Introduction to High Performance Parallel Supercomputing
(Preliminary)

Shao-Jing Dong
Computing Center
University of Kentucky
Lexintong, KY 40506

March 9, 2004

Chapter 1

Introduction to IRIX

1. **Introduction to UNIX**
2. **UNIX basics**
3. **UNIX file system**
4. **File editors**
5. **Processes and Shell**
6. **Laboratory Exercises**

1.1 Introduction to UNIX

1.1.1 What is UNIX

UNIX is a trademark of UNIX Systems Laboratories (USL), a subsidiary of AT&T. IRIX is SGI developed UNIX operating system compliant with UNIX System V Release 4 and Open Group's many standards including UNIX 95, Year 2000 and POSIX.

UNIX is a **multi-user, multi-tasking** operating system, as well as, a machine independent operating system, a software development environment.

UNIX is developed at AT&T Bell Labs in 1970 for a software development environment, and was rewritten using c language in 1973. by University of California Berkeley adds major enhancements, creates Berkeley Standard Distribution (BSD) in 1984. Many Berkeley features incorporated into new AT&T version: System V in 1983 to date. Most of the workstation chooses UNIX as the operating system, c as the basic language. Two variations keep popularity today which are AT&T System V and Berkeley Standard Distribution (BSD). IEEE POSIX is developed as a Potable Operating System specification based on UNIX.

AT&T distributes System V for their computers. System V is also the basis for several commercial vendors, such as HP-UX, Apple AUX, IBM AIX, Cray UNICOS, SGI IRIX. BSD also is basis of some commercial vendors too, such as SUN, Apollo, DEC Ultrix. System V and BSD contain a large set of commands in common. Some of these commands support

different options and have different default behaviors and output formats. Each version also has its own unique utilities.

1.1.2 Why UNIX

- UNIX operating system is a REAL share-time multiple processes operating system. That is necessary for the parallel computing. (Windows NT).
- UNIX is hardware independent operating system. The code is written in C language rather than a specific assembly language. Operating system software can be easily moved from one hardware to another. UNIX applications can be easily moved to another UNIX machines.
- Productive environment for software development, rich set of tools, easy command language, programmable.
- UNIX is available at all NSF sponsored supercomputer centers.
- UNIX has excellent stability (run more than 3 months without reset).

1.1.3 Linux — open source software

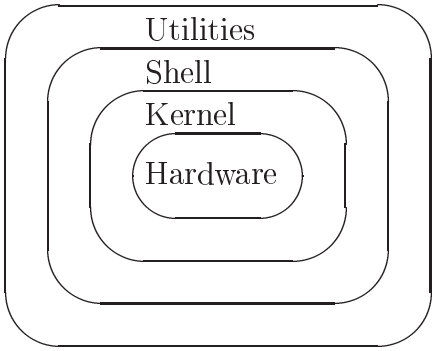
- "Hello every body ... I'm doing a (free) operating system (just a hobby, won't be big and professional ...) "
  Linus Torvaids, creator of Linux, August 25, 1991

- UNIX-like operating system, supported by a global community.
- Lower cost of entry, especially for big cluster
- Good stability. Ran 3 months without restart (IBM East Fishkill, NY)
  Comparing with Windows 2003, logest run without lockup: 5 days.
- 10% better performance than Windows 2003 does.
- Easy administration and management
- Good security.

1.2 UNIX Basics

- Kernel: memory resident control program. Kernel provides service to user applications, device management, process scheduling, etc.

- shell: command interpreter and programming language. Several shells are available, each with its own strengths: Bourne shell (sh), C shell (csh), K shell (ksh), tc shell (tcsh) etc..
- Utilities: several hundreds utility programs provide universal functions: editing, file maintenance, printing, programming support, online information. For example, **ps, grep, more, vi** .....



To use the UNIX commands just needs to type the executable file names. UNIX is case sensitive, remember only type lower case for command name. The **syntax** of commands used to be: **command option(s) argument(s)**. The options is beginning with a dash (-), multiple options can ofter be combined. The arguments are some keywords and filenames. For example,

$$ls$$
$$ls\ -l$$
$$ls\ -la$$
$$ls\ -l\ *.f$$

You can use a lot commands to get information from machine such as:

- man: — man displays on-line manual pages of the utilities or shell. For example,

$$man\ man$$
$$man\ passwd$$
$$man\ -k\ password$$
$$man\ help$$

- who: — who shows who is on the system. For example,

$$who$$
$$who\ am\ i$$

- finger: finger displays information about users, by name or login name. The name could be with remote address. For example,

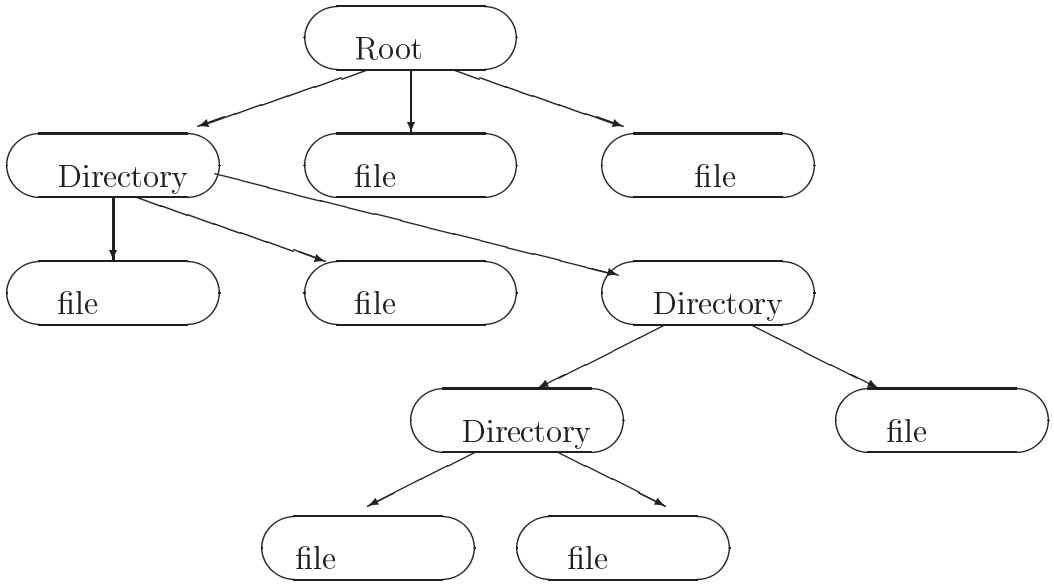
$$finger\ dong$$
$$finger\ super124@ukcc.uky.edu$$

You also need to be familiar with terminal control keys.

- CTRL-u: erase everything you've typed on the command line.
- CTRL-c: stop a command
- CTRL-d: exit from an interactive program
- CTRL-z: suspend a command
- CTRL-s: stop the screen from scrolling
- CTRL-q continue scrolling

1.3 UNIX file system

UNIX files exist in a single shared file system organized into a hierarchy of directories.



UNIX system supports NSF (Networked Shared Files).

- Directory: Used to contain plain files and (or) other directories. Can be used to organize groups of files. Allow parts of the system to be protected from unauthorized users. (Good security).
- root: All directories are descendents of the "root", names "/".
- Working directory: the current default directory for commands and constructing relative pathnames.
- Home directory: The directory associated with your login name, your initial working directory.
- UNIX is sensitive to capital or small characters. You can use up to 256, in general, characters to define your file names.
- Be sure do not use special characters, space, and tabs & ; | ? \ ' " ‘ [ ] < > $ % ! * .
- Extensions: used to identify types of the files. A lot of standard extensions include .c, .f, .f90, .F, .ps, .tex, .jpg, .gif, ...
- Invisible files: It is not listed by "ls", reduce clutter begin with "."
- Pathnames:

1. Absolute Pathnames: refer to files by its descent from the root, always begin with "/". For example, /usr/local/doc/training/sample.f
2. Relative Pathnames: refer to files from the working directory, never begin with "/". For example, training/sample.f

- Special characters can be used to match group of the files ("wild card" characters).

1. ? — matches any single character in a file name. For example, "ls ????" will lists all 4 character filenames.
2. * — matches any number of characters, but not leading periods. For example, "ls *.f *.c " lists all files ending with .f or .c
3. '[ ]' — matches any character in the set withing the brackets. For example, "ls [Mia] " lists files starting with M,i,a

- File Commands:

1. "ls" — lists the contents of a directory. (exp. type "man ls" to see the details of command ls).
2. "cat" — concatenate and display one or more files to the terminal. One can use "$\langle CTRL - s \rangle$" to stop output and use $\langle CTRL - q \rangle$ to quit the stop.
3. "more" — to display and browse long files page by page. One can use "h" for help, spacebar for pageing, q for quitting.
4. "cp" — to copy a file to another file. "cp" will OVERWRITE" existing files, so be careful. (exp. type "man cp" to see the details of command cp)
5. "mv" — to move a file (is equivalent to rename a file). "mv" will OVERWRITE existing files so be careful.
6. "rm" — to remove a file (delete a file). "rm" could not be rescued. For safety, one can use option "-i" as inquire option.

- Directory Commands:

1. Directory identifiers: (directory names)

$\sim$	:	your home directory
$\sim$ *loginname*	:	loginname's home directory
.	:	working directory
..	:	parent directory

2. "pwd" — "Path of Working Directory" displays working directory absolute pathname.
3. "mkdir" — "make a directory" creates a new directory.
4. "cd" — "Change Directory" changes your working directory. item "rmdir" — "remove a directory" (delete a directory). The directory must be empty before you remove it. Or "rm -r" will remove directory recursively.

1.3.1 Access Permissions

UNIX is a multiple user system. You must control who can access your files and how much access rights they have. UNIX system offers good secure way to allow users to control the access.

- Output from "ls -l name": The first character displays the name is file (empty) or directory (d). The next 9 characters displays the permissions of this file (or directory).
- Permissions can be controlled at three levels: User, Group and Others. The number 2,3,4 characters control user's permissions, the number 5,6,7 characters control group's permissions, the number 8,9,10 characters control other's permissions.
- Permissions of a file or directory may be any or all of 3 characters such as:

$$\begin{array}{ll} r & --- \quad \text{read} \\ w & --- \quad \text{write} \\ x & --- \quad \text{execute} \end{array}$$

- A directory must have r and x permissions if permissions are to be granted to files within the directory.
- UNIX allows users to change access permissions for their own files and directories.

$$\begin{array}{l} \text{chmod} \quad [\text{user affected}] \pm [\text{permissions}] \quad \text{file} \\ [\text{user affected}] \text{ are :} \quad \text{u} - \text{user}, \quad \text{g} - \text{group}, \quad \text{o} - \text{others} \\ [\text{permissions}] \text{ are :} \quad \text{r} - \text{read}, \quad \text{w} - \text{write}, \quad \text{x} - \text{execute} \end{array}$$

- Example:

$$\begin{array}{l} \text{ls} \quad -\text{l} \quad \text{sample.f} \\ \text{chmod} \quad \text{o} + \text{rw} \quad \text{sample.f} \\ \text{ls} \quad -\text{l} \quad \text{sample.f} \end{array}$$

1.4 File Editors

1.4.1 UNIX editors

- ed: — the standard line editor.
- ex: — an extended line editor.
- sed: — a stream editor for batch processing of files.

- vi: — a standard visual editor, allows full screen interaction, uses ed/ex line-mode commands for bulk file editing.
- emacs — a full screen editor simulates old vax editor (similar with DOS), is not part of the UNIX package.

1.4.2 The Standard Visual Editor vi

- Editing a file: vi filename
- Exiting vi:

: q – – – quit
: w – – – write edit buffer to disk
: wq – – – write edit buffer to disk and quit
: q! – – – quit without any change

- Positioning:

h – – – left
j – – – downline
k – – – upline
l – – – right
or ← → ↑ ↓
0 – – – first column of current line (zero)
∧ – – – first character of current line
$ – – – last character of current line

- Inserting text:

a – – – append text after cursor
i – – – insert text before cursor

A – – – append text at end of line
I – – – insert text at beginning of line
o – – – open a new line after the cursor line and insert text
O – – – open a new line before the cursor line and insert text
Note : – – –hit ⟨ESC⟩ key when finished inserting

- Inserting file:

: read filename – – – insert contents of filename

- Deleting text:

  x − − − delete character at cursor
  nx − − − delete n characters at cursor
  dd − − − delete line at cursor
  $n$dd − − − delete n lines starting at cursor
  dnw − − − delete n words starting at cursor

- Changing text:

  cw − − − replace word with text
  cc − − − replace line with text
  R − − − overtype existing text until ⟨ESC⟩ is hit
  r − − − overtype only one character
  J − − − join two lines

- Copying lines:

  nyy − − − "yank" copy n lines of text into buffer
  p − − − put contents of buffer after current line

- Moving lines:

  ndd − − − delete n lines (replaced in buffer)
  p − − − put contents of buffer after current line

- Searching / Substituting:

  : /string − − − search forward for string
  :?string − − − search backward for string
  n − − − find next occurrenceofcurrentstring
  : n, ms/old/new − − − from nth line to mth line substitute new for old

- Miscellaneous commands:

  u − − − undo the last command
  . − − − replace the last command
  :!command − − − issuing UNIX command

- vi options

  : set nu − − − show line numbers
  : set showmode − − − show a notice when you are inserting text
  : set ai − − − autoident
  : set wm = 8 − − − wrap lines

  And more commands please see "man vi".

1.5 Process and Shell

Process is a one dimension stream of execution instructions. UNIX is a multitasking operating system. Many different processes exist at the same time. One user may have more than one process running at a time. One user may have more than one simultaneous login session. In general, only multitasking operating system could be used for parallel supercomputing.

1.5.1 Multitasking and shell

1. When you log in, a login process is created by the kernal, which is an operating system process responsible for creating and scheduling processes. The login process verifies your login name and password. It then requests that the kernal create a command line interpreter process that is the shell. The login process then exits.

2. The shell process reads terminal input and creates processes to satisfy user requests. For instance, user enters "ls", the shell reads this from terminal as input, then locates the appropriate executable file. It then asks the kernal to create a new process to run this executable file. At this point user has two processes, a shell process which is sleeping and a "ls" process which is running.

3. The "ls" process lists the directory contents, then exits.

4. The shell process now runs, and gives user a new prompt. The prompt will remain in existence until you log out.

1.5.2 Jobs

The user's running processes are "jobs".

- To run a job:
  User type a executable file name under the shell prompt, it will be submitted to kernal and starts running almost immediately.

- Running a job in the background:
  User type a executable file name with a &, "command &", the shell returns the process ID, and almost immediately returns a new prompt. User can then start a second job. The first job continues to run in the background. But, for some machines, when user logs out, the background job will be terminated by the system.

- Batch jobs: Most of the UNIX system has batch server. Batch server works as an independent operating system, which queues and schedules user's requests whatever the user is login or not. In general, the user's requests is written as a serial set of the shell commands names shell script.

- Managing jobs:

  ps − − − to list processes with process ID
  kill − − − to cancel a running process
  ⟨CTRL − c⟩ − − − to kill a foreground job
  ⟨CTRL − z⟩ − − − to suspend a foreground job
  jobs − − − to list all jobs
  fg − − − to put job in foreground
  bg − − − to put job in background

1.5.3 Standard I/O and Pipes

- Standard input:
  By default, UNIX commands that need input from user expect to receive it from terminal. That is the standard input (type and input on your keyboard).

- Standard output:
  By default, UNIX commands that generate output expect to display their results at the terminal. That is the standard output (show on your screen).

- Redirecting standard I/O:
  User can choose to have input come from, or output go to a file such as:

  cat file1 > file2 − − − redirecting output to file2
  a.out < file − − − redirecting input from file
  tr ISEED 12345 < file1 > file2 − − − input from file1 output to file2

- Pipes:
  A direct connection from the output of one program (command) to the input of another called "pipe". The pipes in UNIX is |. For example,

  who | sort > aa.d

- Filters:
  Filters are some UNIX utilities that read from standard input and write to standard output. So they can be connected in various ways using pipes. Such as:

  grep − − − output lines containing a pattern
  sort − − − sort the lines of a set of lines
  uniq − − − take input stream and output only one copy of any adjacent lines that are identical
  wc − − − return the number of lines(−l), words(−w) and characters(−c) in a file

1.5.4 C Shell

- What is a Shell?
  Shell is a programming language (script), as well as, a command line interpreter. Shell provides user commands and shell variables. Shell variables could be pre-defined by the system also could be defined by user. C shell is a BSD standard, however, most people use the C Shell for interactive work since the commands and language is based on C language.

- We also have the other shells. Each kind of shell has its owe features. Pear language, in fact, is a shell too.

1.5.5 Some C Shell Commands

- history:
  lists the previous commands you have executed. The number of the history, in general, is set in a shell "run control" file names .cshrc. User also can set the number of the history again.

- To repeat commands:

  !! − − − repeats last command

  !*number* − − − repeats numbered command, the number can be find by history

  !*string* − − − repeat last command starting with *string*

- To correct commands:

  ∧*old* ∧ *new* − − − changelastcommand

  !*number* : *s*/*old*/*new*/ − − − changeindicatedcommand

- exercise:
  please type following command and see what happened.

  set history=22

  ls

  history

  pwd

  !!

  men makdir

  !!:s/men/man/

  la -l

  ∧a∧s

  !ls

- Alias:
  Alias allows user to rename a command or modify the default behavior of a command. The usage is:

  alias *string* command – – – create an alias
  unalias *string* removeanalias
  alias – – – lists your aliases

- Exercise:
  please type following command and see what happened.

  alias rm 'rm -i'

  alias

  alias ll 'ls -al \!*| more'

  ll

  unalias ll

- C Shell Variables:
  C shell allows user to set character or numerical variables. The variables are available from current shell only. The shell variables make the c shell much more like a programming language.

- Commands to define the c shell variables:

  set var = *value* – – – assign character *value* to var
  set – – – display all shell variables
  echo $var – – – display value of var
  unset var – – – removes var
  @ imax = 15 – – – assign 15 to numerical variable imax
  @ $imax = $imax – 1 – – – subtract 1 from numerical variable imax

- Pre-defined shell variables:
  When you loged on the machine, type "set" you can see all the pre-defined shell variables and their values.

- Environment Variables:
  A variable from current shell and from all programs started from the shell is the environment variable.

- Commands:

  setenv – – – display all exist environment variables
  setenv var *value* – – – assign *value* to var
  echo $var – – – displays value of var
  unsetenv var – – – remove var

- some environment variables:

  CWD: current working directory

  HOME: user home directory

  USER: userid

  MANPATH: where to look for man pages

  LIBPATH: where to look for libraries

  DISPLAY: where the X server to open a window.

- Your .login and .cshrc files:
  The c shell used to use 2 files to set up user's preferences.

  - .login
    It runs at invocation of login shell. It typically sets terminal characteristics, command search path, and one time shell options.
  - .cshrc
    It runs at each c shell invocation, when user log on, starting a new shell, or opening a new window. It typically sets alias information and some user's preferred environment.
  - user can use "vi" to change .login or .cshrc. After changing, user can type
    source .login
    source .cshrc
    to make the new environment work. Or user can logout and login again.

- A sample of .cshrc file:

```
#
# Default user .cshrc file (/usr/bin/csh initialization).
# Usage: Copy this file to a user's home directory and edit it to
# customize it to taste. It is run by csh each time it starts up.
# Set up default command search path:
#
# (For security, this default is a minimal set.)
set path=(/usr/local/bin /usr/contrib/bin
/usr/etc /usr/local/lib:$path .)
# Set up C shell environment:
if ( $?prompt ) then        # shell is interactive.
set history=20          # previous commands to remember.
set savehist=20          # number to save across sessions.
set system=`hostname`       # name of this system.
set prompt = "$system \!: "          # command prompt.
```

```
# Sample alias:
alias h history alias ri 'rm -i' alias lo 'logout'
# More sample aliases, commented out by default:
endif
xhost + ncx1.uky.edu
xhost + server1.pa.uky.edu
xset +fp /home/dong/xfonts        # add some chinese fonts
```

1.5.6 C Shell Script

That means a C Shell program runs on the machine shell level. Similar program language also as pear. The script allows user to create a set of shell commands and variables to run frequently used set of commands and complete some performance designed by user. So user can submit this script to batch queue and let the machine to complete the job as what the user whats.

- The c shell script must begin with "#!/bin/csh". As well as, it must be made executable.
- C shell allows user to define variables and variable arrays:

  set var = value
  set var = (elem1 elem2 elem2)
  $#var − − − number of elements in var
  $var[n] − − − value of nth element in var

- Using quotes around values user can include space in the value. However, single quotes prevent variable substitution, double quotes allow variable substitution.
- C shell variable can read standard input from terminal by $ <. C shell variable can store the output of a command also.
- Exercise:
  Please use "vi" to create a shell script names "test1.cs" as: #!/bin/csh set opt=-l

```
set x1='ls $opt'
set x2="ls $opt"
echo With single quotes:
echo $x1
echo With double quotes:
echo $x2
echo Enter input
set inputis=$ <
```

```
echo $inputis
set date=‘date‘
echo $date
echo $date[1]
echo $date[2]
echo $date[3]
```

- Numerical variables and evaluations with @ :
- C-like control structures:

$$[\text{if} - \text{then} - \text{else} - \text{endif}] \; - - - \; \text{branch choice}$$
$$[\text{foreach} - \text{end}] \; - - - \; \text{iterative loop}$$
$$[\text{while} - \text{end}] \; - - - \; \text{conditional loop}$$
$$[\text{switch} - \text{case} - \text{endsw}] \; - - - \; \text{case selection}$$
$$[\text{sleep n}] \; - - - \; \text{wait for n seconds}$$

- A sample of c shell script for running a program latqcd.f90 on batch:

```
#!/bin/csh
set name= DONG
set LAT= 32x48_62_gauge
@ imax= 5
@ i= 1
set D= d
set R= .rpt
set TMP='/scratch/super124'
mkdir $TMP
WORK='/usr/super124/MPI/pure_gauge'
echo $name $WORK $WORKW u1$LAT
cd $WORK
if( ! $WORK/xqcd ) then
f90 latqcd.f90 -lmpi -o xqcd
endif
while( $i <= $imax )
@ NUM = $i
mpirun -np 64 $WORK/xqcd < $WORK/in_gauge.d > $WORK/out.d
mv out.d out_$NUM$D
tar -cf $TMP/u$NUM$LAT $WORK/fort.*
@ i= $i + 1
end
exit 0
```

1.6 Laboratory Exercises: Introduction to UNIX

1. Logon to the supercomputer.

2. Try the following commands:
   man passwd

   man -k users

   whatis who

   whatis finger

   whatis finger $\langle CTRL - u \rangle$. Here $\langle CTRL - u \rangle$ means press the $CTRL$ and u keys ar the same time.

   man -k users ¿ aa.d

   cat aa.d

3. Getting information:
   change your password

   determine who is logged on the machine

   display information on all users on the machine.

4. Try following directory commands:
   whatis mkdir

   whatis cd

   whatis pwd

   whatis ls

   mkdir myshell

   cd myshell

   cp ../aa.d aa.d

   pwd

   cd

   cd /usr/bin

   ls -l

   ls -l l*

5. The "vi" editor

   cd myshell

   vi aa.d

Try to move the last line to the beginning of the file.

go to end of the file (G *or* $:\$$)
delete line (dd)
go to beginning of the file ($1G$ *or* $:1$)
put line (p)

Add a blank line after the first line:

insert an line below current line (o)
end insert mode (ESC)

Remove the 7th line then put it back:

move to 7th line ($:7$ *or* $7j$)
delete this line (dd)
put it back (p)

Insert the phrase "we seem like the c-shell" after 10th line 4th word.

move to the position ($j,\ k,\ l,\ h$)
enter insert mode (i)
type the phrase
end insert mode

Ypu can type "u" to "undo" it.

6. Process commands:
   ps to see what jobs do you have.
   sleep 300 & to run a sleeping job in background.
   ps check the process ID of sleep in background
   $\langle CTRL - z \rangle$ to suspend sleep
   jobs to check your jobs
   kill -9 PID to kill the background job sleep.
   ps to make sure you have killed the background job sleep.

7. standard I/O:
   man csh > c_shell_reference
   cat c_shell_reference | more
   cat c_shell_reference | grep shell > bb.d
   more bb.d

8. .login and .cshrc files "vi" your .cshrc file, add "alias ll 'ls -l' " as the last line.
   type "source .cshrc"
   try "ll" to see how the .cshrc control your alias.
   type echo $DISPLAY on your local terminal to see what is your local DISPLAY
   use setenv to set the DISPLAY to your local terminal
   type xterm & to see if the new xterm opened in your terminal or not.

9. The C shell script
   vi test1.cs:

```
#!/bin/csh
set opt=-l
set x1='ls $opt'
set x2="ls $opt"
echo with single quotes:
echo $x1
echo with double quotes:
echo $x2
echo Enter input
set inputis=$<
echo 'What I input is' $inputis
set date=`date`
echo $date
echo $date[1]
echo $date[2]
echo $date[3]
echo 'This c shell script is called' $0
exit 0
```

   chmod u+x test1.cs
   test1.cs and input hello— to see how the c shell script works as a program.

10. A simple computing control c shell script
    vi aa.d

```
&indata
xkappa=0.KP
ran_seed=ISEED
ox=X
oy=Y
```

```
infile=input_N
&end
```

vi test2.cs

```
#!/bin/csh
set KP = 15
set X = 3
set Y = 3
@ IS= 12345
@ imax=5
@ i= 1
while ( $i <= $imax )
@ ISEED= $i * $IS
@ N= $i
tr ISEED $ISEED < aa.d > bb.d
tr X $X < bb.d > cc.d
tr Y $Y < cc.d > bb.d
tr KP $KP < bb.d > cc.d
tr N $N < cc.d > namelist
rm bb.d cc.d
echo 'This is your namelist file as input for your program' $i
cat namelist
@ i= $i + 1
end
exit 0
```

chmod u+x test2.cs

test2.cs — to see how the shell changes your namelist input files 5 times for your 5 times computing.

11. To get your c shell reference
man csh > reference

more reference

Chapter 2

Languages independence and openMP directives

1. **Programming languages**
2. **Parallelization and Data Independence**
3. **Shared Memory Parallelization and openMP**
4. **OpenMP — Application Program Interface Standard Directives**
5. **Shared Memory Examples**
6. **Laboratory exercises**

2.1 Programming Languages

- Assembly:

  The language is the most important bridge between users and the machine. Programming in the early days of computing was extremely tedious and hard. Programmers must know details of the instructions, registers and other aspects of the exact CPU of the computer for which they were writing the codes. The source code itself was written in a numerical notation, so-called *octal code* and can be used on only the exact machine. When time was going on, a readable codes were introduced, named as *machine language* or *assembly language*. This language is depends on the exact CPU, (not portable!). However, it did enable the make the exact CPU be used in a very efficient way. So far, it is still being offered by the industry.

- Fortran:

  In 1950, a team of IBM developed one of the earliest **high-level** language Fortran, the Formula Translation Language. This language is simple to understand, independent on

the exact machine and almost as efficient in execution as the assembly language. The small loss in efficient is programmers must use **compiler** to make the codes executable.

Fortran was a revolutionary development. Programmers were liberated from the tedious working of the assembly language, and were able to concentrate on algorithms, optimizations and exact problem solvers. More important thing, however, was that the computer became accessible to any scientist or engineer willing to use it solve the scientific or engineering problems. The computer were no longer belongs to computer experts only. Fortran spread rapidly as it fulfilled the needs. The application programs were developed very fast. Then a problem appeared, how make the program exchangable, or portable.

In 1960, American Standard Association (later the American National Standard Institute, ANSI) brought out the first ever standard for a programming language, now known as Fortran 66.

In 1978, the Fortran 77 published as a new standard to solve some large-scale problems. It included several new features that were based on vendor extensions or pre-processors. By the mid-1980s, the changeover to Fortran 77 was in full swing. The pre-processors make the Fortran compiler optimizing the original code to fit the exact machine architecture, as well as, to reduce big number of the work which have been done before it repeats. That makes the code runs more and more efficiently. During the time, Fortran 77 involved large math labs and some parallel directives, openMP lab, MPI lab, PVM lab,.....

In the course of time many new languages nad been developed, such as c C++, pascal, Algo68, HPF etc.. They were demonstrably more suitable for a particular type of application. They had been adopted in preference to Fortran for that purpose. Fortran is now in the area of numerical, scientific, engineering, and technical applications. As well, ANSI-accredited technical committee once again prepared new standard, formerly known as Fortran 8x, and now as Fortran 90.

In 1990, first Fortran 90 book issued. In 1991 mid-summer, first Fortran 90 compiler is available. Fortran 90 is different from Fortran 77. It included more features of the other languages. However, as a standard, Fortran 90 involved Fortran 77 as a full subset.

1. Unformed lines. Since we do not need the punched card as a input today, we do not need the 72 columns per line neither. So the line form will be much more like in c language.
2. Introduce array manipulations that would reduce the programming time. Suppose, it would raise running time performance. However, for most of the RISC based supercomputer, it goes to opposite way.
3. Modules — encapsulate data nad subprograms.
4. KIND — standardizes the specification of numerical precision.
5. Overloaded — new assignment statements and operators.

6. Include — statements.
7. Do While loops.
8. Namelist I/O
9. Pointers — which provide data structures that can grow and shrink.
10. dynamical arrays.
11. interface — one interface multiple function.
12. And more.

In November 1995, new standard of Fortran 95 were finalized. Fortran 95 took most of the HPF functions in. It kept Fortran 90 as full subset.

In 1999, the Fortran 2000 Forum issued a standard of Fortran 2000 which involved almost all C++ features and visual functions. It still keeps Fortran 77, Fortran 90 and Fortran 95 as full subset.

- c C++:
  C is a general purpose programming language. Many of the important ideas of C from the language BCPL. The influence of BCPL on C proceeded indirectly through the language B which was written in 1970 for the first UNIX system PDP-7. BCPL and B are "typeless" language. In 1973, C became a new standard language by providing a variety of data types.

  C is a relatively *low level* language. That means C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with arithmetic and logic operators implemented by real computers.

  C++ is the object oriented version of the C.

  Since the dynamical arrays have too much freedom to optimize the performance, the performance of numerical benchmark shows that c and C++ is slower than Fortran in numerical computing area.

- SGI Origin 2000 offers Fortran 90/77 compilers as *f90* and *f77* , ANSI C and ANSI C++ compilers as *cc* and *CC* .

2.2 Parallelization and Data Independence

2.2.1 Why parallel computing?

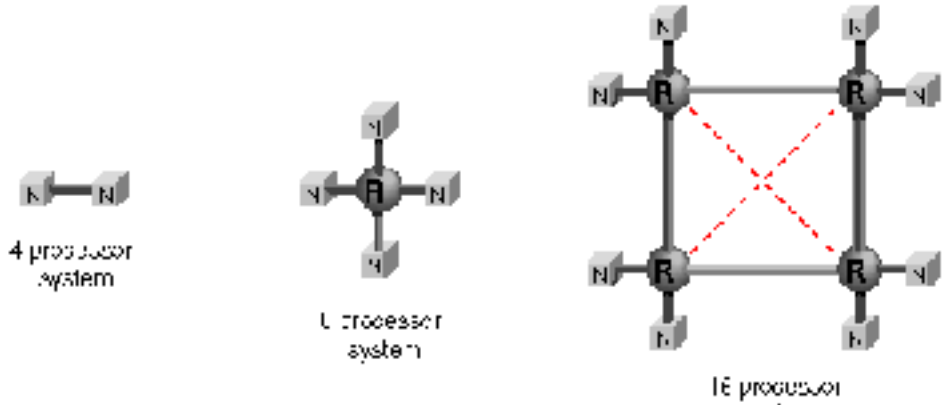
- Parallel computing is to use multiple processors to execute parts of the same program simultaneously.
  GOAL: To Reduce Wall-Clock Time
  The parallel computing can reduce wall-clock time only. In general, it increases the CPU time. Then why we need parallel computing?

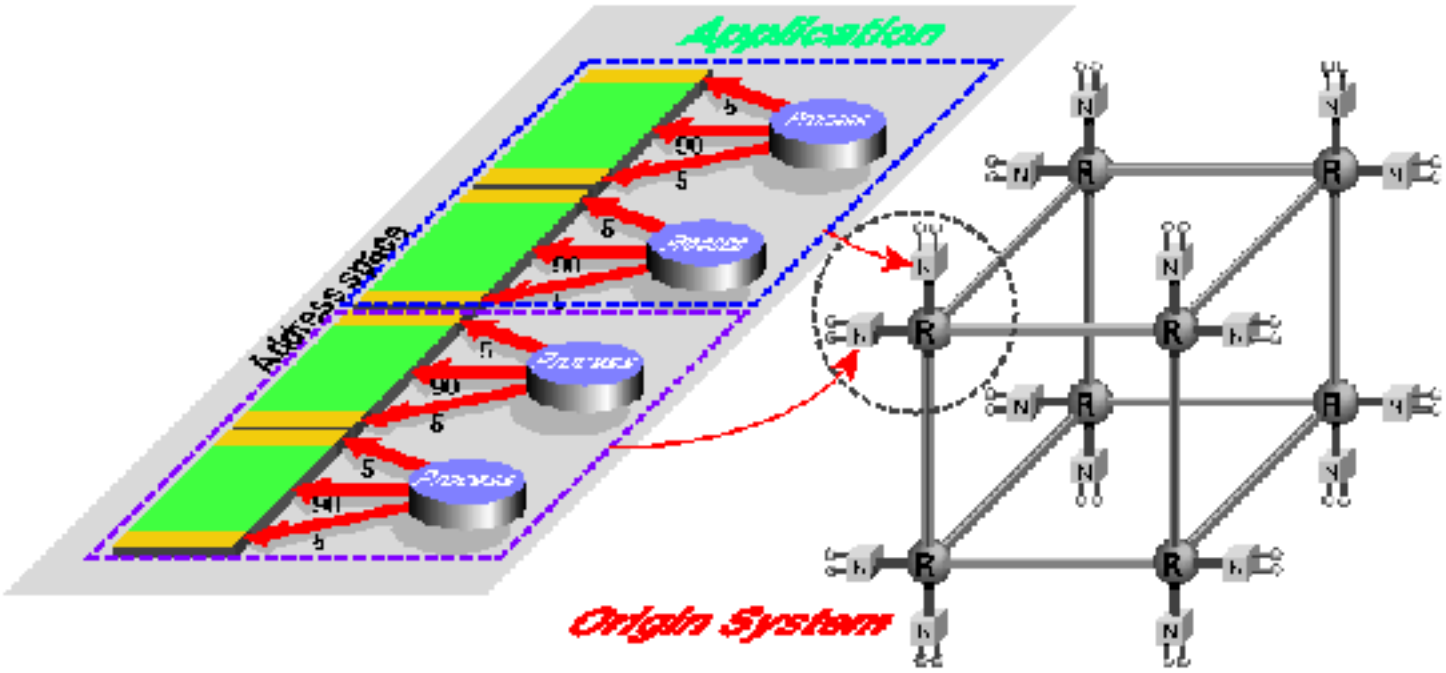
1. The real scientific and engineering computing needs more and more performance. For example, the Lattice QCD with chiral fermions needs average 30Gflops*year performance, many CFD projects ask 1Gflops*years performance, some special material projects use 2Gflops*years performance. And the nuclear physics projects ask 100Gflops*years to 1Teraflople*years performance. The requests are still increasing year by year. So the science and engeneering need multiple teraflop machines. And, in fact, we have began to use such machines.
2. There are physics limits to the speed of a single processor, which almost being reached. That is the speed of the light:

$$\begin{aligned} \text{speed of light c} &= 3.0 \times 10^{10} \text{ cm/sec} \\ \text{wave length for 1GHz} &= 30 \text{ cm} \\ \text{wave length for 1GHz in Copper} &= 9.0 \text{ cm} \end{aligned}$$

   To keep the signal synchronization in a designed processor, the speed of one processor looks could not higher than 100GHz. The quantum computer is still far from us.
3. It is increasingly expensive to make a single processor faster. However, a fairly fast processor is inexpensive.
4. The size of the projects also increase very fast. So far the Lattice QCD uses 2GB memory for one job. Some molecular problem needs 1.5 to 3 GB memory. Good compiler also makes the executable to use more memory, like the IBM f90/f95 compiler. Since the memory hierarchy to use large memory with one processor is very inefficient.

- The SGI Origin 2000 architecture:





2.2.2 Parallel Task, Parallelism, Amdhal's Law

- Parallel task means logically discrete section of computational process which is independent of any other concurrent task in the program. So that all such tasks can be performed simultaneously.
- Not all computational problems can be parallelized.

1. π calculation:

$$\begin{aligned}\frac{\pi}{4} &= \int_0^1 dx\, \frac{1}{x^2+1} \\ &= \int_0^{1/m} dx\, \frac{1}{x^2+1} + \int_{1/m}^{2/m} dx\, \frac{1}{x^2+1} + \int_{2/m}^{3/m} dx\, \frac{1}{x^2+1} + \cdots \int_{(m-1)/m}^{1} dx\, \frac{1}{x^2+1}\end{aligned}$$

   Each integral is independent of the others. You always can set them as parallel tasks.

2. Markov chain:

$$p(n+1) = TP[(n+1) \leftarrow n] \cdot p(n)$$

3. Some program can be totally parallelized with very small overlap. Some program can only partially parallelized. Most of the programs need users to find their parallelism that means to find where and how you can parallelize them.

- A logical theorem of parallel speedup — Amdahl's Law:
  The parallel speedup means the wall-clock time speedup.

$$S_u = \frac{\text{Wall} - \text{clock time running on 1 processor}}{\text{Wall} - \text{clock time running on N processors}}$$

If the computing performance speed would not change when the program size changed, then logically we could write

$$S_u = \frac{1}{1 - f + f/N}$$

were

$$\begin{aligned} N &= \text{Number of the processors} \\ f &= \text{fraction of the code's execution time in parallel} \\ &= \frac{\text{CPU time for parallel execution}}{\text{CPU time for whole execution}} \end{aligned}$$

So that f gives a upper bound of parallel speedup:

$$S_u < \frac{1}{1 - f}$$

For example: $f = 90\%$ (0.9)

$$\begin{aligned} 4 \text{ processors}: \quad S_u &= 1/(1 - 0.9 + 0.9/4) = 3.08 \\ 8 \text{ processors}: \quad S_u &= 1/(1 - 0.9 + 0.9/4) = 4.70 \\ 16 \text{ processors}: \quad S_u &= 1/(1 - 0.9 + 0.9/16) = 6.4 \\ 32 \text{ processors}: \quad S_u &= 1/(1 - 0.9 + 0.9/32) = 7.8 \\ 64 \text{ processors}: \quad S_u &= 1/(1 - 0.9 + 0.9/64) = 8.77 \\ 128 \text{ processors}: \quad S_u &= 1/(1 - 0.9 + 0.9/128) = 9.34 \end{aligned}$$

Because of the memory hierarchy and cache based architecture, the Amdahl's law needs a lot corrections. But the upper bound always is a good reference parameter. It points that the researcher **MUST** find very high parallel faction in the research problem, otherwise he could not turn to the supercomputing. Or one can say he could not do modern computational research.

2.2.3 Data Dependence in Loop

A true dependence results from a store to a memory location followed by a fetch from that location. But only the "loop-carried" dependence will affect the loop level parallelization.

- Example 1: (Backward Loop Carried Dependence B-LCD)

```
DO J = 2, N
A(J) = A(J-1) * 2.0
ENDDO
```

Here, A(J) depends on a value of A(J-1) which computed in the previous iteration. That is the "loop carried" dependence. So that you can not set each iteration of this loop as the parallel task. Otherwise you will get wrong results.

- Example 2: (Forward Loop Carried Dependence F-LCD)

```
DO J=1,N
A(J) = A(J+1) *2.0
ENDDO
```

Here, A(J) depends on a value of A(J+1) which assigned before the loop starts running and will be computed in next iteration. This is also the "loop carried" dependence. You can not set each iteration of this loop as the parallel task. However, if you make an extra copy of array A before the loop starts running, such as:

```
DO J=1,N
B(J) = A(J)
ENDDO
DO J=1,N
A(J) = B(J+1) *2.0
ENDDO
```

Then you get the second loop clearly no loop carried data dependence. You can now parallelize it by adding some "overhead".

- Example 3: (Output Loop Carried Dependence O-LCD)

```
DO J=1,N
A(L(J)) = B(J) + C(J)
ENDDO
```

Here, we do not know the index L(J) contains repeated values or not (i.e. L(3)=L(12) = 7), then 2 different iterations COULD assign a value to the same address. If you set the iterations as parallel tasks, then the values output by the loop into the array A depends on the order in which the iterations are executed. In this case the user MUST very carefully to analyze his program to determine whether the dependence exits or

not. User also can fix this dependence by making an extra output copy of A.

```
DO J=1,N
A_extra(J) = B(J) + C(J)
ENDDO
DO J=1,N
A(L(J))=A_extra(J)
ENDDO
```

However, the second loop could not be parallelize again.

- Example 4: (Apparent Loop Carried Dependence A-LCD)

```
DO J=1,N
A(J) = A(L(J)) + C(J)
ENDDO
```

Here, we do not know the index L(J) is before, after or repeated. The dependence is apparent loop carried dependence. In general, the programmer could not take such a risk to parallelize it except you know the program very clear and firmly know there is no dependence.

2.3 Shared Memory Parallelization and openMP

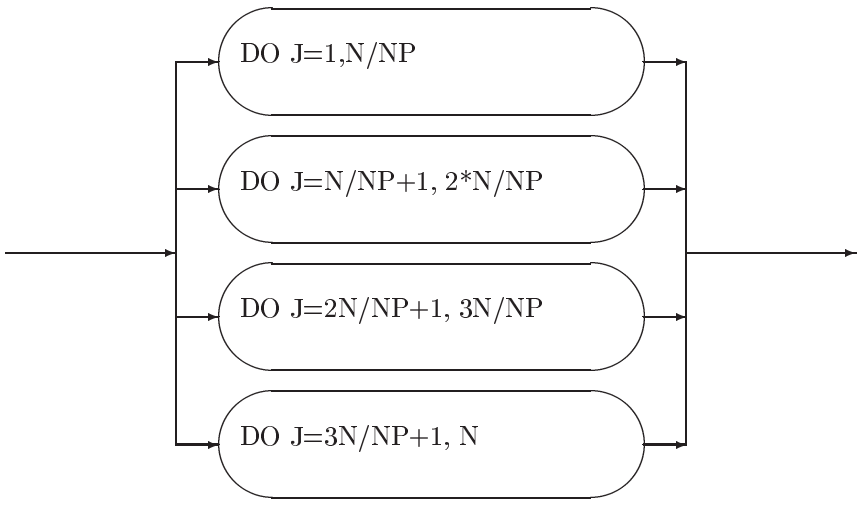
A loop which has not any kind loop carried data dependence can be parallelized by adding some special directives in to your program and let the compiler re-compile it and link the special library. There are a lot of the different directives libraries offered by different vendors. The vendor always says his directives are the best one. That makes parallel programming be very painful. In 1997, KUCK & Associates, Inc. developed a high level directive library which independent of exact machines and works for Shared Multiple Processor (SMP) programming model. So far, it has been accepted by all the major computer industries (IBM, SGI, HP, Compaq, SUN) and became a new "portable" standard. For the portability reason, we just use the openMP directives here.

Let us follow the examples of the loops to see how to make your program be parallelized:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! File kind_module
MODULE kind_spec_module
implicit none
!
! floating point single and double precision
!
```

```
INTEGER, PARAMETER :: high = selected_real_kind(15, 307)
INTEGER, PARAMETER :: low = selected_real_kind(6, 37)
INTEGER, PARAMETER :: short= selected_int_kind(4)
!
END MODULE kind_spec_module
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE SUB_MODULE
USE kind_spec_module
CONTAINS
SUBROUTINE SUB(A,B,C,N)
integer(low), intent(inout) :: N
real(high), dimension(1:N) intent(inout) :: A,B,C
!
!$OMP PARALLEL DO PRIVATE(J), SHARED(A,B,C,N)
DO J=1,N
   A(J) = B(J) + C(J)
ENDDO
!$OMP END PARALLEL DO
!
END SUBROUTINE SUB
!
END MODULE SUB_MODULE
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

After compiler, and setenv OMP_NUM_THREADS 4, what the machine will do? Let $NP = 4$, following figure shows how the openMP makes loop be parallelized.



Some exercises of loop carried dependence

- 1 :

```
DO I = 1, N
J = J + 1
A(I) = B(J)
ENDDO
```

- 2:

```
DO I = 1, N
IF(A(I).LT. 0.0) THEN
J = J + 1
A(I) = B(J)
ENDIF
ENDDO
```

- 3:

```
DO WHILE(A(I) .LT. Z)
I = I + 1
A(I) = B(I)
ENDDO
```

- 4:

```
DO WHILE(I. LT. N)
I = I + 1
A(I) = B(I)
ENDDO
```

- 5:

```
DO I = 1, N
S= C(I) * B(I)
IF( S.GT. 0.0d00) THEN
T = S + 5.0d00
ELSE
T = S + 4.0d00
ENDIF
A(I) = A(I) + T
ENDDO
```

- 6:

```
DO I = 1, N
A(J(I)) = A(K(I)) + 1.0
ENDDO
```

2.4 openMP — Application Program Interface Standard Directives

- Using openMP directives in proper place of your program:

  !$OMP *directive* [clauses] Fortran90

  C$OMP *directive* [clauses] Fortran77

  #pragma omp *directive* [clauses] C or C++

  where *clauses* mean one or more directive clauses. Clauses can appear in any order after the directive name and can be repeated as needed, subject to the restrictions listed in the description of each clause.

- To compile source code with openMP directives

  f90 -mp source.f90

  f77 -mp source.f

  cc -mp source.c

  CC -mp source.C

  The SGI Fortran and c compilers have been set a default *-MP open_mp=on*, and *-mp* used to make the compiler to honor tne *-MP* options.

- Setting the environment to make the parallel code run on multiple processors.

  OMP_NUM_THREADS : to set the number of the threads then your job running on your account could open such number of the processes.

  set OMP_NUM_THREADS 4

  There are also the other openMP environment variables, *OMP_SCHEDULE, OMP_DYNAMIC, OMP_NESTED* . You can get the detail from Fortran 90 manual.

- To run the executable and timming, user just need to type:

  time a.out

  where *time* is a shell command which will show you the wall-clock time and CPU time.

2.4.1 OpenMP Architecture

- openMP involves directives, runtime library and environment variables.
- directives:

1. Parallel Region: *PARALLEL* and *END PARALLEL*

   !$ OMP PARALLEL [clause[[,] clause]...]
   block of code
   !$ OMP END PARALLEL

   clauses:
   - PRIVATE(var[, var]...)
   - SHARED(var[, var]...)
   - DEFAULT(PRIVATE | SHARED | NONE)
   - FIRSTPRIVATE(var[, var]...)
   - REDUCTION({operator | intrinsic}:var[, var]...)
   - IF(scalar_logical_expression)
   - COPYIN(var[, var]...)

   The END PARALLEL is an implied *barrier* here. When a thread encounters a parallel region, it asked shell to create a team of threads, and it becomes the master of the team with a thread number of 0. At the END PARALLEL barrier, the computing stream waiting for all the threads to complete their work. After all the threads finished their work, only the master thread of the team continues execution pass the end of parallel region.

   The number of threads it can create is depends on the environment variable.

2. Work-sharing Constructs 1: *!$ OMP DO* and *!$ OMP END DO*

   !$ OMP DO [clause[[,] clause]...]
   do_loop
   [!$ OMP END DO [NOWAIT]]

   clauses:
   - PRIVATE(var,[, var]...)
   - FIRSTPRIVATE(var[, var]...)
   - LASTPRIVATE(var[, var]...)
   - REDUCTION({operator | intrinsic}: var[, var]...)
   - SCHEDULE(type[, chunk])

- ORDERED

The OMP DO directive specifies that the iterations of the immediately following do_loop must be divided among the threads in the parallel region. If there is not enclosing parallel region, the OMP DO will not work. The loop following OMP DO can not be a do while or a Do loop without loop control.

3. Work-sharing Constructs 2 : *!$ OMP SECTIONS* and *!$ OMP END SECTION*

!$ OMP SECTIONS [clause[[,] clause]...]
[!$OMP SECTION]
block of code
[!$ OMP SECTION]
block of code
...
!$ OMP END SECTIONS [NOWAIT]

clauses:

- PRIVATE(var[, var]...)
- FIRSTPRIVATE(var[, var]...)
- LASTPRIVATE(var[, var]...)
- REDUCTION({operator | intrinsic}: var[, var]...)

The OMP SECTION directive specifies that the enclosed sections of code are to be divided among threads in the team. It is a noniterative work-sharing construct. Each section is executed once by a thread in the team.

The OMP END SECTION is a barrier here. Threads that complete execution of their sections wait at this point until all the threads finished their execution. If a NOWAIT is specified, the waiting could be canceled.

4. Work-sharing Constructs 3 : *OMP SINGLE* and *OMP END SINGLE*

!$OMP SINGLE [clause[[,] clause]...]
block
!$OMP END SINGLE [NOWAIT]

clauses:

- PRIVATE(var[, var]...)
- FIRSTPRIVATE(var[, var]...)

The OMP SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team. The OMP END SINGLE is a barrier. Threads in the team which are not executing wait at this point unless NOWAIT is specified.

5. Combined parallel work-sharing constructs:

- !$OMP PARALLEL DO and !$ OMP END PARALLEL DO
- !$OMP PARALLEL SECTIONS, !$ OMP SECTION, ... !$ OMP END PARALLEL SECTIONS

6. Synchronization Constructs:

   - !$OMP BARRIER: to synchronize all the threads in a team. When any thread encounters a barrier, it waits until all other threads in the team have reached the same point.

   - !$OMP MASTER and !$ OMP END MASTER: The code enclosed is executed by master thread only. The other threads in the team skip the enclosed code and continue next execution. There is not barrier.

   - !$OMP CRITICAL and !$ OMP END CRITICAL: These directives restrict access to the enclosed code to one thread at a time. A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section. The critical section could has a name.

   - !$OMP ATOMIC: The OMP ATOMIC ensures that a specific memory location is updated atomically.

   - !$OMP FLUSH: The OMP FLUSH identifies synchronization points at which thread-visible variables are written back to memory.

   - !$OMP ORDERED and !$ OMP END ORDERED: The code enclosed within these directives is executed in the orde in which it would be executed in a sequential execution.

7. Data Environment Constructs — *!$OMP THREADPRIVATE(/cb/[,/cb/]...)*

   The OMP THREADPRIVATE makes named common blocks private to a thread. Each thread executing a THREADPRIVATE directive receives its own private copy os named common block, which is available to it in any routines within the scope of a application.

8. Data Scope Constructs: exactly they are attribute clauses:

   - PRIVATE(var[, var]...): The private clause declares variables to be private to each thread in a team.

   - SHARED(var[, var]...): The shared clause makes variables shared among all threads in the team. All threads access the same storage area for the shared

data.

- DEFAULT(PRIVATE | SHARED | NONE): The default clause allows user to specify a PRIVATE, SHARED or NONE default scope attribute for all variables in a lexical extent of any parallel region.

- FIRSTPRIVATE(var[, var]...): The firstprivate clause provides a superset of the functionality provided by the PRIVATE clause.

- LASTPRIVATE(var[, var]...): When the lastprivate clause appears on a DO or SECTIONS, the thread that executes the sequentially last iteration (or last section) updates the version of the objects it had before the structs.

- REDUCTION({operatro | intrinsic}: var[, var]...) : This clause performs a reduction on the variables specified with the operator (+, -, *, ...), or intrinsic function (MAX, MIN, IAND, IOR...).

- COPYIN(var[, var]...): The COPYIN only used for THREADPRIVATE common blocks. A copyin clause on a parallel region specifies that the data in the master thread of the team be copied to the threadprivate copies of the common block at the beginning of the parallel region.

- OpenMP environment variables:

  - OMP_NUM_THREADS: To set the number of the threads which you can open.
    setenv OMP_NUM_THREADS 4
  - OMP_SCHEDULE: To set the schedule type.
    setenv OMP_SCHEDULE "dynamic"
  - OMP_DYNAMIC: To set the throughput mode in dynamic.
    setenv OMP_DYNAMIC TRUE
  - OMP_NESTED: To set the nested parallelism.
    setenv OMP_NESTED TRUE

- Runtime openMP functions:

  - OMP_SET_NUM_THREADS()
    OMP_GET_NUM_THREADS()
    OMP_GET_MAX_THREADS()
    OMP_GET_THREAD_NUM()

OMP_GET_NUM_PROCS()
OMP_INPARALLEL()

- OMP_SET_DYNAMIC()
  OMP_GET_DYNAMIC()
- OMP_SET_NESTED()
  OMP_GET_NESTED()

- debug: User can use -g option in compiling time to allow the debug. User follows the fortran 90 manual can do it at all.
- Performance analysis: Most of the machine offer performance analyzer. On the Origin 2000 the performance analyzer is "perfex". User can read the manual to get the details.

2.5 Examples

1. **To fell the parallel programming:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!File kind_module
MODULE kind_spec_module
implicit none
!
! floating point single and double precision
!
INTEGER, Parameter::high = selected_real_kind(15,307)
INTEGER, Parameter::low = selected_real_kind(6,37)
INTEGER, Parameter::short= selected_int_kind(4)
!
END MODULE kind_spec_module
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Program fell
Use kind_spec_module
integer(short) :: I, myid, Num
integer(short), Parameter:: N=16
integer(short), dimension(1:N):: ID
!
!$OMP PARALLEL DO Private(I,myid,Num), SHARED(ID)
DO I = 1, N
myid = OMP_GET_THREAD_NUM()
```

```
ID(I) = myid
ENDDO
!$ OMP END PARALLEL DO
write(6,*) ( I, ID(I), I=1,N)
!
END PROGRAM feel
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

This program will show you which iterations are performed by the thread with thread ID myid.

2. **π calculation: loop parallelization and REDUCTION clause:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! File kind_module
MODULE kind_spec_module
!
INTEGER, PARAMETER:: high = selected_real_kind(15, 307)
INTEGER, PARAMETER:: low = selected_real_kind(6,37)
INTEGER, PARAMETER:: short= selected_int_kind(4)
INTEGER, PARAMETER:: long = selected_int_kind(12)
!
END MODULE kind_spec_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
PROGRAM pi
USE kind_spec_module !
! Use 1000000 as the integral steps. You can change it to see
! what kind accuracy of pi you can get
!
INTEGER(long), PARAMETER:: N=1000000
INTEGER(long) :: I
real(high):: x, w, sum, pi
real(high), function:: f(x) = 4.0_high / ( 1.0_high + x*x)
!
w = 1.0_high / N
sum = 0.0_high
!$OMP PARALLEL DO private(x,I), SHARED(N)
!$OMP& REDUCTION(+:sum)
DO I = 1, N
```

```
x = w * ( I - 0.5_high )
sum = sum + f(x)
ENDDO
!$OMP END PARALLEL DO
pi = w * sum
write(6,*) pi, N
end program pi
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

This program calculates the integral

$$\pi = 4.0 \cdot \int_0^1 dx \, \frac{1}{1+x^2}$$

Changing N use can get different accuracy of π.

3. **π calculation 2: — independent routines parallelization**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! File kind_module
MODULE kind_spec_module
implicit none
!
INTEGER, parameter:: high = selected_real_kind(15,307)
INTEGER, parameter:: low = selected_real_kind(6,37)
INTEGER, parameter:: short= selected_int_kind(4)
INTEGER, PARAMETER:: long = selected_int_kind(12)
!
END MODULE kind_spec_module
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
MODULE SUB
use kind_spec_module
CONTAINS
function f(x)
real(high) :: f, x
!
f = 4.0_high/(1.0_high + x*x)
```

```
end function f
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
SUBROUTINE SUB_INTEGRAL(N0, NF, w, sum)
real(high), intent(inout):: w, sum, x
real(high), function :: f(x) = 4.0_high / ( 1.0_high + x*x )
INTEGER(long), intent(inout):: N0, NF, I
!
DO I = N0, NF
x = w*( I - 0.5_high )
sum = sum + f(x)
ENDDO
!
END SUBROUTINE SUB_INTEGRAL
!
END MODULE SUB
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
PROGRAM PI
USE kind_spec_module
USE SUB
INTEGER(long), PARAMETER:: N = 1000000
INTEGER :: myid, IN
INTEGER(long) :: N0, NF
real(high) :: w, sum, pi
w = 1.0_high/N
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(N,w),LASTPRIVATE(IN) &
!$OMP & REDUCTION(+:sum)
IN = OMP_GET_THREADS_NUM()
myid = OMP_GET_NUM_THREAD()
N0 = myid * N/IN + 1
NF = N0 + N/IN
CALL SUB_integral(N0, NF, w, sum)
!$OMP END PARALLEL
pi = w*sum
print*, pi, N, IN
END
```

This program shows that the PARALLEL region directive can be used for exploiting *coarse-grained* parallelism. It is important in the real problem solution.

4. **Matrix multiplication:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! File kind_spec_module
MODULE kind_spec_module
implicit none
!
INTEGER, PARAMETER:: high = selected_real_kind(15, 307)
INTEGER, PARAMETER:: low = selected_real_kind(6,37)
INTEGER, PARAMETER:: short= selected_int_kind(4)
!
END MODULE kind_spec_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
PROGRAM mm
USE kind_spec_module
INTEGER(short), PARAMETER :: NP=4
INTEGER(short), PARAMETER:: maxrows=1000, maxcols = 1000, maxc=maxcols/NP
real(high), dimension(1:maxrows, 1:maxcols):: a
real(high), dimension(1:maxrows, 1:maxc, 1:NP):: b,c
INTEGER(short) :: n, l, i, j, k
!
!$OMP PARALLEL SHARED(a, b, c)
!$OMP DO PRIVATE(i,j,l,n)
DO n=1, NP
l=(n-1)*maxc
DO j=1,maxc
DO i=1,maxcols
a(i,j+l) = i + 0.023_high*j
ENDDO
ENDDO
ENDDO
!$OMP END DO NOWAIT
!
!$OMP DO PRIVATE(i,j,n)
DO n=1, NP
DO j=1,maxc
DO i=1, maxrows
c(i,j,n) = 0.0_high
```

```
b(i,j,n)=0.11_high*i + 1.19_high*(j+(n-1)*maxc)
ENDDO
ENDDO
ENDDO
!$OMP BARRIER
!
!Parallel multiplication
!
!$OMP DO PRIVATE(i,j,k,n)
DO n=1,NP
DO j=1,maxc
DO k=1,maxcols
DO i=1,maxrows
c(i,j,n)=c(i,j,n)+a(i,k)*b(k,j,n)
ENDDO
ENDDO
ENDDO
ENDDO
!$OMP END PARALLEL
write(6,*) c(10,20,1), c(20,30,1), c(40,50,1)
END Program mm
```

This program multiplies 2 matrices. It shows the NOWAIT clause, and OMP DO. You'd better do not use OMP_NESTED environment.

2.6 Lab exercises

1. Using "vi" to write the first example as feel.f

   ```
   f90 -O2 -mp feel.f
   setenv OMP_NUM_THREADS 4
   a.out
   ```

   To get the feeling how the machine parallelly executes your job.

2. Using "vi" to write the loop parallel π calculation program as pi_1.f.

   ```
   f90 -O2 -mp pi_1.f
   setenv OMP_NUM_THREADS 4
   time a.out
   ```

   To see how about the parallel speed-up. Then increase the integral steps into 10000000 to see the change of accuracy and speed-up.

3. Using "vi" to write the 4th example as mm.f

   f90 -O2 -mp feel.f

   setenv OMP_NUM_THREADS 4

   time a.out

   setenv OMP_NUM_THREADS 1

   time a.out

   To see how the parallelization speeds up the performance.

Chapter 3

Distributed Memory Parallelization and MPI

1. **Parallel Programming modes and parallelism levels.**
2. **Message Passing and the Message Passing Interface Standard (MPI).**
3. **Distributed Memory Parallel Examples**
4. **Laboratory exercises**

3.1 Parallel Programming Modes and message passing

There are many different architectures of the supercomputers, some supercomputers are just "clusters", that means they are one of many almost independent machines connected together, even PC clusters. According supercomputing 2000, the cluster becomes a major machine architectures now. These machines have distributed memory, user can not use data which stored on a machine from the other machines directly.

3.1.1 The theoretical parallel programming modes

Let us first to see what kind parallel programming modes we can have. Since the program is built by the instructions and data, we can assume:

- **SISD mode:**

  Single Instruction with Single data. This mode is used in most serial programming and shared memory parallel programming. User can determine the results always correct.

- **MISD mode:**

  Multiple Instruction with Single Data. This is the typical shared memory mode. In previous chapter we parallelized π calculation by using this mode. In this mode user

needs to take care of data independence, instruction synchronization. However, since all the data are used the same address table, the program almost can be interpreted from a serial program. So it is relatively simple to program.

- **SIMD mode:**

  Single Instruction with Multiple Data. This is a typical distributed memory mode. Each machine (or node) runs the same instruction, with different data. So that the user must separate his exact problem into several small problem and write a program to solve each small problem on each node. That is so-called data decomposition. In runtime when one node needs the data located on the other nodes, user needs to consider the get (send) the data by "message passing" software. This mode is relatively complicated since the data decomposition and message passing. However, since each node is used to solve a small similar problem, the program is similar to the serial program. Since each node runs the same instruction, the synchronization is relatively simple.

- **MIMD mode:**

  Multiple Instruction with Multiple Data. This is also typical distributed memory mode. User needs to take care of the synchronization of the instruction, as well as, take care of the message passing. This is more complicated mode.

The data decomposition will give the users a big benefits which are come from the memory hierarchy and threads spawning-joining procedure.

3.1.2 Fine-Grain, Coarse-Grain parallelism and parallelism levels

We can observe different parallelism levels in a program. We must chose what kind parallelization we want to use.

- **instruction level parallelism:**

  In general, if your program has independence in less than 50 instruction level, we call it instruction level parallelism. It is not used for running on multiple processors, that will really wast your computer resource. But it is used to feed the special (RISC or VLIW) CPU architecture. We will discuss it later (chapter 5).

- **loop level parallelism:**

  In the previous chapter we discussed about the loop level parallelism which is very easy to find in a special program. However, in the exercises we also see that, the speed up is depends on the size of the loop. For example, if the matrix to matrix multiplication only for 20×20 matrix, you will get low speed up and wast some computer resource.

In most of the cases different iterations of the same loop use the same data set. So it is very nice to be considered as a SISD or MISD mode.

Most of the Automatic Parallelizing Option (APO) are working on the loop level parallelism. For most of the small loops which exist in a real program APO will increase the wall clock time and can not guarantee the results. That is why currently I do not encourage users to use the APO even in the small example programs it works well.

- **routine level parallelism:**

  In previous chapter we also discussed a routine level parallel example — a π calculation program. Whatever in the C C++ or Fortran language, the routine always is a independent block of the whole program both in instruction and data. So that the routine level parallelism is easy to be a SIMD mode. Or if you declared the **SHARED(...)**, it also could be a SISD mode.

- **multiple routines level parallelism:**

  We can also find multiple routines parallelism in many applications. That is a good SIMD parallelization mode, or if you declared the **SHARED(...)**, it also could be a SISD mode.

- **program level parallelism:**

  If your scientific-engineering problem could be separated into multiple small pieces and then calculate them to get whole results, then your program will have a program level parallelism. For example, most of the physics interactions are nearest neighbor interactions, so most of the motion equations (Partial Differential Equations) can be separated into small grids with the neighbor's boundary together. You can solve the equations on each computer node need only small grid data set with neighbor's boundary data set together. So that your whole program has the parallelism and data independence. The program level parallelism is a good SIMD mode, even it is almost a SPMD (Single Program Multiple Data) mode. However, the neighbor's boundary exchanging needs message passing. Each node still correlated with the physics neighbors.

  This mode is widely be used to solve real scientific and engineering problems. In next chapter we will discuss this mode with an example of partial differential equations.

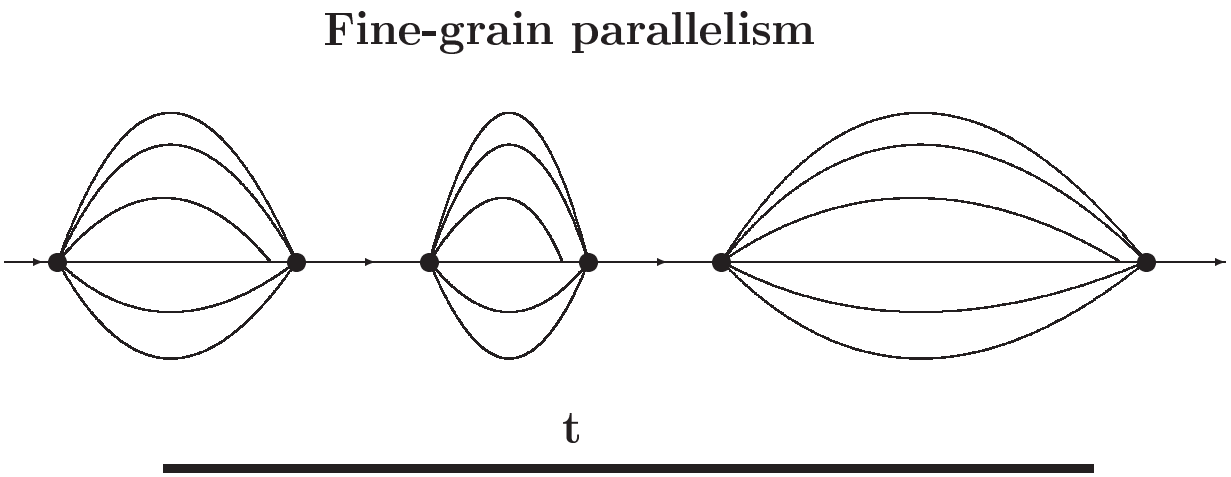
- **job level parallelism:**

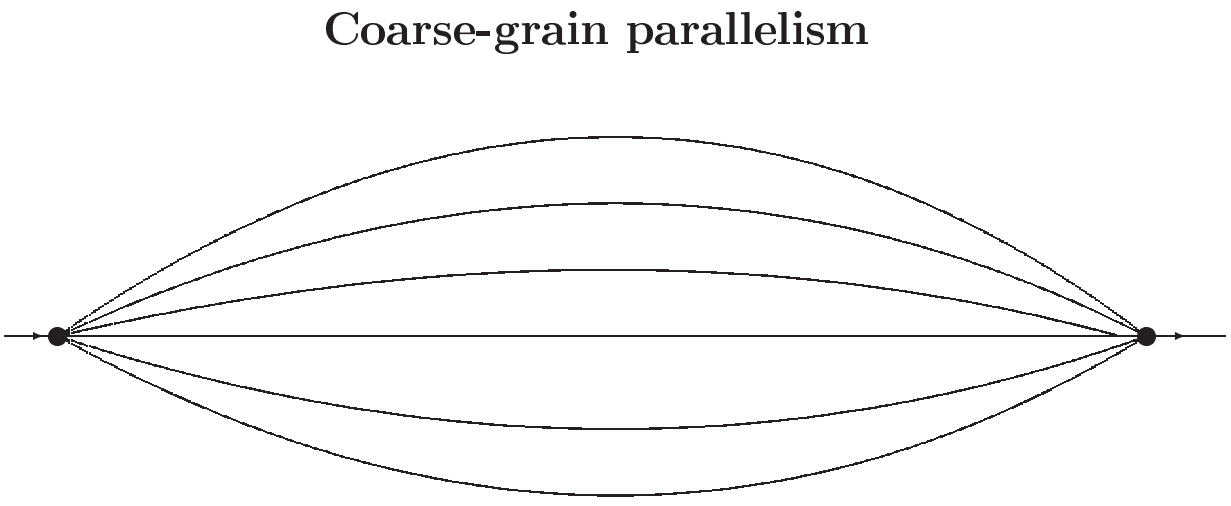
  Some long distance interaction problem, (such as gravity), researchers used to run multiple jobs on the shell, and each job is serial (not correlated with others). It is typical MIMD mode.

- **Fine-Grain and Coarse-Grain parallelism:**

  In general, if the parallelism is larger than 50 instructions, but relatively not too large, we call it fine-grain parallelism. According to Amdahl's law, you MUST have a lot of the fine-grain parallelism you could get the parallel speed up. Most of the loop level parallelism belong to fine-grain parallelism. Instruction level parallelism is a very fine-grain parallelism.

  If the parallelism is really large, we call it coarse-grain parallelism. The Amdahl's law predicts that the coarse-grain parallelism could get better speed up.





3.2 Message Passing and the Message Passing Interface Standard (MPI)

Message passing is a paradigm used widely on certain classes of parallel machines especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. In last ten years, many vendors have developed their own shared memory directives and message passing softwares. The existing message passing systems are, for example, NX/2, Express, Vertex, Canopy, P4, PARMACS, PICI, and PVM. The variations make the distributed parallel programming be very painful. It often happened that even one did invest long time to write a parallel program for some computer, before one get much use to get his job done, either the environment changed or the computer company goes out. So the users always dreams that one can write a program which works on any kind supercomputer. That means we need a real "standard" which must be accepted by all vendors.

The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Thus the MPI has been strongly influenced by the previous message passing softwares and has been accepted by most of the supercomputer vendors.

A preliminary draft proposal of MPI1 was put forward by Dongarra, Hempel, Hey and Walker in November 1992, and a revised version was completed in February 1993. The MPI working group met every 6 weeks for two days throughout the first 9 months of 1993 and presented the draft MPI standard at the supercomputing 93 conference in November 1993.

3.2.1 MPI — Large and small

- What is included in the standard:

  1. **Point-to-point communication**
  2. **Point-to-group communication**
  3. **Group-to-point communication**
  4. **Collective operations and process groups**
  5. **Communication contexts**
  6. **Process topologies and environmental management**
  7. **Profiling interface**

- MPI is built for high level programming languages, such as C, C++ and Fortran. All the routines work as a library in the languages. All the messages use the same data types as in the languages. So we have MPI_INTEGER, MPI_REAL, MIP_REAL8, MPI_LOGICAL, MPI_CHARACTER, MPI_COMPLEX ....

- **Large:**

  There are 128 MPI library routines in MPI1, there are more routines in MPI2.

- **Small:**

  6 to 9 of the 128 routines are enough for most of the applications.

  1. MPI_INIT(arguments)
  2. MPI_COMM_RANK(arguments)
  3. MPI_COMM_SIZE(arguments)
  4. MPI_FINALIZE(arguments)
  5. MPI_SEND(arguments)
  6. MPI_RECV(arguments)
  7. MPI_BCAST(arguments)
  8. MPI_REDUCE(arguments)
  9. MPI_GATHER(arguments)

- Buffer: MPI uses buffer — a piece of the memory to store the messages. The message will be collected to buffer before it being sent. As well as, the received messages also collected in buffer before it being contributed to the corresponding memory addresses.

3.2.2 To learn the 8 basic routines:

1. **MPI_INIT(Ierror) for Fortran**

   where Ierror is a Integer. All MPI programs MUST contain a call to MPI_INIT; This routine initialized the multiple threads by communicating with shell. This routine must be called before any other MPI routine is called.

2. **MPI_COMM_RANK(COMM, RANK, Ierror)**

   where Ierror, Rank and Comm are integer. This routine accesses the communicator's group. The COMM = MPI_COMM_WORLD is a special integer describes the whole group, the RANK = myid, shows rank of the working thread from 0 to number of threads - 1.

3. **MPI_COMM_SIZE(COMM, SIZE, Ierror)**

   where COMM, SIZE and Ierror are integer. This routine used to check the communicator's group size to see if it is the same as expected. The COMM = MPI_COMM_WORLD describes the communicator's group, SIZE = number of the threads exist now.

4. **MPI_FINALIZE(Ierror)**

   where Ierror is a integer. All the fortran routines have a Ierror as one of the arguments just as the C routine's return. If the routine completed correctly, the Ierror returns zero, otherwise, non-zero. This routine cleans up all MPI states. Once this routine is called, no MPI routine may be called. The user must ensure that all pending communications involving a process complete before the process calls MPI_FINALIZE.

   In other hand, user must use MPI_FINALIZE to finish the MPI processes when the program completed, otherwise the machine will keep MPI processes run and wast the supercomputer resources.

5. **MPI_SEND(buf, count, datatype, dest, tag, comm, Ierror)**

   where

   buf: choice, initial address of send buffer (address)

   count: integer, number of elements in send buffer (no negative)

   datatype: handle(MPI data type), the data type of each send buffer element

   dest: integer, rank (ID) of destination

   tag: integer, the message tag for recognization

   comm: handle (i.e. MPI_COMM_WORLD), the communicators

   Ierror: integer, the return value.

   This is the basic point-to-point communication sending routine. The send buffer specified by the MPI_SEND operation consists of *count* successive entries of the type indicated by *datatype* starting with the entry at address *buf*.

   In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information called *message envelope* which include:

   source, determined by the identity of the message sender

   destination, specified by *dest*

   tag, specified by *tag*. This integer can be used to distinguish different type of the messages.

   communicator, specified by *comm*, the communication group.

6. **MPI_SECV(buf, count, datatype, source, tag, comm, status, Ierror)**

   where:

   buf: address, initial address of receive buffer (choice)

count: integer, number of elements in receive buffer (nonegative)

datatype: handle (MPI data type), datatype of each receive buffer element

source: integer, the rank (ID) of the source

tag: interger, message tag for recognization

comm: handle (i.e. MPI_COMM_WORLD), communicator

status: status object, can be read by MPI_GET_COUNT.

Ierror: integer, the routine return

This is one of the basic point-to-point communication receive routine. The receive buffer consists of storage containing *count* consecutive elements of type specified by *datatype*, starting at the address *buf*. The length of the received message must be less than or equal to the length of the receive buffer. The selection of a message by a receive operation is governed by the value of the message envelope. A message can be received by a receive operation if its envelope matches the *source, tag*, and *comm* values specified by the receive operation. The receiver may specify a wild card MPI_ANY_SOURCE value for *source*, and/or MPI_ANY_TAG value for *tag*, indicating that any source and/or tag are acceptable.

The *status* is an array of integers of size MPI_STATUS_SIZE. The two constants MPI_SOURCE and MPI_TAG are the indices of the entries. Using MPI_GET_COUNT(status, datatype, count) can read the receive status.

7. The **MPI_SEND(...) and MPI_RECV(...)** are blocking communication mode. **It does not return until the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer.** The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. Message buffering decouples the send and receive operations. On the other hand , message buffering can be expensive, as it entails additional memory to memory copying, and it requires the allocation of memory for buffering.

   MPI offers the choice of several communication modes that allow user to control the choice of the communication protocol.

8. **MPI_BCAST(buf, count, datatype, root, comm, Ierror)**

   where:

   buf: address, starting address of the buffer

   count: integer, number of the entries in buffer

   datatype: MPI data type in the buffer

   root: integer, the rank of the broadcast root

comm: handl, the communicator — communication group.

MPI_BCAST is a point to group communication routine. It broadcasts a message from the process with rank *root* to all processes of the group, itself included. It is called by all members of group using the same arguments for *comm, root*. On return, the contents of root's communication buffer has been copied to all processes.

9. **MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, Ierror)**

   where:

   sendbuf: address, the address of send buffer

   recvbuf: address, the address of receive buffer, significant only at root

   count: integer, the number of elements in send buffer

   datatype: MPI data type, the data type of the elements in send buffer

   op: operator, the reduce operation (MPI_SUM, MPI_PROD, MPI_MAX...)

   root: integer, rank of root (sender) process

   comm: handle, communicator

   Ierror: integer, the return

   MPI_REDUCE is a group to point communication routine. It combines the elements sent from the send buffer of each process in the group *comm*, using the operation *op*, and return the combined value in the output buffer *recvbuf* of the process with the rank root.

   User also could consider some other basic group to point communication routines such as MPI_GATHER(...).

10. **MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, Ierror)**

    where:

    sendbuf: address, the address of send buffer

    sendcount: integer, the number of elements in send buffer

    sendtype: MPI data type, the data type of the elements in send buffer

    recvbuf: address, the address of receive buffer, significant only at root

    recvcount: integer, the number of elements in receive buffer

    recvtype: MPI data type, the data type of the elements in receive buffer

    root: integer, rank of root (receiver) process

    comm: handle, communicator

Ierror: integer, the return

Each process (root process included) send the contents of its send buffer to the root process. The root process receives the messages and stores them **in rank order**.

3.3 Distributed Memory Parallel Examples

Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point to point communication operations are MPI_SEND(...) and MPI_RECV(...).

Before you can use MPI routines, you MUST initiate the MPI first.

1. **Example 1:** Initiate and Ranks

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Program Hello
!
INCLUDE 'mpif.h'
Character(46,1) :: x
integer :: myid, num_proc, ierror, ierr
! CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, num_proc, ierr)
!
print*, 'number of the processors are: ', num_proc
x = 'Hello every body please distinguish yourselves'
print*, x, myid
END PROGRAM hello
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Using the commands to compile and run it (on SGI 2000):

f90 hello.f90 -lmpi

mpirun -np 4 a.out

You will get the print such as:

```
The number of processors are: 4
The number of processors are: 4
The number of processors are: 4
The number of processors are: 4
Hello every body please distinguish yourselves 1
Hello every body please distinguish yourselves 2
```

```
Hello every body please distinguish yourselves 0
Hello every body please distinguish yourselves 3
```

The *mpirun* command makes the mpi executable run on the machine. The option *-np 4* means you require number of the processors is 4. Of course, you can claim more processors. However, the number of the processors MUST match your data decomposition.

Here the *myid* shows the ID, or RANK of each processor or thread.

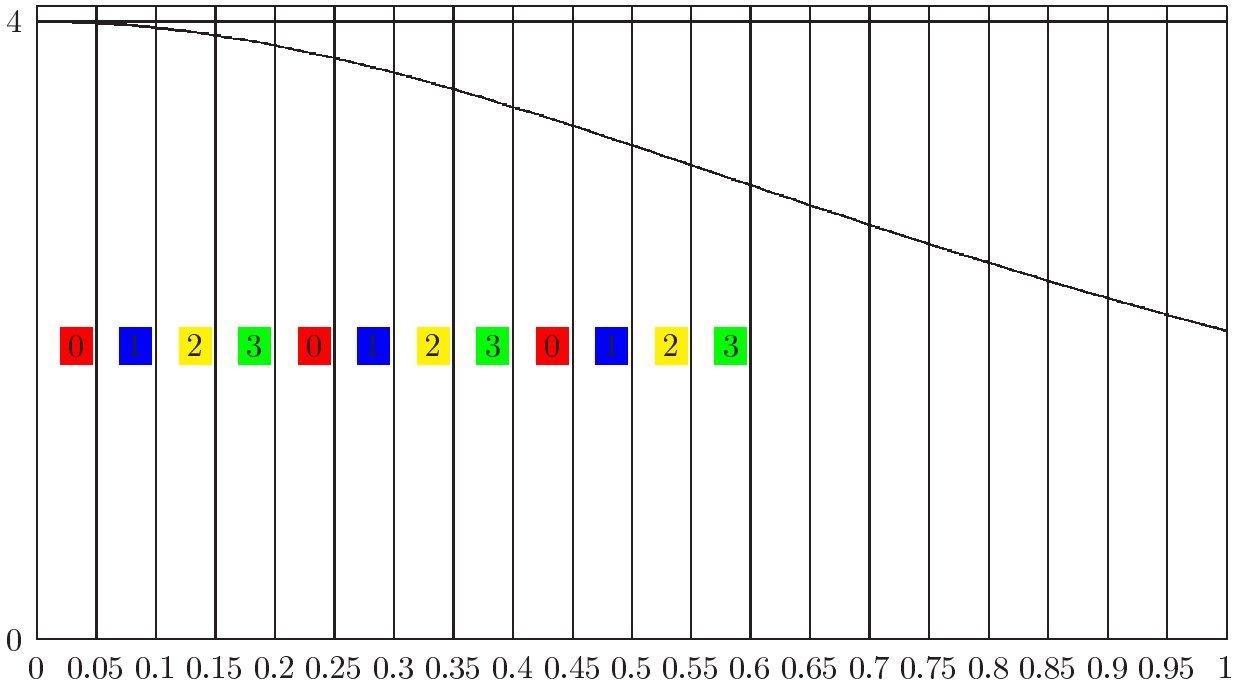
2. **Example 2:** π calculation and integral

$$\pi = \int_0^1 dx f(x) = \int_0^1 dx 4.0 * \frac{1}{1+x^2}$$

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! File kind_spec_module
MODULE kind_spec_module
implicit none
!
INTEGER, PARAMETER:: high = selected_real_kind(15, 307)
INTEGER, PARAMETER:: low = selected_real_kind(6,37)
INTEGER, PARAMETER:: short= selected_int_kind(4)
!
END MODULE kind_spec_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE FUNC
use kind_spec_module
CONTAINS
function f(x)
real(high) :: f, x
!
f = 4.0_high/(1.0_high + x*x)
end function f
END MODULE FUNC
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
PROGRAM PI
USE kind_spec_module
USE FUNC
INTEGER, PARAMETER:: N_init = 1000000
```

```
INTEGER :: myid, N, numprocs ierr
real(high) :: w, sum, xpi, h,mypi ,x
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
n=numprocs*N_init
call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
! calculate the interval size
h = 1.0_high / n
sum = 0.0_high
do i = myid + 1, n, numprocs
x = h * (i - 0.5_high)
sum = sum + f(x)
enddo
mypi = h * sum
print*, mypi, myid
!
! collect all the partial sums
!
call MPI_REDUCE(mypi, xpi, 1, MPI_REAL8, &
MPI_SUM, 0, MPI_COMM_WORLD, ierr)
if (myid .eq. 0) then
write(6, *) xpi, abs(xpi - 3.14159265358979323846264 3 )
endif
call MPI_FINALIZE(ierr)
END PROGRAM PI
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

The same program runs on each processor, with "myid" to compute in different area. Then sum the results together.



Using the shell commands to compile and run it (on SGI 2000):

f90 hello.f90 -lmpi

mpirun -np 4 a.out

You can get:

0.785398100897486 2

0.785397975897464 3

0.78539822589745 1

0.78539835089743 0

3.14159265358983 8.742274371087433E-08

Using more processors you can much higher precision.

3. **Example 3:** Data decomposition — computing matrix-matrix multiplication A·B=C

   - data decomposition 1:

$$\left(\begin{array}{c} \\ A \\ \\ \end{array}\right) \cdots \left(\begin{array}{c|c|c|c} & & & \\ B_1 & B_2 & B_3 & \cdots \\ & & & \end{array}\right)$$

$$C = A \cdot B_1 \oplus A \cdot B_2 \oplus \cdots \oplus A \cdot B_n$$

   - data decomposition 2:

$$\left(\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right) \cdot \left(\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right)$$

$$\begin{aligned} C = &(A_1 \cdot B_1 + A_2 \cdot B_3) \oplus (A_1 \cdot B_2 + A_2 \cdot B_4) \\ &\oplus (A_3 \cdot B_1 + A_4 \cdot B_3) \oplus (A_3 \cdot B_2 + A_4 \cdot B_4) \end{aligned}$$

Program mm_1.f90

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! File kind_spec_module
MODULE kind_spec_module
implicit none
!
INCLUDE 'mpif.h'
INTEGER, PARAMETER:: high = selected_real_kind(15, 307)
INTEGER, PARAMETER:: low = selected_real_kind(6,37)
INTEGER, PARAMETER:: short= selected_int_kind(4)
!
```

```
END MODULE kind_spec_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
PROGRAM mm
USE kind_spec_module
INTEGER, PARAMETER:: root=0, nrow=1000, ncol=1000, NP=4, NC=ncol/NP
INTEGER :: myid, N, numprocs, ierr, i, j, NB0
real(high), dimension(1:nrow, 1:ncol) :: A, C
real(high), dimension(1:nrow, 1:nc) :: b, ans
call MPI_INIT(ierr)
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!
! To create the matrices A and B which is not necessary in a exact routine
!
NB0 = myid*NC
DO j=1,ncol
DO i=1,nrow
a(i,j) = i + 0.023_high*j
ENDDO
ENDDO
DO j = 1,nc
DO i = 1, nrow
b(i,j) = 0.11_high*i + 1.19_high*(j+NB0)
ans(i,j) = 0.0_high
ENDDO
ENDDO
!
! Start Matrix to matrix multiplication
!
DO j=1,nc
DO i = 1, nrow
DO k = 1, nrow
ans(i,j) = ans(i,j) + a(i,k)*b(k,j)
ENDDO
ENDDO
ENDDO
!
! Passing ans to root and directly put them together
```

```
!
ncount = nrow*nc
call MPI_GATHER(ans,ncount,MPI_REAL8,C,ncount,MPI_REAL8, &
root, MPI_COMM_WORLD,ierr)
!
! Write out to check the results
!
IF(myid .EQ. 0) THEN
write(6,*) c(10,20), c(20,30), c(40,50)
ENDIF
!
call MPI_FINALIZE(ierr)
!
END PROGRAM mm
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Using the shell commands to compile and run this program

f90 mm.f90 -lmpi

mpirun -np 4 a.out

You can get:

1907122.455 3070659.305 6111733.005

It must be the same as you get in the shared memory case by mm.f90

4. **Example 4:** Program mm_2.f90 the second data decomposition algorithm — blocking multiplication:

$$C = (A_1 \cdot B_1 + A_2 \cdot B_3) \oplus (A_1 \cdot B_2 + A_2 \cdot B_4)$$
$$\oplus (A_3 \cdot B_1 + A_4 \cdot B_3) \oplus (A_3 \cdot B_2 + A_4 \cdot B_4)$$

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! File kind_spec_module
MODULE kind_spec_module
implicit none
!
INCLUDE 'mpif.h'
INTEGER, PARAMETER:: high = selected_real_kind(15, 307)
INTEGER, PARAMETER:: low = selected_real_kind(6,37)
```

```
INTEGER, PARAMETER:: short= selected_int_kind(4)
!
END MODULE kind_spec_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
PROGRAM mm
USE kind_spec_module
!
! NP = NPL * NPR
!
INTEGER, PARAMETER:: root=0,nrow=1000,ncol=1000,NP=4,NPL=2,NPR=2, &
NC=ncol/NPL,NR=nrow/NPR
INTEGER :: myid, N, numprocs, ierr, i, j, NCB, NRA
real(high), dimension(1:nr, 1:nc, NP) :: C
real(high), dimension(1:nr, 1:nc) ::a1,a2, b1, b2, ans
!
call MPI_INIT(ierr)
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!
! To create the matrices A and B which is not necessary in a exact routine
! Exact routine needs only to input matrix correctly.
! where node 0 uses A1, A2 and B1,B3
!        node 1 uses A1, A2 and B2,B4
!        node 2 uses A3, A4 and B1,B3
!        node 3 uses A3, A4 and B2,B4
NRA = 0
IF( myid .GE. 2) NRA = NR
NCB = mod(myid,2)*NC
!
DO j=1,nc
DO i=1,nr
a1(i,j) = i+NRA + 0.023_high*j
a2(i,j) = i+NRA + 0.023_high*(j+NR)
ENDDO
ENDDO
!
DO j = 1,nc
DO i = 1, nr
```

```
b1(i,j) = 0.11_high*i + 1.19_high*(j+NCB)
b2(i,j) = 0.11_high*(i+NR) + 1.19_high*(j+NCB)
ans(i,j) = 0.0_high
ENDDO
ENDDO
!
! Start Matrix to matrix multiplication
!
DO j=1,nc
DO i = 1, nr
DO k = 1, nr
ans(i,j) = ans(i,j) + a1(i,k)*b1(k,j) + a2(i,k)*b2(k,j)
ENDDO
ENDDO
ENDDO
!
! Passing ans to root and directly put them together
!
ncount = nr*nc
call MPI_GATHER(ans,ncount,MPI_REAL8,C,ncount,MPI_REAL8, &
root, MPI_COMM_WORLD,ierr)
!
! Write out to check the results.
! User'd better to understand the structure of result C
!
IF(myid .EQ. 0) THEN
write(6,*) c(10,20,1), c(20,30,1), c(40,50,1)
ENDIF
!
call MPI_FINALIZE(ierr)
!
END PROGRAM mm
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Using the shell commands to compile and run this program you can get the same results as the previous program.

f90 mm.f90 -lmpi

mpirun -np 4 a.out

You can get:

1907122.455 3070659.305 6111733.005

3.4 Lab Exercises

1. Using "vi" to write the first example as hello.f90

   f90 -O2 -lmpi hello.f90

   mpirun -np 4 a.out

   To get the feeling how the MPI opens 4 processes for you.

   Change -np 4 to -np 6, to see how many processes you get.

2. Using "vi" to write the loop parallel π calculation program as pi_2.f90.

   f90 -O2 -lmpi pi_2.f

   time mpirun -np 4 a.out

   To see how the parallelization speeds up the performance. Change -np 4 into -np 6 to see how the multiple processes can increase the accuracy.

3. sing "vi" to write the first example 3 as mm_1.f90

   f90 -O2 -lmpi mm_1.f

   time mpirun -np 4 a.out

   To see how the parallelization speeds up the performance. Compare the results with the openMP program.

Chapter 4

Parallel Programming — To Solve Partial Differential Equations:

1. **Partial Differential Equations and Algorithms.**
2. **Jacobi's method and loop level parallelization (openMP).**
3. **Gauss-Seidel method and routine level parallelization (openMP)**
4. **Gauss-Seidel method and message passing — distributed programming (MPI)**
5. **Laboratory exercises**

4.1 Partial Differential Equations and Algorithms

4.1.1 Computational Science and PDE

Most of the computational sciences are based on the Partial Differential Equations (PDE). Solving PDE with some initial conditions or some boundary conditions we get the understanding and predictions of scientific phenomena. For example:

- Navier Stokes equations (CFD):

$$\begin{aligned} \frac{\partial \vec{v}}{\partial t} &= \vec{v} \times \vec{w} - \nabla \Pi + \nu \nabla^2 \vec{v} \\ \nabla \vec{v} &= 0 \end{aligned}$$

- Diffusion equation (Continuous medium) :

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}(D \frac{\partial u}{\partial x})$$

- Schrodinger equation (Quantum Mechanics):

$$i\frac{\partial\psi}{\partial t} = -\nabla^2\psi + V(\vec{x})\psi$$

- Langevin equation (Molecular dynamics):

$$\frac{\partial\phi}{\partial t} = -\frac{\partial V[\phi]}{\partial\phi} + \eta(t)$$

- Lagrange equation (Fractal)

$$\frac{\partial L}{\partial\phi} - \sum_{i=1}^{3}\frac{\partial}{\partial x_i}\frac{\partial L}{\partial\phi} - \frac{\partial}{\partial t}\frac{\partial L}{\partial\dot{\phi}} = 0$$

  After some analytical steps most of the scientific PDE will be reduced to the typical elliptic equations.

- elliptic equation (e.g. Poisson):

$$\hat{\mathcal{L}}\cdot\phi(\vec{x}) = \rho(\vec{x}) \quad \{ \text{ e.g. } \nabla^2\phi(\vec{x}) = \rho(\vec{x}) \ \}$$

4.1.2 Relaxation algorithm and loop parallelization

The relaxation method used to be employed to solve the elliptic equation. It is wildly used since most of the physics equations could be reduced to a elliptic equations

- Poisson Equations and a Simple Navier-Stokes Equation Example:

  Let us consider a simple 2 dimension CFD basic equations — 2-D Navier-Stokes equations such as:

$$\begin{aligned}\frac{\partial\vec{U}}{\partial t} + \vec{U}\cdot\nabla\vec{U} &= \frac{1}{R_e}\Delta\vec{U} \\ \nabla\cdot\vec{U} &= 0\end{aligned}$$

  The time direction evolution of the velocity field is simple:

$$\begin{aligned}u_n &= u_o(u_o\frac{\partial u_o}{\partial x} + v_o\frac{\partial u_o}{\partial y})\cdot dt + \frac{1}{R_e}(\frac{\partial^2 u_o}{\partial x^2} + \frac{\partial^2 u_o}{\partial y^2})\cdot dt \\ v_n &= v_o(v_o\frac{\partial v_o}{\partial x} + u_o\frac{\partial v_o}{\partial y})\cdot dt + \frac{1}{R_e}(\frac{\partial^2 v_o}{\partial x^2} + \frac{\partial^2 v_o}{\partial y^2})\cdot dt\end{aligned}$$

  So if we know a initial state $\{u_o(x,y), v_o(x,y)\}$ and set a vary small time direction skip step dt, we can get the new state $\{u_n(x,y), v_n(x,y)\}$. However, that is not enough. The new state MUST satisfy the continuity condition:

$$\frac{\partial u_n}{\partial x} + \frac{\partial v_n}{\partial y} = 0$$

Since the change of the velocity filed $\{\delta u, \delta v\}$ is a vector filed, we can introduce a scale field ϕ to describe this vector field such as:

$$\begin{aligned} \nabla\phi &= \{\delta u, \delta v\} \\ u_n &= u_o + \delta u \\ v_n &= v_o + \delta v \end{aligned}$$

Substituting this equation into the new state continuity condition equation we get:

$$\nabla^2\phi = 0$$

That is a simple Poisson equation. So that finally, to solve Navier-Stokes equation goes to solve a Poisson equation. Repeat this procedure you get the evolution of the velocity field.

- Solving equation $\longrightarrow$ minimization problem:

$$\begin{aligned} & \hat{\mathcal{L}} \cdot \phi(\vec{x}) = \rho(\vec{x}) \quad \{ \text{ e.g. } \nabla^2\phi(\vec{x}) = \rho(\vec{x}) \ \} \\ \Rightarrow \ & \hat{\mathcal{L}} \cdot \phi(\vec{x}) - \rho(\vec{x}) = 0 \\ \Rightarrow \ & \frac{\partial\phi(\vec{x})}{\partial\tau} = \hat{\mathcal{L}} \cdot \phi(\vec{x}) - \rho(\vec{x}) \qquad \text{artificial diffusion equations} \end{aligned}$$

where τ is an artificial time (for artificial diffusion equations). Starting at initial configuration $\phi(\vec{x})$ relaxes to an equilibrium as $\tau \to \infty$ by some proper iterations. The equilibrium means $\phi(\vec{x})$ will not change when time changes (artificial time). So that:

$$\frac{\partial\phi(\vec{x})}{\partial\tau} = 0$$

Then the right hand side goes to zero, will satisfy the original equations.

- FTCS (Forward Time Centered Space) representation and the relaxation procedure.

We have several choices for representing the time derivative term. The obvious way is to set:

$$\frac{\partial\phi(j)}{\partial\tau} = \frac{\phi_j^{n+1} - \phi_j^n}{\Delta\tau} + O(\Delta\tau)$$

That is called *Forward Euler Differencing* . While forward Euler is only first-order accurate in $\Delta\tau$, it is convenient that one can calculate quantities at timestep $n+1$ in terms of only the quantities known at timestep n.

For space derivative, we can use a second-order representation still using only quantities known at timestep n:

$$\begin{aligned} \phi_j &= \frac{1}{2} \cdot (\phi_{j+1} + \phi_{j-1}) \\ \frac{\partial\phi(j)}{\partial x} &= \frac{\phi_{j+1}^n - \phi_{j-1}^n}{2\Delta x} + O(\Delta x^2) \\ \frac{\partial^2\phi(j)}{\partial x^2} &= \frac{(\phi_{j+1}^n - \phi_j^n) - (\phi_j^n - \phi_{j-1}^n)}{\Delta x^2} + O(\Delta x^3) \end{aligned}$$

For a 2-D ∇^2 operator we have:

$$\nabla^2\phi(x,y) = \frac{\phi^n_{j+1,i} + \phi^n_{j-1,i} + \phi^n_{j,i+1} + \phi^n_{j,i-1} - 4\phi^n_{j,i}}{\Delta x^2}$$

where suppose $\Delta x = \Delta y$.

So that to solve a typical Poisson equation could become to solve such a relaxation equation:

$$\begin{aligned} \frac{\partial\phi}{\partial\tau} &= \nabla^2\phi - \rho \\ \frac{\phi^{n+1}_{j,i} - \phi^n_{j,i}}{\Delta\tau} &= \frac{\phi^n_{j+1,i} + \phi^n_{j-1,i} + \phi^n_{j,i+1} + \phi^n_{j,i-1} - 4\phi^n_{j,i}}{\Delta x^2} - \rho_{j,i} \end{aligned}$$

- Using 2-D regular grid and (FTCS) differencing solving 2-D Poisson equation becomes a iteration procedure:

$$\begin{aligned} \phi^{n+1}_{j,i} &= \phi^n_{j,i} + \frac{\Delta\tau}{\Delta x^2}(\phi^n_{j+1,i} + \phi^n_{j-1,i} \\ &+ \phi^n_{j,i+1} + \phi^n_{j,i-1} - 4\phi^n_{j,i}) - \rho_{j,i}\Delta\tau \end{aligned}$$

Iteration stability analysis allows $\Delta\tau/\Delta x^2 = 1/4$

$$\begin{aligned} \phi^{n+1}_{j,i} &= \frac{1}{4}(\phi^n_{j+1,i} + \phi^n_{j-1,i} + \phi^n_{j,i+1} + \phi^n_{j,i-1}) \\ &- \frac{\Delta x^2}{4}\rho_{j,i} \end{aligned}$$

So the question becomes to write a program which does this iterations until the filed ϕ gets equilibrium.

4.2 Jacobi's method and loop level parallelization (openMP)

4.2.1 A Poisson Equation Example:

Let us consider a Poisson equation example such as:

$$\begin{aligned} \nabla^2\phi(x,y) &= \rho(x,y) \\ \rho(x,y) &= -2\pi^2 \cdot sin(\pi x) \cdot sin(\pi y) \\ \text{with boundary } \phi(x_b,y_b) &= 0 \end{aligned}$$

This Poisson equation has an analytical solution of

$$\phi(x,y) = sin(\pi x) \cdot sin(\pi y)$$

So that we can easyly compare the numerical results and analytical solution to check the program.

- **To set the data types:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE kind_spec_module
implicit none
INTEGER, PARAMETER :: high = selected_real_kind(15,307)
INTEGER, PARAMETER :: low = selected_real_kind(6, 37)
INTEGER, PARAMETER :: short= selected_int_kind(4)
!!
END MODULE kind_spec_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **To build a function to get a real maximum values:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE fmax_module
USE kind_spec_module
CONTAINS
!!
real(high) function fmax(a,n) result(value)
integer, intent(in) :: n
real(high), dimension(0:n), intent(in) :: a
real(high) :: x
integer :: i
!!
x=dabs(a(0))
DO i=1,n-1
if(dabs(a(i).gt.x) x=a(i)
ENDDO
value = x
!!
END function fmax
!!
END MODULE fmax_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

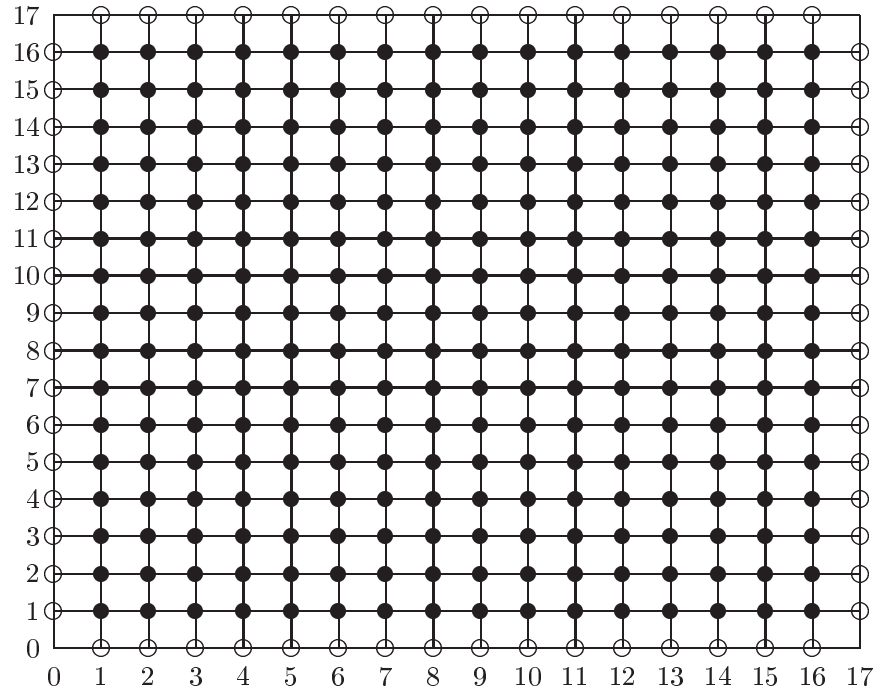
- **To set necessary arrays and to set the geometry, boundary conditions:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
PROGRAM test_f90_loop
!!
USE kind_spec_module
USE sor_module, ONLY: sor
!!
INTEGER, PARAMETER :: nx=161, ny=161, NCPU=4, ny2=ny/NCPU
real(high), dimension(0:nx, 0:ny) :: t1
real(high), dimension(1:nx, 1:ny) :: sol, aw, ae, as, an, ap, dd
real(low), dimension(2) :: tarray
real(low) :: utime1,utime2,stime1,stime2,etime
integer, dimension(3) :: iarray
!!
integer :: i,j,k, nxi,nyi, iter, nxf,nyf
real(high) :: hx,hy,pi,x,y,error
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

where we set the grid is 160×160. Since the **Centeral Space** technique, the boundary is at center of $(0, 1)$ and $(160, 161)$. So we need to set the grid $0 \to 161$. The arrays aw, ae, as, an restore the grid geometry $\Delta x(x, y)/dx$ etc., w,e,s and n mean west, east, south and north directions. In our simple geometry, all $\Delta x = \Delta y = hx = hx = 1/160$.

Where $ap = -(aw + ae + as + an)$, the $1/ap$ is the coefficients of the $\rho(x, y)$ in the iteration procedure expression. In our simple case $1/ap = 1/4$ as we have shown in the iteration procedure expression. **t1** is the main array to store the filed solution of $\phi(x, y)$, the array **sol** is used in the subroutine module sor as a working space.

Where dd is used to store the right hand side of the Poisson equations $\rho(x, y)$



- **To initiate the grid and field:**

```
!
! To set the Δx = hx, Δy = hy and π = 2 * sin^-1(1.0)
!
hx=1.0_high /(dfloat(nx)-1.0_high)
hy=1.0_high /(dfloat(ny)-1.0_high)
pi=dasin(1.0_high) * 2.0_high
!
! To set the geometry amd initiate field values including boundary
!
DO i=1,ny
DO j=1,nx
aw(j,i) = hx/hy
ae(j,i) = aw(j,i)
as(j,i) = hy/hy
an(j,i) = as(j,i)
ap = -(aw(j,i) + ae(j,i) + as(j,i) + an(j,i))
t1(j,i) = 0.0_high
ENDDO
ENDDO
!
```

```
! To set the function ρ(x,y) * Δx^2 on the grid.
!
DO i=1, nx-1
DO j=1, ny-1
x = (i-0.5_high)*hx
y = (j-0.5_high)*hy
dd(i,j) = -2.0_high*pi**2*dsin(pi*x) * dsin(pi*y) *hx*hy
ENDDO
ENDDO
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **To set the boundary geometry:**

```
!
! Boundary is located at 0 → 1 and nx − 1 → nx, ny − 1 → ny.
! West and East
DO j=1,ny
ap(1,j) = ap(1,j) - aw(1,j)
aw(i,j) = 0.0_high
ap(nx-1,j) = ap(nx-1,j) - ae(nx-1,j)
ae(nx-1,j) = 0.0_high
ENDDO
! South and North
DO i=1,nx
ap(i,1) = ap(i,1) - as(i,1)
as(i,1) = 0.0_high
ap(i,ny-1)=ap(i,ny-1)-an(i,ny-1)
an(i,ny-1)=0.0_high
ENDDO
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **Start iteration solver and write the results:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
! Use lib-U77 routine for timming
!
call itime(iarray)
stime1 = etime(tarray)
```

```
utime1 = tarray(1)
!
! The physics grid is 1 → nx − 1, 1 → ny − 1
! nx1=1
nyi=1
nxf=nx-1
nyf=ny-1
error=1.0e-09_high
CALL sor(nx1,nxf,nyi,nyf,aw,ae,as,an,ap,dd, &
t1,sol,error,iter)
!!
! Finish the lib-U77 timming routines
!
stime2=etime(tarray)
utime2=tarray(1)
print*, 'sor is done with iteration number ', iter
print*, 'sor is done with CPU time of ', utime2-utime1, ' seconds'
print*, 'sor is done with system clock ', stime2-stime1, ' seconds'
print*, 'The wall clock started : ', iarray
CALL itime(iarray)
print*, 'The wall clock stoped : ', iarray
!
! Write out the numerical solution comparing with the analytical solution
!
open(8, file='out.dat',status='unknown')
DO i=1,nx-1
DO j=1,ny-1
x=(j-0.5_high)*hx
y=(i-0.5_high)*hy
write(8,'1x, 2I5, 2E15.7') j,i,t1(j,i), dsin(pi*x)*dsin(pi*y)
ENDDO
ENDDO
close(8)
!
END program test_f90_loop
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **The iteration procedure — Jacobi's algorithm:**

  We need the routine of performing the iteration procedure. The previous iteration using $\tau = n$ values to get $\tau = n + 1$ values is called Jacobi's algorithm.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE sor_module
USE kind_spec_module
USE fmax_module
INTEGER, PARAMETER :: nx=161, NY=161, NCPU=4, ny2=ny/NCPU
CONTAINS
SUBROUTINE sor(nxo,nxf,nyi,nyf,aw,ae,as,an, &
ap,dd,solu,sol,error,iter)
real(high), dimension(1:nx,1:ny),intent(inout)::aw,ae,as,an,ap,dd,sol
real(high), dimension(0:nx,0:ny),intent(inout):: solu
integer,intent(inout)::iter,nxi,nxf,nyi,nyf
real(high),intent(inout)::error
real(high),dimension(0:ny) :: err
real(high) :: errors,erro
integer :: JC, j,i
!!
iter=0
erros=error+1.0_high
DO WHILE(erros.GT.error)
err=0.0_high
iter=iter+1
!!
!$OMP parallel do default(none), &
!$OMP& shared(nxi,nxf,sol,dd,aw,ae,as,an,ap,dd,err), &
!$OMP& private(j,i,erro)
!
DO J=nyi,nyf
DO i=nxi,nxf
sol(i,j)=(dd(i,j) &
-aw(i,j)*solu(i-1,j) &
-ae(i,j)*solu(i+1,j) &
-as(i,j)*solu(i,j-1) &
-an(i,j)*solu(i,j+1)) &
/ap(i,j)
erro=dabs(sol(i,j)-solu(i,j))
```

```
if(erro.gt.err(j)) err(j)=erro
ENDDO
ENDDO
!$OMP END parallel DO
!!
errors=fmax(err,ny)
!!
DO i=1,nyf
DO j=1,nxf
solu(j,i)=sol(j,i)
ENDDO
ENDDO
!
! Endo do while loop
!
ENDDO
!!
ENDO SUBROUTINE sor
!!
END MODULE sor_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **Loop Parallelization:**

  The Jacobi's algorithm offered that there is not loop carried dependence within a iteration procedure. So we can easily to use openMP to make the loop within iteration be parallelized.

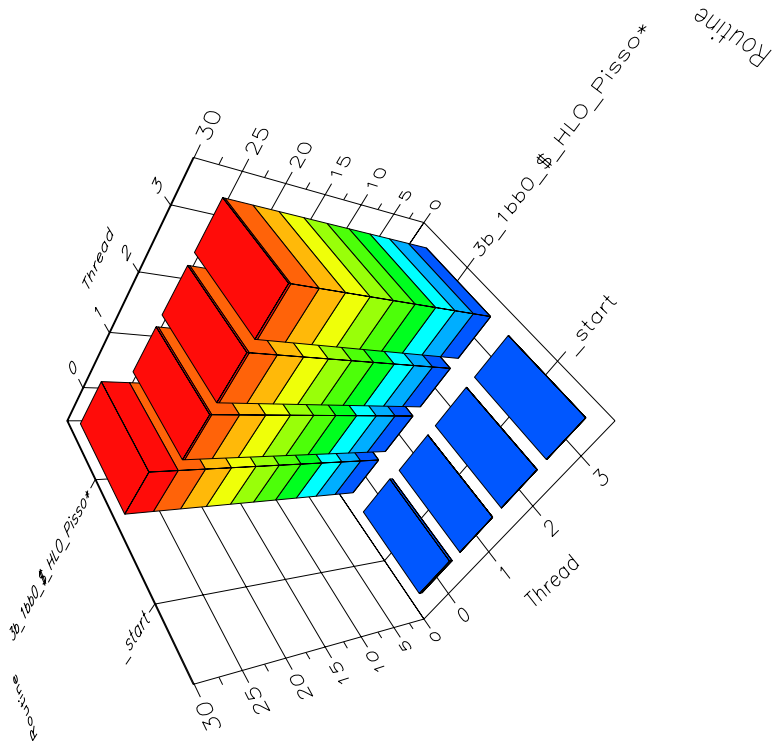
Input $\rho(x, y) = -2\pi^2 sin(\pi x) sin(\pi y)$ we have an analytical solution of

$$\phi(x, y) = sin(\pi x) sin(\pi y)$$

which could be used to compare with numerical solution. The comparison is written in the file names 'out.dat'. Performance analysis with 4 processors as follows (HP/N_class):
Parallelization speed up: $\frac{\text{CPU\_P}}{\text{WALL\_P}} = 3.8$
Schedule speed up: $\frac{\text{WALL\_S}}{\text{WALL\_P}} = 3.7$
CPU overhead: $\frac{\text{CPU\_P}}{\text{CPU\_S}} - 1 = 3.75\%$
The 3-D visual reports as:

Table 4.1: Loop parallelization test (Jacobi's)

items	serial	parallel code with 4 processors
iteration	63127	63127
Wall clock	132.0s	35.9s
CPU Time	131.9s	136.9s



CPU (seconds) for Routines

4.3 Gauss-Seidel method and routine level parallelization (openMP)

4.3.1 Gauss-Seidel Algorithm, Over-Relaxation and Data Decomposition

- **Gauss-Seidel Algorithm:**

  The Jacobi's algorithm is not a good algorithm. Since it uses 'previous' results to estimate 'current' results the speed of this algorithm is bad, the accuracy is bad too.

  Another classical method is the Gauss-Seidel algorithm. This algorithm uses updated values of ϕ as input values as soon as they become available.

$$\begin{aligned} \phi_{j,i}^{n+1} &= \frac{1}{4}(\phi_{j+1,i}^{n} + \phi_{j-1,i}^{n+1} + \phi_{j,i+1}^{n} + \phi_{j,i-1}^{n+1}) \\ &- \frac{\Delta x^2}{4}\rho_{j,i} \end{aligned}$$

  By this way we can get fast converges. However, there is obviously data dependence within an iteration. So that the parallelization is not as simple as the Jacobi's algorithm.

- **Over-Relaxation:**

  Since all the algorithm of solving the Partial Differential Equations are based on the relaxation procedure, we can consider that, when we get the change of the field $\Delta\phi$, it always go forward to the equilibrium status — so-called relaxing. Could we make the relaxing a little bit faster? Such as after each Gauss-Seidel iteration, make the new configuration as:

$$\begin{aligned} \phi'^{n+1}_{j,i} &= \phi^{n}_{j,i} + w \cdot \{\phi^{n+1}_{j,i} - \phi^{n}_{j,i}\} \\ 1.0 \leq\ & w \ \leq 2.0 \end{aligned}$$

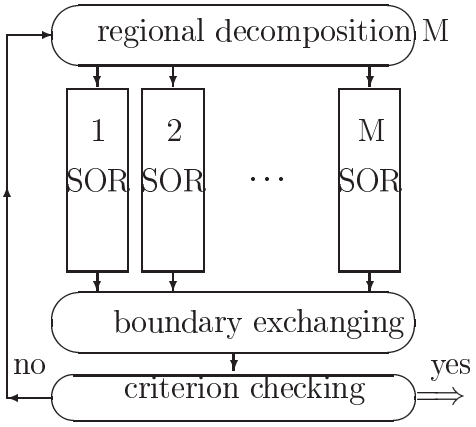
  The experiments show that this **Over-Relaxation** gives us fastest converges adds more data dependence. The optimal choice of w is

$$w = \frac{2}{1 + \sqrt{1 - \rho^2_{Jacobi}}}$$

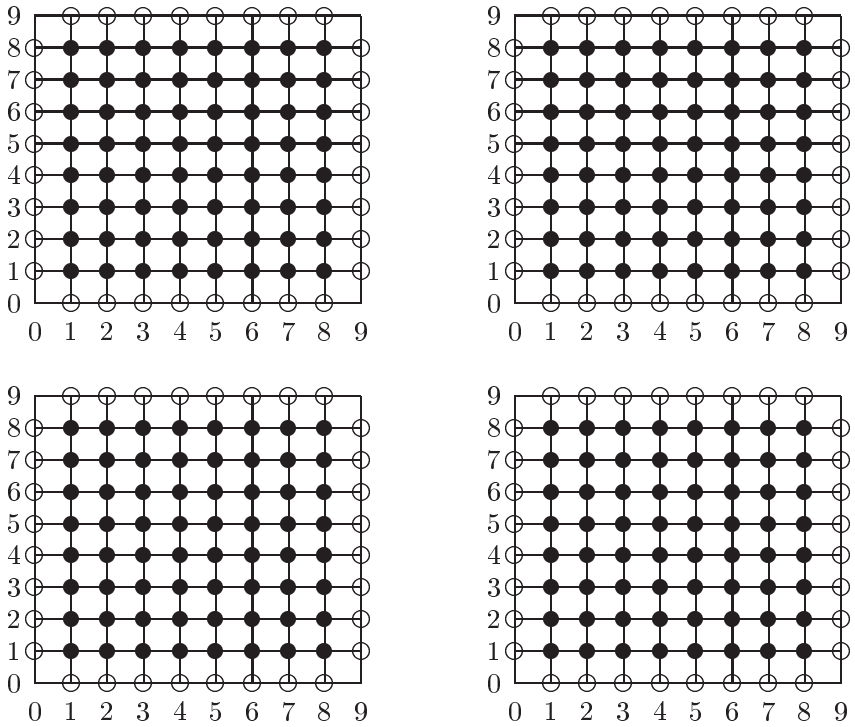
  where ρ_{Jacobi} is the spectral radius of Jacobi's iteration. The detail please check any numerical algorithm text book. However, the experiments show that user can simplely choose $w = 1.5$ *or* 1.6 to get good enough speed.

- **Natural Parallelism and SIMD model:**

  Most of the physics systems are nearest neighbor interacting systems which give us some natural parallelism. We can use regional data decomposition method to manage our data streams within physics space. That gives us a SIMD parallelism model.



The regional data decomposition is simple to separate the 2-D space into several pieces including the additional boundaries such as:



- **The main Gauss-Seidel Routin with Over-Relaxation:**

At this time we first set the number of processors we want to use in the kind_spec_module. We first do the data decomposition for y-direction only and leave the x-direction decomposition as a exercise for the readers.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE kind_spec_module
!
implicit none
INTEGER, PARAMETER :: high = selected_real_kind(15, 307)
INTEGER, PARAMETER :: low = selected_real_kind(6, 37)
INTEGER, PARAMETER :: short= selected_int_kind(4)
INTEGER, PARAMETER :: nx=161, ny=161, NCPU=4, ny2=ny/NCPU
!
END MODULE kind_spec_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

The Gauss-Seidel routine works only on a sub-space such as:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
MODULE sor_module
!
CONTAINS
!
SUBROUTINE sor_task(nx,ny2,aw,ae,as,an,ap,dd,solu,err,w)
!
USE kind_spec_module, ONLY : high
integer, intent(in) :: nx, ny2
real(high), dimension(1:nx,1:ny2+1), intent(inout) :: aw, ae, as, an, ap, dd
real(high), dimension(0:nx,0:ny2+1), intent(inout) :: solu
!
real(high), intent(inout) :: err
real(high), intent(in) :: w
real(high) :: snew
integer :: j, i
!
do j=1,ny2
do i=1,nx
snew = (dd(i,j) &
```

```
-aw(i,j)*solu(i-1,j) &
-ae(i,j)*solu(i+1,j) &
-as(i,j)*solu(i,j-1) &
-an(i,j)*solu(i,j+1)) &
/ap(i,j)
err = dabs(snew-solu(i,j))
solu(i,j) = solu(i,j) + w*(snew-solu(i,j))
enddo
enddo
!
end subroutine sor_task
END MODULE sor_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Then we build a TASKING_module to use the sor_task routine on a **shared memory** machine.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE TASKING_module
USE kind_spec_module
!
real(high), dimension(0:nx, 0:ny2+1) :: t1, t2, t3, t4
real(high), dimension(1:nx, 1:ny2) :: aw1, ae1, as1, an1, &
ap1, dd1, aw2, ae2, as2, an2,ap2,dd2, aw3,ae3,as3,an3,ap3,dd3,aw4,ae4,as4, &
an4,ap4,dd4
!
CONTAINS
SUBROUTINE TASKING(iter,crite,w)
!
! To separate the field data array into NCPU .
! To initialize the boundary condition, source input, and regional
! initialization.
USE sor_module, ONLY : sor_task
!
real(high) :: error,err1,err2,err3,err4
real(high), intent(in) :: crite,w
integer, intent(inout) :: iter
integer :: i, j
!
```

```
! Start iteration
!
iter=0
error = 1.0_high
!
do WHILE(error .GT. crite)
! iter=iter+1 !
! Start the openMP parallel section parallelization
!
!$OMP parallel shared(w, nx, ny2)
!$OMP SECTIONS
!$OMP SECTION
call sor_task(nx,ny2,aw1,ae1,as1, &
an1,ap1,dd1,t1,err1,w)
!$OMP SECTION
call sor_task(nx,ny2,aw2,ae2,as2, &
an2,ap2,dd2,t2,err2,w)
!$OMP SECTION
call sor_task(nx,ny2,aw3,ae3,as3, &
an3,ap3,dd3,t3,err3,w)
!$OMP SECTION
call sor_task(nx,ny2,aw4,ae4,as4, &
an4,ap4,dd4,t4,err4,w)
!$OMP END SECTIONS
!
! Start communication to exchange the inner boundary values
!
!$OMP DO private(j)
DO j=0,nx
t1(j,ny2+1)=t2(j,1)
t2(j,ny2+1)=t3(j,1)
t3(j,ny2+1)=t4(j,1)
t2(j,0) = t1(j,ny2)
t3(j,0) = t2(j,ny2)
t4(j,0) = t3(j,ny2)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

```
!
! Check the criterion condition
!
error=err1
if(err2.gt.error) error=err2
if(err3.gt.error) error=err3
if(err4.gt.error) error=err4
!
if(mod(iter,1000).EQ.0) write(6,*) 'iteration error and criterion ',&
iter,error,crite
end do
!
END SUBROUTINE TASKING
!
END MODULE TASKING_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Performance analysis on HP/N_class shows ($err = 10^{-10}$):

Table 4.2: Multiple section parallelization test (SOR_TASK)

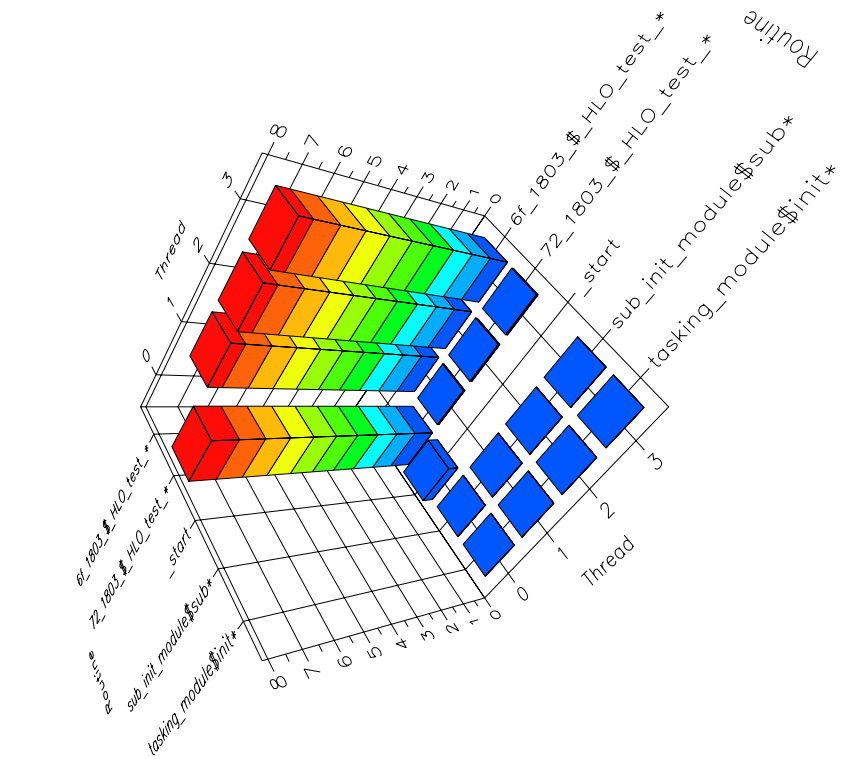
items	serial code	parallel code with 4 processors
iteration	9383	9383
Wall clock	32.1s	8.6 s
CPU Time	32.0s	33.17s

Parallelization speed up: $\frac{\text{CPU\_P}}{\text{WALL\_P}} = 3.86$

Schedule speed up: $\frac{\text{WALL\_S}}{\text{WALL\_P}} = 3.7$

CPU overhead: $\frac{\text{CPU\_P}}{\text{CPU\_S}} - 1 = 3.8\%$

The 3-D visual reports as:


CPU (seconds) for Routines
Routine
6f_1803_$_HLO_test_*
72_1803_$_HLO_test_*
_start
sub_init_module$sub*
tasking_module$init*
Thread
0
1
2
3
0
1
2
3
4
5
6
7
8

4.4 Gauss-Seidel method and message passing — distributed programming (MPI)

Since Gauss-Seidel algorithm needs to use the natural regional parallelism, and needs the communication among all the threads. All the data regions with corresponding boundary are independent of each other within an iteration. So we can send the communication work to the message passing interface and write the distributed memory program. At this case, the program will be a SPMD model. Each processor works with the same program and passes message to each other. That kind program can be run on even very simple PC cluster, without very expensive processors and softwares. The program will has not any directives, neither openMP, nor APO.

- **Set simple data for one processor:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! File kind_module
MODULE kind_spec_module
!
implicit none
!
INTEGER, PARAMETER :: high = selected_real_kind(15, 307)
INTEGER, PARAMETER :: low = selected_real_kind(6, 37)
INTEGER, PARAMETER :: short= selected_int_kind(4)
INTEGER, PARAMETER :: nx=161, ny=161, NCPU=4, ny2=ny/NCPU
!
END MODULE kind_spec_module
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **The Gauss-Seidel Iteration Routine for one processor:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
MODULE sor_module
!
CONTAINS
subroutine sor(nx,ny2,aw,ae,as,an,ap,dd,solu,err,w)
! Use high only, do not use the nx,ny,ny2,NCPU
USE kind_spec_module, ONLY : high
!
```

```
real(high), dimension(1:nx, 1:ny2), intent(inout) :: aw,ae,as,an,ap,dd
real(high), dimension(0:nx, 0:ny2+1), intent(inout) :: solu
real(high), intent(in) :: w
real(high), intent(inout) :: err
real(high) :: snew
integer, intent(in) :: nx, ny2
integer :: i,j
!
do j=1,ny2
do i=1,nx-1
snew = (dd(i,j) &
-aw(i,j)*solu(i-1,j) &
-ae(i,j)*solu(i+1,j) &
-as(i,j)*solu(i,j-1) &
-an(i,j)*solu(i,j+1)) &
/ap(i,j)
err =dabs(snew-solu(i,j))
solu(i,j) = solu(i,j) + w*(snew-solu(i,j))
enddo
enddo
!
end subroutine sor
END MODULE sor_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **Specail Initiation considering the whole large grid:**

  We must initiate a *physics state* on the grid, and prepare to use the message passing interface. So the initiation are little bit different from the original simple one (in section 2).

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE initial_module
!
CONTAINS
!
SUBROUTINE INITIAL(t,aw,ae,as,an,ap,dd,hx,hy,myid,idu,idd)
USE kind_spec_module
real(high), dimension(1:nx,1:ny2), intent(inout) :: aw,ae,as, an, ap, dd
```

```
real(high), dimension(0:nx, 0:ny2+1), intent(inout) :: t
real(high), intent(in) :: hx,hy
real(high) :: pi, y2hy, x, y
integer, intent(in) :: myid
integer, dimension(1:NCPU), intent(inout) :: idu, idd
integer :: npiece, i, j
!
! Input source function
!
npiece = myid + 1
pi=datan(1.0_high)*4.0_high
y2hy=hy*dfloat(ny2)*dfloat(npiece-1)
!
! The neighbors id
!
DO i=1,NCPU
idu(i) = i
idd(i) = i - 2
ENDDO
idd(1) = NCPU-1
idu(NCPU) = 0
!
DO j=1,ny2
DO i=1,nx-1
x=hx*(dfloat(i)-0.5_high)
y=hy*(dfloat(j)-0.5_high)+y2hy
dd(i,j)=-2.0_high*(pi**2)*dsin(pi*x)*dsin(pi*y)*hx*hy
ENDDO
ENDDO
!
! Input boundary and mesh information
!
DO i=1,ny2
DO j=1,nx
aw(j,i)=hy/hy
ae(j,i)=aw(j,i)
as(j,i)=hx/hy
an(j,i)=as(j,i)
```

```
ap(j,i)=-aw(j,i)-ae(j,i)-as(j,i)-an(j,i)
ENDDO
ENDDO
!
! Fix the physics boundary and multi-region boundary condition
!
! X-direction all regions have X-direction physics boundary.
!
DO j=1,ny2
ap(1,j)=ap(1,j)-aw(1,j)
ap(nx-1,j)=ap(nx-1,j)-ae(nx-1,j)
aw(1,j)=0.0_high
ae(nx-1,j)=0.0_high
ENDDO
!
! Y-direction only 1 and 4 have Y-direction physics boundary.
!
IF(npiece.eq.1) THEN
DO i=1,nx
ap(i,1)=ap(i,1)-as(i,1)
as(i,1)=0.0_high
ENDDO
ENDIF
!
IF(npiece.eq.NCPU) THEN
DO i=1,nx
ap(i,ny2)=ap(i,ny2)-an(i,ny2)
an(i,ny2)=0.0_high
ENDDO
ENDIF
!
! Initialize the solution to be zero
!
DO i=0,ny2+1
DO j=0,nx
t(j,i)=0.0_high
ENDDO
ENDDO
```

```
!
end SUBROUTINE INITIAL
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
SUBROUTINE btoex
USE kind_spec_module
!
real(high), dimension(0:nx, 0:ny2+1) :: t
real(high), dimension(1:nx, 1:ny2) :: aw,ae,as,an,ap,dd
integer :: myid, idummy
real(high) :: xerr, err
integer, dimension(1:NCPU) :: idu, idd
!
COMMON / FIELD / t,aw,ae,as,an,ap,dd
!
COMMON / mpidata / myid,idummy, xerr, err,idu,idd
!
IF(myid.NE.(NCPU-1)) THEN
DO j=0,nx
t(j, 0) = t(j, ny2)
ENDDO
ENDIF
!
IF(myid.NE.0) THEN
DO j=0,nx
t(j, ny2+1) = t(j, 1)
ENDDO
ENDIF
!
END SUBROUTINE btoex
!
END MODULE initial_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **For using the message passing interface we build a module to include most of the communication procedures:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
MODULE COMMU_module
```

```
!
CONTAINS !
SUBROUTINE COMMU_replace
!
USE kind_spec_module
include 'mpif.h'
INTEGER ISTATUS(MPI_STATUS_SIZE)
!
real(high), dimension(0:nx, 0:ny2+1) :: t
real(high), dimension(1:nx, 1:ny2) :: aw,ae,as,an,ap,dd
real(high) :: xerr, err
integer :: myid, idummy
integer, dimension(1:NCPU) :: idu, idd
integer :: iid, irr
!
COMMON / FIELD / t,aw,ae,as,an,ap,dd
!
COMMON / mpidata / myid,idummy, xerr, err,idu,idd
!
iid = myid + 1
CALL MPI_Sendrecv_replace(t(1, 0), nx+1, MPI_REAL8, idu(iid), &
myid, idd(iid), idd(iid), MPI_COMM_WORLD, ISTATUS, irr)
!
CALL MPI_BARRIER(MPI_COMM_WORLD, irr)
!
CALL MPI_Sendrecv_replace(t(1, ny2+1), nx+1, MPI_REAL8, idd(iid), &
myid, idu(iid), idu(iid), MPI_COMM_WORLD, ISTATUS, irr)
!
CALL MPI_BARRIER(MPI_COMM_WORLD, irr)
!
CALL MPI_Allreduce(err, xerr, 1, MPI_REAL8, MPI_SUM, &
MPI_COMM_WORLD, irr)
!
END SUBROUTINE COMMU_replace
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
SUBROUTINE OUTPUT(myid, t, hx, hy)
USE kind_spec_module
include 'mpif.h'
```

```
!
real(high), dimension(0:nx, 0:ny2+1), intent(inout) :: t
real(high), dimension(0:nx, 0:ny2, 1:NCPU) :: tc
real(high), intent(in) :: hx,hy
real(high) :: pi, y2hy, x, y
integer , intent(in) :: myid
integer :: ncount, jm, jid, i, j
!
pi=4.0_high*datan(1.0_high)
y2hy=dfloat(ny2)*hy
ncount = (nx+1)*(ny2+1)
!
! Gether all solution to root processor in the array tc
!
IF(myid .EQ. 0 ) OPEN(unit=8, file='out.d')
CALL MPI_Gather(t,ncount,MPI_REAL8, tc,ncount,MPI_REAL8,0, &
MPI_COMM_WORLD, ir)
!
IF(myid.eq.0) THEN
DO jm=1,NCPU
jid = jm - 1
DO j=1,ny2
DO i=1,nx-1
x = (dfloat(i)-0.5_high)*hx
y = (dfloat(j)-0.5_high)*hy
!
write(8,'(1x, 2i5, 2E18.10)') i,j+jid*ny2,tc(i,j,jm),dsin(pi*x) &
*dsin(pi*(y+dfloat(jid)*y2hy))
ENDDO
ENDDO
ENDDO
ENDIF
!
END SUBROUTINE OUTPUT
!
END MODULE COMMU_module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

- **Main program used to call different routines and output the results:**

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program test_MPI
!
USE kind_spec_module
USE initial_module
USE COMMU_module
USE sor_module
!
include 'mpif.h'
!
real(high), PARAMETER :: crite=1.0d-10, w=1.5_high
real(high), dimension(0:nx, 0:ny2+1) :: t
real(high), dimension(1:nx, 1:ny2) :: aw,ae,as,an,ap,dd
real(high) :: xerr, err, hx,hy
real(low), dimension(1:2) :: tarray
real(low) :: utime1,utime2,stime1,stime2
integer :: myid, idummy,numprocs,ierr
integer, dimension(1:NCPU) :: idu, idd
integer, dimension(1:3) :: iarray
COMMON / FIELD / t,aw,ae,as,an,ap,dd
!
COMMON / mpidata / myid,idummy, xerr, err,idu,idd
!
! To initiate the MPI
!
CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
hx=1.0_high/(dfloat(nx)-1.0_high)
hy=1.0_high/(dfloat(ny)-1.0_high)
!
! To initiate the physics state
!
CALL INITIAL(t,aw,ae,as,an,ap,dd,hx,hy,myid,idu,idd)
!
! To initiate U77 Library
```

```
!
print *, 'ok Ax=B is ready', myid
call itime(iarray)
stime1=etime(tarray)
utime1=tarray(1)
stime1=tarray(2)
!
! To solve the PDE by Gauss-Seidel iteration algorithm.
! Start iteration
!
iter = 0
xerr = 1.0_high
!
DO while ( xerr .gt. crite )
!
iter = iter + 1
!
CALL SOR(nx,ny2,aw,ae,as,an,ap,dd,t,err,w)
!
! Start communication to exchange the inner boundary values
!
CALL btoex
CALL COMMU_replace
!
! To check the criterion condition
!
IF( myid.eq.0.and.mod(iter, 1000).EQ.0 ) &
print *, 'iteration and error', iter, xerr
!
ENDDO ! end the do while loop
!
! To output the numerical results into a disk file
!
CALL OUTPUT(myid, t, hx, hy)
!
! To finish the MPI
!
CALL MPI_FINALIZE(ir)
```

end program test_MPI

!!

- The program looks longer, but it is not true for the machine performance.

Table 4.3: MPI parallelization test (SOR)

items	serial code	parallel code with 4 processors
iteration	9558	9558
Wall clock	34.8	8.9 sec
CPU Time	34.4	34.8s

4.5 Laboratory exercises

1. Please write yourself the whole program of Jacobi's loop parallel solver to the Poisson equations name it Poisson_omp.f90.

   You can use following shell commands to compile it and run it.

   f90 -mp -O2 Poisson_omp.f90

   setenv OMP_NUM_THREADS 4

   a.out

   If you meet any troubles, you can use *f90 -mp -g -O2 Poisson_omp.f90* to compile it and use *gb a.out* to debug the executable. Please follow the document of the *gb*. You can do it anymore.

2. Please read the program Poisson_MPI.f90, which is the MPI message passing program. I added some c-preprocessing directives in there. If you understood the program you can try to compile and run it by following shell commands:

   f90 -O2 -DMPI Poisson_MPI.f90 -lmpi

   mpirun -np 4 a.out

   Then using vi to change the NCPU=1, using following commands to run the job serially. Change the *crite* to make the iteration number very close to the parallel run. You can check the parallel speed-up.

   f90 -O2 -DSingle Poisson_MPI.f90 -lmpi

   a.out

Chapter 5

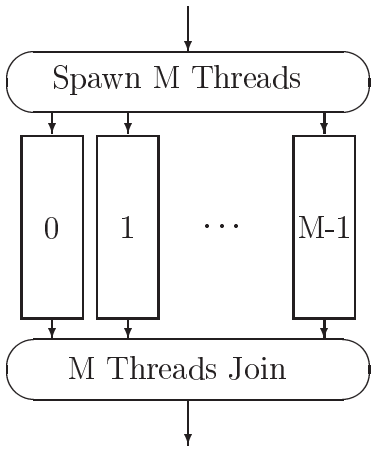
How to Write a Fast and Portable Parallel Program

1. **Spawn and Join Overhead — Size Problem**
2. **Memory Hierarchy, Cache Missing Latency — Locality**
3. **Message Passing Latency — Limit Message Passing**
4. **Synchronization Waiting — Coarse Parallelism and Fine Parallelism**
5. **RISC Processor Pipe Line — Instruction level parallelism**
6. **Portability**

A fast and portable program is a dream of most of the high performance parallel computing programmers. There is a lot of the factors which affect the speed and portability. Most of them are related to the architecture of the exact machines. Some also related the exact scientific problems which you want to solve. We just discuss some major projects here.

5.1 Spawn and Join Overhead — Size Problem

We have see in the chapter2 that in the loop level parallelization the program goes to "spawn" multiple threads, send all "private" variables to each "child" threads as well as send the "code" which going to be executed to the "child" threads, then starts to execute all the block of the code. When any thread finished the executing, it MUST wait at the "end do", until all the threads finished their work. Then the master thread collects all the necessary (like LASTPRIVATE etc.) data back to main stream, and asks system to close the additional threads (joining).



You can get benefits from multiple thread performance, but you also HAVE TO expense the overhead. For example, the matrix-matrix multiplication program mm_omp.f90 which we shown in chapter2:

Table 5.1: Matrix-Matrix Multiplication Test (HP/N_class)

NxN	1 Processor		2 Processor		4 Processor	
	CPU	Wall	CPU	Wall	CPU	Wall
100x100	.00072	.00072	.0022	.0016	.0087	.0038
200x200	0.025	0.025	0.042	0.028	0.114	0.047
500x500	0.912	0.914	1.075	0.553	1.550	0.420
1000x1000	8.740	8.743	10.091	5.093	14.240	3.647
2000x2000	70.259	70.267	85.981	43.159	114.211	28.909

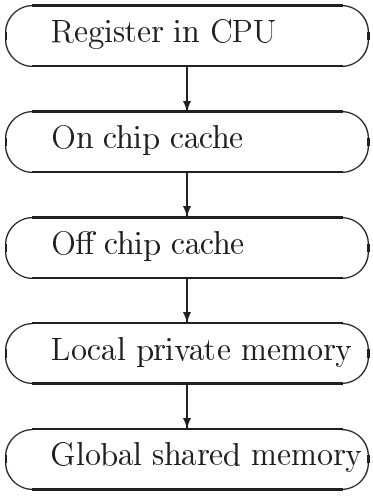
The experiment shows that when the matrix size less than 200x200, the parallelization only gets no speed up but slow down. Only the matrix size is larger than 500x500, the parallelization gets speed up. As well, we can see that, more processors do not means higher speed. In the 2000x2000 case, 2 processors make a factor 1.6 speed up, but 4 processors make only a factor 2.4 speed up. We paied twice computer resources, get only 80% to 60% is useful.

So that the loop size is a very important factor to determine whether we parallelize it or not. For small loop the parallelization will only wast the computer resource. As well as, only for large enough loop we can use more processors.

Almost all the supercomputers have the compiler with Automatic Parallelizing Option (APO). On the Origin 2000, it is -O3 option. In the last 10 years, not even one vendor can make the APO success. Since APO check all the loops in your program and parallelize them all. In most of the case you can not guarantee speed up, in some of the complicated case, even you can not guarantee the numerical results are correct. So that **DO NOT DEPEND ON APO,** depend on yourself!!

5.2 Memory Hierarchy, Cache Missing Latency — Hierarchical Algorithm and Locality

Memory hierarchy is an important technique used on all supercomputers to increase the capacity of the memory and decrease the cost. However, it will also decrease the data fetching speed. A typical 5 level memory hierarchy for a shared-distributed machine, such as Origin 2000, is shown that:



5.2.1 Cache Missing Latency and Hierarchical Algorithm

The designed processor speed (peak speed) is very high, for example, the 200MHz Origin 2000, the peak speed is 400Mflops/sec/processor.

Let us consider s very simple operation such as

```
DO j=1,1000000
a(j) = b(j) + c(j)
ENDDO
```

for 400Mflops/sec/processor speed, each processor **wants** 2*8*400=12800MBytes data fetched from memory. From the SGI.com web we get the information of the bandwidth is only 780MBytes/sec/processor peak which is less than 10% of the data wanted.

This conflict strongly shows that the processor uses more than 90% time for waiting the data and does nothing in this case. We call the wasted time when the processor is waiting for data as the **cache missing latency**.

Cache coherence is used to reduce this conflict. The processor also designed to have more INTEGER UNITS to interpret code and make the data being fetched before it is wanted.

However, it does not solve it at all.

The programmer MUST consider how to use the fast cache to speed up his program. Since cache is a very high speed memory, if we can let the data fetched from memory stay in cache and do more performances, them we can dramatically increase the speed. Logically, let us consider a Cache Use Ratio (CUR) such that

$$CUR = \frac{\text{number of performance}}{\text{number of data reference}}$$

For most of linear algebra computing we have:

Table 5.2: BLAS Class and Cache Use Ratio

Class	computing	CUR = $\frac{\text{number of performance}}{\text{number of data reference}}$
0	vector (or matrix) assignment	$\sim 1/2$
1	vector-vector V = B + aV	$\sim 3/3$
2	V-Matrix multiplication V=MV	~ 2
3	M-M multiplication M = LR	$\sim$ n

Table 5.3: BLAS level and cache miss latency on HP/SPP2200

routine	Function	class	Latency/CPU
DGEMM	M1 = M2 * M3	3	21.8%
DGEMV	V = M * V1	2	44.8%
DAXPY	V = a*V1 + V	1	72.1%
DGTHR	V(i)=V1(ind(i))	0	79.9%
SGEMM	M1 = M2 * M3	3	16.7%
SGEMV	V = M * V1	2	43.2%
SAXPY	V = a*V1 + V	1	74.5%

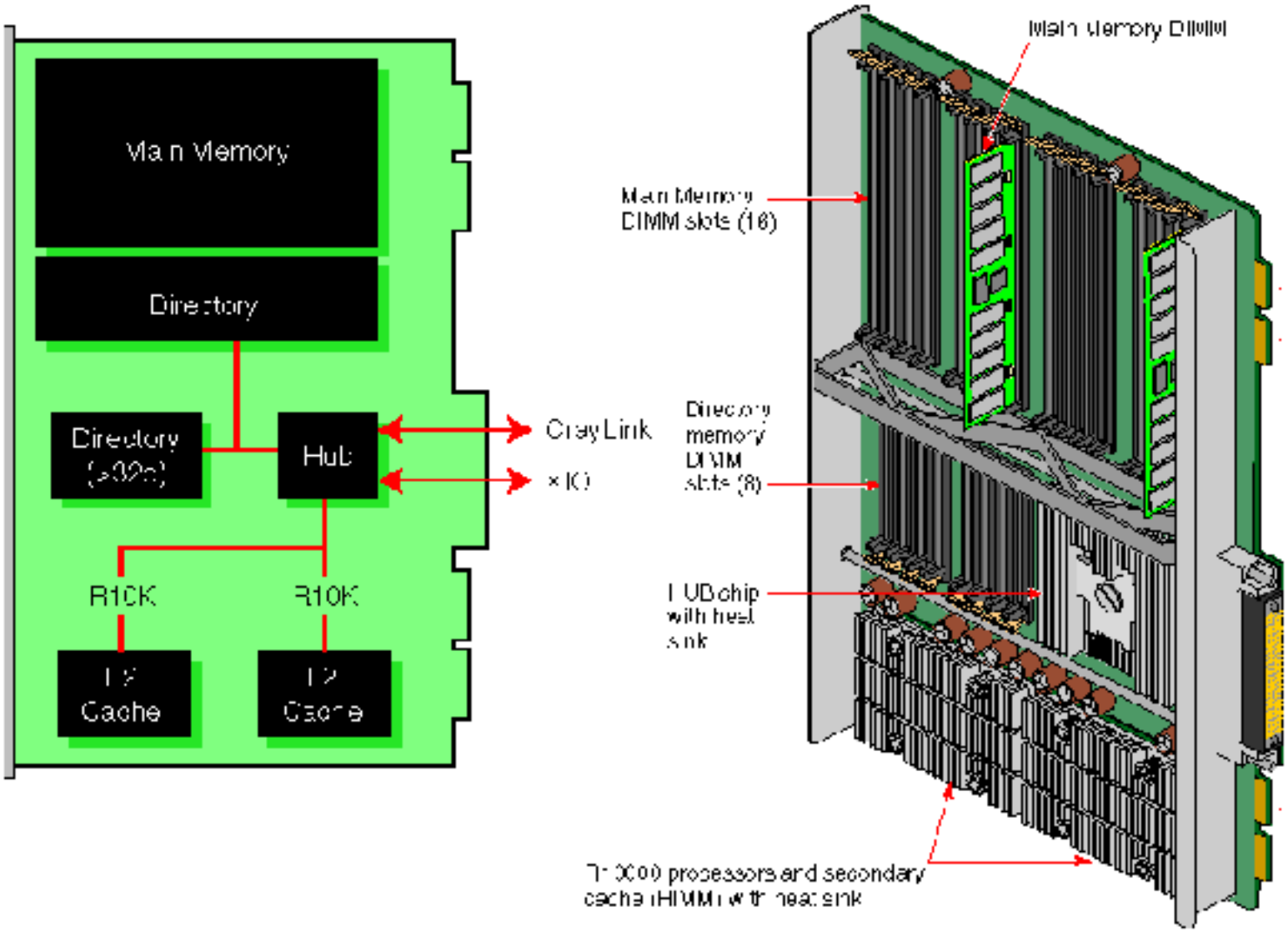
Recently, the numerical algorithm community developed a lot of so-called 'hierarchical algorithms'. The basic idea is using less data to do more computing. In the linear area it is to use more matrix matrix multiplication less lower class computing — that is the *Block Algorithms*. The LAPACK library is a good example.

However, the cache length depends on the exact machine, and L2 cache (off chip) is slower than the L1 cache (on chip). The fatest cache is L0 cache (CPU register). Only very good programmer can consider all such factors to design his program and get very nice speed. However, such a program also will lose the portability too. So we suggest here that:

- If it is possible use the vendor offered library (the same vendor offered machine), which is optimized on the same machine. You will get the best cache use ratio as well as the best performance speed.
- Re-check your programming idea, to see if it does satisfy the *Hierarchical Algorithm* idea or *Block Algorithm* idea.

5.2.2 Private Memory and Global Memory — Locality

The main memory also has 2-level hierarchy — private memory and global memory. For example, SGI Origin 2000 architecture offers 2 processors join one private memory as one board each processor has his own private memory 258MB to 1GB. Two processors on one board connected with multiple cross-bar (CRAY Connection Technique) build one physical address space. The vendor used to call this physical space as the local memory. The data fetching speed within the local memory could reach the peak (about 780MB/sec/processor).



Different processor boards connected by optical fibers. The bandwidth of the connection is about 100MB/sec. When one processor wants data which located on the other processor board, the data MUST fetched through the optical fibers. The cache missing latency will getting much worse.

When the vendor develop the compiler which works on the exact machine they have

considered this problem. So the compiler will distribute the data variables as wide as possible and make them to meet the best fetching status. However, the high level programming language define the data array as that the fortran define array name is an address, C or C++ used to use pointer to the address too. All the array are managed as one by one series (one dimension line in the memory). If your program has some large array, it always located on one processor or one physical address space. That will make the data fetching from the other processor board very slow and the compiler can not help you. So we suggest that:

- To define more relatively small arrays instead of one big array and put them in multiple data blocks (like common block). So that the intelligent compiler can help you to distribute the array location.
- If it is possible, decompose your data design a SPMD mode program. So that the working data arrays always located on one exact processor's main memory.
- To avoid to use single processor with vary large memory, for example > 2GB. It can work but very slow.

5.3 Message Passing Latency — Limit Message Passing

Another important speed delay is the Message Passing Latency. For distributed memory machine programming we always need the message passing. For synchronization reason, the bandwidth of message passing is similar with the global memory data fetching. So we always say that message passing is very expensive. Any message passing slow down your performance. You must design to use the message passing as little as possible. For example, the π calculate is a perfect distributed memory program. It even was used as an advertisement of "networked supercomputer" — thousands workstations connected by internet and got almost linear speed up. Since there is only few data need to be passed.

Other example — the matrix to matrix multiplication:

```
DO j = 1, N
DO i = 1, N
DO k = 1, N
c(i, j) = c(i, j) +a(i, k) * b(k, j)
ENDDO
ENDDO
ENDDO
```

You can just parallelize the most outside loop DO j. So you can send piece of b to different processors, however, you MUST send whole matrix a to each processor too.

$$\left(\quad A \quad \right) \cdots \left(\begin{array}{c|c|c|c} & & & \\ B_1 & B_2 & B_3 & \cdots \\ & & & \end{array} \right)$$

$$C = A \cdot B_1 \oplus A \cdot B_2 \oplus \cdots \oplus A \cdot B_n$$

If you choose BLOCK algorithm you need only send piece of a to each processor.

$$\left(\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right) \cdot \left(\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right)$$

$$\begin{aligned} C = &(A_1 \cdot B_1 + A_2 \cdot B_3) \oplus (A_1 \cdot B_2 + A_2 \cdot B_4) \\ &\oplus (A_3 \cdot B_1 + A_4 \cdot B_3) \oplus (A_3 \cdot B_2 + A_4 \cdot B_4) \end{aligned}$$

The message passing latency will be reduced the performance speed will be getting better.

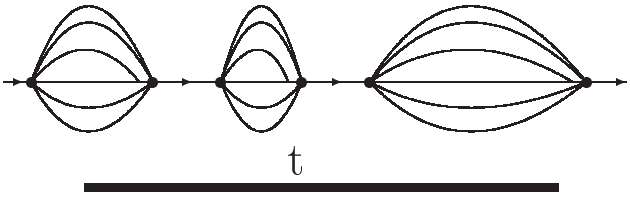
5.4 Synchronization Waiting — Coarse Parallelism and Fine Parallelism

Most of the program is designed as almost the same number of the performance for each processors. So it can get completed simultaneously before the next serial test or parallel step starts. However, the UNIX system is a multiple task multiple user system. Machine works not for one user. There are always multiple jobs run on it. So you can not guarantee each thread you opened could work with the same speed. But the physics parallelism is limited, out this real limit, you have data dependence. For example, to solve a Partial Differential Equation, before one iteration totally completed, you can not start the second iteration, since the second iteration uses the data from the first iteration.

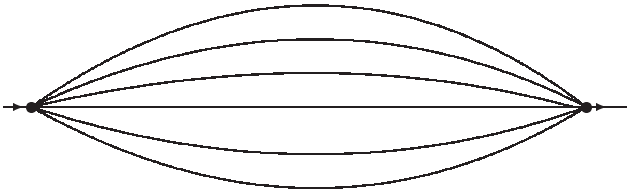
In a exact program, you must put a synchronization point there. If any processor finished his job, MUST wait at the synchronization point until all the processors finished their jobs.

This is an additional overhead to the parallelization, we call it synchronization latency. Logically it is easy to understand that, there are more users on the machine, there is higher probability for waiting. As well as, you user more processors, there is higher probability for waiting too.

Fine-grain parallelism



Coarse-grain parallelism



So that we sugget that:

- When you plan to use Massive Multiple Processors, (used to mean more than 32 processors), you'd better choose coarse parallelism.
- When you plan to use not too many processors and want to simply use shared memory model, you'd can use fine parallelism.
- The mixed coarse-fine parallelism.

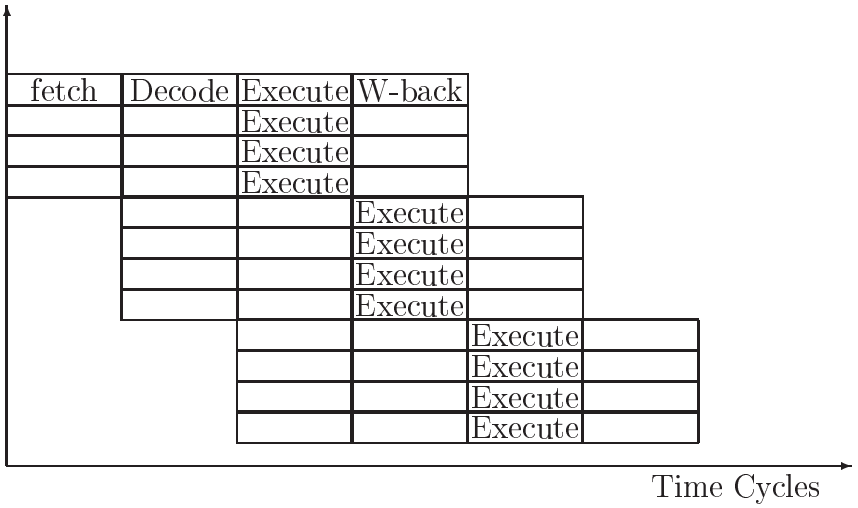
Since the parallel speed up used to get down when user try to use more than 1024 processors, so that if the programmer can parallelize the fine parallelism within the program, to make the fine parallelism works as shared memory few processor parallelized, then the number of processors he can use will be a factor increased.

$$NP_{mix} = NP_{coarse} * NP_{fine}$$

That makes user to get more parallel benefits.

5.5 RISC Processor Pipe Line — Instruction level parallelism

All the current supercomputers are based on the RISC processor (Reduced Instruction Set Computing). The RISC processor is a superscalar processor which allow multiple instructions to be issued simultaneously during each clock cycle. RISC processor has pipeline architecture such that:



The RISC processor peak speed is:

$$\text{(Peak Speed)} = \text{(Frequency)} \times \text{(Number of Floating Units)}$$

Multiple floating units and the additional integer units make the RISC processor very fast. For example, Origin 2000 uses RX1000 processor which has 2 functional units within the processor. So the peak speed is 200x2=400Mflops. The HP/PA8600 processor has 4 floating units within the processor, So the peak speed for 200MHz is 200x4=800Mflops.

However, it is almost impossible to reach the peak speed, since mostly of the cache missing latency. As well as, if your program has not the very fine parallelism which can fill all the floating units full, you can not get high speed too.

- **Loop unrolling:**

  For a simple loop such as:

  DO I=1,N
  A(I)=A(I)+B(I)*C
  ENDDO

We can make it as:

```
DO I=1,N,4
A(I)=A(I)+B(I)*C
A(I+1)=A(I+1)+B(I+1)*C
A(I+2)=A(I+2)+B(I+2)*C
A(I+3)=A(I+3)+B(I+3)*C
ENDDO
```

Since there is not data dependence, the compiler will send each line to each floating unit to run. So the second program can fill all the 2 or 4 floating units fully work. Since this parallelism used to run within a processor, that means all the code and data MUST can stay within L1, even L0 cache. So it is a VERY FINE parallelism (it is used to less than 50 instructions) called instruction level parallelism. This loop change procedure used to be called as **loop unrolling**. A example of the loop unrolling effect can be shown as:

Table 5.4: 100 x 100 matrix to matrix multiplication on HP/SPP2200

Unroll factor	performance(Mflops)	percentage of peak
1	209	26.1%
2	361	45.1%
3	424	53.0%
4	515	64.4%
Veclib	773	96.6%

- Some machine has compiler directives to claim loop unrolling. Some machine has auto-unrolling. However, all of them are not portable. So a good parallel program must include instruction level parallelization. That is called instruction level parallel programming: to unroll the most important loop as fine as possible by programmer's hand.

  For example, in Lattice QCD program there is a : SU(3)×SU(3) routine

```
INTEGER, PARAMETER :: N=1000000
COMPLEX(16), dimension(3,3,N):: w, u, v
w=0.0_high
DO I=1,N
K=J(I)
DO J1=1,3
DO J2=1,3
```

```
DO J3=1,3
w(J2,J1,I)=w(J2,J1,I)+u(J2,J3,I)*v(J3,J1,K)
ENDDO
ENDDO
ENDDO
ENDDO
```

We change this program into a very ugly program such as:

```
INTEGER, PARAMETER :: n=1000000
real(8),dimension(18,m) :: u, v, w
w = 0.0_high
DO i = 1, n
k=j(i)
w(1 ,i) = u(1 ,i) * v(1 ,k) - u(2 ,i) * v(2 ,k) &
+ u(3 ,i) * v(7 ,k) - u(4 ,i) * v(8 ,k) &
+ u(5 ,i) * v(13,k) - u(6 ,i) * v(14,k)

w(2 ,i) = u(1 ,i) * v(2 ,k) + u(2 ,i) * v(1 ,k) &
+ u(3 ,i) * v(8 ,k) + u(4 ,i) * v(7 ,k) &
+ u(5 ,i) * v(14,k) + u(6 ,i) * v(13,k)

w(3 ,i) = u(1 ,i) * v(3 ,k) - u(2 ,i) * v(4 ,k) &
+ u(3 ,i) * v(9 ,k) - u(4 ,i) * v(10,k) &
+ u(5 ,i) * v(15,k) - u(6 ,i) * v(16,k)

w(4 ,i) = u(1 ,i) * v(4 ,k) + u(2 ,i) * v(3 ,k) &
+ u(3 ,i) * v(10,k) + u(4 ,i) * v(9 ,k) &
+ u(5 ,i) * v(16,k) + u(6 ,i) * v(15,k)

w(5 ,i) = u(1 ,i) * v(5 ,k) - u(2 ,i) * v(6 ,k) &
+ u(3 ,i) * v(11,k) - u(4 ,i) * v(12,k) &
+ u(5 ,i) * v(17,k) - u(6 ,i) * v(18,k)

w(6 ,i) = u(1 ,i) * v(6 ,k) + u(2 ,i) * v(5 ,k) &
+ u(3 ,i) * v(12,k) + u(4 ,i) * v(11,k) &
+ u(5 ,i) * v(18,k) + u(6 ,i) * v(17,k)

......
```

```
ENDDO
```

In this new program, each line is independent of the others. Each line can be performed by each floating unit within one processor. So it included the instruction level parallelization.

5.6 Portability

Portability is a dream of numerical programmers. It used to happen to the programmer that, when you finished a program with very hard tunning and make it run well on one supercomputer, then you find that it works bad even does not want to work on the other supercomputers. The reason is obvious. The program is not **Portable**.

It often happened that even one did invest long time to write a parallel program for some supercomputer, before one get much use to get his job done, either the environment changed or the computer company goes out. Then the program becomes useless.

There are many different architectures of the supercomputers. Some supercomputers used a shared-memory model, others used a distributed-memory, or shared-distributed model. Some used a special parallel languages or special language extensions (directives), while others used standard Fortran or c with special message passing package. Some computers used large cache with high bandwidth, the others used small. ····· . How we can make the program running on different architectures with very high user's effective speed?

Portability has not a detail definition yet. However, it means some machine independent properties.

Portablility

- Machine independence:
  - without special language or directives dependence;
  - without special library dependence
- less dependence on the special architecture
  - cache lenth,
  - number of the processors,
  - number of the functional units within processor.
- works on both shared memory machines and distributed memory machines.
- reasonable efficiency.

It needs experiments to get better portability. We only can give some suggestions here:

1. Using standard Fortran, C or C++ language with no special functions.

2. Using MPI (has been accepted by all major supercomputer vendors as a standard) message passing interface with data decomposition for the coarse parallelism.

3. Using openMP (almost being accepted by major supercomputer vendors) for the fine parallelism.

4. Do not use APO; Do not use special directives; Do not use special library.

5. Use high BLAS algorithms and take care of locality.

6. Take care of instruction level parallelism by hand.

After you took care all of the factors, the program could be very portable. For example, a portable QCD Monte-Carlo program test shown:

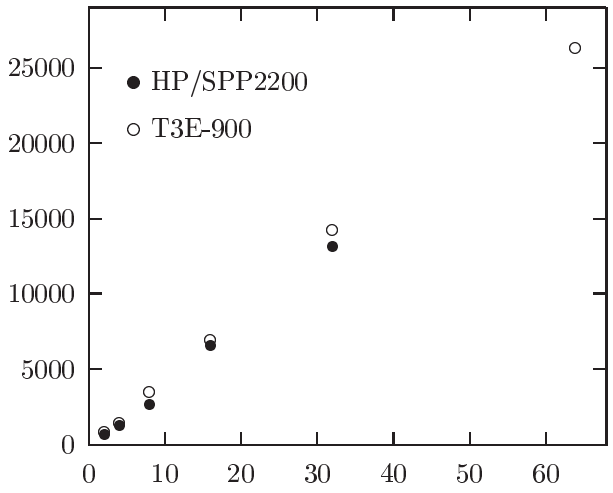


Table 5.5: Monte Carlo code performance efficiency with 8 processors

	T3E900	Or. 2000	Exe. 1200	Exe. 2000	SPP2200
Peak (Mflops)	900	390	120	400	800
Dimension m	2	2	2	4	4
Effective (Mflops)	439.1	170.9	52.1	172.8	363.2
Bandwidth (Mb/sec)	97.2	9.4	14.5	33.0	54.5
Efficiency	48.8%	43.8%	43.4%	43.5%	45.3%

Chapter 6

About LINUX PC Cluster

- The PC Clusters become a very important type in the high performance parallel supercomputing.
- In June 2003, the Top500 list includes 29.7of the clusters are academic/research made, only 8.9
- After the CPU frequency gets more than 1GHz, the Beowulf ethernet communication makes parallel supercomputing being nonsense. General rule of thumb is 1GHz requires, at least, 1Gbit/s communication network.
- P4 processors with SSE technique addes cheap gigbit network goes to PMS (Poor Men's Supercomputer).

1. Mother Board: P4SAA-E7205, 533FSB,DDR-266,6PCI $196
2. 2 CPU: XEON 2.0GHz, 400FSB, 512K Cache 2x$134
3. Memory: 1GB DDR 3200, 400MHz $134
4. Case: incl. Power 500W $55
5. Disk: EIDE 80GB $66
6. GigE: 4xPCI cards 4x$29 $126
7. Summation: Cost per dual node (16Gflops peak with SSE) $845
8. < $1.0/Mflops measured, < $0.5/Mflops sustained, < $0.1/Mflops peak.
9. Waste heat: about 110W per node.

   110KW/sustain Tflops

- The best running time measured performance on the P4 cluster in 2002 is 500Mflops/second/processo
  It is not too bad! (The peack of the P4 2GHz processor with SSE technique is 8GFlops/second/processor)

- The frequency of the FSB (Front Side Bus) together with the memory type and frequency (RAMBUS - DDR) determines the memory to cache data rate. So far, ADM, Q2/2003 Athlon is 400MHz; Intel, Q2/2003 XEON 533MHz; P4 800MHz.
- Memory speed: DDR-SDRAM: 16bytes x clock (x channels) RAMBUS: 16bits x clock (x channels)
- After PCI bus bottlenek PMS will be a very good choise.