Technical Note 1: Porting Fortran Software to a Beowulf Cluster

Paul Michaels <pm@cgiss.boisestate.edu>
Amit Jain <amit@cs.boisestate.edu>

5th November 2004

Abstract

There are many ways to convert a serial Fortran language program to a parallel program needed for execution on a Beowulf cluster. Here, we focus on a method which can be implemented in either Parallel Virtual Machine (PVM) or Message Passing Interface (MPI). We step through the basic processes assuming that the Single Program Multiple Data (SPMD) paradigm will be implemented in PVM. Translation into MPI is straight forward for this paradigm. The steps are demonstrated on a trivial Fortran 77 program which has been inspired by the experience gained on converting a more complicated code which measures body wave dispersion in a down-hole geophysical survey.

The trivial program, *spmd_0.f*, distills the essence of the problem without the burden of a highly detailed computation. The program computes the cube of a series of numbers (1 to *npts*, where *npts* is specified on the command line as an argument). To maintain maximum portability, we avoid using special extensions which may be found on some commercial compilers. Where special functions are needed, we demonstrate solutions using standard C-language functions, linked to the Fortran 77 code.

Contents

List of Figures

1 Converting a Serial Program

A listing of the serial program, *spmd_0.f*, is given in Appendix A at the end of this note. The SPMD paradigm involves only a single program which is concurrently spawned on many processing nodes, each node taking a portion of the task. It is different from the other Master-Slave paradigms which can only be implemented on PVM. The steps can be summarized as follows:

1. **Sketch the program architecture**. The sketch need not be as detailed as a flow chart, but should focus on the major loops.
2. **Instrument the Code**. This involves adding timing calls to examine portions of the code which might be split up among processing nodes. Portions of the code may execute slower, depending on the data being processed. One looks for trends. For example, in our dispersion problem we asked the question, "do high frequencies take longer than low frequencies?" In our case, it did not matter. This is very problem specific, and often data specific. In this example problem, one might ask if large numbers take longer to cube than small numbers.
3. **Profile the Code**. This is done by compiling with the -pg option, and then running the *gprof* program on the result of a test case.
4. **Decide on how to divide up the work.** Often, this involves identifying a major loop which can be split up among the processing nodes which may be available. One looks for a large chunk of the program, since this will minimize communications between tasks. The ideal case might be a major outer loop where no step in the loop is dependent on any other step.
5. **Modify the code.** Here, one begins by adding commands which initialize and spawn copies of the program. We will need to determine of how many instances are spawned, and identification of each process so that the sibling tasks know who they are and what their share of the work is. One task is selected as the master, or zero process. The zero process will have responsibility for sending any thing needed to siblings, and for receiving each siblings contribution, followed by an assembly of the results. Scratch disk space can be used if needed on each processing node, but the final results are assembled on the zero process node.
6. **Debug** the code modifications.
7. **Add pre-processor code** which creates a single code base. This is an optional step. The idea is to be able to compile both the original serial and the modified parallel versions from a single file.

1.1 Sketching the Architecture

The source code for *spmd_0.f* and the corresponding architecture is given in Figure 1. One should stand back and look at the big picture when sketching the program architecture. Try to block out large sections that do not depend on results determined elsewhere in the program. This problem is of the embarrassingly parallel type, and one naturally focuses on the main computational loop for the place to divide up the labor.

The next question would be how sensitive the main computational loop is to variations in the data. For example, do large numbers take longer to cube than small numbers? To answer that question, we can instrument the main loop.

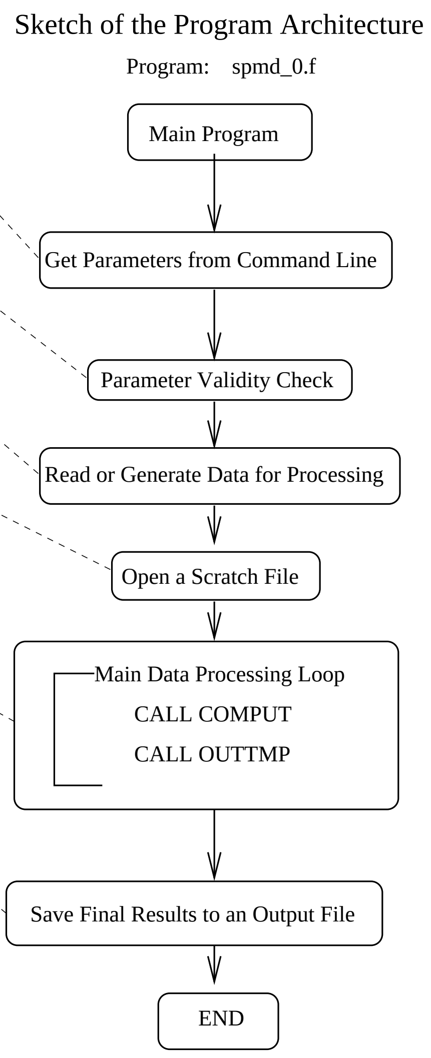
1.2 Instrumenting the Code

What we would like to do is a test which captures computation time within the loop. We will use the modulo function to avoid printing every step (and hence obtain an averaged value which may be more representative and easier to look at). The modified section of the Fortran code is as follows:

Figure 1: Program Listing and Architecture

spmd_0.f   Fri Jul 02 13:40:31 2004   1

```
c  P. Michaels  <pm@cgiss.boisestate.edu>  27 June 2004
c  Example program spmd_0.f:  serial program for conversion
c                             to pvm, beowulf cluster
      program main
      parameter (NDIM=10000000)
c
      character*80      arg1
      integer*1         bz
      character*80      infil , outfil,  outlst
      real*4            values(NDIM),results(NDIM)
c
c---get input parameter
      nargsx=iargc()
      if(nargsx.ge.1) then
      call getarg(1,arg1)
      read(arg1,'(i10)') npts
      else
      write(*,*) 'USAGE: spmd_0 <npts>'
      stop
      endif
c
c---check for valid parameter
      if(npts.gt.NDIM) then
      write(*,*) 'ABORT: dimensions exceeded, NDIM'
      stop
      endif
c
c---generate some data
      bz=0
      do 50, j=1,npts
      values(j)=float(j)
      results(j)=0.0
  50  continue
c
c---define a scratch file
      write(outfil,'(80(1h ))')
      write(outfil,'(''data.txt'')')
c---open it for I/O
      open(unit=3,file=outfil,access='sequential')
c
c---process the data-----------------------
      do 100, j=1,npts
      call comput(values(j),results(j))
c
c---perhaps write results to temporary file for additional
c   processing or possibly debugging
      call outtmp(j,results(j))
c
 100  continue
c  end of data processing loop-------------
c
c---close temporary file
      close(3)
c
c
c...save final results in file in working directory
      call finio(npts,results)
      stop
      end
c-----------------------------------------------------------
      subroutine comput(valuex,resultx)
      real*4 valuex,resultx
      resultx=valuex*valuex*valuex
      end
c-----------------------------------------------------------
      subroutine outtmp(j,resultx)
      integer*4         j
      real*4            results
      write(3,'(1x,i10,1x,e14.7)') j,resultx
      end
c-----------------------------------------------------------
      subroutine finio(npts,results)
      integer*4         npts
      real*4            results(*)
      character*80      outfil
c
      write(outfil,'(''spmd_0-results.txt'')')
      open(unit=3,file=outfil,access='sequential')
      write(3,*) 'Final Results for spmd_0'
c
      do 620, j=1,npts
      write(3,'(1x,e14.7,1x)') results(j)
  620 continue
      close(3)
      end
```


Sketch of the Program Architecture
Program: spmd_0.f
Main Program
Get Parameters from Command Line
Parameter Validity Check
Read or Generate Data for Processing
Open a Scratch File
Main Data Processing Loop
CALL COMPUT
CALL OUTTMP
Save Final Results to an Output File
END

Code Listing 1.2: Instrumentation of the code with gettime()

```
c---process the data----------------------
      jdel=npts/5
      tmr0=gettime()
      do 100, j=1,npts
      if (mod(j,jdel).eq.0) then
      tmr=gettime()
      tmr1=tmr-tmr0
      tmr0=tmr
      write(*,'(I10," time=",f18.4)') j, tmr1
      endif
      call comput(values(j),results(j))
c
c---perhaps write results to temporary file for additional
c processing or possibly debugging
      call outtmp(j,results(j))
c
  100 continue
c end of data processing loop-------------
```

The function, *gettime()*, is defined in a C-language function which is then compiled as an object module, added to a local static library, *sublibc.a*, and then linked to the Fortran code. There are a number of tricks required to make this work. While the C-language function is *gettime()*, Fortran will add an underscore to the name, *gettime_()*. Thus, in the C-language file, a line is added which defines the underscored function as returning the non-underscored version. Another point is that the C-language has no concept of a subroutine, everything is a function. This is no problem here, since we are treating it like a floating point function in the Fortran code. One can transfer arguments and call C-functions as if they were subroutines, and we will show that later.

1.2.1 How *gettime()* works

This C-language function calls a C-library routine called *gettimeofday*(). The data structure returned by gettimeofday has two parts, the number of seconds and the number of microseconds since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC). This is a big number, so our function, gettime() computes an offset which reduces the result to seconds measured from the beginning of the year. An additional offset subtracts a value to bring the result in the range from 0 to 10,000 seconds. This second offset avoids an underflow, since we will be returning a single precision value to the Fortran program. A single code base for our C-language timing routines is kept in *timing.c*. The relevant parts are shown below:

Code Listing 1.2.1: *gettime()* components of *timing.c*

```
/* Suitable for Fortran77   timing.c   */
#include <stdio.h>
#include <sys/times.h> /* for times system call */
#include <sys/time.h> /* for gettimeofday system call */
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
float gettime();
float gettime_() { return gettime(); }
#define SECONDS_IN_A_YEAR 365*24*3600

float gettime()
{
  struct timeval now;
  double offset;
  long years;
  double valu;
  float value;
  gettimeofday(&now, (struct timezone *)0);
  years = (long)now.tv_sec/(SECONDS_IN_A_YEAR);
/*offset to get seconds into the year*/
  offset = years*SECONDS_IN_A_YEAR - 1;
  valu = (double) (((double) now.tv_sec +
                (double)now.tv_usec/1000000.0) - offset);
/*offset to avoid underflow when converting to float*/
  offset = (double) floor((valu/10000.0))*10000.0;
  value = (float) (valu-offset); return value;
}
```

1.2.2 The Fortran Makefile

The make file is show below. It builds the C-subroutine library (by calling a Makefile in that directory), compiles the Fortran, and links the whole thing together. One only needs to type *make* from within the source directory where the Fortran main program is located. The directory tree looks something like this:

```
Fort
      include
      sublibc
           timing.c
           Makefile
           sublibc.a
      fortran
           spmd_0.f
           spmd_0a.f
           Makefile
```

Both the *sublibc* directory, and the *fortran* directory have their own make files. The program *spmd_0a.f* is the instrumented version of the original serial program.

Code Listing 1.2.2: Makefile for Fortran programs

```
#Date:  June 2004
# <pm@cgiss.boisestate.edu>
BIN=$(HOME)/bin/
#
#compile flags
FLAGS=-Wall -pg -g
EXEC=spmd_0 spmd_0a
FC=g77
SUBLIBC=../sublibc/sublibc.a
MAKEC=../sublibc
INCL=../include
all:  $(EXEC) $(SUBLIBC)
     spmd_0:  spmd_0.o $(SUBLIBC)
     $(MAKE) -C $(MAKEC);
     $(FC) $(FLAGS) -o spmd_0 spmd_0.o \
     $(SUBLIBC)
spmd_0.o:  spmd_0.f
     $(FC) $(FLAGS) -c -o spmd_0.o \
     spmd_0.f
spmd_0a:  spmd_0a.o $(SUBLIBC)
     $(MAKE) -C $(MAKEF);
     $(MAKE) -C $(MAKEC);
     $(FC) $(FLAGS) -o spmd_0a spmd_0a.o \
     $(SUBLIBC)
spmd_0a.o:  spmd_0a.f
     $(FC) $(FLAGS) -c -o spmd_0a.o \
     spmd_0a.f
$(SUBLIBC):
     $(MAKE) -C $(MAKEC);
install:  $(EXEC)
     mkdir -p $(BIN)
     install -m 775 spmd_0 $(BIN)spmd_0
     install -m 775 spmd_0a $(BIN)spmd_0a
     rm $(EXEC)
clean:
     $(MAKE) -i -C $(MAKEC) clean;
     rm *.o;
     for i in $(EXEC) ; do rm $$i; done
```

1.2.3 The SUBLIBC Makefile

For completeness, the Makefile in the sublibc directory (called by the Fortran Makefile) is shown below. It is assumed that the reader is familiar with the *make* program (which would include where tab's are mandatory and the fact that a continuation character, '\', must be the very last character on a line). This make file will build both *timing.c* and *runcmd.c*. We will discuss runcmd.c a bit later (it is used to execute system commands from a running Fortran program). It will also build an example test program, *prog.f*, to be discussed later. The program, *prog.f* illustrates how to link a built in C-function to a Fortran executable.

Code Listing 1.2.3: Makefile for the local C-subroutine library

```
# P. Michaels <pm@cgsiss.boisestate.edu>
# July 2004
#SUBLIBC    Makefile
CC=gcc
INCL=../include
CFLAGS=-Wall -g -c -pg
SUBC=sublibc.a
OBJC= timing.o runcmd.o
all:  $(OBJSC) $(SUBC)
$(SUBC): $(OBJC)
     rm sublibc.a ; \
     ar ruv sublibc.a *.o ; \
     ranlib sublibc.a ; \
%.o:  %.c
     $(CC) $(CFLAGS) -o $@ $< -I$(INCL)
clean:
     rm *.o *.a
prog:  prog.f
     g77 -Wall prog.f sublibc.a -lc -o prog
```

1.2.4 The output from the instrumented version

The following standard output was observed on running the program *spmd_0a*.

```
[pm@penguin testing]$ spmd_0a 2000000
400000 time= 3.2695
800000 time= 3.3789
1200000 time= 3.4600
1600000 time= 3.5430
2000000 time= 3.4717
```

There is not enough of a change in execution times to make any additional program complexity necessary. Not only is each step in the loop independent of any other step, the amount of time does not vary significantly with the data. Had it varied, one could compensate in the step where we divide up the labor among the different spawned tasks.

1.3 Profi le the Code

In a larger program, there may be many subroutines and function calls. One naturally wants to focus the improvements on portions of the program that account for a significant portion of the computational and I/O time. Profiling the code is done by compilation with the -pg option. This option should be included with the compilation flags. See code listing 1.2.2 for example, below the comment "`#compile flags`".

1.3.1 Running *gprof*, and Examining the Output

After compiling spmd_0.f with the -pg option, one runs the program. In addition to the normal output, an additional file, *gmon.out*, is created. This file contains profiling information required by the program *gprof*. From the directory with the *gmon.out* file, one types something like the following:

```
gprof  ~/bin/spmd_0  | less
```

Here, the program name, *spmd_0*, is only an example. In general, one would replace *spmd_0* with the name of the executable being profiled. You do not have to specify *gmon.out* for the profile file name, as this is the default. For our example program, the top of the listing looks like this:

```
Flat profile:
Each sample counts as 0.01 seconds.
% cumulative self self total
time  seconds seconds   calls  ms/call ms/call    name
48.48    0.16    0.16       1   160.00  330.00    MAIN__
36.36    0.28    0.12 2000000     0.00    0.00    comput_
12.12    0.32    0.04       1    40.00   40.00    finio_
3.03     0.33    0.01 2000000     0.00    0.00    outtmp_

% the percentage of the total running time of the
time program used by this function.
```

From this portion of the listing, we see that 36.36% of the time is spent in the subroutine, *comput*. We also note that the subroutine, comput, was called 2 million times.

1.4 Dividing Up the Work

After examining the *gprof* output and the source code for *spmd_0.f*, we identify the main do-loop, which includes calls to both *comput* and *outtmp* subroutines, as being a place to divide the labor and temporary disk space usage. The two subroutine calls within this loop account for nearly 40% of the execution time. The *outtmp* subroutine was included here as an example of how to write to temporary disk space on sibling nodes, and is not really essential to this example for any other reason.

1.5 Modifying the Code

There are two topics to be addressed at this point.

1. Calling functions not included in Fortran77 which will permit us to make system calls and take advantage of the wide range of C-language functions.

2. Adding the code to spawn sibling tasks on the cluster's additional processing nodes and disks. For this example, this will be done using the PVM subroutine calls in Fortran.

1.5.1 Calling System and C-language Functions

Our first example is a function, *runcmd(string_argument)*. This is a local C-language function which executes the the built in C-language function, *system(string_argument)*. From the Fortran point of view, we just call a subroutine, "`call runcmd(string_argument)`", and the system executes the character string argument. Of course the string argument must be a valid system command for Unix. We have chosen this example to illustrate some of the tricks required in passing strings between Fortran and C-language programs. Further, it is essential to our problem, since we will need to make a child directory under */tmp* on each sibling node.

Example runcmd The source code for runcmd() is shown below. It provides an interface to map the underscore assumption of the Fortran compiler (runcmd_() in Fortran link, as opposed to runcmd() as written in C). The other odd thing is that an additional argument is passed to the C-function which is not included in the argument list of the Fortran call statement. Do not put this argument in the Fortran call, it is done by magic, and is available to the C-function. This argument is the length of the character array in the Fortran program. In this specific case, we only need to include the argument in the C-function definition, even though we choose not to use it in this case.

The other odd thing to remember is that C-language expects a null (zero) as the byte which terminates any string (Fortran has no such assumption). What we must do is write that zero byte at the end of the string on the Fortran side, so it will help the C-function know where the command string stops. If you don't do this, expect disaster. It is very important! If you don't do it, then a simple make directory command may make a bunch of directories all over your system with weird binary and other non-printable names, depending on what is in the memory that has not been initialized. We show how to add this zero byte in the listing of the main Fortran calling program which follows the *runcmd.c* listing.

Code Listing 1.5.1a: runcmd.c

```
/* Suitable for Fortran77 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
void runcmd(char *cmd, int n); /* prototype */

/* interface to map underscore */
void runcmd_(char *cmd, int n) { runcmd(cmd, n);}

/* the function runcmd() */
void runcmd(char *cmd, int n)
{
  system(cmd);
}
```

The above function is is located in our sublibc directory, compiled with the Makefile in that directory, and then added to the static subroutine library, *sublibc.a*. Note that the C-function *runcmd.c* has two arguments, but the Fortran77 calling program below has only one. The extra integer argument, n, is created by the magic of the linker. The main program which tests *runcmd* is:

Code Listing 1.5.1b: running a system command, prog.f

```
      program main
      integer*1 bz
      character*1024 cmd
c
c assign zero to bz, terminates a character string for C
      bz=0
      write(cmd,'(" ls -al ",a)') bz
      call runcmd(cmd)
      stop
      end
```

In this case, we have tested the program with a simple listing of the current directory. The one-byte integer, bz, is zero (NULL in C), and is written with a Fortran “a” format at the end of the string in the write command. Later, we will write a make directory command into the character string, cmd, and use this in our spmd_0.f example.

Bonus Example While it has nothing to do with our current needs, we include another example which calls a standard C-function directly, and also passes a string argument back from C to Fortran. Calling a C-function directly will require linking to the dynamic C-library. This amounts to a command line argument, “-lc” (lower case L before the c). See the Makefile (code listing 1.2.3). The following code is needed in our *timing.c* code base:

Code Listing 1.5.1c: relevant code in *timing.c* for strlen() and timeofday()

```
#include <stdio.h>
#include <sys/times.h>
#include <sys/time.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
/* interface to map underscore link of strlen_ to standard
strlen()
and convert a C-function result to an argument in a subroutine
call */
void strlen_(const char *str, int *len1 ) { *len1=strlen(str);}

void timeofday(char *tod1)
{
  time_t now;
  struct tm *ptr;
  static char *tt;
/* get current time and convert to string */
  time(&now);
  ptr=localtime(&now);
  tt=asctime(ptr);
  sprintf(tod1,"%s",tt);
}
```

The interface could be located in an include file that would act as a stub for other C-function calls needed by Fortran. We have added it to *timing.c* for simplicity of presentation. The local function, timeofday(), was created to generate a string that we could pass back (almost anything would do here). We thought this might be useful to someone who wishes to capture the date and time in a Fortran program).

The calling Fortran program is shown below:

Code Listing 1.5.1d: testing program, prog.f for bonus example
```
      program main
      external strlen
      integer*4 len1
      character*120 tod
c
c zero out the character string (optional)
      write(tod,'(120(1h0))')
c
c call a function which generates a character string with date and
time
      call timeofday(tod)
c
c call a standard c-library function (interface in timing.c)
      call strlen(tod,len1)
c
c write out the string, len1 is the length of the string returned
from C
      write(*,*) len1 write(*,'(a)') tod(1:len1)
      stop
      end
```

1.5.2 Adding PVM calls to spawn Sibling Tasks

We assume that PVM has been installed and has reference material on PVM, or at a minimum, can view the man pages. Our first step is to create spmd_1.f from our existing serial code. We will take a very minor step forward, and demonstrate how to initiate PVM from inside your Fortran code, and then to spawn identical clones which run on the separate processors. There is no computational advantage in this, since each clone will do all the work, and we are merely concurrently running the same job on other processors. This will be followed by dividing the labor up among the spawned tasks. But first, we will just try to get the same process running concurrently on several nodes.

Tasks vs. Processors It should be noted that all of this can be done on a single desktop machine, you don't need a cluster to try this out. The reason is that a single processor can run multiple tasks. There need not be a 1 to 1 correspondence between tasks and processors. For example, consider the following PVM setup:

```
[pm@penguin testing]$ pvm
pvm> add node02
1 successful
HOST DTID node02 80000
pvm> conf
2 hosts, 1 data format
                  HOST    DTID  ARCH      SPEED   DSIG
                  penguin 40000 LINUXI386 1000    0x00408841
                  node02  80000 LINUXI386 1000    0x00408841
pvm>
```

Here we have two processors (each is a single processor machine, if they were dual processor machines, we would have 4 processors). If we spawn a zero process and one sibling process (master and a slave), then each processor is handling only one task. We could spawn 32 tasks, and each processor would handle 16 tasks simultaneously, or we could specify that all 32 tasks run on a single machine of our own specification. Of course, the execution would be much faster if we have 32 processors available, and spawn one task on each processor (compared to 32 tasks on two processors). None of these decisions will impact our code modifications. With the SPMD paradigm, we write a single code, and then execute it any way we like, using the cluster that happens to be available at the time.

Step 1. Clone the serial code, run concurrently as multiple tasks First we set up two Fortran subroutines which will make the initialization process convenient. It is a good coding practice to write modular code, and we do that here with two subroutines, *startpvm.f* and *mypvmid.f*. The first of these subroutines enrolls the task, and then determines how many other siblings have been enrolled. All three arguments are returned, and they are the task id of the current process, the number of tasks enrolled, and an array containing the task id's. There are two very important points to make:

1. The *tids* array must have a base of 0. That is, the subscript range needs to be forced to start from zero. This is done in the `integer*4 tids(0:1024)` declaration.

2. Never use a dummy dimension in a subroutine for the *tids* array. Even though the main program specifies a zero base, we must also do the same thing in this subroutine. DO NOT use a statement like `integer*4 tids(*)` in this or any other subroutine which uses the *tids* array. Dummy dimensions always assume a base of 1.

The second subroutine, *mypvmid.f,* obtains an alternative ID for the current process. This can be used as a subscript and in if block tests to direct each sibling's activities. One usually chooses zero to be the zero process which acts like a master, and often will assemble the results from the sibling processes. In point of fact, any sibling can be the master, and the master does not need to be on the gateway machine, or the machine from which you are launching pvm. This is facilitated by using NFS mounts of the pvm3 directory tree (a typical setup will actually cross-mount the */home* directories on all nodes). Thus, any process can be chosen as the master, writing output to the NFS mounted directories.

We have located these subroutines in a directory, *sublibf*, and then archive them into a static library, *sublibf.a*. It is a simple matter then to modify the make files to add this subroutine library to the others for building an executable.

Code Listing 1.5.2a: subroutine *startpvm.f*

```
c |---------------------------------------------------------------|
c | startpvm:         enrolls process into pvm, gets task id's |
c | <pm@cgiss.boisestate.edu>    P. Michaels July 2004         |
c |---------------------------------------------------------------|
c
      subroutine startpvm(mytid,ntids,tids)
      include 'fpvm3.h'
      integer*4 tids(0:1024), ntids, mytid, i, idum
c
c enroll in PVM
      call pvmfmytid(mytid)
c
c find number of siblings
      call pvmfsiblings(ntids,-1,idum)
c
      do 10, i=0,ntids-1
      call pvmfsiblings(ntids,i,tids(i))
   10 continue
      end
```

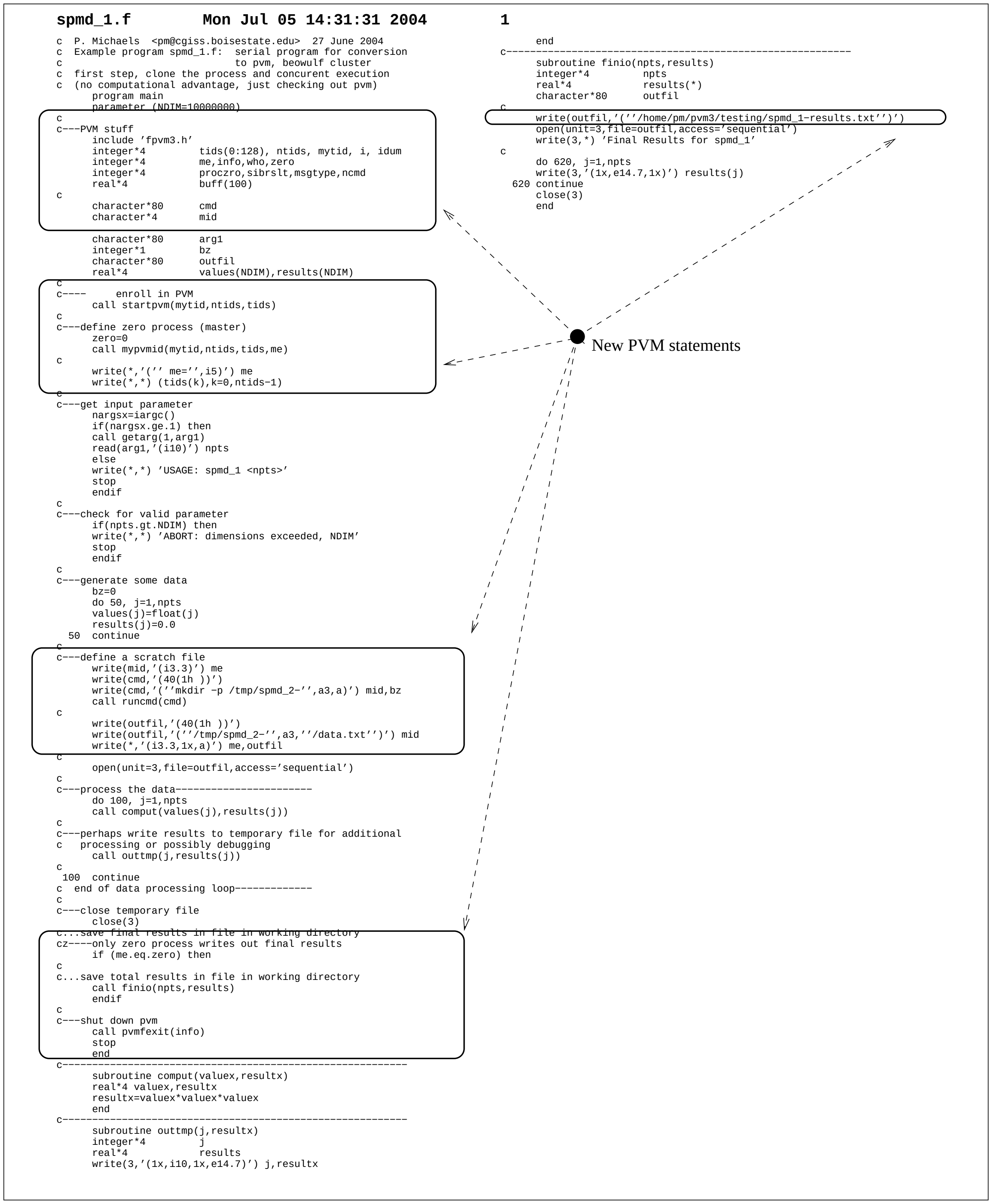
Code Listing 1.5.2b: subroutine *mypvmid.f*

```
c |-------------------------------------------------------------|
c |            mypvmid:  get my task id, defines who I am =me |
c | <pm@cgiss.boisestate.edu>    P. Michaels July 2004        |
c |-------------------------------------------------------------|
c
      subroutine mypvmid(mytid,ntids,tids,me)
      integer*4 tids(0:1024), ntids, mytid, i, me
c
      do 20, i=0,ntids-1
      if(tids(i).eq.mytid) then
      me=i
      go to 30
      endif
   20 continue
   30 continue
      end
```

The modified version of spmd_0.f is show as spmd_1.f in the following listing. The PVM additions have been circled to direct the reader's attention to the changes.

Using Architecture Independent Make (aimk) A convenient wrapper for the make program, aimk, allows one to build executables for different architectures with a single make file, *Makefile.aimk*.

Figure 2: Modified code with PVM calls


New PVM statements

spmd_1.f Mon Jul 05 14:31:31 2004 1

```
c  P. Michaels  <pm@cgiss.boisestate.edu>  27 June 2004
c  Example program spmd_1.f:  serial program for conversion
c                             to pvm, beowulf cluster
c  first step, clone the process and concurent execution
c  (no computational advantage, just checking out pvm)
      program main
      parameter (NDIM=10000000)
c
c---PVM stuff
      include 'fpvm3.h'
      integer*4         tids(0:128), ntids, mytid, i, idum
      integer*4         me,info,who,zero
      integer*4         proczro,sibrslt,msgtype,ncmd
      real*4            buff(100)
c
      character*80      cmd
      character*4       mid

      character*80      arg1
      integer*1         bz
      character*80      outfil
      real*4            values(NDIM),results(NDIM)
c
c----     enroll in PVM
      call startpvm(mytid,ntids,tids)
c
c---define zero process (master)
      zero=0
      call mypvmid(mytid,ntids,tids,me)
c
      write(*,'('' me='',i5)') me
      write(*,*) (tids(k),k=0,ntids-1)
c
c---get input parameter
      nargsx=iargc()
      if(nargsx.ge.1) then
      call getarg(1,arg1)
      read(arg1,'(i10)') npts
      else
      write(*,*) 'USAGE: spmd_1 <npts>'
      stop
      endif
c
c---check for valid parameter
      if(npts.gt.NDIM) then
      write(*,*) 'ABORT: dimensions exceeded, NDIM'
      stop
      endif
c
c---generate some data
      bz=0
      do 50, j=1,npts
      values(j)=float(j)
      results(j)=0.0
  50  continue
c
c---define a scratch file
      write(mid,'(i3.3)') me
      write(cmd,'(40(1h ))')
      write(cmd,'(''mkdir -p /tmp/spmd_2-'',a3,a)') mid,bz
      call runcmd(cmd)
c
      write(outfil,'(40(1h ))')
      write(outfil,'(''/tmp/spmd_2-'',a3,''/data.txt'')') mid
      write(*,'(i3.3,1x,a)') me,outfil
c
      open(unit=3,file=outfil,access='sequential')
c
c---process the data-----------------------
      do 100, j=1,npts
      call comput(values(j),results(j))
c
c---perhaps write results to temporary file for additional
c   processing or possibly debugging
      call outtmp(j,results(j))
c
 100  continue
c  end of data processing loop-------------
c
c---close temporary file
      close(3)
c...save final results in file in working directory
cz----only zero process writes out final results
      if (me.eq.zero) then
c
c...save total results in file in working directory
      call finio(npts,results)
      endif
c
c---shut down pvm
      call pvmfexit(info)
      stop
      end
c-----------------------------------------------------------
      subroutine comput(valuex,resultx)
      real*4 valuex,resultx
      resultx=valuex*valuex*valuex
      end
c-----------------------------------------------------------
      subroutine outtmp(j,resultx)
      integer*4         j
      real*4            results
      write(3,'(1x,i10,1x,e14.7)') j,resultx
      end
c-----------------------------------------------------------
      subroutine finio(npts,results)
      integer*4         npts
      real*4            results(*)
      character*80      outfil
c
      write(outfil,'(''/home/pm/pvm3/testing/spmd_1-results.txt'')')
      open(unit=3,file=outfil,access='sequential')
      write(3,*) 'Final Results for spmd_1'
c
      do 620, j=1,npts
      write(3,'(1x,e14.7,1x)') results(j)
  620 continue
      close(3)
      end
```

Code Listing 1.5.2c: Makefile.aimk

```
# $Id:  Makefile.aimk.1,v 1.1 2004/07/04 19:57:06 pm Exp $
SHELL = /bin/sh
PVMDIR = $(HOME)/pvm3
SDIR = $(PWD)/..
BDIR = $(HOME)/pvm3/bin
XDIR = $(BDIR)/$(PVM_ARCH)
SEQ_XDIR = $(HOME)/bin/
SUBLIBF=../../sublibf/sublibf.a
SUBLIBC=../../sublibc/sublibc.a
MAKEF=../../sublibf
MAKEC=../../sublibc
INCL=../../include
CC = g++
F77 = g77
FFLAGS = -g -I$(PVM_ROOT)/include -I$(INCL)
CFLOPTS = -g -gstabs+ -Wall
CFLAGS = $(CFLOPTS) -I$(PVM_ROOT)/include $(ARCHCFLAGS)
PVMLIB = -lpvm3
FPVMLIB = -lfpvm3
PVMHLIB = -lpvm3
FLIBS = $(SUBLIBC) $(SUBLIBF) $(FPVMLIB) $(ARCHLIB) $(PVMLIB)
LIBS = $(PVMLIB) $(ARCHLIB)
HLIBS = $(PVMHLIB) $(ARCHLIB)
GLIBS = -lgpvm3
LFLAGS = $(LOPT) -L$(PVM_ROOT)/lib/$(PVM_ARCH) -L$(PLPLOT)
CPROGS =
FPROGS = spmd_0$(EXESFX) \
spmd_1$(EXESFX) spmd_2a$(EXESFX) spmd_2$(EXESFX)
default:  spmd_0$(EXESFX) spmd_1$(EXESFX) spmd_2$(EXESFX)
all:  f-all
f-all:  $(FPROGS)
c-all:  $(CPROGS)
hostprogs:  $(HOSTCRPROGS) $(HOSTFPROGS)
clean:
     /bin/rm -f *.o $(CPROGS)
     /bin/rm -f *.o $(FPROGS)
$(XDIR):
     - mkdir $(BDIR)
     - mkdir $(XDIR)
$(SUBLIBF):
     $(MAKE) -C $(MAKEF);
$(SUBLIBC):
     $(MAKE) -C $(MAKEC);
spmd_0$(EXESFX): $(SDIR)/spmd_0.f $(XDIR) $(SUBLIBF) $(SUBLIBC)
     $(F77) $(FFLAGS) -o $@ $(SDIR)/spmd_0.f $(LFLAGS) $(FLIBS)
     mv $@ $(XDIR)
spmd_1$(EXESFX): $(SDIR)/spmd_1.f $(XDIR) $(SUBLIBF) $(SUBLIBC)
     $(F77) $(FFLAGS) -o $@ $(SDIR)/spmd_1.f $(LFLAGS) $(FLIBS)
     mv $@ $(XDIR)
```

Running spmd_1 in PVM The details of running a program under PVM will vary from one cluster to the next. At Boise State University, we use a batch scheduler, PBS. At other sites, one might run PVM directly, either from a console or with the graphical TK interface, *xpvm*. The following examples will be done using a PVM console. The following transcript is for execution on 2 nodes.

```
01  [pm@penguin testing]$ pvm
02  pvm> add node02
03  1 successful
04                HOST      DTID
05                node02    80000
06  pvm> conf
07  2 hosts, 1 data format
08            HOST      DTID      ARCH      SPEED      DSIG
09            penguin   40000     LINUXI386 1000       0x00408841
10            node02    80000     LINUXI386 1000       0x00408841
11  pvm> spawn -2 -> spmd_1 10
12  [1]
13  2 successful
14  t80001
15  t40002
16  pvm> [1:t40002]me= 1
17  [1:t40002] 524289 262146
18  [1:t40002] 001 /tmp/spmd_2-001/data.txt
19  [1:t40002] EOF
20  [1:t80001] me= 0
21  [1:t80001] 524289 262146
22  [1:t80001] 000 /tmp/spmd_2-000/data.txt
23  [1:t80001] EOF [1]
24  finished pvm> halt
25  Terminated
```

On line 06, the PVM command, *conf*, displays the configuration of the cluster, in this case just two nodes, *penguin* and *node02*. The job is started on line 11 with the PVM *spawn* command. With the *spawn* command, we instruct PVM to start two instances (`-2` argument) of the program *spmd_1* and to redirect standard output to the console (`->` argument). The display rapidly responds down to line 16. Then one waits for the spawned processes to complete. As they complete, lines 16 through 24 display to the console. We verify that two instances were spawned (me=0 and me=1 with task id's `524289 262146` respectively). We see that two temporary scratch files have been written (line 18 and line 22).

To confirm that the scratch files were written on the two different computers, we can use the parallel shell (pdsh). A transcript is as follows:

```
[pm@penguin testing]$ pdsh -w penguin,node02
pdsh> date
penguin:  Mon Jul 5 11:37:21 MDT 2004
node02:  Mon Jul 5 11:37:21 MDT 2004
pdsh> cat /tmp/spmd*/data.txt
penguin:  1 0.1000000E+01
penguin:  2 0.8000000E+01
penguin:  3 0.2700000E+02
penguin:  4 0.6400000E+02
penguin:  5 0.1250000E+03
penguin:  6 0.2160000E+03
penguin:  7 0.3430000E+03
penguin:  8 0.5120000E+03
penguin:  9 0.7290000E+03
penguin:  10 0.1000000E+04
node02:  1 0.1000000E+01
```

```
node02:  2 0.8000000E+01
node02:  3 0.2700000E+02
node02:  4 0.6400000E+02
node02:  5 0.1250000E+03
node02:  6 0.2160000E+03
node02:  7 0.3430000E+03
node02:  8 0.5120000E+03
node02:  9 0.7290000E+03
node02:  10 0.1000000E+04
pdsh> quit
[pm@penguin testing]$ cat spmd_1-results.txt
Final Results for spmd_1
0.1000000E+01
0.8000000E+01
0.2700000E+02
0.6400000E+02
0.1250000E+03
0.2160000E+03
0.3430000E+03
0.5120000E+03
0.7290000E+03
0.1000000E+04
```

First we give the Unix *date* command, and this is executed on all the mounted nodes. Then we give the Unix listing command (*cat*), using a wild card to focus on the *spmd_1* output. The scratch files (*data.txt*) are identical, since all we have done is clone the job, running two identical, concurrent instances. Exiting the *pdsh* shell, we check to see if the final output contains the cubes of the first 10 integers, and note that it is correct. This is a good check, since it confirms that we have done what we intended. We are now ready to divide the labor among the spawned tasks, and do some real parallel processing.

Dividing the Labor Among Tasks Up to this point, we have only spawned concurrent executions of a single program on the same data, and gained nothing but confirmation that we can spawn concurrent executions. Now, we modify the main loop which has been identified as being the major opportunity. We could do this in contiguous blocks (task 00 does j=1 to 1000, task 01 does j=1001 to 2000, etc.). However, we have chosen to inter-finger the work using the loop increment. For example, in a 2 task run, this amounts to one task taking the even j's, the other the odd j's.

In a more general case, let *ntids*=number of tasks which have been spawned by the PVM spawn command. In our example program, this value is returned by our subroutine, *startpvm.f*. The do-loop changes from

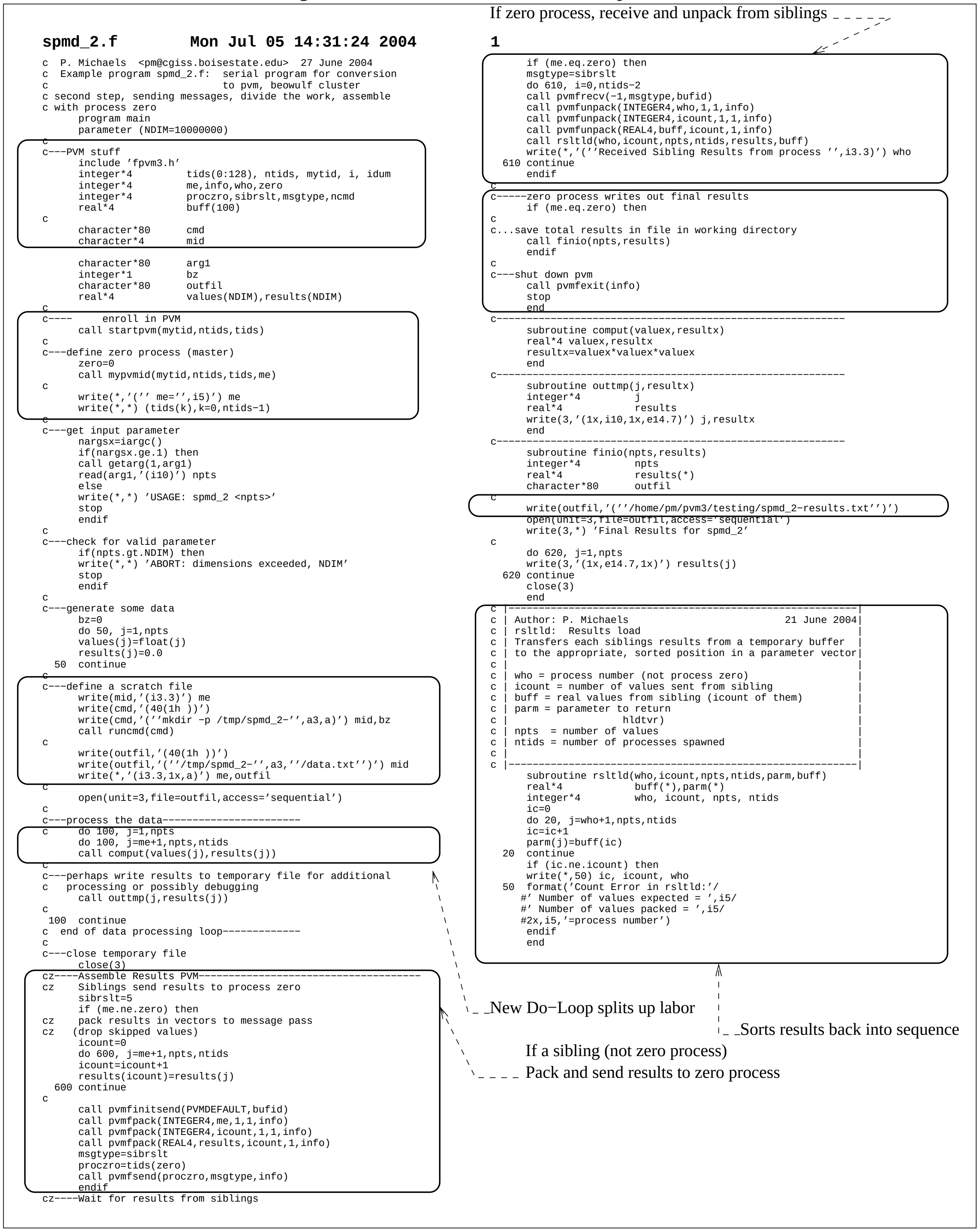
```
do 100, j=1,npts
```

to

```
do 100, j=me+1,npts,ntids
```

Note that both the starting point and the increment have changed. Each task is identified by the integer, *me* (where $0 \leq me \leq (ntids - 1)$). Each spawned task has full access to the command line arguments of the program, as well as any data files that might be read. The only major requirement is that each step of the loop be completely independent of the results from any other step. The modifications have been applied to *spmd_1.f*, and Figure 3 shows the code, *spmd_2.f*

Figure 3: Modified code PVM divides up the labor


If zero process, receive and unpack from siblings
New Do−Loop splits up labor
Sorts results back into sequence
If a sibling (not zero process)
Pack and send results to zero process

spmd_2.f Mon Jul 05 14:31:24 2004 1

```
c  P. Michaels  <pm@cgiss.boisestate.edu>  27 June 2004
c  Example program spmd_2.f:  serial program for conversion
c                              to pvm, beowulf cluster
c second step, sending messages, divide the work, assemble
c with process zero
      program main
      parameter (NDIM=10000000)
c
c---PVM stuff
      include ’fpvm3.h’
      integer*4         tids(0:128), ntids, mytid, i, idum
      integer*4         me,info,who,zero
      integer*4         proczro,sibrslt,msgtype,ncmd
      real*4            buff(100)
c
      character*80      cmd
      character*4       mid

      character*80      arg1
      integer*1         bz
      character*80      outfil
      real*4            values(NDIM),results(NDIM)
c
c----     enroll in PVM
      call startpvm(mytid,ntids,tids)
c
c---define zero process (master)
      zero=0
      call mypvmid(mytid,ntids,tids,me)
c
      write(*,’(’’ me=’’,i5)’) me
      write(*,*) (tids(k),k=0,ntids-1)
c
c---get input parameter
      nargsx=iargc()
      if(nargsx.ge.1) then
      call getarg(1,arg1)
      read(arg1,’(i10)’) npts
      else
      write(*,*) ’USAGE: spmd_2 <npts>’
      stop
      endif
c
c---check for valid parameter
      if(npts.gt.NDIM) then
      write(*,*) ’ABORT: dimensions exceeded, NDIM’
      stop
      endif
c
c---generate some data
      bz=0
      do 50, j=1,npts
      values(j)=float(j)
      results(j)=0.0
   50  continue
c
c---define a scratch file
      write(mid,’(i3.3)’) me
      write(cmd,’(40(1h ))’)
      write(cmd,’(’’mkdir -p /tmp/spmd_2-’’,a3,a)’) mid,bz
      call runcmd(cmd)
c
      write(outfil,’(40(1h ))’)
      write(outfil,’(’’/tmp/spmd_2-’’,a3,’’/data.txt’’)’) mid
      write(*,’(i3.3,1x,a)’) me,outfil
c
      open(unit=3,file=outfil,access=’sequential’)
c
c---process the data-----------------------
c     do 100, j=1,npts
      do 100, j=me+1,npts,ntids
      call comput(values(j),results(j))
c
c---perhaps write results to temporary file for additional
c   processing or possibly debugging
      call outtmp(j,results(j))
c
 100  continue
c  end of data processing loop-------------
c
c---close temporary file
      close(3)
cz----Assemble Results PVM-------------------------------------
cz    Siblings send results to process zero
      sibrslt=5
      if (me.ne.zero) then
cz    pack results in vectors to message pass
cz   (drop skipped values)
      icount=0
      do 600, j=me+1,npts,ntids
      icount=icount+1
      results(icount)=results(j)
  600 continue
c
      call pvmfinitsend(PVMDEFAULT,bufid)
      call pvmfpack(INTEGER4,me,1,1,info)
      call pvmfpack(INTEGER4,icount,1,1,info)
      call pvmfpack(REAL4,results,icount,1,info)
      msgtype=sibrslt
      proczro=tids(zero)
      call pvmfsend(proczro,msgtype,info)
      endif
cz----Wait for results from siblings
```

```
      if (me.eq.zero) then
      msgtype=sibrslt
      do 610, i=0,ntids-2
      call pvmfrecv(-1,msgtype,bufid)
      call pvmfunpack(INTEGER4,who,1,1,info)
      call pvmfunpack(INTEGER4,icount,1,1,info)
      call pvmfunpack(REAL4,buff,icount,1,info)
      call rsltld(who,icount,npts,ntids,results,buff)
      write(*,’(’’Received Sibling Results from process ’’,i3.3)’) who
  610 continue
      endif
c
c-----zero process writes out final results
      if (me.eq.zero) then
c
c...save total results in file in working directory
      call finio(npts,results)
      endif
c
c---shut down pvm
      call pvmfexit(info)
      stop
      end
c-----------------------------------------------------------
      subroutine comput(valuex,resultx)
      real*4 valuex,resultx
      resultx=valuex*valuex*valuex
      end
c-----------------------------------------------------------
      subroutine outtmp(j,resultx)
      integer*4         j
      real*4            results
      write(3,’(1x,i10,1x,e14.7)’) j,resultx
      end
c-----------------------------------------------------------
      subroutine finio(npts,results)
      integer*4         npts
      real*4            results(*)
      character*80      outfil
c
      write(outfil,’(’’/home/pm/pvm3/testing/spmd_2-results.txt’’)’)
      open(unit=3,file=outfil,access=’sequential’)
      write(3,*) ’Final Results for spmd_2’
c
      do 620, j=1,npts
      write(3,’(1x,e14.7,1x)’) results(j)
  620 continue
      close(3)
      end
c |-------------------------------------------------------|
c | Author: P. Michaels                      21 June 2004|
c | rsltld:  Results load                                 |
c | Transfers each siblings results from a temporary buffer |
c | to the appropriate, sorted position in a parameter vector|
c |                                                       |
c | who = process number (not process zero)               |
c | icount = number of values sent from sibling           |
c | buff = real values from sibling (icount of them)      |
c | parm = parameter to return                            |
c |                  hldtvr)                              |
c | npts  = number of values                              |
c | ntids = number of processes spawned                   |
c |                                                       |
c |-------------------------------------------------------|
      subroutine rsltld(who,icount,npts,ntids,parm,buff)
      real*4            buff(*),parm(*)
      integer*4         who, icount, npts, ntids
      ic=0
      do 20, j=who+1,npts,ntids
      ic=ic+1
      parm(j)=buff(ic)
   20 continue
      if (ic.ne.icount) then
      write(*,50) ic, icount, who
   50 format(’Count Error in rsltld:’/
     #’ Number of values expected = ’,i5/
     #’ Number of values packed = ’,i5/
     #2x,i5,’=process number’)
      endif
      end
```

Running spmd_2 in PVM and dividing up the labor A transcript of the PVM run for spmd_2.f is similar to that of spmd_1.f above, except that now we have an additional message indicating that process zero has received some partial results from process 001. Again, to make viewing the results easy, this is a small job (only the first 10 integers are cubed).

```
pvm> spawn -2 -> spmd_2 10
[1]
2 successful
t80001
t40002
pvm> [1:t40002] me= 1
[1:t40002] 524289 262146
[1:t40002] 001 /tmp/spmd_2-001/data.txt
[1:t40002] EOF
[1:t80001] me= 0
[1:t80001] 524289 262146
[1:t80001] 000 /tmp/spmd_2-000/data.txt
[1:t80001] Received Sibling Results from process 001
[1:t80001] EOF [1] finished pvm> halt Terminated
```

Next, we examine the contents of the two scratch files to see if the labor of cubing numbers has been divided up between the two tasks. The parallel shell (*pdsh*) transcript is:

```
pdsh -w penguin,node02
pdsh> cat /tmp/spmd*/data.txt
penguin:  2 0.8000000E+01
penguin:  4 0.6400000E+02
penguin:  6 0.2160000E+03
penguin:  8 0.5120000E+03
penguin:  10 0.1000000E+04
node02:  1 0.1000000E+01
node02:  3 0.2700000E+02
node02:  5 0.1250000E+03
node02:  7 0.3430000E+03
node02:  9 0.7290000E+03
pdsh> quit
[pm@penguin testing]$ cat spmd_2-results.txt
Final Results for spmd_2
0.1000000E+01
0.8000000E+01
0.2700000E+02
0.6400000E+02
0.1250000E+03
0.2160000E+03
0.3430000E+03
0.5120000E+03
0.7290000E+03
0.1000000E+04
```

From the above we can see that penguin processed the even integers, and that node02 processed the odd integers. Then we exit pdsh, and list the contents of the final output file. Process zero has assembled the results and written them in the correct order.

Which node ran process zero? The following transcript answers that question very easily.

```
pdsh -w penguin,node02
pdsh> ls -R /tmp/spmd*
penguin:  /tmp/spmd_2-001:
penguin:  data.txt
node02:  /tmp/spmd_2-000:
node02:  data.txt
pdsh> quit
```

From the above, it is clear that process zero was run on *node02* (even though PVM was initiated and the job spawned from *penguin*). This is evident because we composed the directory names to include the process index (the integer *me* in the program). This can be a helpful trick, since it immediately associates the files with the task that generated them.

With the SPMD paradigm, we can assign any task to be the zero process. The if-statement blocks which guide the processing test against a variable named zero. Examination of *spmd_2.f* reveals that we have set $zero = 0$ in this case. We could have just as easily set $zero = 1$. However, it is probably a good practice to set $zero = 0$, since any other value would require the user to spawn at least that many processes for one to serve as a master task.

1.6 Debug Code

When debugging code in PVM, we recommend running all the tasks on a single node and using the *gdb* program. While it isn't required, running debug sessions on a single node avoids the common problems associated with exporting X-displays. These issues have a solution, but cluster setups vary, and it is just convenient to do it on one machine. For example, one might spawn two tasks (master and one sibling) with the following:

```
[pm@penguin testing]$ pvm
pvm> spawn -2 -?  -penguin -> spmd_2 10
[1]
2 successful
t40002
t40003
pvm> [1:t40002] EOF
[1:t40003] EOF
[1] finished
```

The `"-?"` option triggers the unix *gdb* program. The `-penguin` argument runs both tasks on penguin. A debug program is initiated for each task. One can then engage in an interactive debug session for each task. Usually one sets a breakpoint at main, run, then steps through the program, listing some lines, setting other breakpoints, continuing, printing variable contents, etc.. An example is shown in the transcript taken from the *gdb* xterm window:

Debugging Code: gdb xterm window contents

```
use:  run 10
__________________________________________________
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x80605e6
(gdb) run Starting program:  /home/pm/pvm3/bin/LINUXI386/spmd_2
Breakpoint 1, 0x080605e6 in main ()
(gdb) s
Single stepping until exit from function main,
which has no line number information.
MAIN__ () at /home/pm/devel/beowulf-1.0/Fort/bsegy/LINUXI386/../spmd_2.f:25
warning:  Source file is more recent than executable.
25 call startpvm(mytid,ntids,tids)
Current language:  auto; currently fortran
(gdb) l 20 integer*1 bz
21 character*80 outfil
22 real*4 values(NDIM),results(NDIM)
23 c
24 c---- enroll in PVM
25 call startpvm(mytid,ntids,tids)
26 c
27 c---define zero process (master)
28 zero=0
29 call mypvmid(mytid,ntids,tids,me) (gdb) l
30 c
31 write(*,'(" me=",i5)') me
32 write(*,*) (tids(k),k=0,ntids-1)
33 c
34 c---get input parameter
35 nargsx=iargc()
36 if(nargsx.ge.1) then
37 call getarg(1,arg1)
38 read(arg1,'(i10)') npts
39 else
(gdb) b 35
Breakpoint 2 at 0x8049a34:  file
/home/pm/devel/beowulf-1.0/Fort/bsegy/LINUXI386/../spmd_2.f, line 35.
(gdb) c Continuing.  me= 1
262146 262147
Breakpoint 2, MAIN__ ()
at /home/pm/devel/beowulf-1.0/Fort/bsegy/LINUXI386/../spmd_2.f:35
35 nargsx=iargc()
(gdb) p me $1 = 1
(gdb) p ntids $2 = 2
```

1.7 Adding Pre-processor Code

Once we have a version working in PVM, we can go back and merge both the serial and parallel versions into a single code base. This is done using the C-pre-processor that comes with the GCC compiler. This can be made to work with Fortran77 code, and is implemented with some changes in the make file. The serial version will have a make file option -DSERIAL and the parallel version will have an option -DPARALLEL. Figure 4 shows the merged version. It is in file *spmd_3.fpp* (note the different suffix).

The dashed lines indicate the IF blocks for the pre-processor. Depending on the value of the option specified on the compile line of the file, *Makefile.aimk*, either the serial or parallel version is compiled. Another important point is that the include file must now be the C-language version, not the Fortran version (since it goes through the C-preprocessor).

1.7.1 Makefile.aimk for Pre-processor version

The make file has the following rules added (see Makefile.aimk listed in code listing 1.5.2c)

Code Listing 1.7.1: additions to the Makefile.aimk

```
seq_3$(EXESFX): $(SDIR)/spmd_3.fpp $(XDIR) $(SUBLIBF) $(SUBLIBC)
     $(F77) $(FFLAGS) -DSERIAL -o $@ $(SDIR)/spmd_3.fpp $(LFLAGS)
$(FLIBS)
     mv $@ $(SEQ_XDIR)
spmd_3$(EXESFX): $(SDIR)/spmd_3.fpp $(XDIR) $(SUBLIBF) $(SUBLIBC)
     $(F77) $(FFLAGS) -DPARALLEL -o $@ $(SDIR)/spmd_3.fpp
$(LFLAGS) $(FLIBS)
     mv $@ $(XDIR)
```

One also adds the new program names to the list of executables to build. Notice the definition option (-D<option>) on the F77 compiler instructions. They are either -DSERIAL or -DPARALLEL. This is passed on to the make program and the pre-processor, thus determining what version is compiled. The serial executable is installed in the directory SEQ_XDIR (here, that would be $HOME/bin). The parallel version is installed in XDIR (here, $HOME/pvm3/bin/LINUXI386). If both of these directories are in your PATH environment, then they can be executed from anywhere.

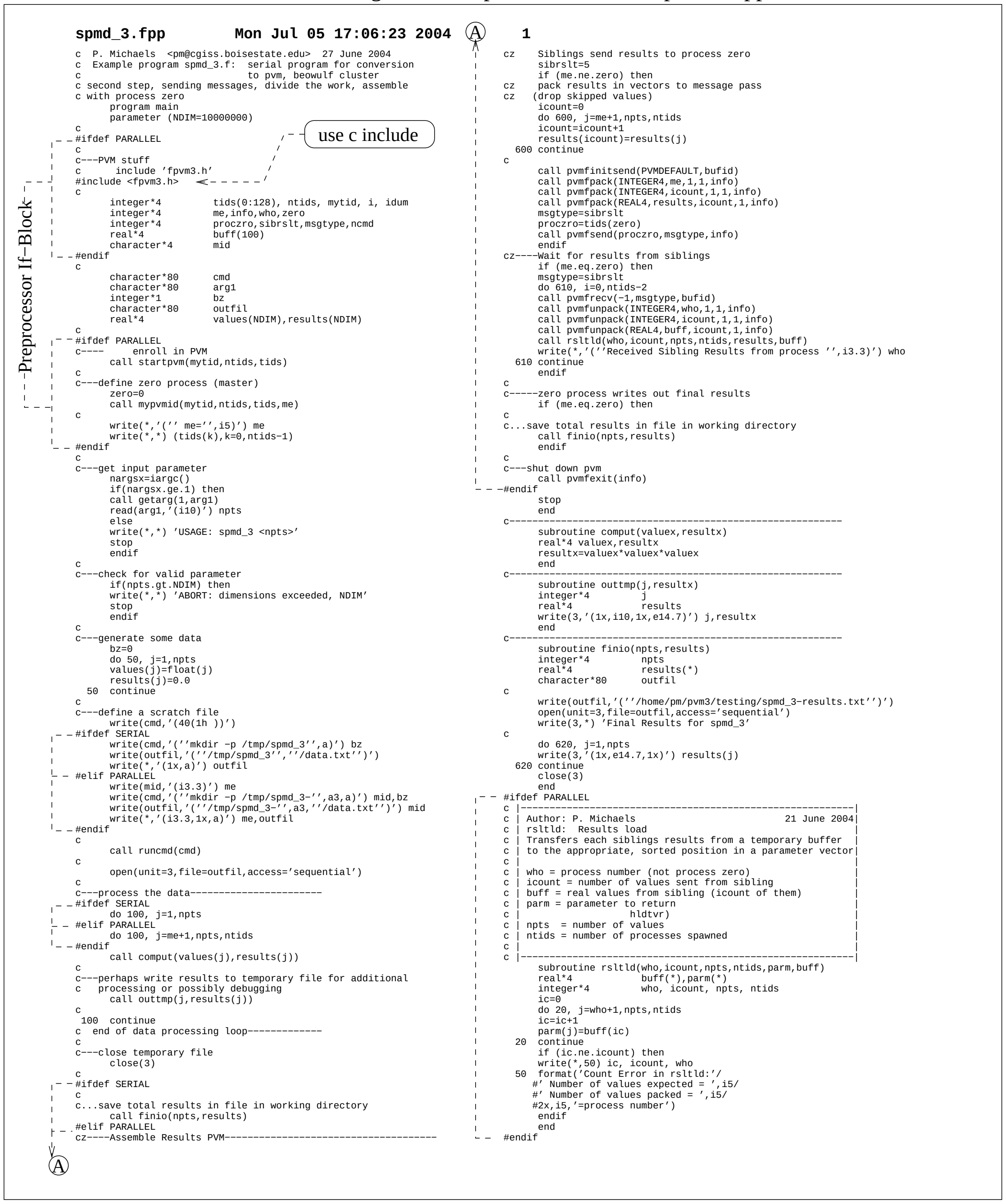
2 Summary

The above example is just a brief representation of what can be done in converting code from serial to parallel. In this specific case, there was no need for the zero process to send any messages to the siblings before computation. This conversion, where possible is about as good as it gets. The reason is that each task spends virtually all its time computing with communications only at the end. This may not be always possible. Figure 5 is a modest expansion of the flow presented above, and includes the possibility of messages being needed before the computation phase.

3 Acknowledgments

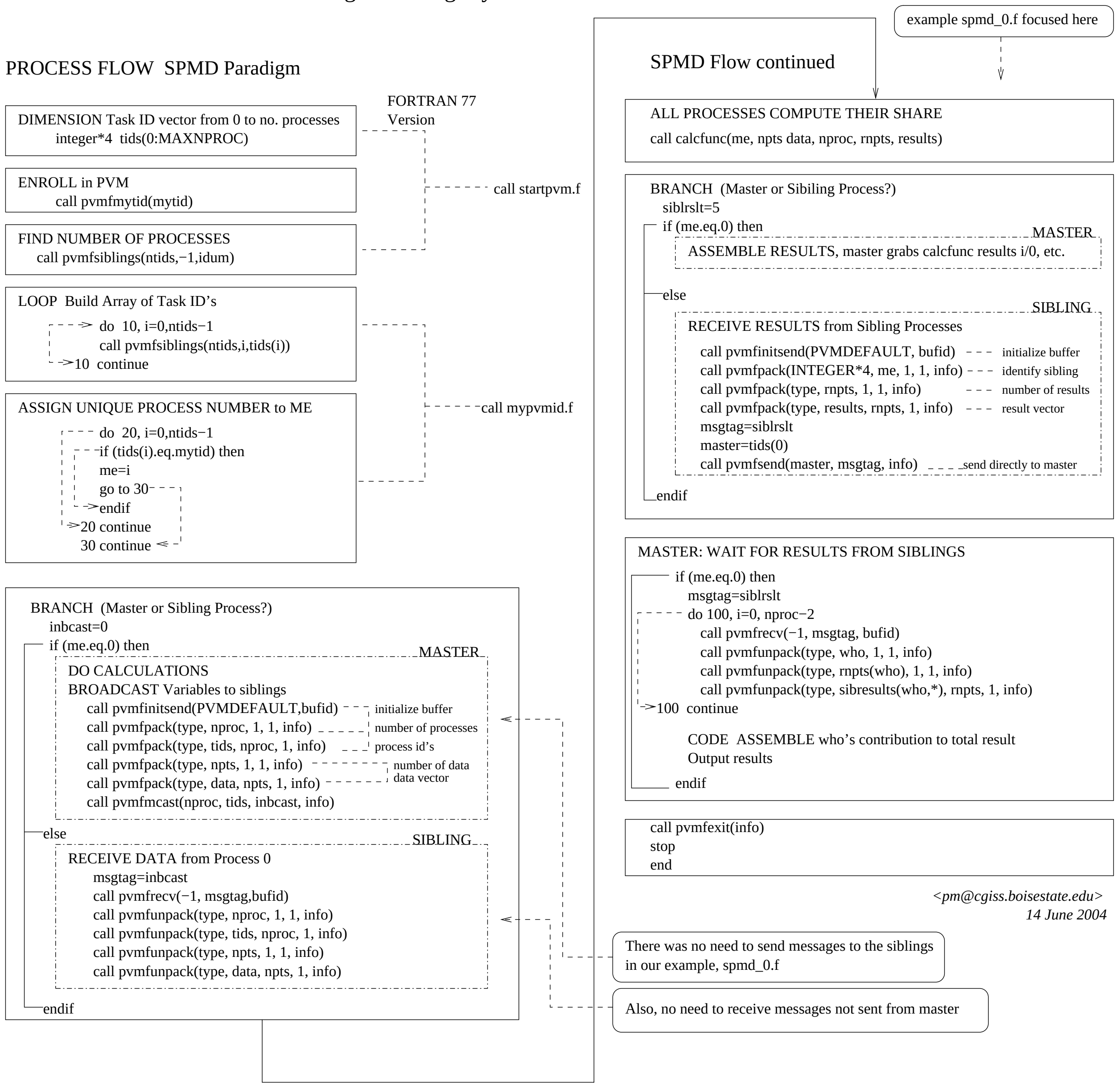
This material is based upon work supported by the National Science Foundation under grant No. 0321233, *"Development of Tools to Enable the Port of Software to a Beowulf Cluster"*. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Figure 4: Pre-processor version spmd_3.fpp


use c include
Preprocessor If–Block
A
A

```
spmd_3.fpp        Mon Jul 05 17:06:23 2004        1
c  P. Michaels  <pm@cgiss.boisestate.edu>  27 June 2004
c  Example program spmd_3.f:  serial program for conversion
c                             to pvm, beowulf cluster
c second step, sending messages, divide the work, assemble
c with process zero
      program main
      parameter (NDIM=10000000)
c
#ifdef PARALLEL
c
c---PVM stuff
c      include 'fpvm3.h'
#include <fpvm3.h>
c
      integer*4         tids(0:128), ntids, mytid, i, idum
      integer*4         me,info,who,zero
      integer*4         proczro,sibrslt,msgtype,ncmd
      real*4            buff(100)
      character*4       mid
#endif
c
      character*80      cmd
      character*80      arg1
      integer*1         bz
      character*80      outfil
      real*4            values(NDIM),results(NDIM)
c
#ifdef PARALLEL
c----     enroll in PVM
      call startpvm(mytid,ntids,tids)
c
c---define zero process (master)
      zero=0
      call mypvmid(mytid,ntids,tids,me)
c
      write(*,'('' me='',i5)') me
      write(*,*) (tids(k),k=0,ntids-1)
#endif
c
c---get input parameter
      nargsx=iargc()
      if(nargsx.ge.1) then
      call getarg(1,arg1)
      read(arg1,'(i10)') npts
      else
      write(*,*) 'USAGE: spmd_3 <npts>'
      stop
      endif
c
c---check for valid parameter
      if(npts.gt.NDIM) then
      write(*,*) 'ABORT: dimensions exceeded, NDIM'
      stop
      endif
c
c---generate some data
      bz=0
      do 50, j=1,npts
      values(j)=float(j)
      results(j)=0.0
  50  continue
c
c---define a scratch file
      write(cmd,'(40(1h ))')
#ifdef SERIAL
      write(cmd,'(''mkdir -p /tmp/spmd_3'',a)') bz
      write(outfil,'(''/tmp/spmd_3'',''/data.txt'')')
      write(*,'(1x,a)') outfil
#elif PARALLEL
      write(mid,'(i3.3)') me
      write(cmd,'(''mkdir -p /tmp/spmd_3-'',a3,a)') mid,bz
      write(outfil,'(''/tmp/spmd_3-'',a3,''/data.txt'')') mid
      write(*,'(i3.3,1x,a)') me,outfil
#endif
c
      call runcmd(cmd)
c
      open(unit=3,file=outfil,access='sequential')
c
c---process the data-----------------------
#ifdef SERIAL
      do 100, j=1,npts
#elif PARALLEL
      do 100, j=me+1,npts,ntids
#endif
      call comput(values(j),results(j))
c
c---perhaps write results to temporary file for additional
c   processing or possibly debugging
      call outtmp(j,results(j))
c
 100  continue
c  end of data processing loop-------------
c
c---close temporary file
      close(3)
c
#ifdef SERIAL
c
c...save total results in file in working directory
      call finio(npts,results)
#elif PARALLEL
cz----Assemble Results PVM-------------------------------------
cz    Siblings send results to process zero
      sibrslt=5
      if (me.ne.zero) then
cz    pack results in vectors to message pass
cz   (drop skipped values)
      icount=0
      do 600, j=me+1,npts,ntids
      icount=icount+1
      results(icount)=results(j)
  600 continue
c
      call pvmfinitsend(PVMDEFAULT,bufid)
      call pvmfpack(INTEGER4,me,1,1,info)
      call pvmfpack(INTEGER4,icount,1,1,info)
      call pvmfpack(REAL4,results,icount,1,info)
      msgtype=sibrslt
      proczro=tids(zero)
      call pvmfsend(proczro,msgtype,info)
      endif
cz----Wait for results from siblings
      if (me.eq.zero) then
      msgtype=sibrslt
      do 610, i=0,ntids-2
      call pvmfrecv(-1,msgtype,bufid)
      call pvmfunpack(INTEGER4,who,1,1,info)
      call pvmfunpack(INTEGER4,icount,1,1,info)
      call pvmfunpack(REAL4,buff,icount,1,info)
      call rsltld(who,icount,npts,ntids,results,buff)
      write(*,'(''Received Sibling Results from process '',i3.3)') who
  610 continue
      endif
c
c-----zero process writes out final results
      if (me.eq.zero) then
c
c...save total results in file in working directory
      call finio(npts,results)
      endif
c
c---shut down pvm
      call pvmfexit(info)
#endif
      stop
      end
c-----------------------------------------------------------
      subroutine comput(valuex,resultx)
      real*4 valuex,resultx
      resultx=valuex*valuex*valuex
      end
c-----------------------------------------------------------
      subroutine outtmp(j,resultx)
      integer*4         j
      real*4            results
      write(3,'(1x,i10,1x,e14.7)') j,resultx
      end
c-----------------------------------------------------------
      subroutine finio(npts,results)
      integer*4         npts
      real*4            results(*)
      character*80      outfil
c
      write(outfil,'(''/home/pm/pvm3/testing/spmd_3-results.txt'')')
      open(unit=3,file=outfil,access='sequential')
      write(3,*) 'Final Results for spmd_3'
c
      do 620, j=1,npts
      write(3,'(1x,e14.7,1x)') results(j)
  620 continue
      close(3)
      end
#ifdef PARALLEL
c |--------------------------------------------------------|
c | Author: P. Michaels                        21 June 2004|
c | rsltld:  Results load                                  |
c | Transfers each siblings results from a temporary buffer |
c | to the appropriate, sorted position in a parameter vector|
c |                                                        |
c | who = process number (not process zero)                |
c | icount = number of values sent from sibling            |
c | buff = real values from sibling (icount of them)       |
c | parm = parameter to return                             |
c |                  hldtvr)                               |
c | npts  = number of values                               |
c | ntids = number of processes spawned                    |
c |                                                        |
c |--------------------------------------------------------|
      subroutine rsltld(who,icount,npts,ntids,parm,buff)
      real*4            buff(*),parm(*)
      integer*4         who, icount, npts, ntids
      ic=0
      do 20, j=who+1,npts,ntids
      ic=ic+1
      parm(j)=buff(ic)
   20  continue
      if (ic.ne.icount) then
      write(*,50) ic, icount, who
   50  format('Count Error in rsltld:'/
     #' Number of values expected = ',i5/
     #' Number of values packed = ',i5/
     #2x,i5,'=process number')
      endif
      end
#endif
```

Figure 5: Slightly More Generalized SPMD Flow



PROCESS FLOW SPMD Paradigm

FORTRAN 77 Version

```
DIMENSION Task ID vector from 0 to no. processes
      integer*4  tids(0:MAXNPROC)

ENROLL in PVM
      call pvmfmytid(mytid)                          call startpvm.f

FIND NUMBER OF PROCESSES
   call pvmfsiblings(ntids,−1,idum)

LOOP  Build Array of Task ID's
   - - ->  do  10, i=0,ntids−1
              call pvmfsiblings(ntids,i,tids(i))
   - ->10  continue

ASSIGN UNIQUE PROCESS NUMBER to ME                   call mypvmid.f
         do  20, i=0,ntids−1
           if (tids(i).eq.mytid) then
             me=i
             go to 30
           endif
      20 continue
      30 continue

BRANCH  (Master or Sibiling Process?)
   inbcast=0
   if (me.eq.0) then                                              MASTER
      DO CALCULATIONS
      BROADCAST Variables to siblings
         call pvmfinitsend(PVMDEFAULT,bufid)       initialize buffer
         call pvmfpack(type, nproc, 1, 1, info)    number of processes
         call pvmfpack(type, tids, nproc, 1, info) process id's
         call pvmfpack(type, npts, 1, 1, info)     number of data
         call pvmfpack(type, data, npts, 1, info)  data vector
         call pvmfmcast(nproc, tids, inbcast, info)
   else                                                           SIBLING
      RECEIVE DATA from Process 0
         msgtag=inbcast
         call pvmfrecv(−1, msgtag,bufid)
         call pvmfunpack(type, nproc, 1, 1, info)
         call pvmfunpack(type, tids, nproc, 1, info)
         call pvmfunpack(type, npts, 1, 1, info)
         call pvmfunpack(type, data, npts, 1, info)
   endif
```

SPMD Flow continued

example spmd_0.f focused here

```
ALL PROCESSES COMPUTE THEIR SHARE
call calcfunc(me, npts data, nproc, rnpts, results)

BRANCH  (Master or Sibiling Process?)
   siblrslt=5
   if (me.eq.0) then                                              MASTER
      ASSEMBLE RESULTS, master grabs calcfunc results i/0, etc.
   else                                                           SIBLING
      RECEIVE RESULTS from Sibling Processes
         call pvmfinitsend(PVMDEFAULT, bufid)          initialize buffer
         call pvmfpack(INTEGER*4, me, 1, 1, info)      identify sibling
         call pvmfpack(type, rnpts, 1, 1, info)        number of results
         call pvmfpack(type, results, rnpts, 1, info)  result vector
         msgtag=siblrslt
         master=tids(0)
         call pvmfsend(master, msgtag, info)           send directly to master
   endif

MASTER: WAIT FOR RESULTS FROM SIBLINGS
   if (me.eq.0) then
      msgtag=siblrslt
      do 100, i=0, nproc−2
         call pvmfrecv(−1, msgtag, bufid)
         call pvmfunpack(type, who, 1, 1, info)
         call pvmfunpack(type, rnpts(who), 1, 1, info)
         call pvmfunpack(type, sibresults(who,*), rnpts, 1, info)
 100  continue

      CODE  ASSEMBLE who's contribution to total result
      Output results
   endif

call pvmfexit(info)
stop
end
```

There was no need to send messages to the siblings in our example, spmd_0.f

Also, no need to receive messages not sent from master

<pm@cgiss.boisestate.edu>
14 June 2004

Index