {
    int myrank, num_procs, err; //variables
    err = MPI_Init (&argc, &argv); //initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); //get processor rank
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs); //get total number of processors
    if(myrank == 0){
        master(myrank, num_procs);
    } else {
        slave(myrank, num_procs);
    }
    err = MPI_Finalize(); //terminate MPI
}

void master(int myrank, int num_procs) {
    printf("Processor %d of %d: Hello World from Master! \n", myrank,
```
/*****************************************************************************/
**
This is HelloW_Again.cpp created on June 29, 2001 by K. Spry for CSCI3145-020. This program is a simple function used to familiarize myself with MPI calls. The program displays the rank of each processor executing the HelloWorld code. Uses master and slave.
*****************************************************************************/

#include <stdio.h> //include files
#include <mpi.h>

void master(int, int); //function prototypes
void slave(int, int);

int main (int argc, char *argv[]) {
    int myrank, num_procs, err; //variables
    err = MPI_Init (&argc, &argv); //initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); //get processor rank
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs); //get total number of processors
    if(myrank == 0){
        master(myrank, num_procs);
    } else {
        slave(myrank, num_procs);
    }
    err = MPI_Finalize(); //terminate MPI
}

void master(int myrank, int num_procs) {
    cout << "Processor " << myrank << " of " << num_procs << " Hello M" <<endl;
}
Other First Assignments

**Graduate:**

1. Write a parallel program that measures the time to send a message. Repeat with the ping-pong method described in Section 2.4.4 of textbook. Experiment with sending groups of multiple messages and messages of different sizes to obtain a good estimate for the time of message transfers. Plot the time of sending a message against the size of the message and fit a line to the results. Estimate the startup time, latency, and the time to send one data item.

2. Make a comparative study of the communication times in both PVM and MPI by passing messages between processes that have been instrumented to measure the communication times.