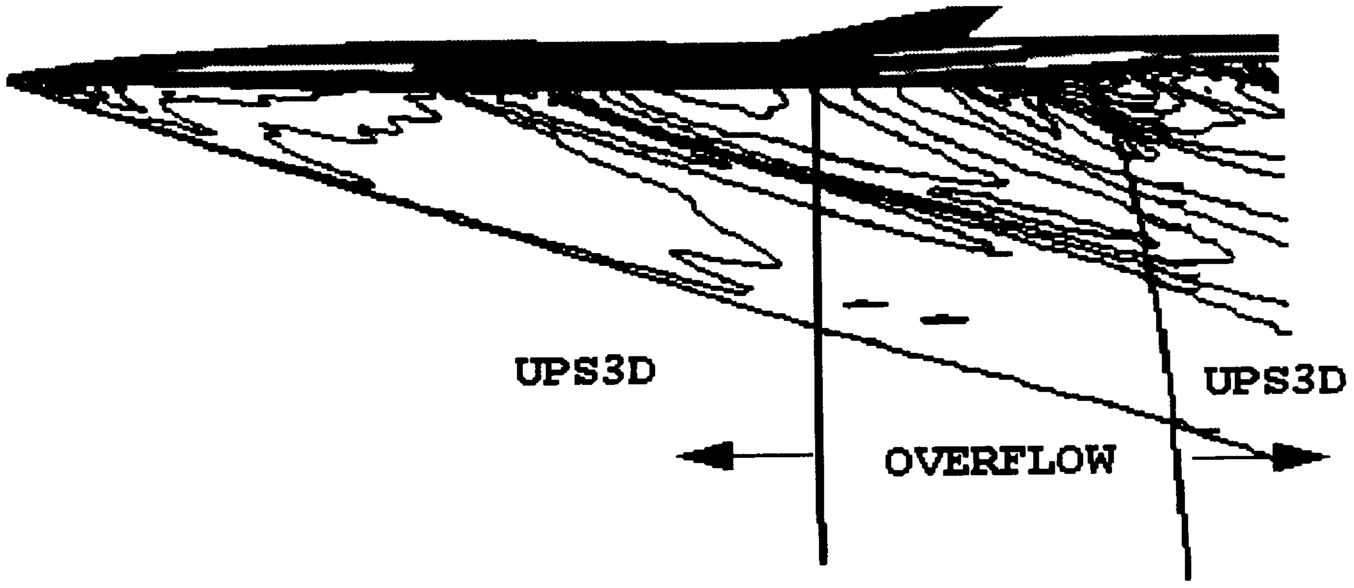
MCAT Institute
Progress Report
94-10

NASA-CR-196397

[illegible]
[illegible]
P. 40

High Speed Civil Transport: Sonic Boom Softening and Aerodynamic Optimization

Samson Cheung



July 1994

NCC2-617

MCAT Institute
3933 Blue Gum Drive
San Jose, CA 95127

Sonic Boom Softening and Aerodynamic Optimization

Samson Cheung

INTRODUCTION

Currently, this research effort is concentrated on a single theme of sharpening the tools for High Speed Commercial Transport (HSCT) design. Three research topics are focused: near-field Computational Fluid Dynamic (CFD) calculation and sonic boom softening of the Boeing Reference-H design, improvement of sonic boom extrapolation, and aerodynamic design on parallel computers.

In order to design and study a complex aircraft, a relatively fast CFD technique has to be developed for optimization environment. Coupling a fast space-marching code with a time-iterative code with overset grids can take the advantage of the speed of the marching code on the fuselage/wing and handle the complex grid near the wing/nacelle region at the same time.

A very efficient wave propagation code for mid-field sonic boom prediction has been developed based on the method of characteristics. This code solves the Euler equations in 1.2 minutes on Cray-YMP; whereas, the axisymmetric CFD method used in 1990 takes 40 minutes on the same computer.

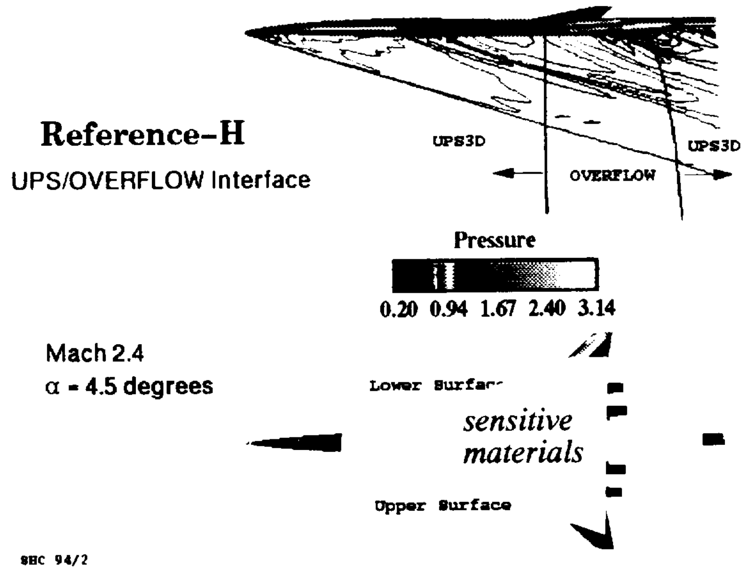
In today's computing environment, large computation intensive problems, like CFD calculations, can now be done efficiently on parallel machines. Such computations may become standard for aerodynamic research and design in the future. In the present research, a nonlinear optimization routine for HSCT design has been developed for a network based parallel computer system in which a cluster of engineering workstations serves as a virtual parallel machine.

Sonic Boom and Performance Study of Reference-H

This research effort has concentrated on low-boom HSCT concepts for the past four years. Even though a new proposed route structure, incorporating supersonic corridors over land and water, has relaxed sonic boom constraints somewhat, they are still an issue. The objective of this study is twofold. First is to exercise the methodology of combining two different CFD codes to solve the near-field solution of a realistic HSCT configuration in an efficient and accurate manner. Second is to reduce the sonic boom of a the Reference-H performance configuration without impacting aerodynamic performance. The basic components of Reference-H are a fuselage, a pair of swept wings, and four nacelles.

Reference-H Near-Field Study

The CFD codes used in this study are the UPS3D code and the OVERFLOW code. The former is an efficient space-marching code. However, it fails in the regions of subsonic flow, especially in the region of the wing/nacelle integration. The latter is a time-iterative code with the Chimera overset grid concept, which makes the code more suitable for solving the region of wing/nacelle integration. In this study, only inviscid flow is considered. The figure below summarizes the results of a near-field calculation.



The flow conditions for the near-field solution above are a Mach number 2.4 and angle of attack 4.5 degrees. Wind-tunnel data of the Reference-H compares well the results of the CFD analysis. These results show that the flow turns significantly over the outer nacelle compared with the flow over the inner nacelle which indicates that nacelle orientation can impact the aerodynamic performance of the configuration.

Sonic Boom Softening

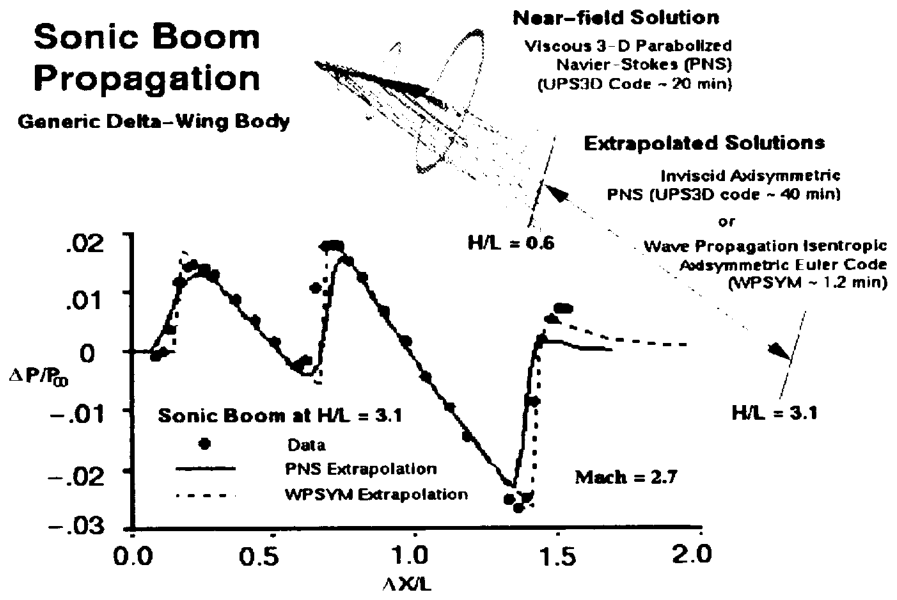
The sonic boom signature of the Reference-H configuration is also obtained. This calculation shows that the sonic boom is an N-wave of 104PLdB with a 2.5psf bow shock on the ground. Details of the sonic boom prediction technique can be found in Ref. 1. Sonic boom softening of performance aircraft is very different from that of low-boom aircraft since cruise Mach number and lift are higher. Therefore, the technique developed previously can not be directly applied to the Reference-H configuration. However, changing the second derivative of the equivalent area distribution can reduce the PLdB to 102. This change in the area is so small that the aerodynamic performance remains basically unaltered according to the CFD calculations. The results of this study were presented in the 4th Sonic Boom Workshop.[2]

An on-going communication between sonic boom analysis personnel and aircraft design personnel is essential to effectively approach the task of sonic boom softening on performance aircraft. A team consisting the principal investigator of this effort, design personnel from Boeing and NASA Langley has been formed to achieve this goal.

Sonic Boom Mid-Field Extrapolation (WPSYM)

In the beginning of 90's, sonic boom extrapolation techniques still relied on the linear theory developed in the 60's since the nonlinear techniques were computationally expensive. A better sonic boom extrapolation technique was needed to accurately and efficiently model sonic boom extrapolation for HSCT design. Therefore, the objective of this study is the development of an efficient and accurate higher-order computational method, solving the Euler equations, for supersonic aero-acoustic wave propagation.

An axisymmetric wave propagation code (WPSYM) has been developed for mid-field sonic boom extrapolation. This propagation code has been demonstrated as an efficient and accurate tool over previous CFD methods[1] on a generic wing-body configuration. The figure below shows the 3-D near-field solution obtained with the UPS3D code; the result is



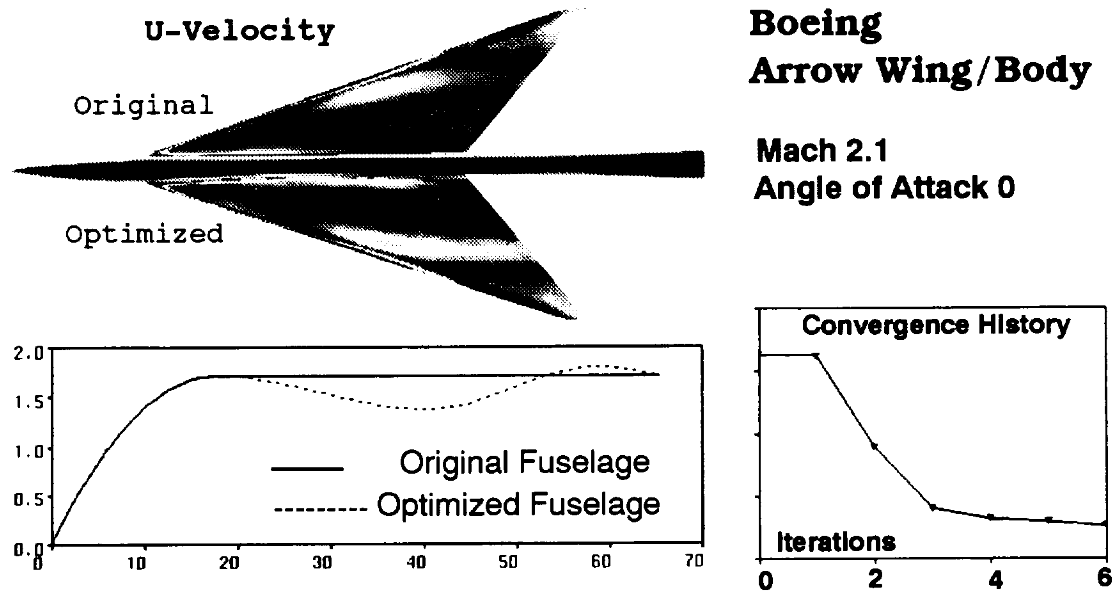
then interfaced to two axisymmetric sonic boom extrapolation codes, namely, the axisymmetric version of UPS3D and the recent wave propagation code (WPSYM). The former takes 40 minutes on Cray-YMP, and the latter takes 1.2 minutes on the same machine. The x-y plot in the figure compares the two numerical extrapolation results to wind-tunnel data. The result has been shown in NASA Technical Highlight and the methodology has been presented in the 4th Annual Sonic Boom Workshop at NASA Langley in June 1994.[2]

Optimizer on PVM (*II*OWA)

Moving to the world of parallel computing, the aerospace industry needs a numeric optimization tool in the parallel environment. One of the promising parallel computing concepts is network-based distributed computing. The Parallel Virtual Machine (PVM) is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. PVM allows users to link up engineering workstations to work as a single distributed-memory (parallel) machine. Merritt

Smith of NASA Ames Research Center and the Principle Investigator for this research compiled a PVM manual for beginning users. A copy of the manual is attached as Appendix A to this report.

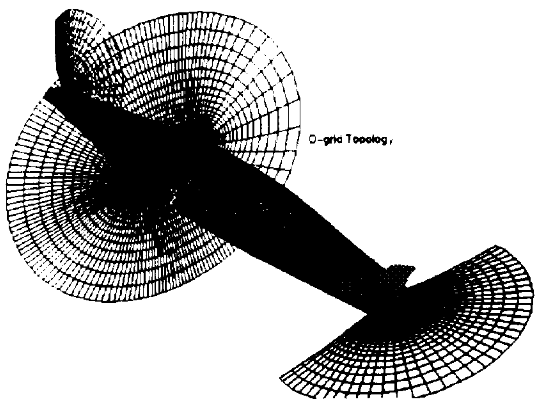
A parallel optimizer based on nonlinear Quasi-Newton method has been developed and coupled with an efficient CFD code for basic aerodynamic design and study. This optimizer is called *//*OWA (parallel Optimizer With Aerodynamics). The figure below is a demonstration of *//*OWA. A Boeing arrow wing/body configuration is chosen in this

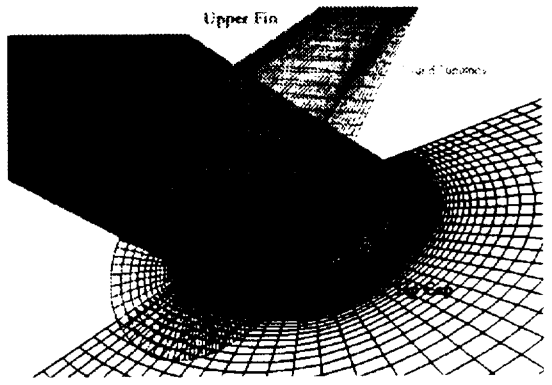


study. The fuselage radius is changed so that the wave drag is minimized. The parallel CFD optimization process takes 24 wall-clock hours on 4 SGI workstations to reduce the wave drag by 6.5%. The optimized result is a "coke bottle" shaped fuselage, as expected by supersonic area rule. The convergence history of the optimization process is also shown in the figure.

Oblique All-Wing (OAW): CFD Support

The OAW design team has asked for CFD support on the latest configuration OAW-3 from which a wind-tunnel model has been built and tested at Ames in June 1994. The figure below shows the chimera grid topology on the OAW-3 with an upper fin. The design team





plans to compare the current CFD results with the results from pressure sensitive paint (PSP). CFD calculations have to be done prior to the wind-tunnel test because the color map from CFD result is need for PSP calibration.

SUMMARY

An improvement in sonic boom extrapolation techniques has been the desire of aerospace designers for years. This is because the linear acoustic theory developed in the 60's is in capable of predicting the nonlinear phenomenon of shock wave propagation. On the other hand, CFD techniques are too computationally expensive to employ on sonic boom problems. Therefore, this research focused on the development of a fast and accurate sonic boom extrapolation method that solves the Euler equations for axisymmetric flow. This new technique has brought the sonic boom extrapolation techniques up to the standards of the 90's.

Parallel computing is a fast growing subject in the field of computer science because of their promising speed. A new optimizer (*II*OWA) for the parallel computing environment has been developed and tested for aerodynamic drag minimization. This is a promising method for CFD optimization making use of the computational resources of workstations, which unlike supercomputers can spend most of their time idle.

Finally, the OAW concept is attractive because of its overall theoretical performance. In order to fully understand the concept, a wind-tunnel model was built and is currently being tested at NASA Ames Research Center. The CFD calculations performed under this cooperative agreement helped to identify the problem of the flow separation (1992 Annual Report), and also aided the design by optimizing the wing deflection for roll trim.

References

1. Cheung, S., Edwards, T., and Lawrence, S., "Application of CFD to Sonic Boom Near and Mid Flow-Field Prediction," J. of Aircraft, Vol. 29, No. 5, 1992.
2. Cheung, S., "Sonic Boom Softening of Reference-H," High Speed Research: Sonic Boom, NASA Langley, June 1994. (NASA CP will be published for limited distribution).

Appendix A

PVM Manual

Manual of PVM

Samson Cheung

Merritt Smith

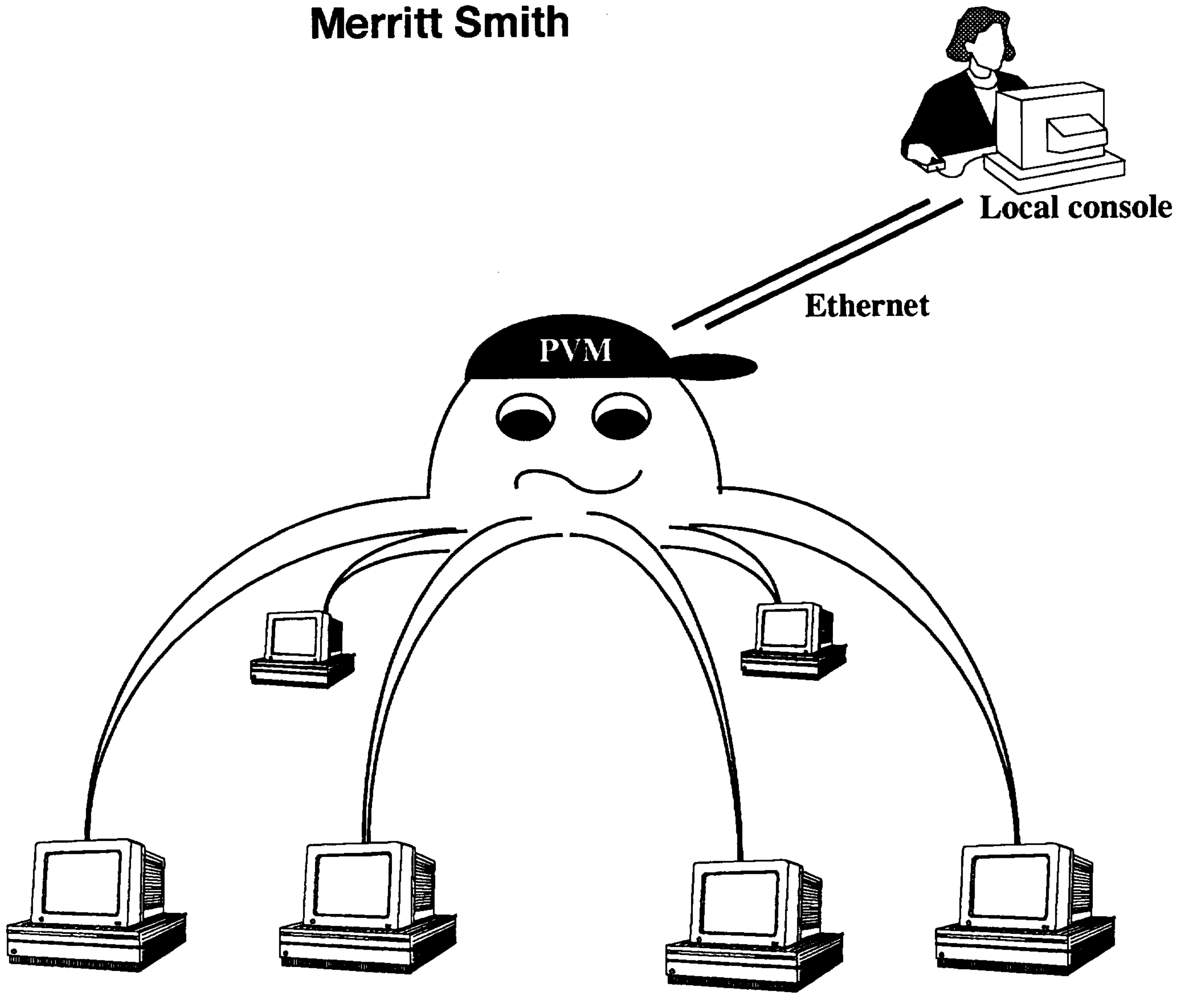


Table of Contents

Preface

This manual serves as a supplementary document for the official reference manual of a relatively new research software, PVM, which has been developed at Oak Ridge National Laboratory. A beginner, who has no previous experience with PVM, would find this manual useful.

We would like to thank you in advance that if you find any problems in PVM or this manual, please contact one of us.

Mr. Merritt Smith
NASA Ames Research Center, MS 258-1
Moffett Field, CA 94035
e-mail: mhsmith@nas.nasa.gov
phone: (415) 604-4462

Dr. Samson Cheung
MCAT Institute
NASA Ames Research Center, MS 258-1
Moffett Field, CA 94035
e-mail: cheung@nas.nasa.gov
phone: (415) 604-4462

1 INTRODUCTION

This manual provides you with an introduction to PVM and provides the fundamentals necessary to write FORTRAN programs in the PVM environment through a tutorial sample. This manual is designed for those who have no previous experience with PVM. However, you should know basic FORTRAN programming and UNIX. If you are ready for an advanced PVM application, please consult the official PVM Reference Manual.

Software Package

PVM stands for Parallel Virtual Machine. It is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. PVM allows you to link up all or some of the computational systems on which you have accounts, to work as a single distributed-memory (parallel) machine. We call this a Virtual Machine.

PVM is useful for the following reasons. Unlike large mainframe computers or vector supercomputers, workstations spend most of the time idle. The idle time on a workstation represents a significant computational resource. PVM links these workstations up to become a powerful multi-processor computational machine. With PVM, the lack of supercomputer resources should not be an obstacle to number crunching computational programs. Furthermore, the annual maintenance costs of a vector supercomputer is often sufficient to purchase the equivalent computing resource in the form of workstation CPU's.

Definitions

Here are some terms we use throughout this document:

Virtual Machine	PVM links different user-defined computers together to perform as one large distributed-memory computer. We call this computer the Virtual Machine.
Host	Individual computer (member) in the virtual machine.
Process	Individual program operating on different computers or hosts.
Processor	The processing unit in computers. A virtual machine can be viewed as a multi-processor computer.

Task	The unit of computation handled by the virtual machine. You may want to think of one processor handling one task.
Tid	Task identification number which is a unique number used by the daemon and other tasks.
Console	A program from which you can directly interact with the virtual machine. (Add hosts, kill processes,...)

Structure of PVM

The PVM software is composed of two parts. The first part is a daemon. We call it *pvmd*. This is the control center of the virtual machine. It is responsible for starting processes, establishing links between processes, passing messages, and many other activities in PVM. Since the daemon runs in the background, you have to use PVM console to directly interact with the virtual machine.

The second part of the system is a library of PVM interface routines located in `libpvm3.a`. This library contains user callable routines for message passing, spawning processes, coordinating tasks, and modifying your machine. In writing your application, you will need to call the routines in this library.

Directory Setup

This setup is for NAS system. Before you use PVM, you need to set up the following directories on *all* the machines that you want PVM to link:

- Make a directory `$HOME/pvm3/bin/ARCH` in all the hosts of the virtual machine.

  Note `ARCH` is used throughout this manual to represent the architecture name that PVM uses for a given computer. The table in the Appendix lists all the ARCH names that PVM supports. For example, for Silicon Graphic IRIS workstations, you should make a directory `$HOME/pvm3/bin/SGI`.

- Make a directory `$HOME/pvm3/include`, and copy the file `fpvm3.h` from `/usr/nas/pkg/pvm3.2/include`. (If you are on different system from NAS, please consult your system consultant.)
- Make a directory `$HOME/pvm3/codes`, and write your application programs in this directory. You can actually put your programs anywhere you like as long as the correct "include" files are includes. The current setup is for clarity.

2 *Programming Concept*

Unlike graphical software or a word-processor, you cannot *see* PVM working by clicking your mouse buttons. In fact, a virtual machine is quite an abstract concept because you don't physically have a multi-processor machine! In this chapter, you will learn a simple concept, which will help you to visualize how PVM works.

Master and Slaves

A common way to work with PVM is a *Master/Slave* relationship. A Master process starts Slave routines and distributes work. However, a Master does not actively participate in the computation. A Master process most often resides on the originating host (user's computer), while the Slave programs are distributed to the hosts of the virtual machine.

You need to distribute executables of Slave programs to the directory `$HOME/pvm3/bin/ARCH` on every host. You can locate this Master program anywhere you like.

Since the Master program spawns Slave programs on each of the hosts to do jobs, it is important to understand the communication (message passing) amoung the hosts in PVM.

Typically, a Master and a Slave have the following logic:

Master

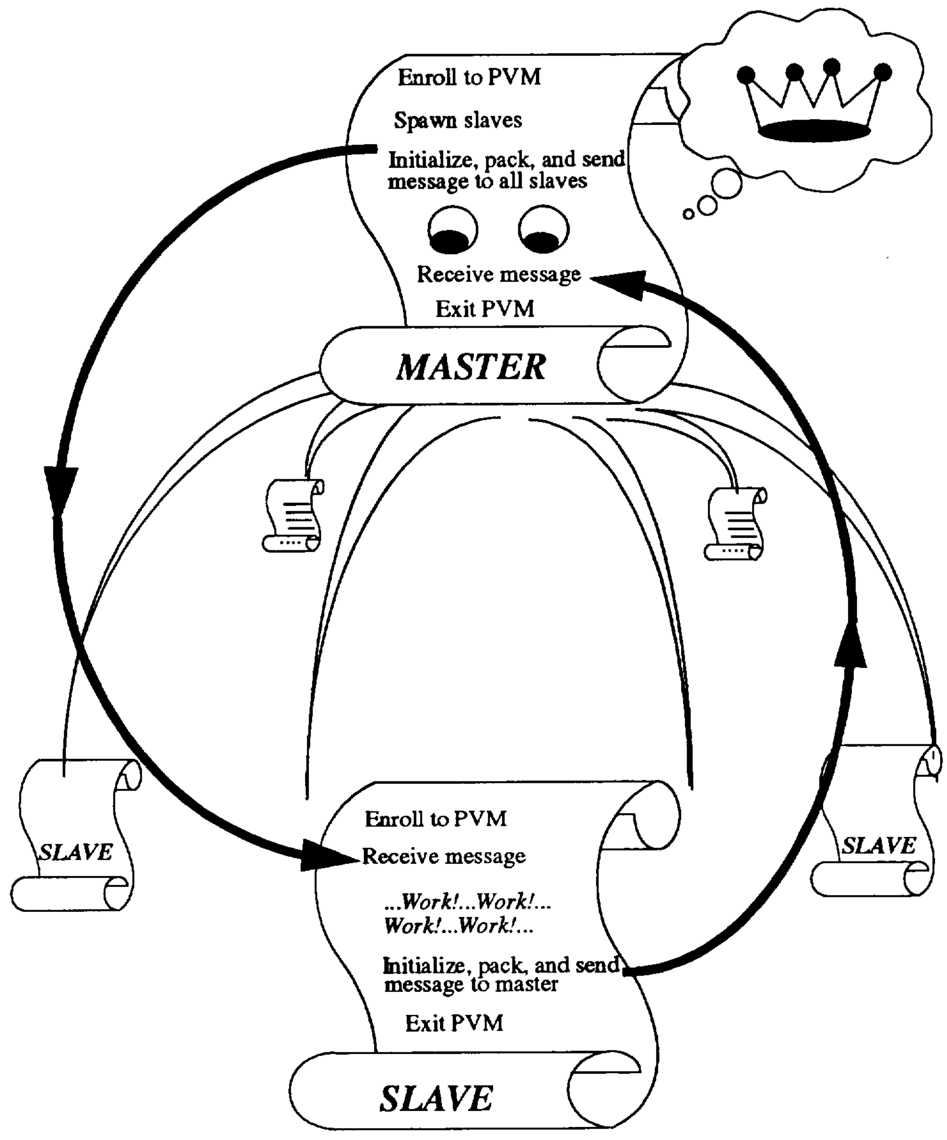
1. Enroll itself to PVM
2. Spawn slave processes
3. Initialize buffer, pack, and send message to all slaves.
4. *...wait for slaves to finish...*
5. Receive message from slave(s)
6. Exit PVM

Slave

1. Enroll itself to PVM
2. Receive message from master
3. *...do something useful...*
4. Initialize buffer, pack, and send message to master
5. Exit PVM

The figure on the opposite page graphically describes a Master/Slave relationship and shows the exchange of information.

FIGURE 1. Communication in *Master/Slave* programs.


Enroll to PVM
Spawn slaves
Initialize, pack, and send message to all slaves
Receive message
Exit PVM
MASTER
SLAVE
Enroll to PVM
Receive message
...Work!...Work!... Work!...Work!...
Initialize, pack, and send message to master
Exit PVM
SLAVE
SLAVE

SPMD

Another common way to work with PVM is the *SPMD*, Single Program Multiple Data model. There is only a single program, and there is no Master program directing the computation. The user starts the first copy of the program and using the routine `pvmfparent()`, this copy can determine that it was not spawned by PVM, and thus must be the first copy (parent). It then spawns multiple copies (children) of itself and passes them the array of *tids*. At this point each copy is equal and can work on its partition of the data in collaboration with the other processes.

Typically, a SPMD program has the following logic:

1. **Enroll in pvm**
2. **If I am the first copy (parent)**
   a) Spawn child processes
   b) Initialize buffer, pack, and send message out
3. **If I am a secondary copy (child)**
   Receive messages
4. **Work!...Work!...Work!**
5. **Exit PVM**

The program on the opposite page describes a SPMD logic and shows the exchange of information. Please spend some time to study the program.

In the next chapter we will introduce the PVM daemon and the fundamentals of message passing.

SPMD Program

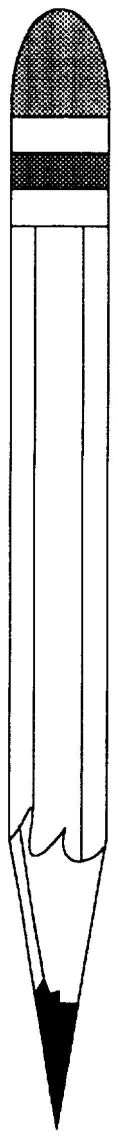
```
   c---------------------------------------
   c  SPMD Fortran example using PVM 3.0
   c---------------------------------------
         program spmd
         include '../include/fpvm3.h'
         PARAMETER( NPROC=4 )
         integer mytid, me, i
         integer tids(0:NPROC)

   c     Enroll in pvm
1        call pvmfmytid( mytid )

   c     --------------------------------------------------
   c     Find out if I am parent or child
   c     --------------------------------------------------
         call pvmfparent(tids(0))
         if( tids(0) .lt. 0 )  then
           tids(0) = mytid
           me = 0
   c       ------------------------
   c       start up copies of myself
   c       ------------------------
           call pvmfspawn('spmd',PVMDEFAULT,'*',
2       * NPROC-1,tids(1), info)
   c       --------------------------
   c       multicast tids array to children
   c       --------------------------
           call pvmfinitsend( PVMDEFAULT, info )
           call pvmfpack( INTEGER4, tids, NPROC, 1, info )
           call pvmfmcast( NPROC-1, tids(1), 0, info )
         else
   c       -------------------------------
   c       receive the tids array and set me
   c       -------------------------------
3          call pvmfrecv( tids(0), 0, info )
           call pvmfunpack( INTEGER4, tids, NPROC,1,info)
           do 30 i=1, NPROC-1
             if( mytid .eq. tids(i) ) me = i
     30    continue
         endif
   c-----------------------------------------------------
   c  all NPROC tasks are equal now
   c  and can address each other by tids(0) thru tids(NPROC-1)
   c  for each process me => process number [0-(NPROC-1)]
   c-----------------------------------------------------
         print*,'me =',me, '  mytid =',mytid
4        call dowork( me, tids, NPROC )

   c     ------------------------
   c     program finished exit pvm
   c     ------------------------
5        call pvmfexit(info)
         stop
         end
```

Notes



```
      subroutine dowork( me, tids, nproc )
      include '../include/fpvm3.h'
c-------------------------------------------------
c Simple subroutine to pass a token around a ring
c-------------------------------------------------
      integer me, nproc
      integer tids( 0:nproc)

      integer token, dest, count, stride, msgtag

      count  = 1
      stride = 1
      msgtag = 4

      if( me .eq. 0 ) then
        token = tids(0)
        call pvmfinitsend( PVMDEFAULT, info )
        call pvmfpack( INTEGER4,token,count,stride,info)
        call pvmfsend( tids(me+1), msgtag, info )
        call pvmfrecv( tids(nproc-1), msgtag, info )
        print*, 'token ring done'
      else
        call pvmfrecv( tids(me-1), msgtag, info )
        call pvmfunpack( INTEGER4,token,count,stride,info)
        call pvmfinitsend( PVMDEFAULT, info )
        call pvmfpack( INTEGER4,token,count,stride,info)
        dest = tids(me+1)
        if( me .eq. nproc-1 ) dest = tids(0)
        call pvmfsend( dest, msgtag, info )
      endif

      return
      end
```

3 PVM Daemon

The PVM daemon is the control center of the virtual machine. You can activate the PVM daemon by starting the PVM console or by invoking the daemon directly with a list of hosts. The latter will be discussed in chapter 6. To start the console, enter `pvm` at UNIX prompt on your local machine. The PVM console prints the prompt

```
pvm>
```

and accepts commands from standard input. The console allows interactive adding and deleting of hosts to the virtual machine as well as interactive starting and killing of PVM processes. Even if the daemon is started directly, the console can be used to modify the virtual machine.

Console Commands

Here are the commands available in the PVM console:

ADD	add other computers (hosts) to PVM
ALIAS	define and list command aliases you set
CONF	show members in virtual machine
DELETE	remove hosts from pvm
ECHO	echo arguments
HALT	stop all pvm processes and exit deamon
HELP	print this information
ID	print console task identity
JOBS	display list of running jobs
KILL	terminate tasks
MSTAT	show status of hosts
PS	list tasks
PSTAT	show status of tasks
QUIT	exit PVM console, but PVM daemon is still activated
RESET	kill all tasks
SETENV	display or set UNIX environment variables
SIG	send signal to task
SPAWN	spawn task
UNALIAS	remove alias commands you previous set
VERSION	show PVM version

Console Usage

Suppose the console is running on workstation *win210*. This computer will automatically be a host in your virtual machine. Here are some examples of using PVM console:

1. **Activate PVM console**

```
win210> pvm
```

2. **Add amelia and fred to your virtual machine**

```
pvm> add amelia
1 successful
    HOST        DTID
    amelia      c0000
pvm> add fred
1 successful
    HOST        DTID
    fred        100000
```

3. **Check the configuration of your virtual machine**

```
pvm> conf
3 host, 1 data format
    HOST        DTID        ARCH        SPEED
    win210      40000       SGI         1000
    amelia      c0000       SGI         1000
    fred        100000      SGI         1000
```

4. **Delete amelia**

```
pvm> delete amelia
1 successful
    HOST        STATUS
    amelia      deleted
```

5. **Exit PVM console, but PVM daemon is still running**

```
pvm> quit
pvmd still running
win210>
```

4 PVM Library

This chapter introduces the PVM library. In writing your application programs, you need to call the subroutines in the library to instruct PVM to control processes, send information, pack/unpack data, and send/receive messages. Many subroutines have pre-defined option values for some arguments. These are defined in the include file `fpvm3.h` and are listed in the Appendix.

Process Control

call pvmfmytid(tid)

This routine enrolls this process with the PVM daemon on its first call, and generates a unique **tid**. You call this routine at the beginning of your program.

call pvmfexit(info)

This routine tells the local PVM daemon that this process is leaving PVM. You call this routine at the end of your program.Values of **info** less than zero indicate an error.

call pvmfkill(tid, info)

This routine kills a PVM task identified by **tid**. Values of **info** less than zero indicate an error.

call pvmfspawn(pname,flag,where,ntask,tids,numt)

This routine starts up **ntask** instances of a single process named **pname** on the virtual machine. Here are the definition of the other arguments:

flag

Option Value	Meaning
PVMDEFAULT (0)	PVM can choose any machine to start task
PVMHOST (1)	**where** specifies a particular host
PVMARCH (2)	**where** specifies a type of architecture
PVMDEBUG (4)	start up processes under debugger
PVMTRACE (8)	processes will generate PVM trace data

where is where you want to start the PVM process. If **flag** is 0, **where** is ignored.

tids contains identification numbers of PVM processes started by this routine.

numt indicates how many processors started; negative values indicate an error.

Note You should always check **tids** and **numt** to make sure all processes started correctly.

call pvmfparent (tid)

This routine returns the **tid** of the process that spawned this task. If the calling process was not created with **pvmfspawn**, then **tid**=PvmNoParent.

Dynamic Configuration

call pvmfaddhost(host, info)

call pvmfdelhost(host, info)

These routines add and delete hosts to the virtual machine respectively. Values of **info** less than zero indicate an error.

Note Both routines are expensive operations that require the synchronization of the virtual machine.

Message Buffers

call pvmfinitsend(encoding, bufid)

This routine clears the send buffer, and creates a new one for packing a new message.

encoding

Encoding Value	Meaning
PVMDEFAULT (0)	XDR encoding if virtual machine configuration is heterogeneous
PVMRAW (1)	no encoding is done. Messages are sent in their original format.
PVMINPLACE (2)	data left in place. Buffer only contains sizes and pointers to the sent items.

This is not implemented in PVM v3.2.

bufid contains the message buffer identifier. Values less than zero indicate an error.

call pvmffreebuf(bufid, info)

This routine disposes the buffer with identifier **bufid**. You use it after a message has been sent, and is no longer needed. Values of **info** less than zero indicate an error.

Packing and Unpacking

call pvmfpack(what, xp, nitem, stride, info)

call pvmfunpack(what, xp, nitem, stride, info)

These routines pack/unpack your message **xp**, which can be a number or a string. You can call these routines multiple times to pack/unpack a single message. Thus a message can contain several arrays, each with a different data type.

Note There is no limit to the complexity of the packed messages, but you must unpack them exactly as they were packed.

what indicates what type of data **xp** is

STRING (0)	REAL (4)
BYTE1 (1)	COMPLEX8 (5)
INTEGER2 (2)	REAL8 (6)
INTEGER4 (3)	COMPLEX16 (7)

nitem is number of items in the pack/unpack. If **xp** is a vector of 5, **nitem** is 5.

stride is the stride to use when packing.

info is status code returned by this routine. Values less than zero indicate an error.

Sending and Receiving

call pvmfsend(tid, msgtag, info)

This routine labels the message with an integer identifier **msgtag**, and sends it immediately to the process **tid**. Values of **info** less than zero indicate an error.

call pvmfmcast(ntask, tids, msgtag, info)

This routine labels the message with an integer identifier **msgtag**, and broadcasts the message to all **ntask** number of tasks specified in the integer array **tids**. Values of **info** less than zero indicate an error.

call pvmfrecv(tid, msgtag, bufid)

This routine blocks the flow of your program until a message with label **msgtag** has arrived from **tid**. A value of -1 in **msgtag** or **tid** matches anything (wildcard). This routine creates a new active receive buffer, and puts the message in it. Values of **bufid** identify the newly created buffer; values less than zero indicate an error.

call pvmfnrecv(tid, msgtag, bufid)

This routine performs in the same way as **pvmfrecv**, except that it does not block the flow of your program. If the requested message has not arrived, this routine returns **bufid**=0. This routine can be called multiple times for the same message to check if it has arrived, while performing useful work between calls. When no more useful work can be performed, the blocking receive **pvmfrecv** can be used for the same message.

call pvmfprobe(tid, msgtag, bufid)

This routine checks if a message has arrived; however, it does not receive the message. If the requested message has not arrived, this routine returns **bufid**=0. This routine can be called multiple times for the same message to check if it has arrived, while performing useful work between calls.

call pvmfbufinfo (bufid, bytes, msgtag, tid, info)

This routine returns information about the message in the buffer identified by **bufid**. The information returned is the actual **msgtag**, source **tid**, and message length in **bytes**. Values of **info** less than zero indicate an error.

5 Tutorial

This chapter shows you how PVM may be applied to your application programs through a simple example. The example chosen is the Golden Section rule for finding the maximum of a function. You may remember it from Math class in high school. Let us review the method and the algorithm.

Golden Section

Suppose we want to find the maximum of a curve y=f(x); where x is between the interval a_1 and a_2. The points a_3 and a_4 are symmetrically placed in this interval, so that

$$a_3 = (1-\alpha)\, a_1 + \alpha\, a_2 \qquad \text{(EQ 1)}$$

$$a_4 = \alpha\, a_1 + (1-\alpha)\, a_2 \qquad \text{(EQ 2)}$$

See Figure 1 at left. Golden Section rule requires α to be 0.382.

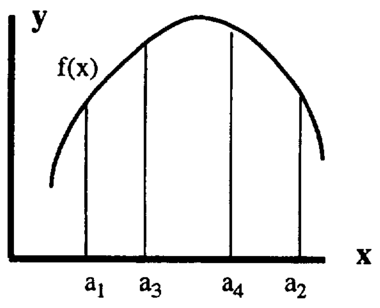


Figure 1. Interval division for Golden Section

The algorithm of finding the maximum is as follow:

If $f(a_4) < f(a_3)$		If $f(a_4) > f(a_3)$	
1	Consider new interval (a_1,a_4)	1	Consider new interval (a_3,a_2)
2	Apply EQ.(1) and (2) again	2	Apply EQ. (1) and (2) again
3	Until maximum is reached	3	Until maximum is reached

If $f(a_3)=f(a_4)$, the maximum is found

The FORTRAN program (Serial Program) on the opposite page is the Golden Section rule that a programmer would write on a normal serial computer. Please spend a few minutes to study the flow of the program. This simple program consists of two parts, the main (calling) program and the function subroutine. The latter has only four lines.

Note Notice that for each interval (a_1,a_2), we need to call the function evaluation four times to find $f(a_1)$, $f(a_2)$, $f(a_3)$, and $f(a_4)$.

Serial Program

```
C       Linear optimization:
C
C       Search for maximum of a x-y curve.
C
        DIMENSION A(4),FN(4)
C
C       Initial interval
        L = 0
        TOL = 1.E-3
        A(1) = 0.4
        A(2) = 1.6
        ALPHA = 0.382
C
C
 10     CONTINUE
C
C       Loop begins:
        L = L + 1

        A(3) = (1.-ALPHA)*A(1) + ALPHA*A(2)
        A(4) = ALPHA*A(1) + (1.-ALPHA)*A(2)
        FN(1) = F(A(1))
        FN(2) = F(A(2))
        FN(3) = F(A(3))
        FN(4) = F(A(4))
        WRITE(10,*) 'A  ',A(1),A(2),A(3),A(4)
        WRITE(10,*) 'F  ',FN(1),FN(2),FN(3),FN(4)
        WRITE(10,*) ' '
        ERR = ABS(FN(2)-FN(3))
        IF(ERR.LE.TOL) GOTO 999
C
C
        IF(FN(4) .GT. FN(3)) THEN
          B1 = A(3)
          B2 = A(2)
          A(1) = B1
          A(2) = B2
          GOTO 10
        ELSEIF(FN(4) .LT. FN(3)) THEN
          B1 = A(1)
          B2 = A(4)
          A(1) = B1
          A(2) = B2
          GOTO 10
        ENDIF
 999    CONTINUE
        STOP
        END

        FUNCTION F(X)
        F = TANH(X)/(1.+X*X)
        RETURN
        END
```

Golden Section rule → ALPHA = 0.382

Equations (1) and (2) → A(3), A(4)

Four function evaluations: FN(1)–FN(4)

If $f(a_4) > f(a_3)$

If $f(a_4) < f(a_3)$

Function evaluation

PVM Master Guideline

Recall that in the procedure of finding a new interval, the program calls the function evaluation four times *serially* to get f(a_1), f(a_2), f(a_3), and f(a_4). We would like to assign four processors to perform the four function evaluations *simultaneously* on the virtual machine. Therefore, we modify the Serial Program by writing the main (calling) program as a Master program, and the function subroutine as a Slave program.

The following steps are general guidelines to writing a Master program. Please study the steps, and compare them with the program on the opposite page. Also compare it with the Serial Program.

1. **Include fpvm3.h**

   Include this file in your program, you are able to use the PVM preset variables; such as `PVMDEFAULT`, `REAL4`, and more, mentioned in Chapter 4 and the Appendix.

2. **Enroll Master to PVM**

   Use `pvmfmytid(mytid)` to enroll.

3. **Assign virtual processors**

   Use the following call to spawn `nproc` function processes.
   `pvmfspawn(pname,PVMDFAULT,where,nproc,tids,numt)`
   Also tell PVM the name of the Slave program (`pname`). PVM returns `tids`, the identifier of the `nproc` processors.

4. **Initialize buffer and pack data**

   Use `pvmfinitsend` to clear buffer.
   Use the following routine to pack a real array `A` of dimension `m`.
   `pvmfpack(REAL4,A,m,1,info)`

5. **Send message**

   Use the following call to send the packed message to the Slave process identified by `tids`.
   `pvmfmcast(nproc,tids,msgtag,info)`

Master Program

```
C     Linear optimization:
C     Search for maximum of a x-y curve.
       PROGRAM MASTER
C
C
```

1

```
      include '../include/fpvm3.h'
      DIMENSION A(4),FN(4)
      integer tids(0:32),who
      character*8 where
      character*12 pname
```

2

```
c     Enroll this program in PVM
      call pvmfmytid(mytid)
c     Start up the four processors
      nproc = 4
```

Assign four processors

```
      where = '*'
      pname = 'function'
```

Slave program's name

3

```
      call pvmfspawn(pname,PVMDEFAULT,where,nproc,tids,numt)
      do 20 i=0,nproc-1
        write(*,*) 'tid', i, tids(i)
 20   continue
C
C     Initial interval
      L = 0
      A(1) = 0.4
      A(2) = 1.6
      ALPHA = 0.382
      TOL = 1.E-3
      ERR = 1.
C
 10   CONTINUE
C
C
C     Loop begins:
      L = L + 1
```

Equations (1) and (2)

```
      A(3) = (1.-ALPHA)*A(1) + ALPHA*A(2)
      A(4) = ALPHA*A(1) + (1.-ALPHA)*A(2)

c
c     Broadcast data to all node programs
c     first pack them, then send them
```

4

Pack nproc, tids, A, and ERR

```
      call pvmfinitsend(PVMDEFAULT,info)
      call pvmfpack(INTEGER4,nproc,1,1,info)
      call pvmfpack(INTEGER4,tids,nproc,1,info)
      call pvmfpack(REAL4,A,4,1,info)
      call pvmfpack(REAL4,ERR,1,1,info)
c
c
```

5

msgtype value matches the one received in Slave program

```
      msgtype = 1
      call pvmfmcast(nproc,tids,msgtype,info)
c
```

6. **Wait until messages come from Slaves**

   Use `pvmfrecv()` to block until Slaves return function values. Make sure value of `msgtype` matches values coming from Slaves.

7. **Receive and Unpack data**

   The sequence of unpacking is the same as the packing in the Slave.

8. **Exit PVM**

   Use `pvmfexit(info)` to exit PVM.

6

```
c     Wait for results from processors
```

msgtype value matches the one sent from Slave program

7

Receive /unpack FN and 'who' from the 4 processors one by one

```
      msgtype = 2
      do 100 i=1,nproc
        call pvmfrecv(-1,msgtype,info)
        call pvmfunpack(INTEGER4,who,1,1,info)
        call pvmfunpack(REAL4,FN(who),1,1,info)
100   continue

      WRITE(10,*) 'A   ',A(1),A(2),A(3),A(4)
      WRITE(10,*) 'F   ',FN(1),FN(2),FN(3),FN(4)
      WRITE(10,*) ' '
      ERR = ABS(FN(2)-FN(3))
      IF(ERR.LE.TOL) GOTO 999
C
C
      IF(FN(4) .GT. FN(3)) THEN
        B1 = A(3)
        B2 = A(2)
        A(1) = B1
        A(2) = B2
        GOTO 10
      ELSEIF(FN(4) .LT. FN(3)) THEN
        B1 = A(1)
        B2 = A(4)
        A(1) = B1
        A(2) = B2
        GOTO 10
      ENDIF
c
c     Program finished leave PVM before exiting
999   continue
      call pvmfexit(info)
      STOP
      END
```

8

PVM Slave Guideline

The Slave program is basically the function evaluation program. In order to do the function evaluation, it needs information from Master. For example, it needs the identity numbers (`tids(1),...,tids(4)`) that PVM assigns, and the values of $a_1,...,a_4$.

The following steps are general guidelines to writing a Slave program. Please study the steps, and compare them with the program on the opposite page. Also try to find the connection with the Master Program. You may find Figure 1 helpful.

1. **Include fpvm3.h**

   Include this file in your program, you are able to use the PVM preset variable names; such as `PVMDEFAULT`, `REAL4`, and more, mentioned in all tables in Chapter 4 and the Appendix.

2. **Enroll Slave with PVM**

   Use `pvmfmytid(mytid)` to enroll.

3. **Identify the parent of this process**

   Use the following call to obtain the task identifyer (`mtid`) of parent process. This is useful for returning solutions to the Master.
   `pvmfparent(mtid)`

4. **Receive and Unpack data**

   Make sure the value of `msgtype` matches the one from Master. The sequence of unpacking is the same as the packing in Master.

5. **Perform function evaluation**

6. **Initialize buffer and pack data**

   Use `pvmfinitsend` to clear buffer.

   Use the following call to pack a real array `F` of dimension `n`.
   `pvmfpack(REAL4,F,n,1,info)`

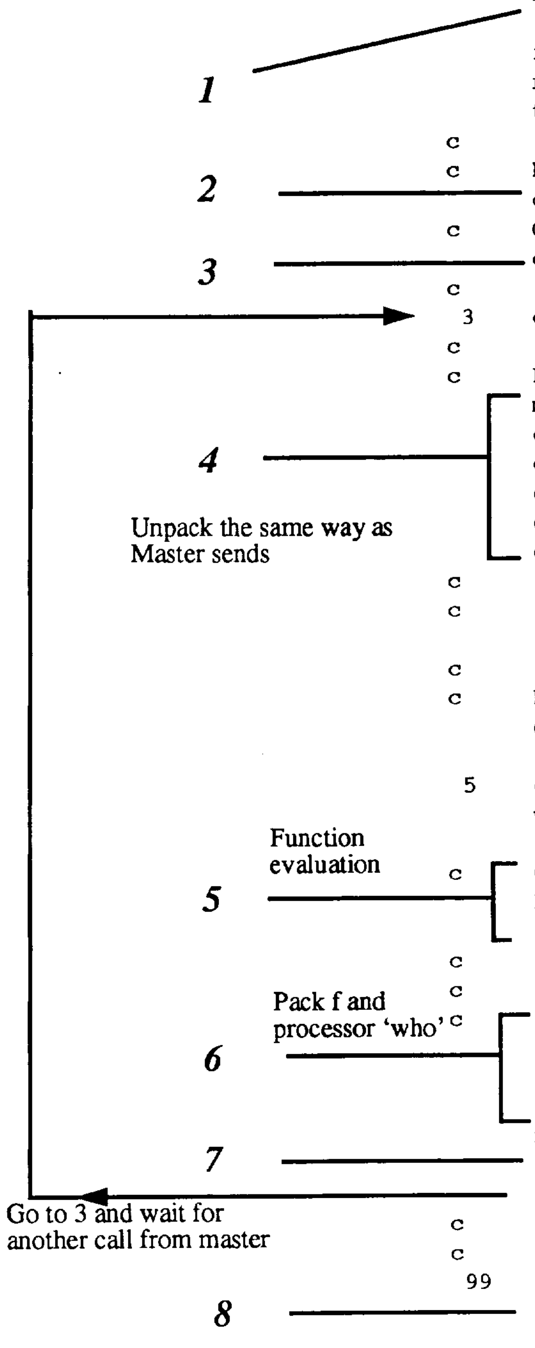
7. **Send data**

   Use the following call to send the packed message to Master:
   `pvmfsend(mtid,msgtag,info)`

8. **Exit PVM**

   Use `pvmfexit(info)` to exit PVM.

Slave Program


1
2
3
4
Unpack the same way as Master sends
Function evaluation
5
Pack f and processor 'who'
6
7
Go to 3 and wait for another call from master
8

```
c
      program function
c
c
      include '../include/fpvm3.h'

      integer tids(0:32),who
      real a(32)
      tor = 1.e-3
c
c     Enroll this program in PVM
      call pvmfmytid(mytid)
c     Get the parent's task id
      call pvmfparent(mtid)
c
 3    continue
c
c     Receive data from host
      msgtype = 1
      call pvmfrecv(mtid,msgtype,info)
      call pvmfunpack(INTEGER4,nproc,1,1,info)
      call pvmfunpack(INTEGER4,tids,nproc,1,info)
      call pvmfunpack(REAL4,A,4,1,info)
      call pvmfunpack(REAL4,ERR,1,1,info)
c
c
      if(err.le.tor) go to 99
c
c     Determine which processor I am
      do 5 i=0,nproc-1
        if(tids(i).eq.mytid) me = i
 5    continue
      who = me + 1
c     Calculate the function
      X = A(who)
      f = TANH(X)/(1.+X*X)
c
c
c     Send the result to Master
      call pvmfinitsend(PVMDEFAULT,info)
      call pvmfpack(INTEGER4,who,1,1,info)
      call pvmfpack(REAL4,f,1,1,info)
      msgtype = 2
      call pvmfsend(mtid,msgtype,info)
      go to 3
c
c     Program finished. Leave PVM before exiting
 99   continue
      call pvmfexit(info)
      stop
      end
```

Compilation and Running

After you finish your program, it is time to compile and run. Follow the steps below to compile your programs.

1. **Make sure you have the correct directory setup**

   Follow the advice from *Directory Setup* in Chapter 1.

2. **Compile the program**

   Use the sample `Makefile` on the opposite page to compile your programs.

Note The `Makefile` links the PVM library, `libfpvm3.a`.

3. **Copy executables to all the hosts**

   Follow the advice from *Directory Setup* in Chapter 1, and distribute the executables to `$HOME/pvm3/bin/ARCH`.

4. **Activate PVM**

   Activate PVM by entering `pvm` at UNIX prompt.

5. **Decide the configuration of the virtual machine**

   Add or delete hosts to the virtual machine. (Chapter 3)

6. **Quit PVM console**

   Leave PVM console (don't halt daemon) by entering `quit` at the `pvm` prompt.

Makefile

PVM Library

Make appropriate changes for your own path

```
#
# Custom section
# Set PVM_ARCH to your architecture type (SUN4, HP9K, RS6K, # SGI,
etc.)
# if PVM_ARCH = BSD386 then set ARCHLIB = -lrpc
# if PVM_ARCH = SGI    then set ARCHLIB = -lsun
# if PVM_ARCH = I860   then set ARCHLIB = -lrpc -lsocket
# if PVM_ARCH = IPSC2  then set ARCHLIB = -lrpc -lsocket
# otherwise leave ARCHLIB blank
#
# PVM_ARCH and ARCHLIB are set for you if you use 'aimk'.
#
PVM_ARCH        =       SGI
ARCHLIB         =       -lsun
# END of custom section - leave this line here
#
PVMDIR   =      /amd/fs02/pub/iris4d_irix4/nas/pkg/pvm3.2
PVMLIB   =      $(PVMDIR)/lib/$(PVM_ARCH)/libpvm3.a
SDIR     =      .
BDIR    =      /u/wk/cheung/pvm3/bin
XDIR     =      $(BDIR)/$(PVM_ARCH)

CFLAGS   =      -g -I../include
LIBS     =      $(PVMLIB) $(ARCHLIB)

F77      =      f77
FFLAGS   =      -g
FLIBS    =      $(PVMDIR)/lib/$(PVM_ARCH)/libfpvm3.a $(LIBS)

default:        master function

$(XDIR):
        - mkdir $(BDIR)
        - mkdir $(XDIR)

clean:
        rm -f *.o bfgs quadfunct

master: $(SDIR)/master.f  $(XDIR)
       $(F77) $(FFLAGS) -o master $(SDIR)/master.f $(FLIBS)
        mv master $(XDIR)

function: $(SDIR)/function.f  $(XDIR)
        $(F77) $(FFLAGS) -o function $(SDIR)/function.f $(FLIBS)
        mv function $(XDIR)
```

6 *Problems and Tips*

PVM is a relatively new piece of software. It is not advanced enough to warn you ahead of time before problems come. Here are a couple of cases that you may encounter as a beginner.

Problems

Can't activate PVM

- If the message you get, after entering `pvm` at UNIX prompt, is
  `libpvm [pid-1]: Console: Can't start pvmd,`
  it is possible that the last time you halted PVM daemon, the daemon created a residual file `/tmp/pvmd.xxxx`; where `xxxx` is an unique number for you. Delete this file, and start PVM again.
- If the daemon is running but the PVM console will not start, it is possible that you have too many processes running. You have to kill all the processes before you re-activate PVM console.

Note Use `ps -ef | username` at UNIX prompt to locate your running processes.

Can't add hosts

It is possible that there are no links between your local computer and the other hosts. Check the following two things:

- Make sure each of your hosts has a `.rhosts` file in the `$HOME` directory, and this file points to your local computer.
- Make sure the `.rhosts` file is "read" and "write" protected from others users.

Host File

You can create the following file to build the virtual machine without activating the PVM console. The addresses must be recognizable by your system.

```
computer1.address  ┐
computer2.address  ├── host file
computer3.address  │
computer4.address  ┘
```

Note The first machine listed must be the initiating host.

Note If tasks are to be spawned on specific systems, the system name contained in `where` (routine **pvmfspawn**) must match the name in the host file <u>exactly</u>.

Note If spawning tasks are on the initiating host, use the truncated host name. For example, if the full address is `win210.nas.nasa.gov` ;
use `win210` instead. This is a bug in PVM v3.2.

Having the host file ready, enter the following at UNIX prompt,

```
win210> pvmd3 host
```

Notes

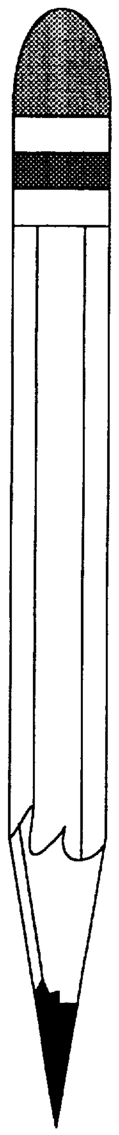
Place to jot down problems.

If encounter problems, please contact:

Merritt Smith: mhsmith@nas.nasa.gov

or

Samson Cheung: cheung@nas.nasa.gov



Appendix

TABLE 1. ARCH names used in PVM.

ARCH	Machine	Note
AFX8	Alliant FX 8	
ALPHA	DEC Alpha	DEC OSF-1
BAL	Sequent Balance	DYNIX
BFLY	BBN Butterfly TC2000	
BSD386	80386/486 Unix box	BSDI
CM2	Thanking Machines CM2	Sun front-end
CM5	Thanking Machines CM5	
CNVX	Convex C-series	
CNVXN	Convex C-series	native mode
CRAY	C-90, YMP, Cray-2	UNICOS
CRAYSMP	Cray S-MP	
DGAV	Data General Aviion	
HP300	HP-9000 model 300	HPUX
HPPA	HP-9000 PA-RISC	
I860	Intel iPSC/860	link-lprc
IPSC2	Intel iPSC/860 host	SysV
KSR1	Kendall Square KSR-1	OSF-1
NEXT	NeXT	
PGON	Intel Paragon	link -lprc
PMAX	DECstation 3100,5100	Ultrix
RS6K	IBM/RS6000	AIX
RT	IBM RT	
SGI	Silicon Graphics IRIS	link -lsun
SUN3	Sun 3	SunOS
SUN4	Sun 4, SPARCstation	
SYMM	Sequent Symmetry	
TITN	Staedent Titan	
UVAX	DEC Micro VAX	

TABLE 2. **Error codes returned by PVM routines**

Error Code	Meaning
PvmOK (0)	All right
PvmBadParam (-2)	Bad parameter
PvmMismatch (-3)	Barrier count mismatch
PvmNoData (-5)	Read past end of buffer
PvmNoHost (-6)	No such host
PvmNoFile (-7)	No such executable
PvmNoMem (-10)	Can't get memory
PvmBadMsg (-12)	Can't decode received massage
PvmSysErr (-14)	Pvmd not responding
PvmNoBuf (-15)	No current buffer
PvmNoSuchBuf (-16)	Bad message identifyer
PvmNukkGroup (-17)	Null group name is illegal
PvmDupGroup (-18)	Already in group
PvmNoGroup (-19)	No group with that name
PvmNotInGroup (-20)	Not in group
PvmNoInst (-21)	No such instance in group
PvmHostFail (-22)	Host failed
PvmNoParent (-23)	No parent task
PvmNoImpl (-24)	Function not implemented
PvmDSysErr (-25)	Pvmd system error
PvmBadVersion (-26)	Pvmd-pvmd protocol mismatch
PvmOutOfRes (-27)	Out of resources
PvmDupHost (-28)	Host already configurated
PvmCantStart (-29)	Fail to execute new slave pvmd
PvmAlready (-30)	Slave pvmd already running
PvmNoTask (-31)	Task does not exist
PvmNoEntry (-32)	No such (group,instance)
PvmDupEntry (-33)	(Group,instance) already exists