Multilevel Parallel Programming

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163

Multilevel Parallel Programming

Table of Contents:

Introduction

- The Debate Rages: Distributed vs. Shared Memory
- Why Not Both?
- Modern Distributed/Shared Memory Parallel Architectures
- Building Codes to Exploit DSM Systems

The Debate Rages: Distributed vs. Shared Memory

- For years, supercomputer vendors have been offering two very different types of machines: *distributed memory systems* and *shared memory systems*.
- In distributed memory systems, such as the IBM SP2 or the Cray T3E, each processor has its own memory system and communicates with other processors using some kind of network interface. Parallel programming is essentially limited to message passing, but efficient use of very large numbers of processors is possible.
- In shared memory machines, such as the Cray Y-MP or the Sun Starfire (UE10k), all CPUs have equal access to a large, shared memory system through a shared memory bus or crossbar switch. Parallel programming can be done in a variety of ways, including compiler parallelization with directives as well as message passing. The effective use of more than modest numbers of processors with shared memory is limited by memory bus bandwidth (and cache coherence strategies on cache-based architectures).

Distributed vs. Shared Memory

Distributed Memory

- Pros:
  - Scales to very large numbers of processors
  - Often parallel I/O paths
- Cons:
  - Difficult to program and debug
  - Limited flexibility for large memory jobs
  - Often requires multiple system images
- Programming models:
  - Message passing (Cray SHMEM, PVM, MPI)

Shared Memory

- Pros:
  - Relatively easy to program and debug
  - Greater flexibility for large memory jobs
  - Single system image
- Cons:
  - Does not scale to very large numbers of processors
  - Often a single I/O path
- Programming models:
  - Compiler directives (HPF, OpenMP)
  - Message passing (Cray SHMEM, PVM, MPI)
  - Hand-coded threads (pthreads, etc.)

Why Not Both?

- Several modern supercomputer systems now take a hybrid approach, where parts of the system look like a shared memory system and other parts look like a distributed memory system. These are typically called *distributed/shared memory systems* or DSM systems.
- These modern architectures typically have a "block" which consists of a modest number of processors attached to a memory system. These blocks are then connected by some type of network.
- The main differences between these modern architectures are:
  - The block granularity (i.e. how many CPUs per block)
  - The network topology (switched, hypercube, ring, mesh, torus, fat tree, etc.)
  - Whether the system looks more like a distributed or shared memory system from a user's point of view

Modern Distributed/Shared Memory Parallel Architectures

- Compaq HPC320
- HP Exemplar
- IBM SP3
- SGI Origin 2000
- Clusters of SMPs

Compaq HPC320

- Maximum of 32 CPUs in a single system image
- Block: 4 DEC Alpha EV6 processors and a memory bank on a crossbar switch
- Network topology: Multi-tiered switch
- User view: Shared memory

HP Exempler

- Maximum of 64 CPUs in a single system image.
- Block: 4 or 8 PA-RISC 7100 processors and a memory bank on a crossbar switch
- Network topology: Ring
- User view: Shared memory

IBM SP3

- Maximum of 16 CPUs in a single system image; large installations can have hundreds of nodes (eg. ASCI Blue Pacific).
- Block: 8 or 16 IBM Power^3 processors and a memory bank on a shared memory bus
- Network topology: Multi-tier switch
- User view: Distributed memory

SGI Origin 2000

- Maximum of 256 CPUs in a single system image
- Block: 2 MIPS R12000 processors and a memory bank on a crossbar switch
- Network topology: “Fat” hypercube
- User view: Shared memory

Clusters of SMPs

- This includes “Beowulf” clusters of commodity Intel or Alpha based SMPs , as well as clusters of larger SMPs or “shared-memory-like” DSM systems (eg. ASCI Blue Mountain).
- Block: Varies with system type
- Network topology: Varies, typically switch or multi-tier switch
- User view: Distributed memory

Note: Technically the IBM SP3 is a cluster of SMPs.

Building Codes to Exploit DSM Systems

- While it is possible to to treat DSM systems as strictly distributed memory systems (or strictly as shared memory systems in some cases), there is great potential for added scalability by writing code to specifically take advantage of both distributed and shared memory.
- The most commonly used approach is to have some sort of multi-threaded computation kernel which runs on a shared-memory "block", and do message passing between blocks.
- This requires two things:
  - Some form of relatively coarse grain domain decomposition which can be parallelized using message passing techniques
  - Loops or other fine-grained structures within each block of the domain-decomposed problem which lend themselves to being parallelized using shared memory techniques.

Interlude 1: Solving Laplace’s Equation in 2D

- Laplace’s Equation
- Finite Difference Approximations for Laplace’s Equation
- Initial and Boundary Conditions
- Domain Layout
- Serial Implementations of a Laplace Solver
- Performance Characteristics of Serial Implementation

Laplace’s Equation

- Laplace’s equation is the model equation for diffusive processes such as heat flow and viscous stresses. It makes a good test case for approaches to numerical programming.
- In two dimensions, Laplace’s equation takes the following forms:

$$\nabla^2 u = 0$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Finite Difference Approximations

- If we assume that we are approximating Laplace's equation on a regular grid (i.e. dx and dy are constant), we can substitute the following finite differences for the partial derivatives:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + O((\Delta x)^2)$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} + O((\Delta y)^2)$$

Finite Difference Approximations (con't)

- If we now substitute the above finite differences into Laplace's equation and rearrange, we can get the following:

$$\frac{u^{n}_{i+1,j} - 2u^{n+1}_{i,j} + u^{n}_{i-1,j}}{(\Delta x)^2} + \frac{u^{n}_{i,j+1} - 2u^{n+1}_{i,j} + u^{n}_{i,j-1}}{(\Delta y)^2}$$

$$= O((\Delta x)^2, (\Delta y)^2)$$

- We can now assume that the $O((\Delta x)^2,(\Delta y)^2)$ terms are small enough to be neglected, and that Δx is equal to Δy.

OSC
Innovations in computing, networking, and education

Finite Difference Approximations (con't)

- If we now solve this equation for $u_{i,j}^{n+1}$ and subtract $u_{i,j}^{n}$ from both sides, we arrive at the following:

$$\Delta u_{i,j}^{n+1} = \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4} - u_{i,j}^{n}$$

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \Delta u_{i,j}^{n+1}$$

Innovations in computing, networking, and education

Initial and Boundary Conditions

- For the purposes of this discussion, we will assume the following initial and boundary conditions:

$$0 \le x \le L$$

$$0 \le y \le L$$

$$u(0, y) = 0$$

$$u(L, y) = u_{\max} \sin(\pi \frac{y}{L})$$

$$u(x,0) = 0$$

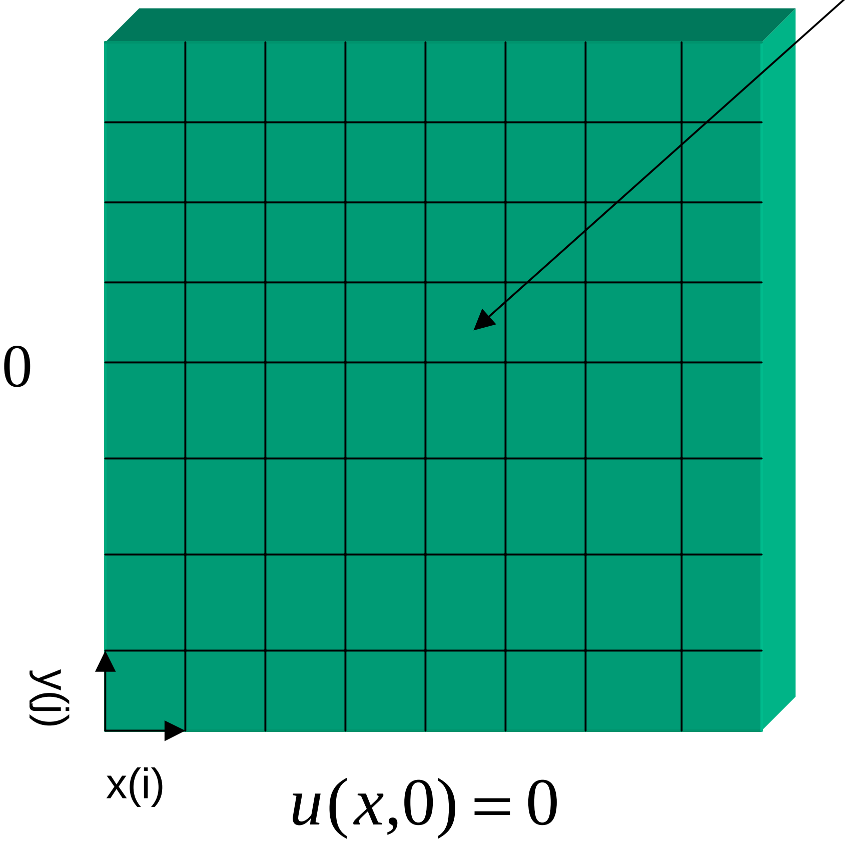
$$u(x, L) = 0$$

$$u_{i,j}^{0} = 0$$

Domain Layout

$$u(x,L)=0$$

$$\nabla^2 u = 0$$



$$u(0,y)=0$$

$$u(L,y)=u_{\max}\sin(\pi\frac{y}{L})$$

$$u(x,0)=0$$

Serial Implementations of a Laplace Solver

- There are two possible approaches in implementing the finite difference solution to Laplace's equation described above:
  - The vectorized approach: all $\Delta u_{i,j}$'s are computed, then a Δu_{max} is found, then all $u_{i,j}$'s are updated. This approach performs well on vector-based architectures such as the Cray T90 series, but also performs very badly on cache-based microprocessor architectures because of memory bandwidth limitations and poor cache reuse.
  - The cache-friendly approach: $\Delta u_{i,j}$ calculations, Δu_{max} comparisons, and $u_{i,j}$ updates are performed incrementally. This approach performs better on cache-based microprocessor architectures because it tends to reuse cache, but it performs very badly on vector architectures because recurrance relationships appear in the inner loops which vectorizing compilers cannot convert into vector operations. Performance on microprocessor architectures is still limited by memory bandwidth.
- Implementations of both approaches are shown below.

Serial Implementation -- Vectorized

```
      program lpvect
      integer imax,jmax,im1,im2,jm1,jm2,it,itmax
      parameter (imax=2001,jmax=2001)
      parameter (im1=imax-1,im2=imax-2,jm1=jmax-1,jm2=jmax-2)
      parameter (itmax=100)
      real*8 u(imax,jmax),du(imax,jmax),umax,dumax,tol,pi
      parameter (umax=10.0,tol=1.0e-6,pi=3.1415)

! Initialize
      do j=1,jmax
         do i=1,imax-1
            u(i,j)=0.0
            du(i,j)=0.0
         enddo
         u(imax,j)=umax*sin(pi*float(j-1)/float(jmax-1))
      enddo
```

Serial Implementation -- Vectorized (con’t)

```
! Main computation loop
      do it=1,itmax
         dumax=0.0
         do j=2,jm1
            do i=2,im1
               du(i,j)=0.25*(u(i-1,j)+u(i+1,j)+
     +                  u(i,j-1)+u(i,j+1))-u(i,j)
            enddo
         enddo
         do j=2,jm1
            do i=2,im1
               dumax=max(dumax,abs(du(i,j)))
            enddo
         enddo
```

Serial Implementation -- Vectorized (con’t)

```
do j=2,jm1
      do i=2,im1
         u(i,j)=u(i,j)+du(i,j)
      enddo
   enddo
   write (1,*) it,dumax
enddo
stop
end
```

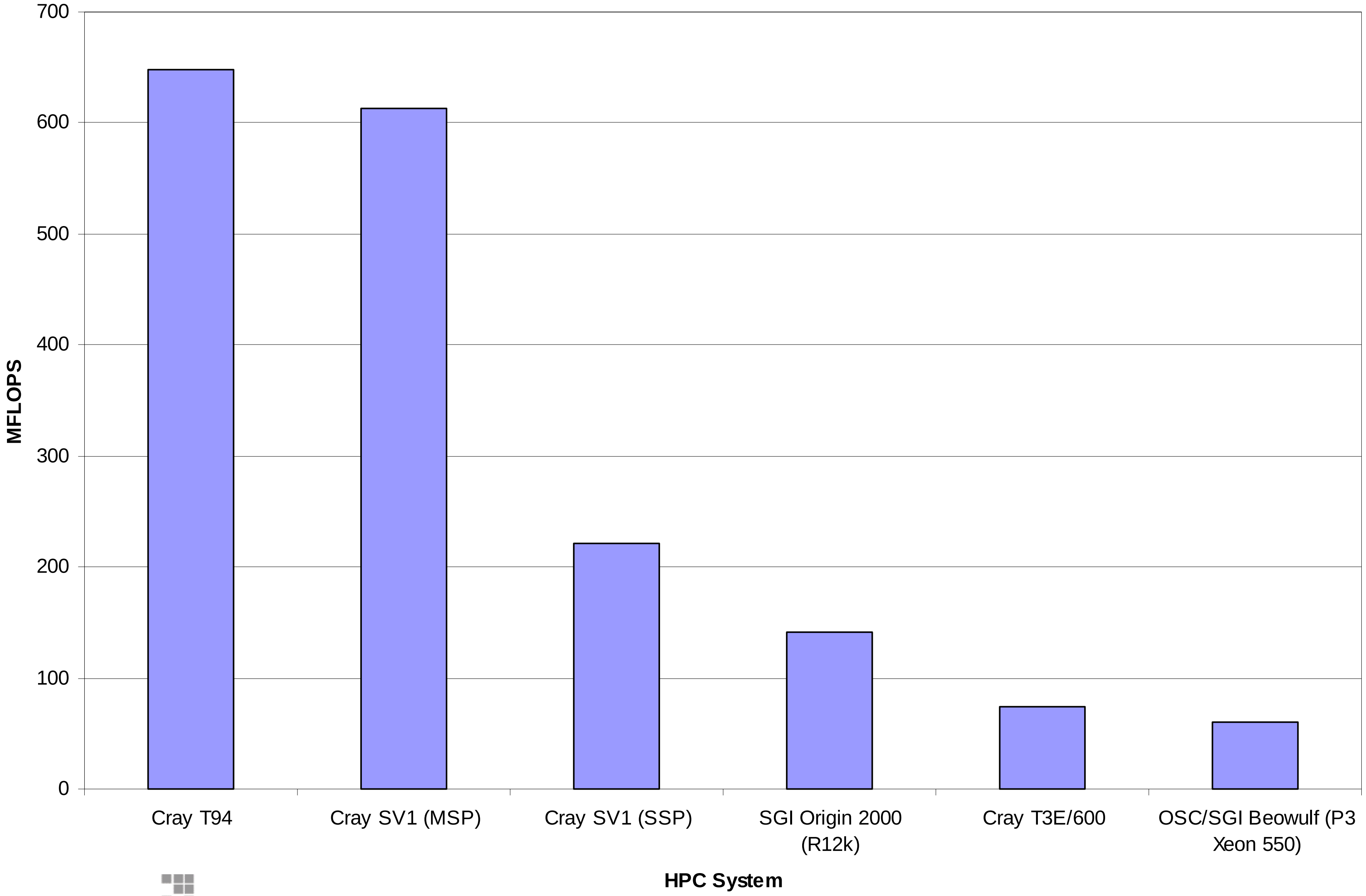
Serial Implementation -- Cache-Friendly

```
 program lpcache
      integer imax,jmax,im1,im2,jm1,jm2,it,itmax
      parameter (imax=2001,jmax=2001)
      parameter (im1=imax-1,im2=imax-2,jm1=jmax-1,jm2=jmax-2)
      parameter (itmax=100)
      real*8 u(imax,jmax),du(imax,jmax),umax,dumax,tol,pi
      parameter (umax=10.0,tol=1.0e-6,pi=3.1415)

! Initialize
      do j=1,jmax
         do i=1,imax-1
            u(i,j)=0.0
            du(i,j)=0.0
         enddo
         u(imax,j)=umax*sin(pi*float(j-1)/float(jmax-1))
      enddo
```

Serial Implementation --Cache-Friendly (con’t)

```
! Main computation loop
      do it=1,itmax
         dumax=0.0
         do j=2,jm1
            do i=2,im1
               du(i,j)=0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)
     +                       +u(i,j+1))-u(i,j)
               dumax=max(dumax,abs(du(i,j)))
               u(i,j)=u(i,j)+du(i,j)
            enddo
         enddo
         write (1,*) it,dumax
      enddo
      stop
      end
```

Serial Performance Characteristics



OSC Innovations in computing, networking, and education

OpenMP Parallel Programming

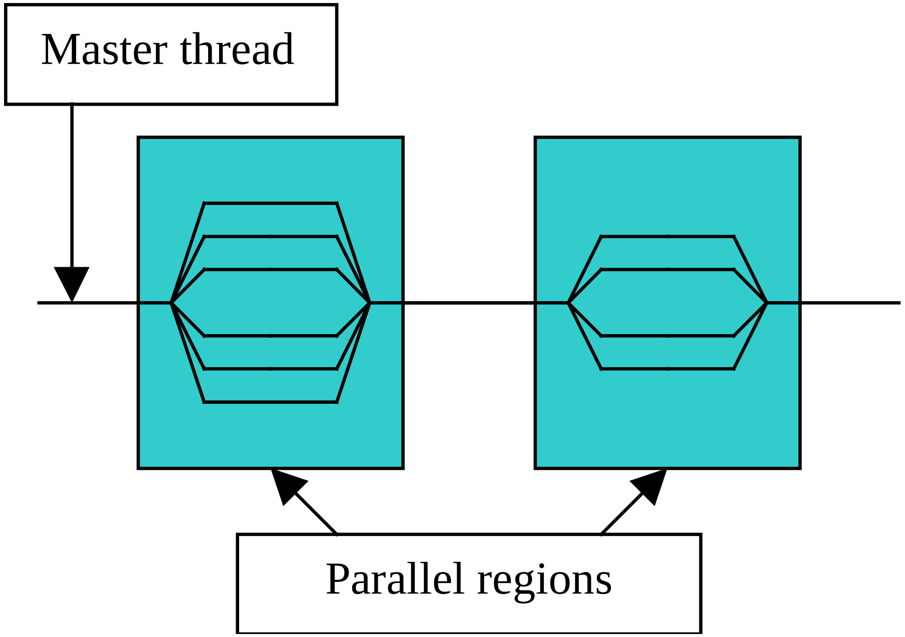
- Introduction
- The OpenMP Programming Model
- The Basics of OpenMP
- Synchronization Constructs
- OpenMP Problem Set

Introduction to OpenMP

- OpenMP is used for writing multithreaded (parallel processing) applications in a shared memory environment
- It consists of a set of compiler directives and library routines
- Relatively easy to create multi-threaded applications in Fortran, C and C++
- Standardizes the last 15 or so years of SMP development and practice
- Currently supported by
  - Hardware vendors
    - Intel, HP, SGI, Compaq, Sun, IBM
  - Software tools vendors
    - KAI, PGI, PSR, APR, Absoft
  - Applications vendors
    - ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software, ...
- Support is common and growing

The OpenMP Programming Model

- A *master* thread spawns *teams* of threads as needed
- Parallelism is added incrementally; the serial program evolves into a parallel program



The OpenMP Programming Model

- Programmer inserts OpenMP directives (Fortran comments, C #pragmas) at key locations in the source code.
- Compiler interprets these directives and generates library calls to parallelize code regions.

Serial:

```
void main(){
  double x[1000];
  for (int i=0; i<1000; i++){
    big_calc(x[i]);
  }
}
```

Parallel:

```
void main(){
  double x[1000];
#pragma omp parallel for
  for (int i=0; i<1000; i++){
    big_calc(x[i]);
  }
}
```

Split up loop iterations among a team of threads

The OpenMP Programming Model

- Number of threads can be controlled from within the program, or using the environment variable **`OMP_NUM_THREADS.`**
- The programmer is responsible for managing synchronization and data dependencies!

The Basics of OpenMP

- General syntax rules
- The parallel region
- OpenMP directive clauses
- Work-sharing constructs
- Environment variables
- Runtime environment routines

General Syntax Rules

- Most OpenMP constructs are compiler directives or C `pragmas`
  - For C and C++, `pragmas` take the form

```
#pragma omp construct [clause [clause]...]
```

  - For Fortran, directives take one of the forms:

```
C$OMP construct [clause [clause]...]
!$OMP construct [clause [clause]...]
*$OMP construct [clause [clause]...]
```

- Since these are directives, compilers that don’t support OpenMP can still compile OpenMP programs (serially, of course!)

The Parallel Region

- The fundamental construct that initiates parallel execution
- Fortran syntax:

```
c$omp parallel
c$omp& shared(var1, var2, …)
c$omp& private(var1, var2, …)
c$omp& lastprivate(var1, var2, …)
c$omp& firstprivate(var1, var2, …)
c$omp& reduction(operator|intrinsic:var1, var2, …)
c$omp& if(expression)
c$omp& default(private|shared|none)

   a structured block of code

c$omp end parallel
```

The Parallel Region

- C/C++ syntax:

```
#pragma omp parallel                                 \
        private (var1, var2, …)                      \
        shared (var1, var2, …)                       \
        firstprivate(var1, var2, …)                  \
        copyin(var1, var2, …)                        \
        reduction(operator:var1, var2, …) \
        if(expression)                               \
        default(shared|none)                         \
{
   …a structured block of code…
}
```

The Parallel Region

- The number of threads created upon entering the parallel region is controlled by the value of the environment variable **`OMP_NUM_THREADS`.**
  - Can also be controlled by a function call from within the program.
- Each thread executes the block of code enclosed in the parallel region.
- In general there is no synchronization between threads in the parallel region!
  - Different threads reach particular statements at unpredictable times.
- When all threads reach the end of the parallel region, all but the master thread go out of existence and the master continues on alone.

The Parallel Region

- Each thread has a thread number, which is an integer from 0 (the master thread) to the number of threads minus one.
  - Can be determined by a call to **`omp_get_thread_num()`**
- Threads can execute different paths of statements in the parallel region
  - Typically achieved by branching on the thread number:

```
#pragma omp parallel
{
   myid = omp_get_thread_num();
   if (myid == 0)
     do_something();
   else
      do_something_else(myid);
}
```

OpenMP Directive Clauses

- **`default(shared|private|none)`**
  - Specifies default scoping for variables in parallel code.
- **`shared(var1,var2,…)`**
  - Variables to be shared among all threads (threads access same memory locations).
- **`private(var1,var2,…)`**
  - Each thread has its own copy of the variables for the duration of the parallel code.
- **`firstprivate(var1,var2,…)`**
  - Private variables that are initialized when parallel code is entered.
- **`reduction(operator|intrinsic:var1,var2…)`**
  - Ensures that a reduction operation (e.g., a global sum) is performed safely.

The `private`, `shared`, and `default` clauses

`private & default`

```
c$omp  parallel shared(a)
c$omp& private(myid,x)
     myid=omp_get_thread_num()
     x = work(myid)
     if (x < 1.0) then
       a(myid) = x
     end if
c$omp end parallel
```

Equivalent is:

```
c$omp parallel do default(private)
c$omp& shared(a)
      …
```

- Each thread has its own private copy of `x` and `myid`
- Unless `x` is made private, its value is indeterminate during parallel operation
- Values for private variables are undefined at beginning and end of the parallel region!
- `default` clause automatically makes `x` and `myid` private.

firstprivate

- Variables are private (local to each thread), but are initialized to the value in the preceding serial code.

```
     program first
      integer myid,c
      c=98
c$omp  parallel private(myid)
c$omp& firstprivate(c)
      myid=omp_get_thread_num()
      print *,’T:’,myid,’ c=‘,c
c$omp end parallel
      end
-----------------------------------
T:1 c=98
T:3 c=98
T:2 c=98
T:0 c=98
```

- Each thread has a private copy of `c`, initialized with the value 98

OpenMP Work-Sharing Constructs

- Parallel `for/DO`
- Placed inside parallel regions
- Distribute the execution of associated statements among existing threads
  - No *new* threads are created.
- No implied synchronization between threads at the start of the work sharing construct!

OpenMP work-sharing constructs - `for/DO`

- Distribute iterations of the immediately following loop among threads in a team

```
#pragma omp parallel shared(a,b) private(j)
{
   #pragma omp for
     for (j=0; j<N; j++)
        a[j] = a[j] + b[j];
}
```

- By default there is a barrier at the end of the loop
  - Threads wait until all are finished, then proceed.
  - Can use the **`nowait`** clause to allow threads to continue without waiting.

Detailed syntax - for

```
#pragma omp for [clause [clause]…]
   for loop
```

where each clause is one of

- private(*list*)
- firstprivate(*list*)
- lastprivate(*list*)
- reduction(*operator: list*)
- ordered
- schedule(*kind [, chunk_size]*)
- nowait

Detailed syntax - DO

```
c$omp do [clause [clause]…]
   do loop
[c$omp end do [nowait]]
```

where each `clause` is one of

- private(*list*)
- firstprivate(*list*)
- lastprivate(*list*)
- reduction(*operator: list*)
- ordered
- schedule(*kind [, chunk_size]*)

- For Fortran 90, use `!$OMP` and F90-style line continuation.

reduction(operator|intrinsic:var1[,var2])

- Allows safe global calculation or comparison.
- A private copy of each listed variable is created and initialized depending on `operator` or `intrinsic` (e.g., 0 for +).
- Partial sums and local mins are determined by the threads in parallel.
- Partial sums are added together from one thread at a time to get gobal sum.
- Local `mins` are compared from one thread at a time to get `gmin`.

```
c$omp do shared(x) private(i)
c$omp&  reduction(+:sum)
      do i = 1, N
         sum = sum + x(i)
      enddo


c$omp do shared(x) private(i)
c$omp&  reduction(min:gmin)
      do i = 1,N
         gmin = min(gmin,x(i))
      end do
```

`reduction(operator|intrinsic:var1[,var2])`

- Listed variables must be `shared` in the enclosing parallel context.
- In Fortran
  - `operator` can be **`+, *, -, .and., .or., .eqv., .neqv.`**
  - `intrinsic` can be **`max, min, iand, ior, ieor`**
- In C
  - `operator` can be **`+, *, -, &, ^, |, &&, ||`**
  - pointers and reference variables are not allowed in reductions!

OpenMP Environment Variables

- **`OMP_NUM_THREADS`**
  - Sets the number of threads requested for parallel regions.
- **`OMP_SCHEDULE`**
  - Set to a string value which controls parallel loop scheduling at runtime.
  - Only loops that have schedule type `RUNTIME` are affected.

OpenMP Environment Variables

- Examples:

```
origin$ export OMP_NUM_THREADS=16

origin$ setenv OMP_SCHEDULE “guided,4”
```

Note: values are case-insensitive!

OpenMP Runtime Environment Routines

- **`(void) omp_set_num_threads(int`** ***`num_threads`*** **`)`**
  - Sets the number of threads to use for subsequent parallel regions.
- **`int omp_get_num_threads()`**
  - Returns the number of threads currently in the team.
- **`int omp_get_thread_num()`**
  - Returns the thread number, an integer from `0` to the number of threads minus `1.`
- **`int omp_get_num_procs()`**
  - Returns the number of physical processors available to the program.

OpenMP Runtime Environment Routines

- In Fortran, routines that return a value (integer or logical) are functions, while those that set a value (*i.e.*, take an argument) are subroutines.

- In C, be sure to **`#include <omp.h>`**

- Changes to the environment made by function calls have precedence over the corresponding environment variables.
  - For example, a call to **`omp_set_num_threads()`**overrides any value that **`OMP_NUM_THREADS`** may have.

Loop Nest Parallelization Possibilities

All examples shown run on 8 threads with `schedule(static)`

- Parallelize the outer loop:

```
!omp parallel do private(i,j) shared(a)
     do i=1,16
       do j=1,16
         a(i,j) = i+j
       enddo
     enddo
```

- Each thread gets two values of `i` (T0 gets `i`=1,2; T1 gets `i`=3,4, etc.) and *all* values of `j`

Loop Nest Parallelization Possibilities

- Parallelize the inner loop:

```
     do i=1,16
!omp parallel do private(j) shared(a,i)
       do j=1,16
         a(i,j) = i+j
       enddo
     enddo
```

- Each thread gets two values of `j` (T0 gets `j`=1,2; T1 gets `j`=3,4, etc.) and *all* values of `i`

OpenMP Synchronization Constructs

- `critical`
- `atomic`
- `barrier`
- `master`

OpenMP Synchronization - `critical` Section

- Ensures that a code block is executed by only one thread at a time in a parallel region
- Syntax:

```
#pragma omp critical [(name)]
        structured block
```

```
!$omp critical [(name)]
      structured block
!$omp end critical [(name)]
```

- When one thread is in the critical region, the others wait until the thread inside exits the critical section.
- *`name`* identifies the critical region.
- Multiple critical sections are independent of one another unless they use the same name.
- All unnamed critical regions are considered to have the same identity.

OpenMP Synchronization - `critical` Section Example

```
integer cnt1, cnt2

c$omp parallel private(i)
c$omp& shared(name1,name2)

c$omp do
   do i = 1, n
      …do work…
      if(condition1)then
c$omp critical (name1)
         cnt1 = cnt1+1
c$omp end critical (name1)
      else
c$omp critical (name1)
         cnt1 = cnt1-1
c$omp end critical (name1)
      endif
      if(condition2)then
c$omp critical (name2)
         cnt2 =cnt2+1
c$omp end critical (name2)
      endif
   enddo
c$omp end parallel
```

OpenMP - Critical Section Problem

Is this correct?

```
…

c$omp parallel do
      do i = 1,n
        if (a(i).gt.xmax) then
c$omp critical
        xmax = a(i)
c$omp end critical
        endif
      enddo
…
```

What about this?

```
…

c$omp parallel do
      do i = 1,n
c$omp critical
        if (a(i).gt.xmax) then
          xmax = a(i)
        endif
c$omp end critical
      enddo
…
```

OpenMP Synchronization - `atomic` Update

- Prevents a thread that is in the process of (1) accessing, (2) changing, and (3) restoring values in a shared memory location from being interrupted at any stage by another thread.
- Syntax:

```
#pragma omp atomic
         statement
```

```
!$omp atomic
      statement
```

- Alternative to using the `reduction` clause (it applies to same kinds of expressions).
- Directive in effect only for the code statement immediately following it.

OpenMP Synchronization - `atomic` Update

```
integer, dimension(8) :: a,index
data index/1,1,2,3,1,4,1,5/

c$omp parallel private(i),shared(a,index)
c$omp do
  do i = 1, 8
c$omp  atomic
    a(index(I)) = a(index(I)) + index(I)
  enddo
c$omp end parallel
```

OpenMP Synchronization - `barrier`

- Causes threads to stop until all threads have reached the barrier.
- Syntax:

```
!$omp barrier
```

```
#pragma omp barrier
```

- A red light until all threads arrive, then it turns green.
- Example:

```
c$omp parallel
c$omp do
      do i = 1, N
        <assignment>
c$omp barrier
        <dependent
   work>
      enddo
c$omp end parallel
```

OpenMP Synchronization - `master` Region

- Code in a `master` region is executed only by the master thread.
- Syntax:

```
#pragma omp master
        structured block
```

```
!$omp master
        structured block
!$omp end master
```

- Other threads skip over entire master region (no implicit barrier!).

OpenMP Synchronization - `master` Region

```
!$omp parallel shared(c,scale)
  &
!$omp private(j,myid)
      myid=omp_get_thread_num()
!$omp master
   print *,’T:’,myid,’ enter
  scale’
   read *,scale
!$omp end master
!$omp barrier
!$omp do
   do j = 1, N
      c(j) = scale * c(j)
   enddo
!$omp end do
!$omp end parallel
```

OpenMP Problem Set

- Write a program where each thread prints the message ‘Hello World!’, along with its thread ID number and the total number of threads used. Run with 8 threads and run your program several times. Does the order of the output change? Repeat using 4,16, 33 and 50 threads
- Modify your solution to Problem 1 so that only even-numbered threads print out the information message.
- Write a program that declares an array A to have 16000 integer elements and initialize A so that each element has its index as its value. Then create a real array B which will contain the running average of array A. That is,

  ```
  B(I)=(A(I-1) + A(I) +A(I+1)/3.0
  ```

  except at the end points. Your code should do the initialization of A and the running average in parallel using 8 threads. Experiment with all four of scheduling types for the running average loop by timing the loop with different schedules.

OpenMP Problem Set (con’t)

- Write a program so that the parallel threads print out ‘Backwards’ and their thread ID number in **reverse order** of thread number. That is, each time your program is run the last thread prints out first, then the second to last and so on. There are at least five different ways to solve this problem. Find as many as you can.

Interlude 2: Laplace Solver Using OpenMP

- Candidate Loops for OpenMP Parallelization
- Applying OpenMP to the Serial Laplace Solver
- OpenMP Implementation of Laplace Solver
- Performance Characteristics of OpenMP Implementation

Candidate Loops for OpenMP Parallelization

- If we examine the cache-friendly version of the serial Laplace solver shown earlier, there are two loop structures which can and should be parallelized using OpenMP:
  - *The initialization loop in `j` at the beginning of the program*: While this loop is only run once, it is necessary to run it in parallel to ensure the proper memory placement on ccNUMA systems like the SGI Origin 2000.
  - *The `j` loop within the main iterative loop*: This loop comprises the bulk of the work in the program, and clearly should be parallelized if possible.
- The inner loops in `i` within the above loops should <u>not</u> be parallelized, as this run the risk of causing unnecessary cache misses by breaking up stride-1 memory accesses.
- The main iterative loop should not be parallelized because it is essentially a time-stepping loop and thus has a recurrance relationship (each iteration depends on the one before it).

Applying OpenMP to the Serial Laplace Solver

- The entirety of the program can be wrapped in an OpenMP parallel region (`!$OMP PARALLEL…!$OMP END PARALLEL`), so long as private variables such as the counters `i` and j are declared private and sequences which should only be done by one thread are wrapped in `!$OMP SINGLE` or `!$OMP MASTER` directives.
- The outer `j` loop of the initialization can be wrapped with a simple `!$OMP DO` directive.
- The j loop in the main computation can also be wrapped with a `!$OMP DO` directive, but it requires an additional `REDUCTION` clause due to the use of the `max()` instrinsic.

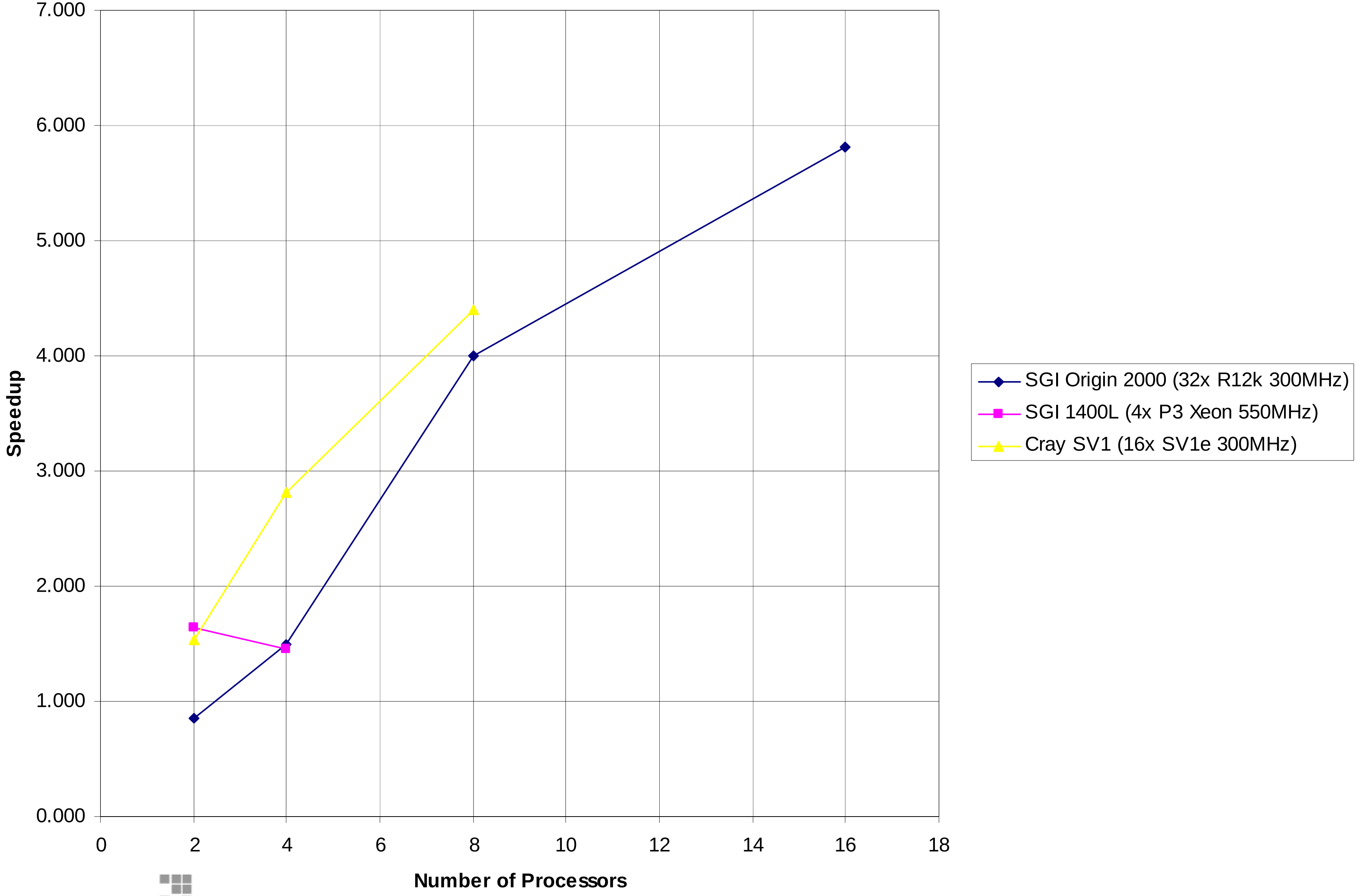
OpenMP Implementation of a Laplace Solver

```
      program lpomp
      integer imax,jmax,im1,im2,jm1,jm2,it,itmax
      parameter (imax=2001,jmax=2001)
      parameter (im1=imax-1,im2=imax-2,jm1=jmax-1,jm2=jmax-2)
      parameter (itmax=100)
      real*8 u(imax,jmax),du(imax,jmax),umax,dumax,tol,pi
      parameter (umax=10.0,tol=1.0e-6,pi=3.1415)
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,j)
! Initialize -- done in parallel to force "first-touch”
! distribution on ccNUMA machines (i.e. O2k)
!$OMP DO
      do j=1,jmax
         do i=1,imax-1
            u(i,j)=0.0
            du(i,j)=0.0
         enddo
         u(imax,j)=umax*sin(pi*float(j-1)/float(jmax-1))
      enddo
!$OMP END DO
```

OpenMP Implementation (con’t)

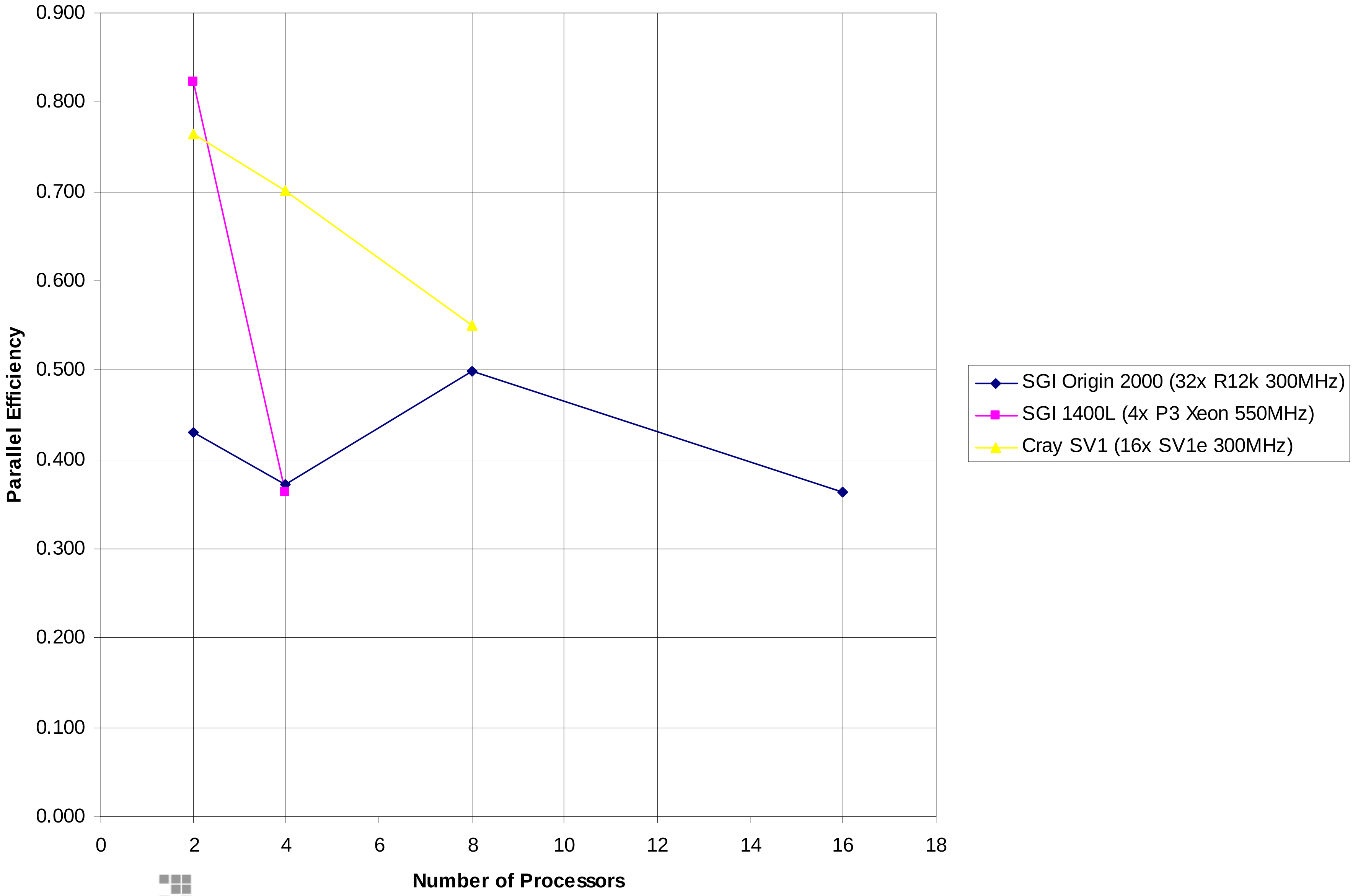
```
! Main computation loop
      do it=1,itmax
!$OMP MASTER
         dumax=0.0
!$OMP END MASTER
!$OMP DO REDUCTION(max:dumax)
         do j=2,jm1
            do i=2,im1
               du(i,j)=0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)
     +                       +u(i,j+1))-u(i,j)
               dumax=max(dumax,abs(du(i,j)))
               u(i,j)=u(i,j)+du(i,j)
            enddo
         enddo
!$OMP END DO
!$OMP MASTER
         write (1,*) it,dumax
!$OMP END MASTER
      enddo
      stop
      end
```

Performance Characteristics of OpenMP Implementation



Innovations in computing, networking, and education

Performance Characteristics of OpenMP Implementation (con’t)



OSC
Innovations in computing, networking, and education

MPI Programming Review

- Introduction
- MPI Program Structure
- Message Passing
- Point-to-Point Communications
- Collective Communication
- Virtual Topologies

Introduction to MPI

- MPI is an *standard* library of message passing routines which allows transfer of data between individual processes, collective communication among all processes, collective calculations among all processes, the design of sophisticated data types for program use and data transfer, and the creation of virtual topologies, among other capabilities.
- The prime goals of of MPI are:
  - To provide **source-code portability**
  - To allow efficient implementation
  - Support for heterogeneous parallel architectures

MPI Program Structure

- The MPI communicator is defined to be the group of processes that are allowed to communicate with each other

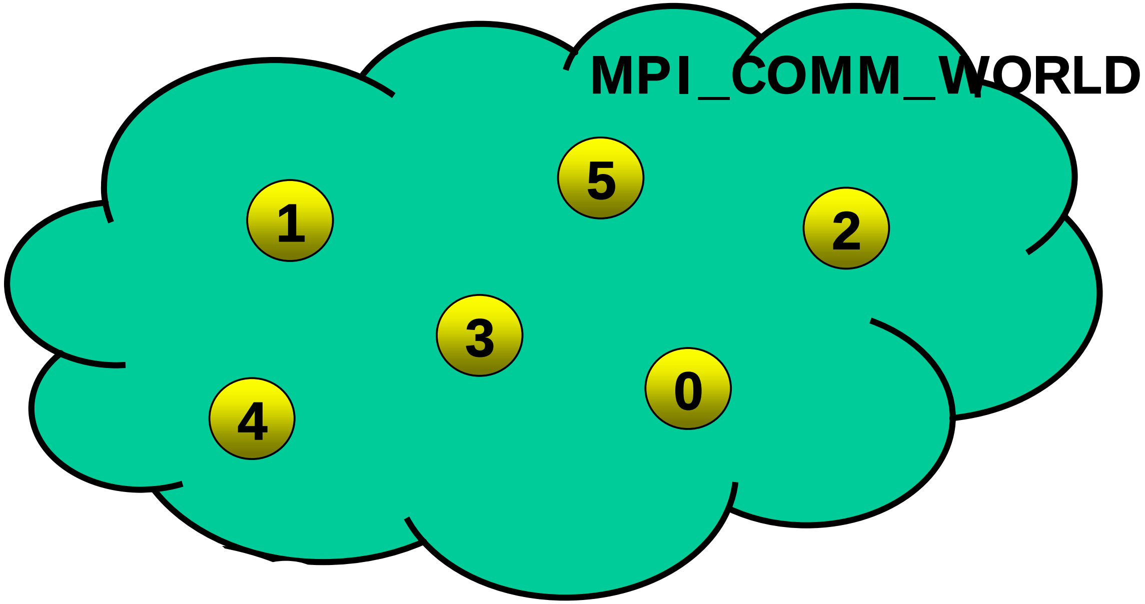
- All MPI communication calls have a communicator argument

- Most often you will use `MPI_COMM_WORLD`

  - Defined when you call `MPI_Init`

  - It is all of your processors...

MPI_COMM_WORLD Communicator



Header Files

- MPI constants and internal variables are defined here

C:

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

Fortran:

```
include ‘mpif.h’
```

MPI Function Format

C:

```
error = MPI_Xxxxx(parameter,…);
MPI_Xxxxx(parameter,…);
```

Fortran:

```
CALL MPI_XXXXX(parameter,…,IERROR)
```

Initializing MPI

- Must be the first routine called (only once)

C:

```
int MPI_Init(int argc, char***argv)
```

Fortran:

```
CALL MPI_INIT(IERROR)

INTEGER IERROR
```

Communicator Size

- How many processes are contained within a communicator

C:

```
MPI_Comm_size(MPI_Comm comm,int *size)
```

Fortran:

```
CALL MPI_COMM_SIZE(COMM, SIZE, IERROR)

INTEGER COMM, SIZE, IERROR
```

Process Rank

- Process ID number within the communicator
  - Starts with zero and goes to (n-1) where n is the number of processes requested

- Used to identify the source and destination of messages

C:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

```
CALL MPI_COMM_RANK(COMM, RANK, IERROR)

INTEGER COMM, RANK, IERROR
```

Exiting MPI

- Must be called last by "all" processes

C:

```
MPI_Finalize()
```

Fortran:

```
CALL MPI_FINALIZE(IERROR)
```

Bones.c

```
#include<mpi.h>
void main(int argc, char *argv[]) {
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
/* … your code here … */
   MPI_Finalize ();
}
```

Bones.f

```
      PROGRAM skeleton
      INCLUDE ‘mpif.h’
      INTEGER ierror, rank, size
      CALL_MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
C     … your code here …
      CALL MPI_FINALIZE(ierror)
      END
```

Exercise #1: Hello World

- Write a minimal MPI program which prints “hello world”
- Run it on several processors in parallel
- Modify your program so that only the process ranked 2 in MPI_COMM_WORLD prints out “hello world”
- Modify your program so that each process prints out its rank and the total number of processors

Message Passing

- Messages
- MPI Basic Datatypes - C
- MPI Basic Datatypes - Fortran
- Rules and Rationale

Messages

- A message contains an array of elements of some particular MPI datatype
- MPI Datatypes:
  - Basic types
  - Derived types
- Derived types can be build up from basic types
- C types are different from Fortran types

MPI Basic Datatypes - C

MPI Datatype	C Datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed log int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

MPI Basic Datatypes - Fortran

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

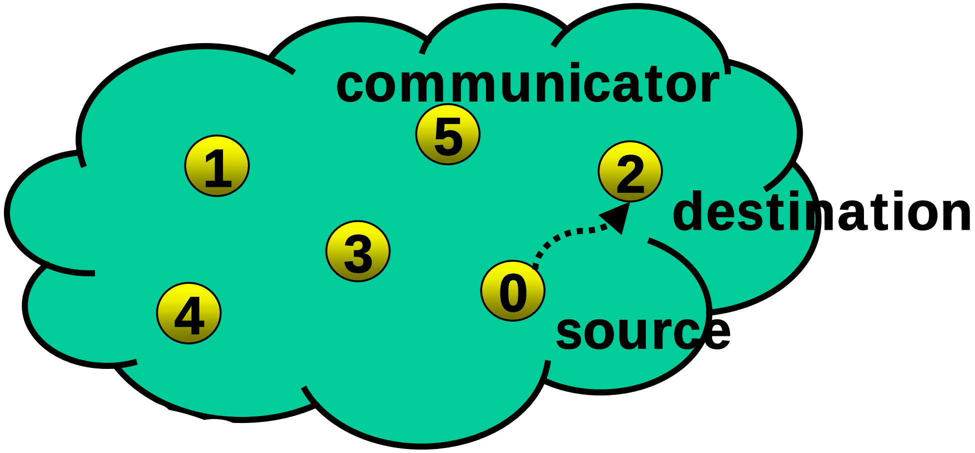
Rules and Rationale

- Programmer declares variables to have “normal” C/Fortran type, but uses matching MPI datatypes as arguments in MPI routines
- Mechanism to handle type conversion in a heterogeneous collection of machines
- General rule: MPI datatype specified in a receive must match the MPI datatype specified in the send

Point-to-Point Communications

- Definitions
- Sending a Message
- Memory Mapping
- Standard Send
- Receiving a Message
- Wildcarding
- Communication Envelope
- Received Message Count
- Sample Programs
- Class Exercise: Processor Ring
- Extra Exercise 1: Ping Pong
- Timers
- Extra Exercise 2: Broadcast

Point-to-Point Communication



- Communication between two processes
- Source process *sends* message to destination process
- Destination process *receives* the message
- Communication takes place within a communicator
- Destination process is identified by its rank in the communicator

Definitions

- "Completion" of the communication means that memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can now be used

Sending a Message

C:

```
int MPI_Send(void *buf, int count,MPI_Datatype datatype,
         int dest,int tag, MPI_Comm comm)
```

Fortran:

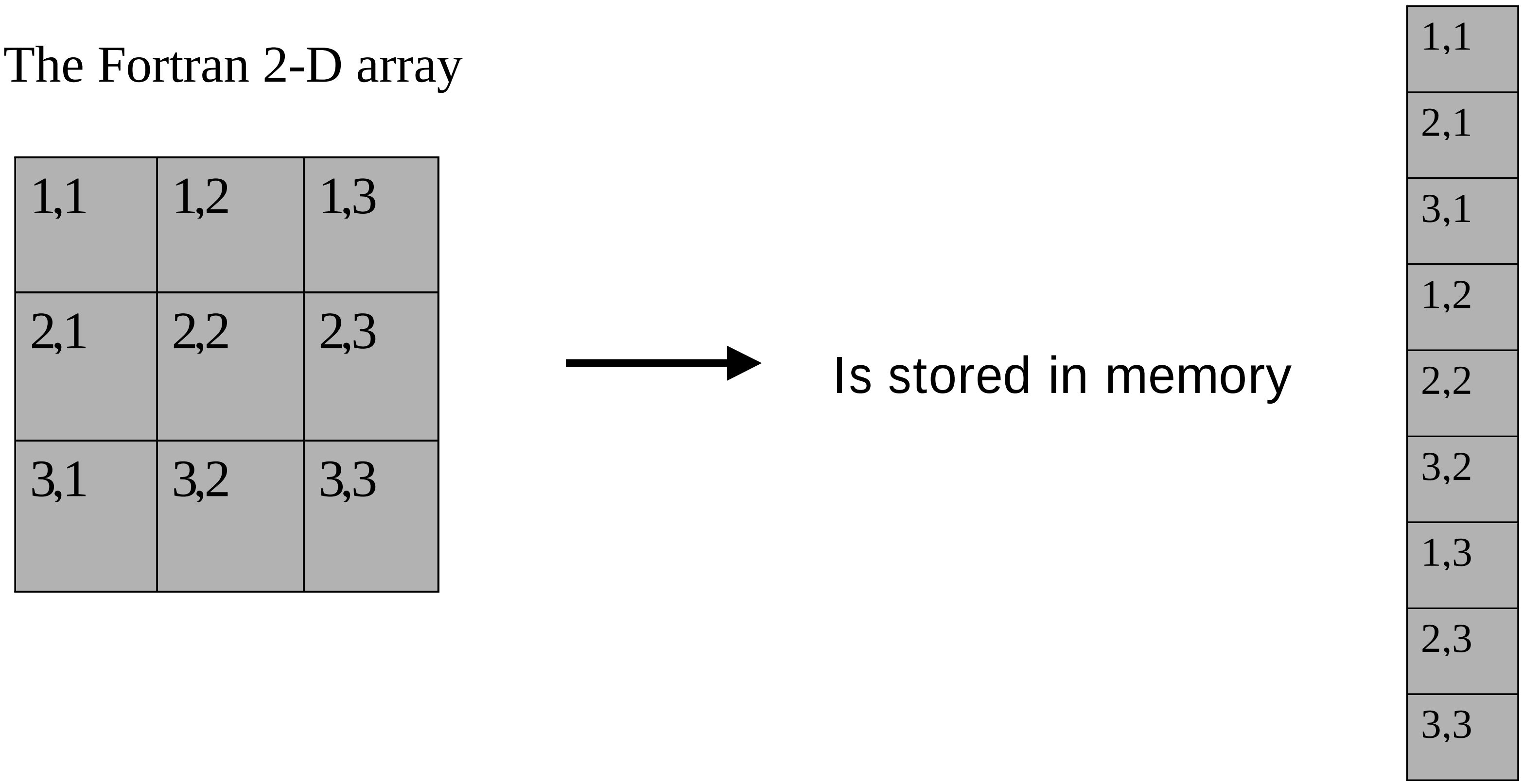
```
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG,COMM,IERROR)
type BUF(*)

INTEGER COUNT, DATATYPE, DEST, TAG
INTEGER COMM, IERROR
```

Arguments

`buf`	starting address of the data to be sent
`count`	number of elements to be sent
`datatype`	MPI datatype of each element
`dest`	rank of destination process
`tag`	message marker (set by user)
`comm`	MPI communicator of processors involved

```
MPI_SEND(data,500,MPI_REAL,6,33,MPI_COMM_WORLD,IERROR)
```

Memory Mapping



The Fortran 2-D array

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

Is stored in memory

1,1
2,1
3,1
1,2
2,2
3,2
1,3
2,3
3,3

Standard Send

- Completion criteria:

  Completes when message has been sent

- May or may not imply that message has arrived at destination

- Don’t make any assumptions (implementation dependant)

Receiving a Message

C:

```
int MPI_Recv(void *buf, int count,MPI_Datatype datatype, \
      int source, int tag, MPI_Comm comm,MPI_Status *status)
```

Fortran:

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

type BUF(*)
INTEGER         COUNT, DATATYPE, DEST, TAG
INTEGER         COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

For a communication to succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver’s buffer must be large enough

Wildcarding

- Receiver can wildcard
- To receive from any source

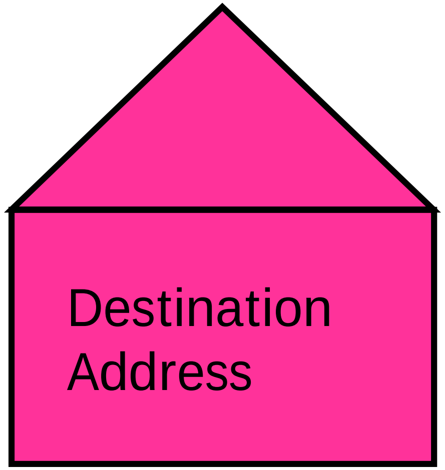
```
MPI_ANY_SOURCE
```

To receive with any tag

```
MPI_ANY_TAG
```

- Actual source and tag are returned in the receiver’s `status` parameter

Communication Envelope



Destination Address

Sender’s Address

For the attention of:

Data

- Item 1
- Item 2
- Item 3

Communication Envelope Information

- Envelope information is returned from `MPI_RECV` as `status`

- Information includes:
  - Source: `status.MPI_SOURCE` or `status(MPI_SOURCE)`
  - Tag: `status.MPI_TAG` or `status(MPI_TAG)`
  - Count: `MPI_Get_count` or `MPI_GET_COUNT`

Received Message Count

- Message received may not fill receive buffer
- `count` is number of elements actually received

C:

```
int MPI_Get_count (MPI_Status, *status,
                   MPI_Datatype datatype, int *count)
```

Fortran:

```
CALL MPI_GET_COUNT(STATUS,DATATYPE,COUNT,IERROR)
INTEGER          STATUS(MPI_STATUS_SIZE), DATATYPE
INTEGER          COUNT,IERROR
```

Sample Program - C

```
#include<mpi.h>

/* Run with two processes */

 void main(int argc, char *argv[]) {
    int rank, i, count;
    float data[100],value[200];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    if(rank==1) {
       for(i=0;i<100;++i) data[i]=i;
       MPI_Send(data,100,MPI_FLOAT,0,55,MPI_COMM_WORLD);
    } else {
       MPI_Recv(value,200,MPI_FLOAT,MPI_ANY_SOURCE,55,MPI_COMM_WORLD,&status);
       printf("P:%d Got data from processor %d \n",rank, status.MPI_SOURCE);
       MPI_Get_count(&status,MPI_FLOAT,&count);
       printf("P:%d Got %d elements \n",rank,count);
       printf("P:%d value[5]=%f \n",rank,value[5]);
    }
    MPI_Finalize();
 }
```

```
        Program Output
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=5.000000
```

Sample Program - Fortran

```
      PROGRAM p2p
C Run with two processes
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      real data(100)
      real value(200)
      integer status(MPI_STATUS_SIZE)
      integer count
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
      if (rank.eq.1) then
         data=3.0
         call MPI_SEND(data,100,MPI_REAL,0,55,MPI_COMM_WORLD,err)
       else
         call MPI_RECV(value,200,MPI_REAL,MPI_ANY_SOURCE,55,
     &                 MPI_COMM_WORLD,status,err)
         print *, "P:",rank," got data from processor ",
     &                 status(MPI_SOURCE)
         call MPI_GET_COUNT(status,MPI_REAL,count,err)
         print *, "P:",rank," got ",count," elements"
         print *, "P:",rank," value(5)=",value(5)
      end if
      CALL MPI_FINALIZE(err)
      END
```

```
           Program Output
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=3.
```

Class Exercise: Processor Ring

- A set of processes are arranged in a ring
- Each process stores its rank in `MPI_COMM_WORLD` in an integer
- Each process passes this on to its neighbor on the right
- Each processor keeps passing until it receives its rank back

Extra Exercise 1: Ping Pong

- Write a program in which two processes repeatedly pass a message back and forth
- Insert timing calls to measure the time taken for one message
- Investigate how the time taken varies with the size of the message

Timers

- Time is measured in seconds

- Time to perform a task is measured by consulting the timer before and after

C:

```
double MPI_Wtime(void);
```

Fortran:

```
DOUBLE PRECISION MPI_WTIME()
```

Extra Exercise 2: Broadcast

- Have processor 1 send the same message to all the other processors and then receive messages of the same length from all the other processors
- How does the time taken vary with the size of the messages and the number of processors?

Collective Communication

- Collective Communication
- Barrier Synchronization
- Broadcast
- Global Reduction Operations
- Predefined Reduction Operations
- MPI_Reduce
- Class Exercise: Last Ring

Collective Communication

- Communications involving a group of processes
- Called by *all* processes in a communicator
- Examples:
  - Broadcast, scatter, gather (Data Distribution)
  - Global sum, global maximum, etc. (Collective Operations)
  - Barrier synchronization

Characteristics of Collective Communication

- Collective communication will not interfere with point-to-point communication and vice-versa
- All processes must call the collective routine
- Synchronization not guaranteed (except for barrier)
- No tags
- Receive buffers must be exactly the right size

Barrier Synchronization

- Red light for each processor: turns green when all processors have arrived
- Slower than hardware barriers (example: Cray T3E)

C:

```
int MPI_Barrier (MPI_Comm comm)
```

Fortran:

```
CALL MPI_BARRIER (COMM,IERROR)
INTEGER COMM,IERROR
```

Broadcast

- One-to-all communication: same data sent from root process to all the others in the communicator

- C:

```
int MPI_Bcast (void *buffer, int, count,
  MPI_Datatype datatype,int root, MPI_Comm comm)
```

- Fortran:

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM IERROR)

<type> BUFFER (*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

- All processes must specify same root rank and communicator

Sample Program - C

```
#include<mpi.h>
void main (int argc, char *argv[]) {
  int rank;
  double param;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if(rank==5) param=23.0;
  MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
  printf("P:%d after broadcast parameter is %f\n",rank,param);
  MPI_Finalize();
}
```

```
                  Program Output
P:0 after broadcast parameter is 23.000000
P:6 after broadcast parameter is 23.000000
P:5 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000
P:7 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:4 after broadcast parameter is 23.000000
```

Sample Program - Fortran

```
PROGRAM broadcast
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
if(rank.eq.5) param=23.0
call MPI_BCAST(param,1,MPI_REAL,5,MPI_COMM_WORLD,err)
print *,"P:",rank," after broadcast param is ",param
CALL MPI_FINALIZE(err)
END
```

```
             Program Output
P:1 after broadcast parameter is 23.
P:3 after broadcast parameter is 23.
P:4 after broadcast parameter is 23
P:0 after broadcast parameter is 23
P:5 after broadcast parameter is 23.
P:6 after broadcast parameter is 23.
P:7 after broadcast parameter is 23.
P:2 after broadcast parameter is 23.
```

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes
- Examples:
  - Global sum or product
  - Global maximum or minimum
  - Global user-defined operation

Example of Global Reduction

- Sum of all the `x` values is placed in `result` only on processor 0

C:

```
MPI_Reduce(&x,&result,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)
```

Fortran:

```
CALL MPI_REDUCE(x,result,1,MPI_INTEGER,MPI_SUM,0,
                   MPI_COMM_WORLD,IERROR)
```

Predefined Reduction Operations

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

General Form

- `count` is the number of “*ops*” done on consecutive elements of `sendbuf` (it is also size of `recvbuf`)

- `op` is an associative operator that takes two operands of type `datatype` and returns a result of the same type

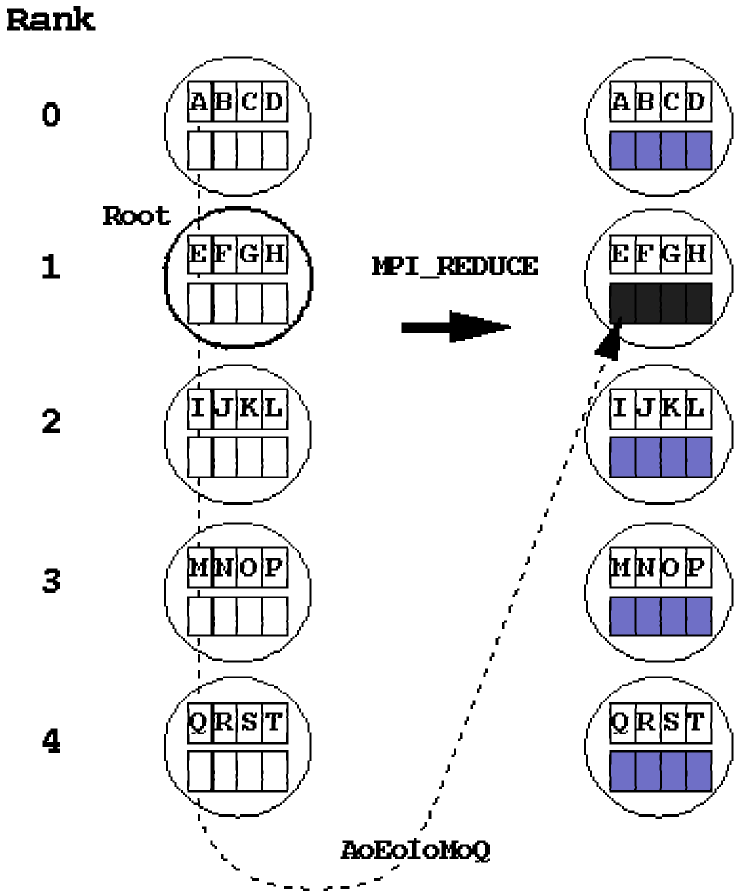
C:

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)
```

Fortran:

```
CALL MPI_REDUCE(SENDBUF,RECVBUF,COUNT,DATATYPE,OP,ROOT,COMM,IERROR)
<type> SENDBUF(*), RECVBUF(*)
```

MPI_Reduce


Rank
0
A B C D
Root
1
E F G H
MPI_REDUCE
2
I J K L
3
M N O P
4
Q R S T
AoEoIoMoQ

Class Exercise: Last Ring

- Rewrite the “Structured Ring” program to use MPI global reduction to perform its global sums
- Extra credit: Rewrite it so that each process computes a product
- Extra extra credit: Rewrite this so that each process prints out its product in rank order

Virtual Topologies

- Virtual Topologies
- Topology Types
- Creating a Cartesian Virtual Topology
- Cartesian Example
- Cartesian Mapping Functions
  - MPI_CART_RANK*
  - MPI_CART_COORDS*
  - MPI_CART_SHIFT*
- Exercise

*includes sample C and Fortran programs

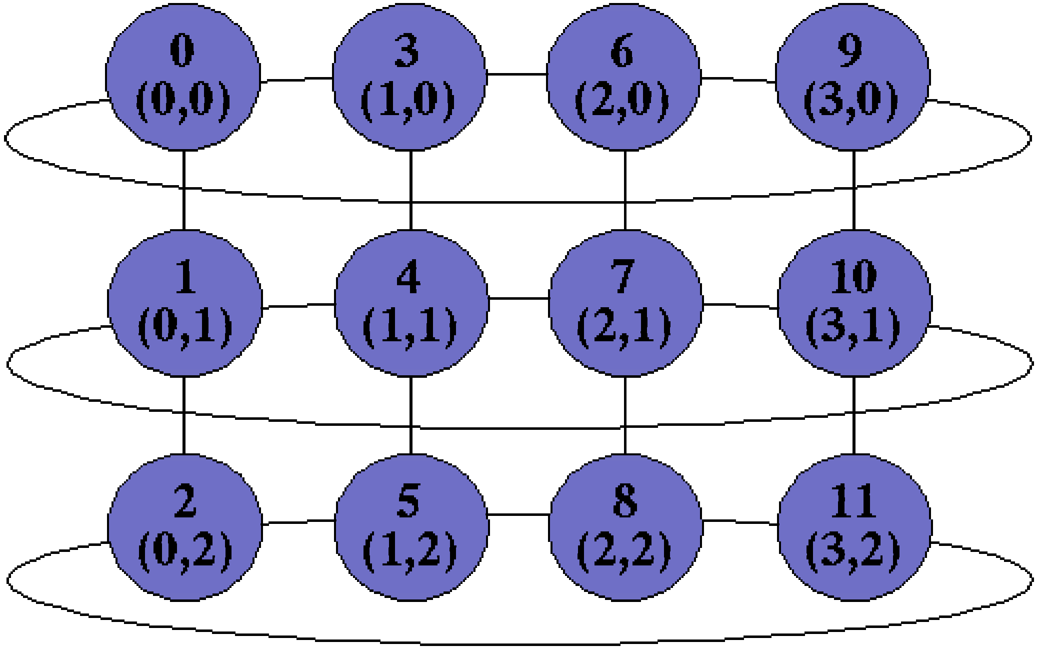
Virtual Topologies

- Convenient process naming
- Naming scheme to fit the communication pattern
- Simplifies writing of code
- Can allow MPI to optimize communications
- Rationale: access to useful topology routines

How to use a Virtual Topology

- Creating a topology produces a new communicator
- MPI provides “mapping functions”
- Mapping functions compute processor ranks, based on the topology naming scheme

Example - 2D Torus



Topology types

- Cartesian topologies
  - Each process is connected to its neighbors in a virtual grid
  - Boundaries can be cyclic
  - Processes can be identified by cartesian coordinates
- Graph topologies
  - General graphs
  - Will not be covered here

Creating a Cartesian Virtual Topology

- C

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
             int *dims, int *periods, int reorder,
             mPI_Comm *comm_cart)
```

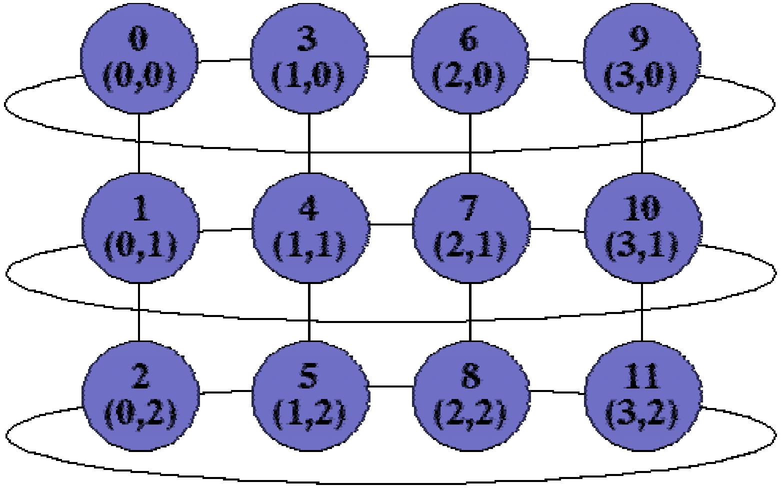
- Fortran

```
CALL MPI_CART_CREATE(COMM_OLD,NDIMS,DIMS,PERIODS,
              REORDER,COMM_CART,IERROR)

INTEGER COMM_OLD,NDIMS,DIMS(*),COMM_CART,IERROR
LOGICAL PERIODS(*),REORDER
```

Arguments

comm_old	existing communicator
ndims	number of dimensions
periods	logical array indicating whether a dimension is cyclic (TRUE=>cyclic boundary conditions)
reorder	logical (FALSE=>rank preserved) (TRUE=>possible rank reordering)
comm_cart	new cartesian communicator

Cartesian Example



```
MPI_Comm vu;
int dim[2], period[2], reorder;

dim[0]=4; dim[1]=3;
period[0]=TRUE; period[1]=FALSE;
reorder=TRUE;

MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
```

Cartesian Mapping Functions

Mapping process grid coordinates to ranks

C:

```
int MPI_Cart_rank (MPI_Comm comm, init *coords, int *rank)
```

Fortran:

```
CALL MPI_CART_RANK(COMM,COORDS,RANK,IERROR)

INTEGER       COMM,COORDS(*),RANK,IERROR
```

Cartesian Mapping Functions

Mapping ranks to process grid coordinates

C:

```
int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims,
                        int *coords)
```

Fortran:

```
CALL MPI_CART_COORDS(COMM,RANK,MAXDIMS,COORDS,IERROR)

INTEGER       COMM,RANK,MAXDIMS,COORDS(*),IERROR
```

Sample Program - C

```
#include<mpi.h>
/* Run with 12 processes */
 void main(int argc, char *argv[]) {
    int rank;
    MPI_Comm vu;
    int dim[2],period[2],reorder;
    int coord[2],id;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=FALSE;
    reorder=TRUE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    if(rank==5){
      MPI_Cart_coords(vu,rank,2,coord);
      printf("P:%d My coordinates are %d %d\n",rank,coord[0],coord[1]);
    }
    if(rank==0) {
      coord[0]=3; coord[1]=1;
      MPI_Cart_rank(vu,coord,&id);
      printf("The processor at position (%d, %d) has rank %d\n",coord[0],coord[1],id);
    }
    MPI_Finalize();
 }
```

```
                 Program output
The processor at position (3,1) has rank 10
P:5 My coordinates are 1 2
```

Sample Program - Fortran

```
      PROGRAM Cartesian
C
C Run with 12 processes
C
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      integer vu,dim(2),coord(2),id
      logical period(2),reorder
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
      dim(1)=4
      dim(2)=3
      period(1)=.true.
      period(2)=.false.
      reorder=.true.
      call MPI_CART_CREATE(MPI_COMM_WORLD,2,dim,period,reorder,vu,err)
      if(rank.eq.5) then
        call MPI_CART_COORDS(vu,rank,2,coord,err)
        print*,'P:',rank,' my coordinates are',coord
      end if
      if(rank.eq.0) then
        coord(1)=3
        coord(2)=1
        call MPI_CART_RANK(vu,coord,id,err)
        print*,'P:',rank,' processor at position',coord,' is',id
        end if
      CALL MPI_FINALIZE(err)
      END
```

```
              Program Output
P:5 my coordinates are 1, 2
P:0 processor at position 3, 1 is 10
```

Cartesian Mapping Functions

Computing ranks of neighboring processes

C:

```
int MPI_Cart_shift (MPI_Comm comm, int direction, int disp,
                        int *rank_source, int *rank_dest)
```

Fortran:

```
CALL MPI_CART_SHIFT(COMM,DIRECTION,DISP,RANK_SOURCE,
                        RANK_DEST,IERROR)

INTEGER        COMM,DIRECTION,DISP,RANK_SOURCE,RANK_DEST,IERROR
```

MPI_Cart_shift

- Does not actually shift data: returns the correct ranks for a shift which can be used in subsequent communication calls
- Arguments:
  - `direction` dimension in which the shift should be made
  - `disp` length of the shift in processor coordinates (+ or -)
  - `rank_source` where calling process should receive a message **from** during the shift
  - `rank_dest` where calling process should send a message **to** during the shift
- If shift off of the topology, `MPI_Proc_null` is returned

Sample Program - C

```
#include<mpi.h>
#define TRUE 1
#define FALSE 0
int main(int argc, char *argv[]) {
    int rank;
    MPI_Comm vu;
    int dim[2],period[2],reorder;
    int up,down,right,left;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=FALSE;
    reorder=TRUE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    if(rank==9){
      MPI_Cart_shift(vu,0,1,&left,&right);
      MPI_Cart_shift(vu,1,1,&up,&down);
      printf("P:%d My neighbors are r: %d d:%d 1:%d u:%d\n",rank,right,down,left,up);
    }
    MPI_Finalize();
    return(0);
 }
```

```
                Program Output
P:9 my neighbors are r:0 d:10 1:6 u:-1
```

Innovations in computing, networking, and education

Sample Program - Fortran

```
     PROGRAM neighbors
C
C Run with 12 processes
C
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      integer vu
      integer dim(2)
      logical period(2),reorder
      integer up,down,right,left
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
      dim(1)=4
      dim(2)=3
      period(1)=.true.
      period(2)=.false.
      reorder=.true.
      call MPI_CART_CREATE(MPI_COMM_WORLD,2,dim,period,reorder,vu,err)
      if(rank.eq.9) then
        call MPI_CART_SHIFT(vu,0,1,left,right,err)
        call MPI_CART_SHIFT(vu,1,1,up,down,err)
        print*,'P:',rank,' neighbors (rdlu)are',right,down,left,up
      end if
      CALL MPI_FINALIZE(err)
      END
```

Program Output

```
P:9 neighbors (rdlu) are 0, 10, 6, -1
```

Class Exercise: Ring Topology

- Rewrite the “Calculating Ring” exercise using a Cartesian Topology
- Extra credit: Extend the problem to two dimensions. Each row of the grid should compute its own separate sum

Interlude 3: Laplace Solver Using MPI

- Topologies for Domain Decomposition
- Applying MPI at the Boundaries
- MPI Implementation of Laplace Solver
- Performance Characteristics of MPI Implementation

Topologies for Domain Decomposition

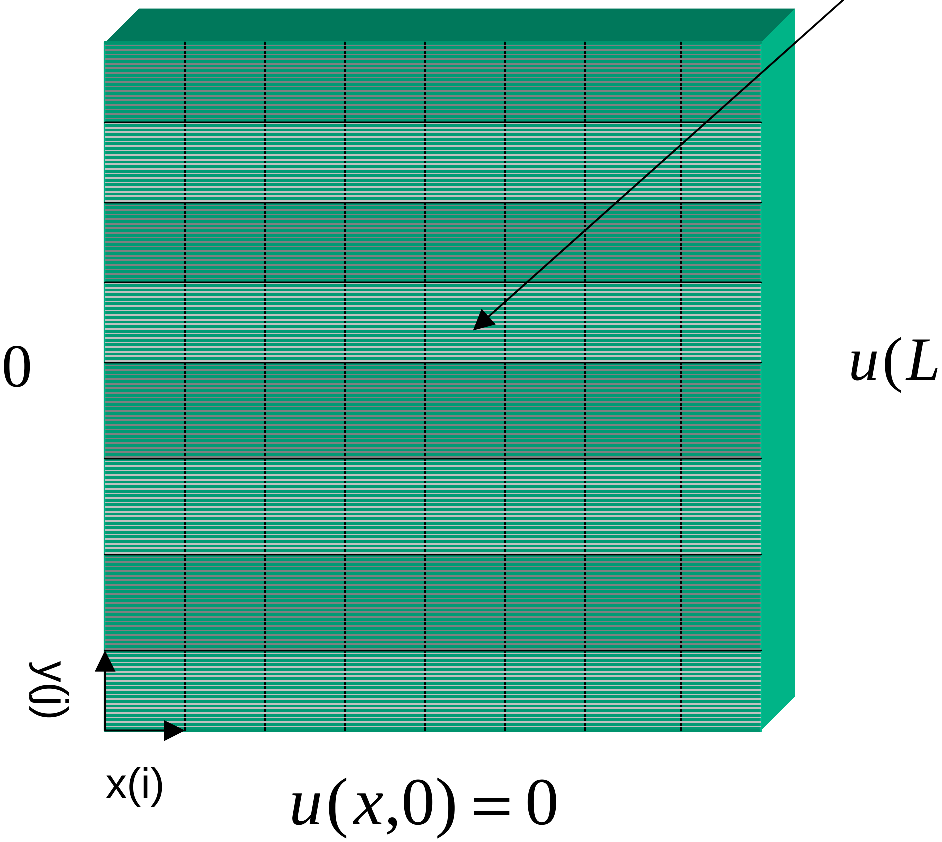
- To use a message passing approach such as MPI, we first need to think about domain decomposition -- how to divide work between the MPI processes.
- In the case of the 2D Laplacian, we can divide the grid into segments and assign a process to each segment. Each segment also needs to maintain “ghost cells”, which contain values solution values at points on the boundaries of neighboring processes; the ghost cells are kept up to date by passing messages between processes containing the boundary values.
- There are two natural ways to decompose the domain:
  - A 1D decomposition (i.e. each MPI process computes the solution for all values in the `i` index and a fixed range of values in the `j` index), in which each process must make up to two communications per iteration.
  - A 2D decomposition (i.e. each MPI process computes the solution for fixed ranges of both `i` and `j`), in which each process must make up to four communications per iteration.

1D Domain Decomposition

$$u(x,L)=0$$

$$\nabla^2 u = 0$$

$$u(0,y)=0$$



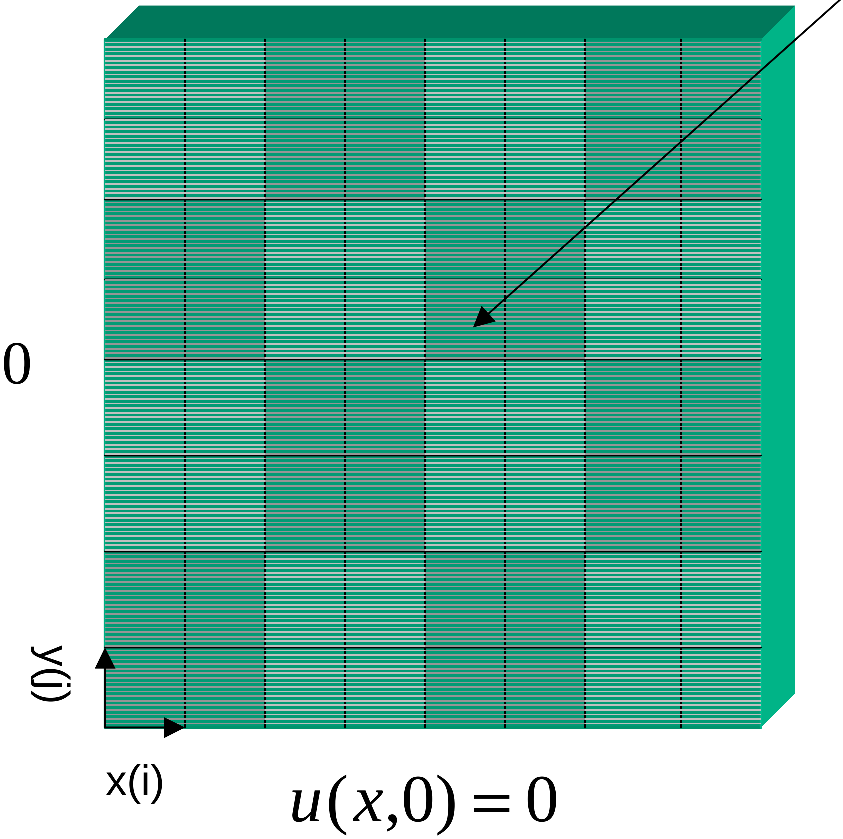
$$u(L,y)=u_{\max}\sin(\pi\frac{y}{L})$$

$$u(x,0)=0$$

2D Domain Decomposition

$$u(x, L) = 0$$

$$\nabla^2 u = 0$$



$$u(0, y) = 0$$

$$u(L, y) = u_{\max} \sin(\pi \frac{y}{L})$$

$$u(x,0) = 0$$

Applying MPI at the Boundaries

- If we take the case of the 1D decomposition, then we must maintain ghost cells for the edge values of `u(i,j)` for the processes directly to each process's left and right (or top and bottom, depending on your point of view) as part of the enforcement of boundary conditions.
- A typical communication algorithm for this type of data exchange might look something like this:
  - *Send phase:*
    - if a left neighbor exists, then send the solution values on the left edge of this segment to the left neighbor
    - if a right neighbor exists, then send the solution values on the right edge of this segment to the right neighbor
  - *Receive phase:*
    - if a left neighbor exists, then receive a message from the left neighbor and place the values in the ghost cells to the left of this segment
    - if a right neighbor exists, then receive a message from the right neighbor and place the values in the ghost cells to the right of this segment

MPI Implementation of a Laplace Solver

```
      program lpmpi
      include 'mpif.h'
      integer imax,jmax,im1,im2,jm1,jm2,it,itmax
      parameter (imax=2001,jmax=2001)
      parameter (im1=imax-1,im2=imax-2,jm1=jmax-1,jm2=jmax-2)
      parameter (itmax=100)
      real*8 u(imax,jmax),du(imax,jmax),umax,dumax,tol
      parameter (umax=10.0,tol=1.0e-6)

! Additional MPI parameters
      integer istart,iend,jstart,jend
      integer size,rank,ierr,istat,mpigrid,length
      integer grdrnk,dims(1),left,right,isize,jsize
      real*8 tstart,tend,gdumax
      logical cyclic(1)
      real*8 lbuf(jmax),rbuf(jmax),ubuf(imax),dbuf(imax)
      common /T3EHACK/ istart,iend,jstart,jend
```

Innovations in computing, networking, and education

MPI Implementation (con’t)

```
! Initialize MPI
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
! 1D linear topology
      dims(1)=size
      cyclic(1)=.FALSE.
      call MPI_CART_CREATE(MPI_COMM_WORLD,1,dims,cyclic,.true.,
     +     mpigrid,ierr)
      call MPI_COMM_RANK(mpigrid,grdrnk,ierr)
      call MPI_CART_SHIFT(mpigrid,0,1,grdrnk,right,ierr)
      call MPI_CART_SHIFT(mpigrid,0,-1,grdrnk,left,ierr)
! Compute index bounds
      isize=imax
      jsize=jmax/dims(1)
      istart=loc(1)*isize+1
      if (istart.LT.2) istart=2
      iend=(loc(1)+1)*isize
```

MPI Implementation (con’t)

```
      if (iend.GE.imax) iend=imax-1
      jstart=loc(2)*jsize+1
      if (jstart.LT.2) jstart=2
      jend=(loc(2)+1)*jsize
      if (jend.GE.jmax) jend=jmax-1

! Initialize
      do j=jstart-1,jend+1
         do i=istart-1,iend+1
            u(i,j)=0.0
            du(i,j)=0.0
         enddo
         u(imax,j)=umax
      enddo

! Main computation loop
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
```

MPI Implementation (con’t)

```
 do it=1,itmax
        dumax=0.0
         do j=jstart,jend
            do i=istart,iend
               du(i,j)=0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))-u(i,j)
               dumax=max(dumax,abs(du(i,j)))
               u(i,j)=u(i,j)+du(i,j)
            enddo
         enddo
! Compute the overall residual
         call MPI_REDUCE(dumax,gdumax,1,MPI_REAL8,MPI_MAX,0
     +        ,MPI_COMM_WORLD,ierr)

! Pass the edge data to neighbors
! Using buffers
```

MPI Implementation (con’t)

```
! Send phase
         if (left.NE.MPI_PROC_NULL) then
            i=1
            do j=jstart,jend
               lbuf(i)=u(istart,j)
               i=i+1
            enddo
            length=i-1
            call MPI_SEND(lbuf,length,MPI_REAL8,left,it,mpigrid,
     +                    ierr)
         endif
         if (right.NE.MPI_PROC_NULL) then
            i=1
            do j=jstart,jend
               rbuf(i)=u(iend,j)
               i=i+1
            enddo
            length=i-1
            call MPI_SEND(rbuf,length,MPI_REAL8,right,it,mpigrid,
     +                    ierr)
         endif
```

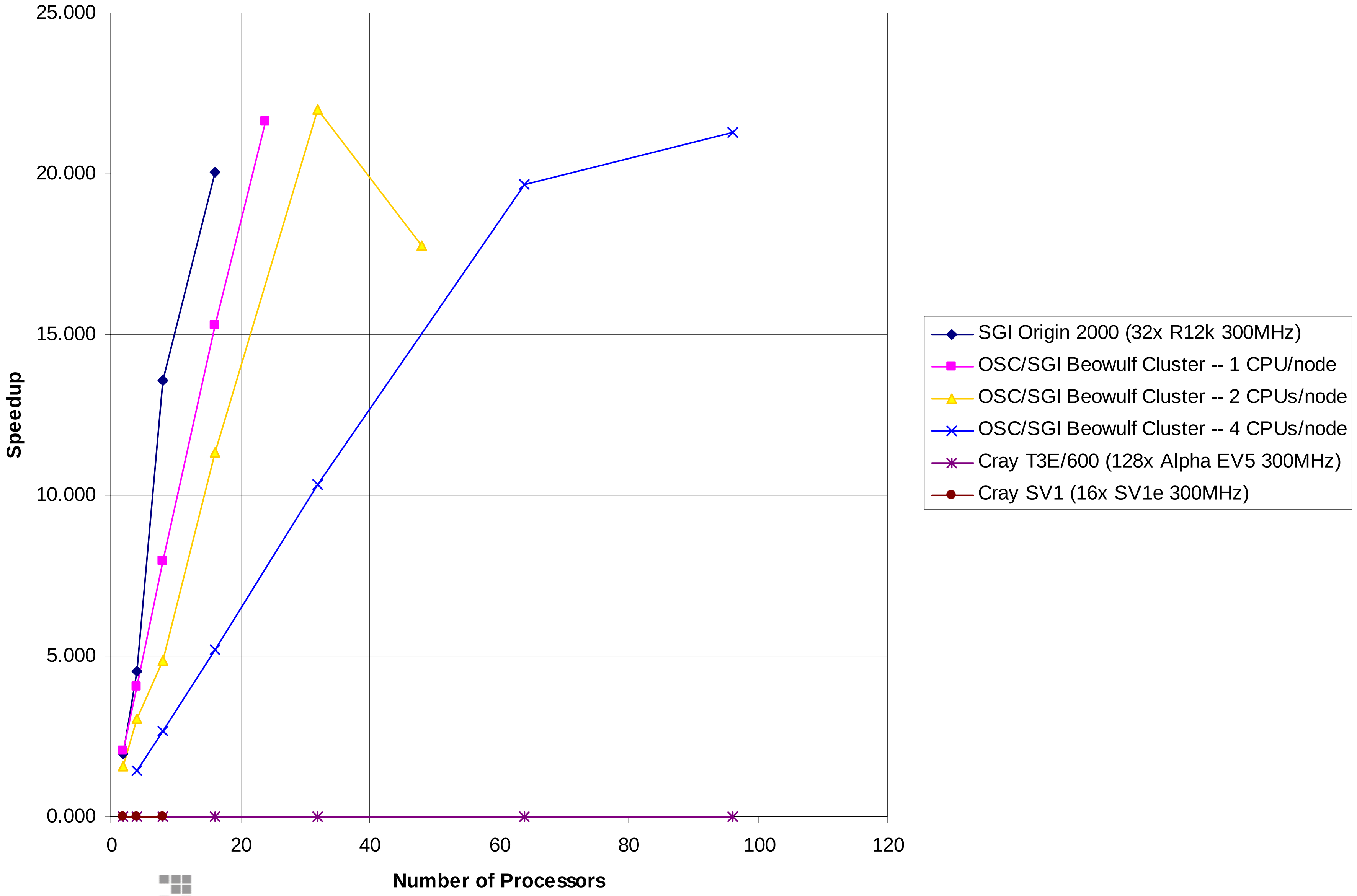
MPI Implementation (con’t)

```
! Receive phase
         if (left.NE.MPI_PROC_NULL) then
            length=jend-jstart+1
            call MPI_RECV(lbuf,length,MPI_REAL8,left,it,
     +           mpigrid,istat,ierr)
            i=1
            do j=jstart,jend
               u(istart-1,j)=lbuf(i)
               i=i+1
            enddo
         endif
         if (right.NE.MPI_PROC_NULL) then
            length=jend-jstart+1
            call MPI_RECV(rbuf,length,MPI_REAL8,right,it,
     +           mpigrid,istat,ierr)
            i=1
            do j=jstart,jend
               u(iend+1,j)=rbuf(i)
               i=i+1
            enddo
         endif
```

MPI Implementation (con’t)

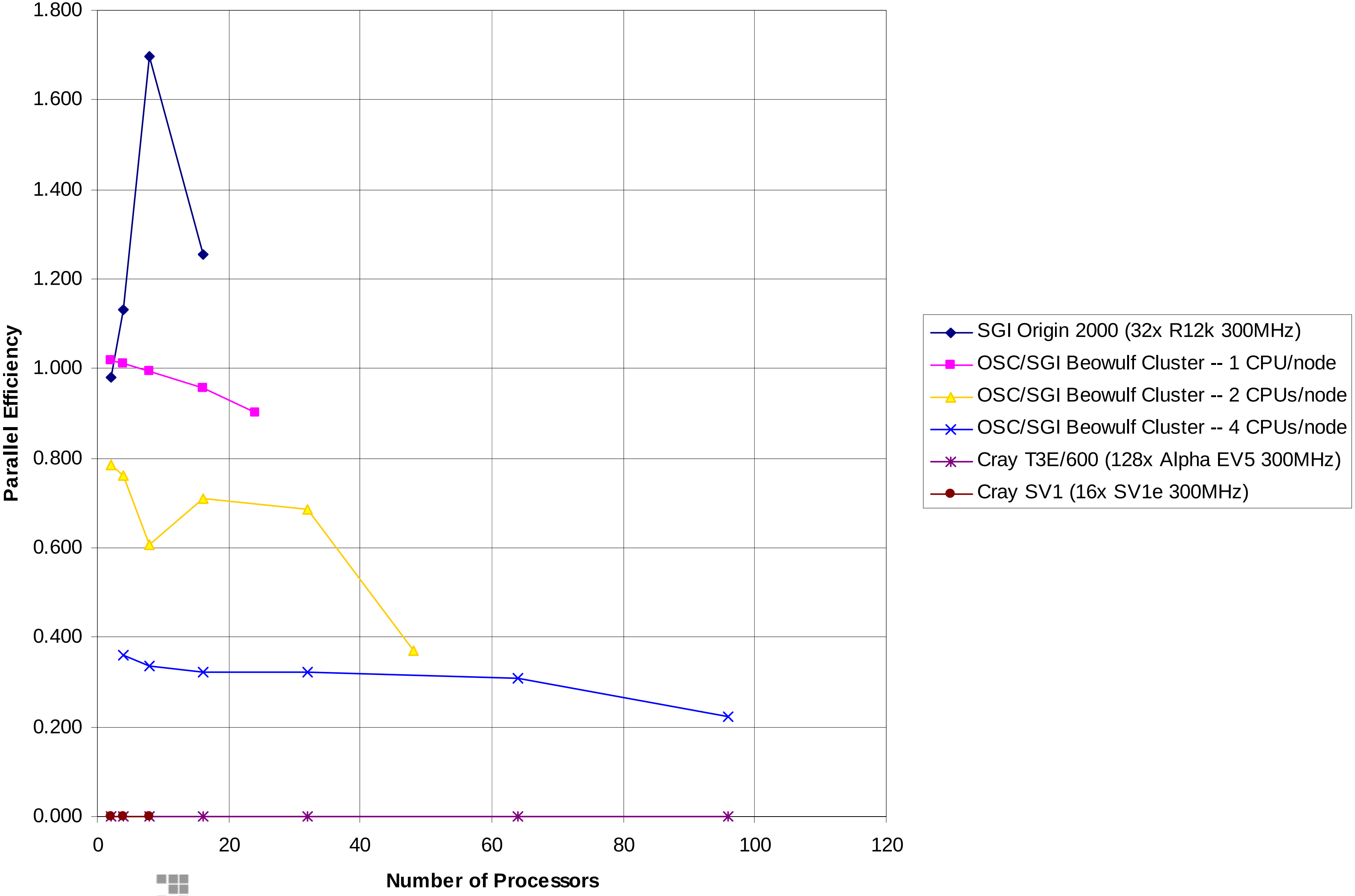
```
         write (rank+10,*) rank,it,dumax,gdumax
         if (rank.eq.0) write (1,*) it,gdumax
      enddo

! Clean up
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
      if (rank.EQ.0) then
         write(*,*) 'Calculation took ',tend-tstart,'s. on ',size,
     +        ' MPI processes'
      endif
      call MPI_FINALIZE(ierr)
      stop
      end
```

Performance Characteristics of MPI Implementation



OSC
Innovations in computing, networking, and education

Performance Characteristics of MPI Implementation (con't)



Innovations in computing, networking, and education

Mixing OpenMP and MPI

- Multilevel Parallel Programming Approach
- Demonstration of Multilevel Parallel Programming
- Sample Search Program and Output
- Conclusions
- MLP Problem Set

Multilevel Parallel Programming Approach

- In a Multi-Level Parallel (MLP) program both the message passing and loop-level parallelism approaches are combined in one program.
  - Merging of what was once taught as two separate parallel paradigms
- MLP Strategy:
  - Use message-passing commands to initiate parallel processes each of which will be responsible for the calculations required on large sections of the global data (domain decomposition)
  - Each message-passing process will hold its local domain data in its own local memory. When necessary, processes will transfer data between local memories in the form of messages.
  - For the actual computations each message-passing process performs, spawn parallel threads which will execute (simultaneously) sets of iterations values of critical loops.
- MLP Procedure:
  - Use the exact same functionality and syntax of the message-passing commands and directive-based loop parallelism shown above in programs *only* using one of the parallel paradigms. **Just combine the two parallel library calls into one program**

Multilevel Parallel Programming Approach (con’t)

- Be sure to call all the include files, initialization procedures, and closing procedures of both the message-passing and loop-level parallel libraries.
- When compiling an MLP program just combine whatever options were used for each separate library.
- For example, all the codes shown in this workshop use MPI as the message-passing library and OpenMP as the parallel loop library. On the OSC Origin 2000, the combined compilation could look like this:

```
f90 -mp search.f -lmpi
```

- **WARNING:** Use of MPI processes and OpenMP threads has to be limited to what they are designed to do. For example, an OpenMP thread cannot send or receive a message. (This is explained further in the next section.)

How do we know MLP programming works?

- How can a user know directly that both MPI parallel processes and OpenMP threads are *both* being created and run on actual processors?
- Answer: Call the the UNIX `ps` command internally from the code at different points. The `ps` command output will show process creation and use (along with much other information).
- The following program demonstrates this approach. It uses the classic master-slave parallel algorithm to search a large 1-D array for the locations (indices) of a certain value called the mark. First, the master MPI process (rank=0) sends different thirds of the array two three slave MPI processes. Then, each slave MPI process parallelizes the search loops with OpenMP threads.
- The first internal `ps` call is made at the near the beginning of code when only the MPI processes have been created. The second is toward the end of the code, where each MPI process has four parallel threads working.

```
program search
  INCLUDE 'mpif.h'
  parameter (N=3000000)
  INTEGER err,rank,size,thread
  integer i,mark
  integer b(N),sub_b(N/3)
  integer status(MPI_STATUS_SIZE)
  integer*8 seed
  character*42, command

  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, err)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, err)
  call omp_set_num_threads(4)

  mark=10

  if(rank==0) then
    ! WARNING:  ISHELL subroutine not in Fortran 77 standard!
    print *,"R:",rank," ps using MPI only"
    command="ps -u dje -o pid,ppid,stime,etime,cpu,comm"
    call ISHELL(command)

    seed=78946230
    call ranset(seed)
    do i=1,N
      b(i)=nint(100.0*ranf())-50
    end do
```

```
          CALL MPI_SEND(b(1),N/3,MPI_INTEGER,1,51,MPI_COMM_WORLD,err
          CALL MPI_SEND(b(N/3+1),N/3,MPI_INTEGER,2,52,MPI_COMM_WORLD,err)
          CALL MPI_SEND(b(2*N/3+1),N/3,MPI_INTEGER,3,53,MPI_COMM_WORLD,err)

      else if(rank.eq.1) then
          CALL MPI_RECV(sub_b,N/3,MPI_INTEGER,0,51,MPI_COMM_WORLD,status,err)

c$omp parallel private(i,thread,command,j)
c$omp&firstprivate(rank,mark,sub_b)
         thread=omp_get_thread_num()
c$omp     do
          do i=1,N/3
              if (abs(sub_b(i)).eq.mark) then
                j=(rank-1)*(N/3)+i
                write(11,*)"R:",rank,"T:",thread," j=",j
              end if
          end do
c$omp end parallel

        else if(rank.eq.2) then
          CALL MPI_RECV(sub_b,N/3,MPI_INTEGER,0,52,MPI_COMM_WORLD,status,err)

c$omp parallel private(i,thread,command,j)
c$omp&firstprivate(rank,mark,sub_b)
         thread=omp_get_thread_num()
         ! WARNING:  ISHELL subroutine not in Fortran 77 standard!
         command="ps -u dje -o pid,ppid,stime,etime,cpu,comm"
         if(thread.eq.0) then
          print *,"R:",rank,"T:",thread," ps within OpenMP"
          call ISHELL(command)
         end if
```

```
c$omp     do
          do i=1,N/3
              if (abs(sub_b(i)).eq.mark) then
                j=(rank-1)*(N/3)+i
                write(12,*)"R:",rank,"T:",thread," j=",j
              end if
          end do
c$omp end parallel
        else if(rank.eq.3) then
          CALL MPI_RECV(sub_b,N/3,MPI_INTEGER,0,53,MPI_COMM_WORLD,
     &                    status,err)

c$omp parallel private(i,thread,command,j)
c$omp&firstprivate(rank,mark,sub_b)
         thread=omp_get_thread_num()

c$omp     do
          do i=1,N/3
              if (abs(sub_b(i)).eq.mark) then
                j=(rank-1)*(N/3)+i
                write(13,*)"R:",rank,"T:",thread," j=",j
              end if
          end do
c$omp end parallel


        end if

        CALL MPI_FINALIZE(err)
      end
```

Output of first internal ps call

```
R: 0  ps using MPI only
      PID       PPID    STIME     ELAPSED  P COMMAND
  1255106    1265106 15:55:22        0:01  * sh
  1120158    1135850 10:28:28     5:26:55  * sh
  1247528    1256632 15:33:53       21:30  * sh
  1264741    1255106 15:55:23        0:00  * mpirun
  1264866    1264945 15:55:23        0:00  7 mark
  1264945    1264741 15:55:23        0:00  * mark
  1265077    1264945 15:55:23        0:00  * mark
  1265106    1265399 15:55:21        0:02  * sh
  1265121    1264945 15:55:23        0:00  3 mark
  1265333    1264945 15:55:23        0:00  0 mark
  1265406    1265077 15:55:23        0:00 16 ps
```

Output of second internal ps call

```
R: 2 T: 0  ps within OpenMP
       PID       PPID    STIME     ELAPSED  P COMMAND
   1255106    1265106 15:55:22        0:02  * sh
   1120158    1135850 10:28:28     5:26:56  * sh
   1263527    1264866 15:55:24        0:00  2 mark
   1247528    1256632 15:33:53       21:31  * sh
   1264741    1255106 15:55:23        0:01  * mpirun
   1264817    1264866 15:55:24        0:00 21 mark
   1264866    1264945 15:55:23        0:01 14 mark
   1264945    1264741 15:55:23        0:01  * mark
   1265077    1264945 15:55:23        0:01 18 mark
   1265106    1265399 15:55:21        0:03  * sh
   1265121    1264945 15:55:23        0:01  * mark
   1265274    1264866 15:55:24        0:00  7 mark
   1265296    1265121 15:55:24        0:00  * mark
   1265297    1265121 15:55:24        0:00 26 ps
   1265299    1265121 15:55:24        0:00 13 mark
   1265312    1265333 15:55:24        0:00  3 mark
   1265314    1265121 15:55:24        0:00 27 mark
   1265333    1264945 15:55:23        0:01 16 mark
   1265338    1264866 15:55:24        0:00  * mark
   1265392    1265333 15:55:24        0:00 17 mark
   1265407    1265121 15:55:24        0:00  0 mark
   1265421    1265333 15:55:24        0:00  * mark

   1251653    1265333 15:55:24        0:00  4 mark
```

Conclusions

- At the time of the first internal `ps` call only MPI processes had been created. They are color-coded in blue and all have PPID=1264945
- At the time of the second internal `ps` call - from within a OpenMP parallel region- 12 (3x4) OpenMP parallel threads have come into existence and are marked in purple. The four threads executing in the three OpenMP parallel regions have PPIDS equal to 1264866, 1265333, and 1265121.

MLP Problem Set

① The first three problems are modifications and extensions to the `search` program shown in this chapter. For both these problems remove the code that performs the `ISHELL` command (the lines marked in purple.)

You may have noticed in the `search` program that the master process sends the appropriate array subsections then does nothing. First, modify the `search` code so the size of the array `b` is 2500000. Then, Have the master process send out the second, third, and fourth fourths of `b` to processes 1, 2, 3 respectively. It should keep the first fourth of `b` for itself and perform the same search as the slaves. The master process should output the locations of the `mark` number to file fortran.10.

② Modify the `search` program so that each process counts the number times the `mark` number appears in the array `b`, not the indices where it can be found. For this problem, just output to the monitor.

MLP Problem Set (con't)

③ Do three separate RECV statements have to be made in the search program? If not, rewrite it so that only one RECV statement is used. Use the status array to insure that the proper reception occurred.

④ Write a program in which data are exchanged between the rank1 MPI process and the rank3 MPI process. The rank1 MPI processes should calculate the square of the first 200 integers. The resulting values should then be transferred to the rank3 MPI process. The rank3 MPI process should calculate the square root of 100 real values between 20.0 and 70.0. The resulting values should then be transferred to the rank1 MPI process. THE CALCULATIONS DONE BY EACH PROCESS SHOULD BE PERFORMED IN PARALLEL WITH OPENMP THREADS. Output whatever you feel necessary to confirm that the assigned data transfers occurred correctly.

⑤ Write a program in which all the MPI processes generate and exam different sets of 2500 random integers. In addition, *only* the rank4 MPI process should use a variable called EvenCount which keeps a

MLP Problem Set (con't)

A running total of all the even numbers found by the MPI processes. That is, if any MPI processes finds an even number it should increment (safely) the EvenCount variable. The processing of the random numbers used by each MPI process should be done in parallel with OpenMP threads. After all the MPI processes are finished, rank3 MPI process should output the fraction of integers found and we will see if it is close to 50%.

Interlude 4: Laplace Solver Using OpenMP and MPI

- Applying OpenMP to the MPI Implementation
- Keeping OpenMP and MPI Separate
- Multilevel MPI/OpenMP Implementation of Laplace Solver
- Performance Characteristics of Multilevel Implementation

Applying OpenMP to the MPI Implementation

- If we look closely at the MPI implementation of the Laplace solver shown earlier, we notice that it has the same two loop structures as we parallelized in the OpenMP implementation, only with different index bounds:
  - *The initialization loop in `j` at the beginning of the program*: Again, it is necessary to run this in parallel to ensure proper memory placement on ccNUMA systems like the SGI Origin 2000.
  - *The `j` loop within the main iterative loop*: This still comprises the bulk of the computation.
- We can wrap these two loop structures in `!$OMP DO…!$OMP END DO` directives as in the OpenMP case.
- However, rather than having a single parallel region, we should have a parallel region for each section of thread-parallel code. The reason for this is to eliminate the possibility of multiple OpenMP threads calling MPI routines simultaneously.
- We also need to hardcode the number of OpenMP threads in the program, as many cluster environments don’t propagate environment variables such as OMP_NUM_THREADS to MPI processes when they are spawned.

Keeping OpenMP and MPI Separate

It is **critical** to keep OpenMP and MPI calls separate, i.e. don't call MPI routines in OpenMP parallel regions:

- Many MPI implementations, such as MPICH, are not thread-safe on all platforms.
- Calling MPI routines inside an OpenMP parallel region can result in deadlock, race conditions, or incorrect results unless wrapped in `!$OMP SINGLE` or `!$OMP MASTER` directives.
- Keeping the two cleanly separated will make an easier to understand (and debug) program.

Multilevel MPI/OpenMP Implementation of a Laplace Solver

```
      program lpmlp
      include 'mpif.h'
      integer imax,jmax,im1,im2,jm1,jm2,it,itmax
      parameter (imax=2001,jmax=2001)
      parameter (im1=imax-1,im2=imax-2,jm1=jmax-1,jm2=jmax-2)
      parameter (itmax=100)
      real*8 u(imax,jmax),du(imax,jmax),umax,dumax,tol
      parameter (umax=10.0,tol=1.0e-6)

! Additional MPI parameters
      integer istart,iend,jstart,jend
      integer size,rank,ierr,istat,mpigrid,length
      integer grdrnk,dims(1),left,right,isize,jsize
      real*8 tstart,tend,gdumax
      logical cyclic(1)
      real*8 lbuf(jmax),rbuf(jmax),ubuf(imax),dbuf(imax)
      integer nthrds
      common /T3EHACK/ istart,iend,jstart,jend
```

Multilevel Implementation (con’t)

```
! Initialize MPI
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
! 1D linear topology
      dims(1)=size
      cyclic(1)=.FALSE.
      call MPI_CART_CREATE(MPI_COMM_WORLD,1,dims,cyclic,.true.,
     +     mpigrid,ierr)
      call MPI_COMM_RANK(mpigrid,grdrnk,ierr)
      call MPI_CART_SHIFT(mpigrid,0,1,grdrnk,right,ierr)
      call MPI_CART_SHIFT(mpigrid,0,-1,grdrnk,left,ierr)
! Compute index bounds
      isize=imax
      jsize=jmax/dims(1)
      istart=loc(1)*isize+1
      if (istart.LT.2) istart=2
      iend=(loc(1)+1)*isize
```

Multilevel Implementation (con’t)

```
      if (iend.GE.imax) iend=imax-1
      jstart=loc(2)*jsize+1
      if (jstart.LT.2) jstart=2
      jend=(loc(2)+1)*jsize
      if (jend.GE.jmax) jend=jmax-1
      nthrds=2
      call omp_set_num_threads(nthrds)
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,j)
! Initialize -- done in parallel to force "first-touch" distribution
! on ccNUMA machines (i.e. O2k)
!$OMP DO
      do j=jstart-1,jend+1
         do i=istart-1,iend+1
            u(i,j)=0.0
            du(i,j)=0.0
         enddo
         u(imax,j)=umax
      enddo
!$OMP END DO
!$OMP END PARALLEL
```

Multilevel Implementation (con’t)

```
! Main computation loop
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
      do it=1,itmax
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,j)
!$OMP MASTER
         dumax=0.0
!$OMP END MASTER
!$OMP DO REDUCTION(max:dumax)
         do j=jstart,jend
            do i=istart,iend
               du(i,j)=0.25*(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))-u(i,j)
               dumax=max(dumax,abs(du(i,j)))
               u(i,j)=u(i,j)+du(i,j)
            enddo
         enddo
!$OMP END DO
!$OMP END PARALLEL
! Compute the overall residual
         call MPI_REDUCE(dumax,gdumax,1,MPI_REAL8,MPI_MAX,0
     +        ,MPI_COMM_WORLD,ierr)
```

Multilevel Implementation (con’t)

```
! Send phase
         if (left.NE.MPI_PROC_NULL) then
            i=1
            do j=jstart,jend
               lbuf(i)=u(istart,j)
               i=i+1
            enddo
            length=i-1
            call MPI_SEND(lbuf,length,MPI_REAL8,left,it,mpigrid,
     +                       ierr)
         endif
         if (right.NE.MPI_PROC_NULL) then
            i=1
            do j=jstart,jend
               rbuf(i)=u(iend,j)
               i=i+1
            enddo
            length=i-1
            call MPI_SEND(rbuf,length,MPI_REAL8,right,it,mpigrid,
     +                       ierr)
         endif
```

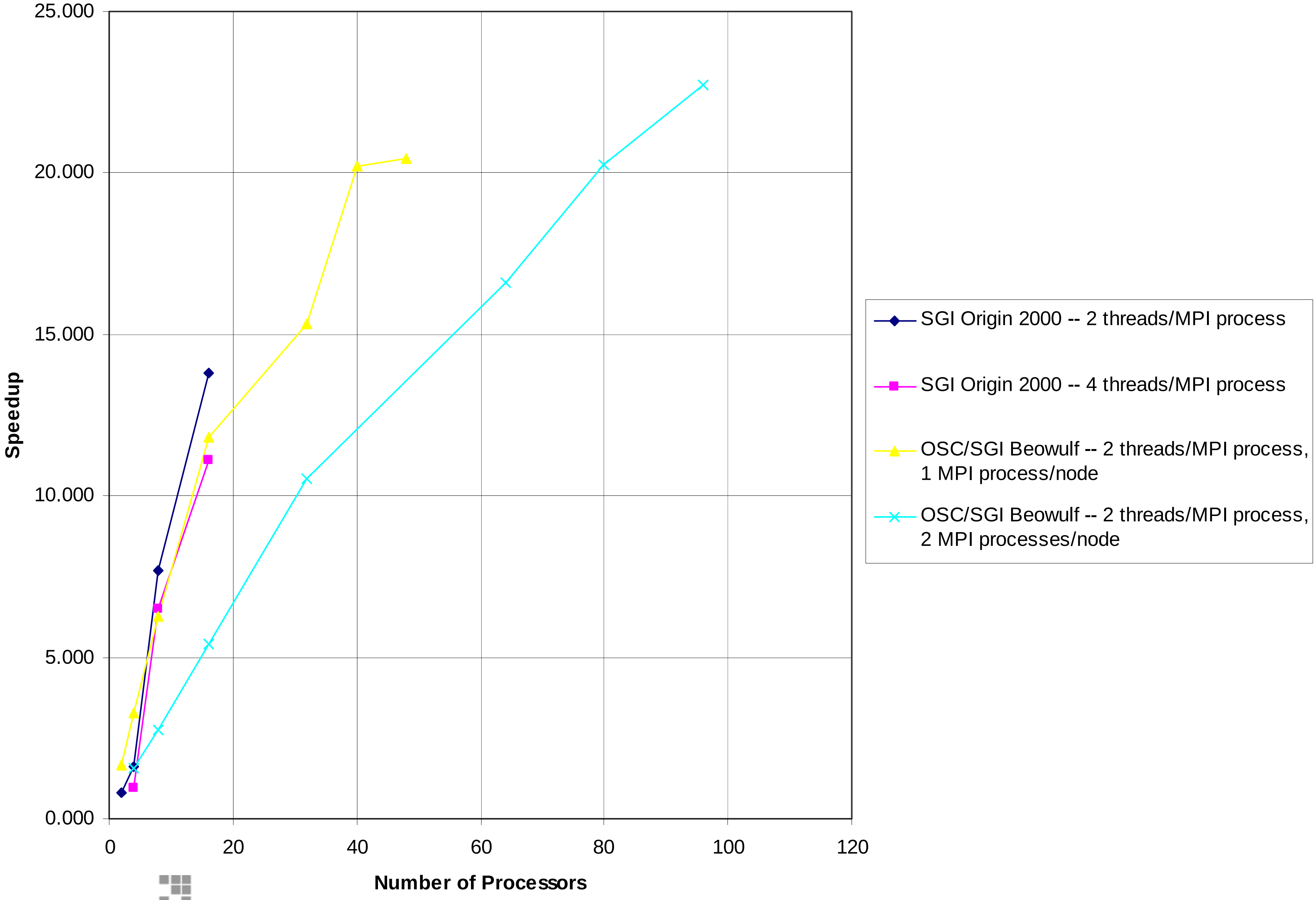
Multilevel Implementation (con’t)

```
! Receive phase
         if (left.NE.MPI_PROC_NULL) then
            length=jend-jstart+1
            call MPI_RECV(lbuf,length,MPI_REAL8,left,it,
     +           mpigrid,istat,ierr)
            i=1
            do j=jstart,jend
               u(istart-1,j)=lbuf(i)
               i=i+1
            enddo
         endif
         if (right.NE.MPI_PROC_NULL) then
            length=jend-jstart+1
            call MPI_RECV(rbuf,length,MPI_REAL8,right,it,
     +           mpigrid,istat,ierr)
            i=1
            do j=jstart,jend
               u(iend+1,j)=rbuf(i)
               i=i+1
            enddo
         endif
```

Multilevel Implementation (con’t)

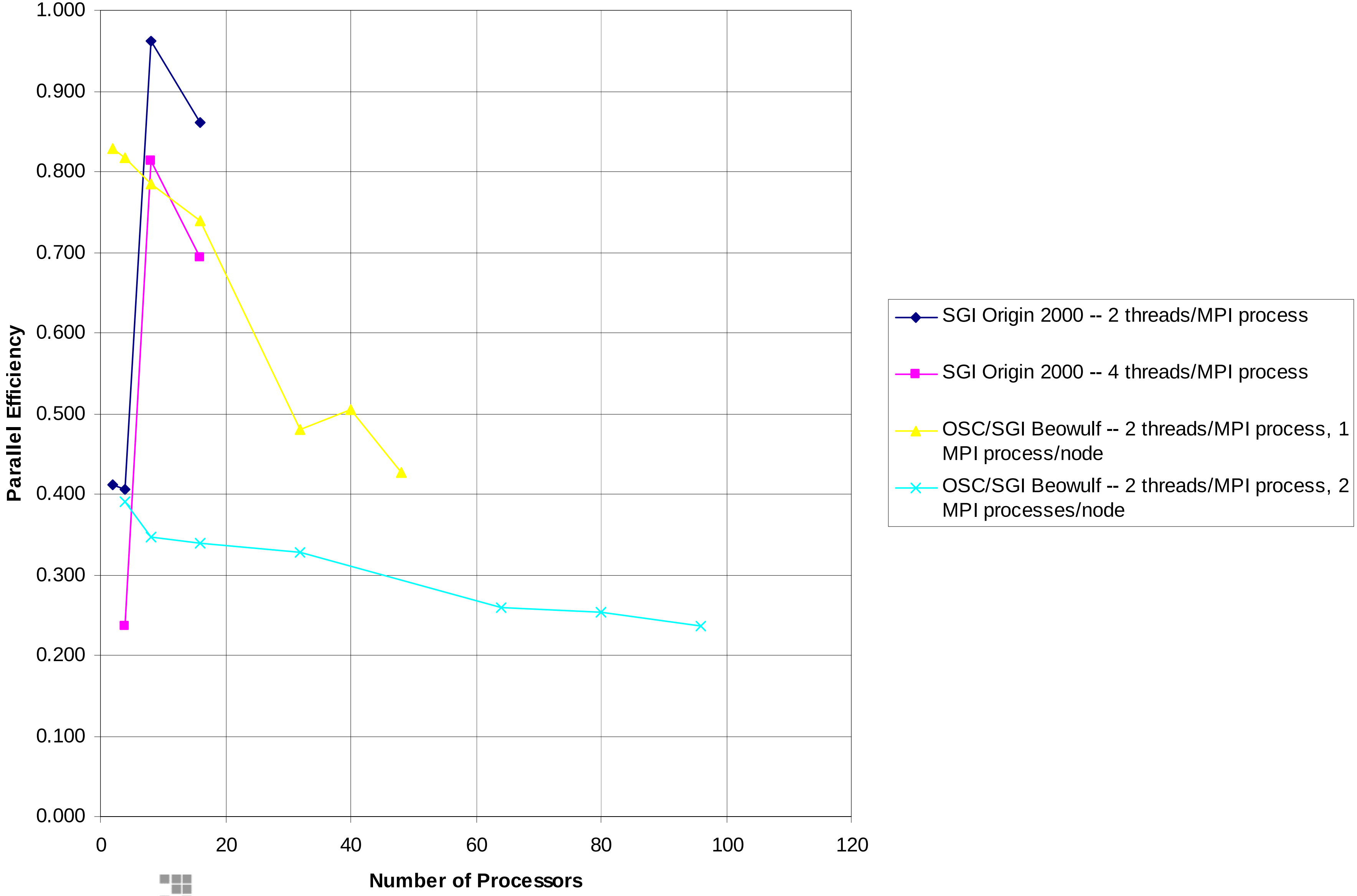
```
         write (rank+10,*) rank,it,dumax,gdumax
         if (rank.eq.0) write (1,*) it,gdumax
      enddo

! Clean up
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
      if (rank.EQ.0) then
         write(*,*) 'Calculation took ',tend-tstart,'s. on ',size,
     +        ' MPI processes’
     +        ,' with ',nthrds,' OpenMP threads per process'
      endif
      call MPI_FINALIZE(ierr)
      stop
      end
```

Performance Characteristics of Multilevel Implementation



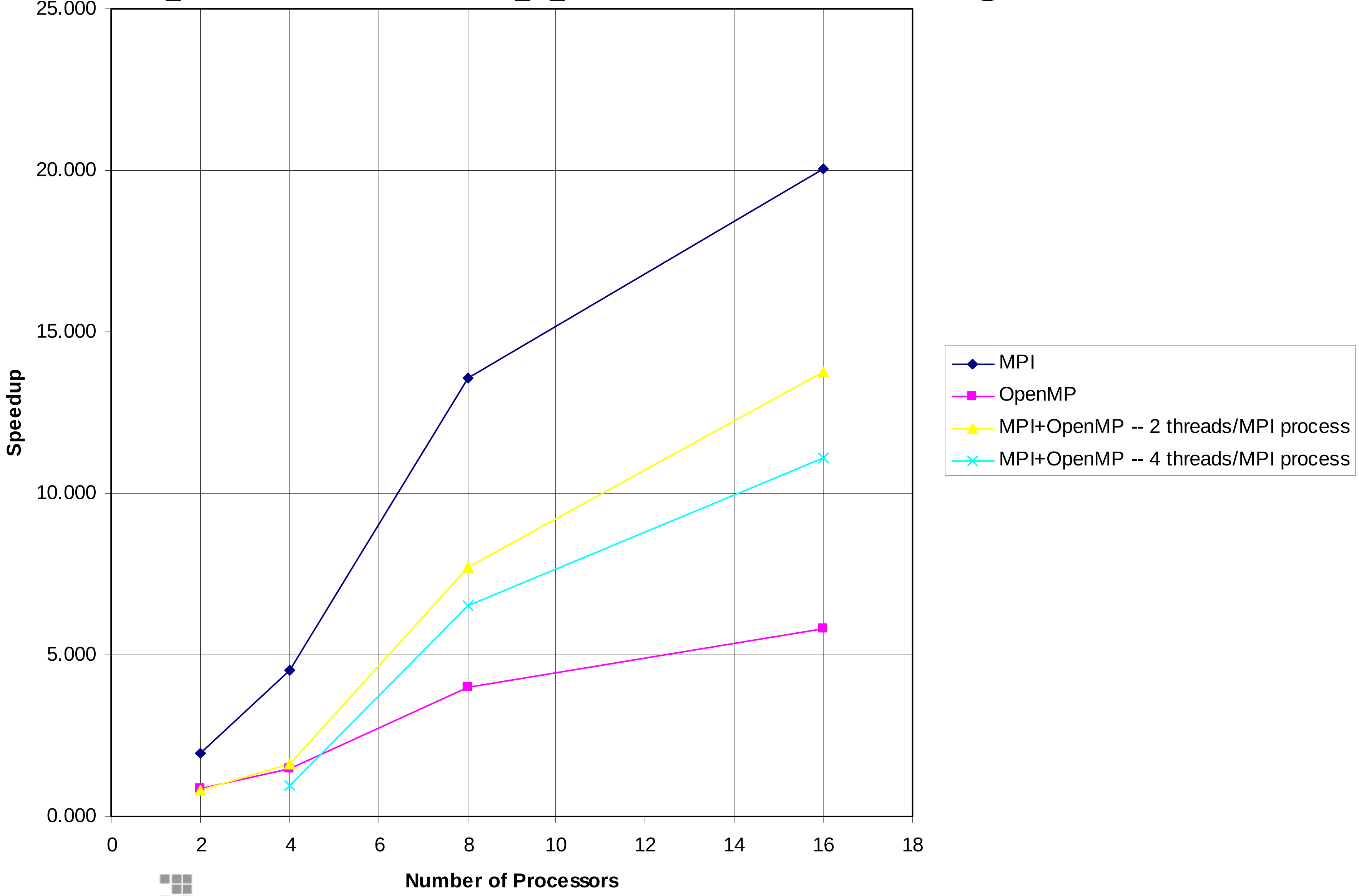
OSC
Innovations in computing, networking, and education

Performance Characteristics of Multilevel Implementation (con't)



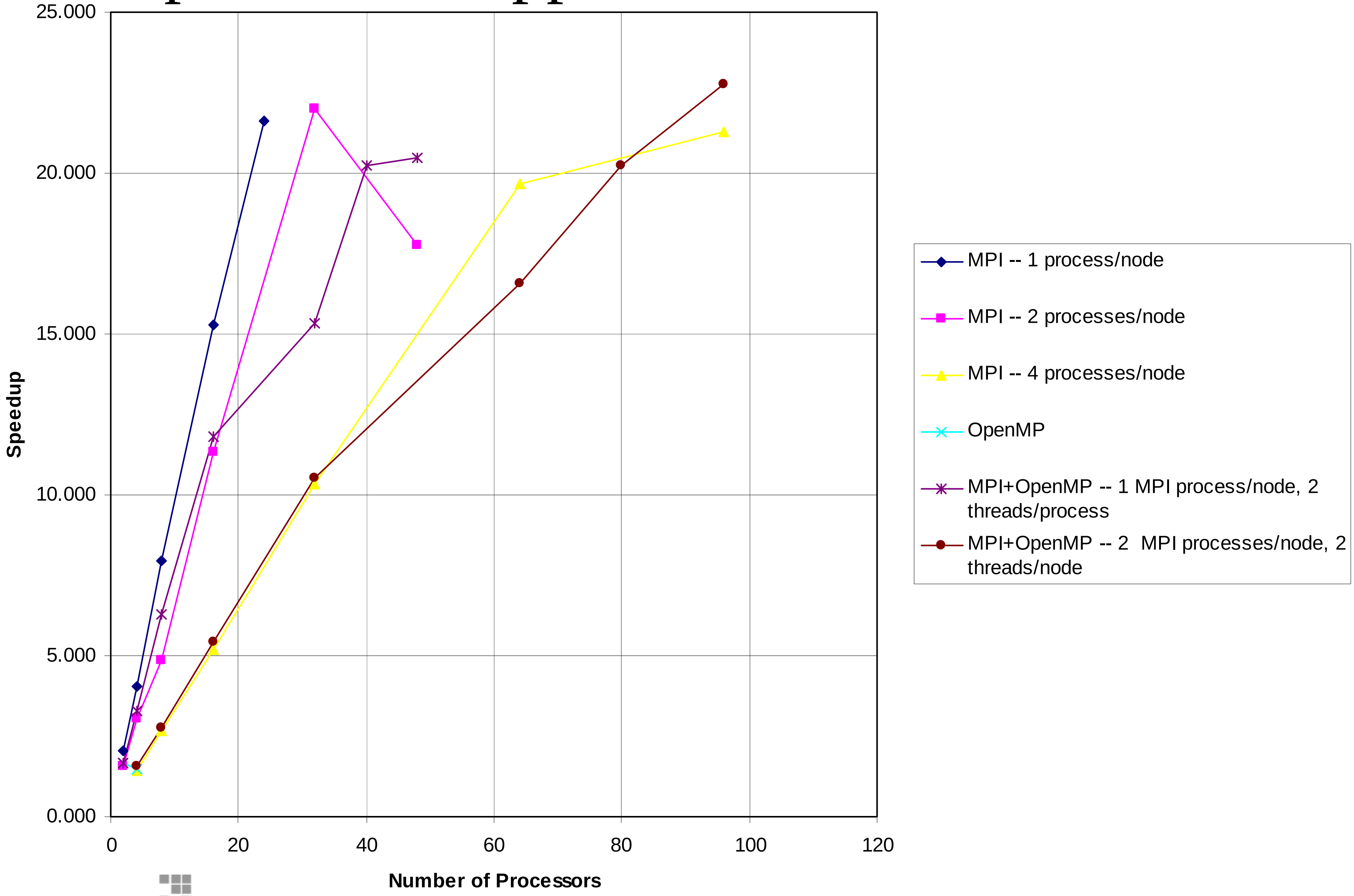
Innovations in computing, networking, and education

Comparison of Approaches -- Origin



OSC Innovations in computing, networking, and education

Comparison of Approaches -- Beowulf



OSC
Innovations in computing, networking, and education

Variations on a Theme

- POSIX Threads and Other Exercises in Masochism
- Using Threaded Libraries Within MPI Programs
- Load Balancing in Message Passing Programs (a.k.a. "Power Balancing")
- Applications
  - Adaptive Mesh Refinement
  - Multiple Levels of Detail
  - Multiblock grids

POSIX Threads and Other Exercises in Masochism

- It is possible to use a more explicit threading library such as `pthreads` (POSIX Threads) in place of the OpenMP directive-based approach.
- However, in our experience this is **extremely** painful and not for the faint of heart, especially if you are retrofitting an existing serial or MPI code.

Using Threaded Libraries Within MPI Programs

- A more implicit way of doing multilevel parallel programming is to have an MPI program make calls to multithreaded libraries such as
  - Multithreaded FFT libraries (eg. SGI’s SCSL, FFTw)
  - Multithreaded BLAS libraries (eg. SGI’s SCSL, the Intel dual P6 implementation from University of Tennessee at Knoxville)
  - Vendor supplied libraries (eg. SGI’s parallel sparse matrix solver library)
- This is considerably easier to code than mixing MPI and OpenMP, but it is also less likely to be portable between platforms.

Load Balancing Message Passing Programs

- Also known as “power balancing” in some circles.
- A technique where each MPI processes spawns a variable number of threads based on the volume of computation assigned to that process.
- This is mainly for message passing codes on large SMP-like systems or clusters of medium sized SMPs.

Applications: Adaptive Mesh Refinement

- One potential application for multilevel parallel programming is adaptive mesh refinement (AMR), a technique where grid resolutions are dynamically increased in regions where there are strong gradients.
- Here's one way a multilevel parallel AMR code could be structured:
  - Each MPI process would be assigned a segment of the domain. Each region of the domain would be allowed to enhance or de-enhance the resolution at a given location; a variable number of threads would be spawned from each process based on the volume of the computation to achieve better load balancing.
  - The solution values at the boundaries between segments would need to be kept consistent; this could easily be done using message passing.

Applications: Multiple Levels of Detail

- Another application which would likely benefit from multilevel parallel programming approach is mulitple levels of detail (multi-LOD) analysis.
- This is somewhat similar to adaptive mesh refinement; in a multi-LOD analysis, a macro-scale solution is computed, then a finer grained micro-scale analysis is applied in selected regions of interest.
- In this case, threading the micro-scale analysis could result in better overall load balancing.

Applications: Multiblock Grids

- A natural application of multilevel parallel programming techniques is multiblock grid applications, such as “chimera” CFD schemes.
- In these applications, there are several interacting grid “blocks” or “zones” which have overlapping regions.
- This type of application maps quite nicely to multilevel approachs:
  - The interaction between zones can be handled by message passing between MPI processes as part of boundary condition enforcement.
  - Each zone contains a computationally intensive loop structure which can be parallelized using OpenMP directives.

References

- D. Anderson, J. Tannehill, and R. Pletcher. *Computational Fluid Mechanics and Heat Transfer*, Hemisphere Publishing, 1984.
- S. Bova, C. Breshears, C. Cuicchi, Z. Demirbilek, and H. Gabb. “Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP”, CEWES MSRC/PET Technical Report 99-18, U.S. Army Engineering Research and Development Center (ERDC) Major Shared Resource Center (MSRC), 1999.
- W. Boyce and R. DiPrima. *Elementary Differential Equations and Boundary Value Problems*, 4th edition, John Wiley and Sons, 1986.
- R. Buyya, ed. *High Performance Cluster Computing*, Prentice-Hall, 1999.
- K. Dowd and C. Severance. *High Performance Computing*, 2nd edition, O’Reilly and Associates, 1998.

References (con't)

- D. Ennis. "Parallel Programming Using MPI," http://oscinfo.osc.edu/training/mpi/, Ohio Supercomputer Center, 1996.
- D. Ennis and D. Robertson. "Parallel Programming Using OpenMP," http://oscinfo.osc.edu/training/omp/, Ohio Supercomputer Center, 1999.
- W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*, 1st edition, MIT Press, 1994.
- N. MacDonald, E. Minty, M. Antonioletti, J. Malard, T. Harding, and S. Brown. "Writing Message-Passing Parallel Programs with MPI," http://www.epcc.ed.ac.uk/epic/mpi/notes/mpi-course-epic.book_1.html, Edinburgh Parallel Computing Centre, 1995.
- *MPI: a Message Passing Interface Standard*, version 1.1, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html, MPI Forum, 1995.

References (con’t)

- B. Nichols, D. Buttlar, and J.P. Farrell. *Pthreads Programming*, O’Reilly and Associates, 1996.
- *OpenMP C and C++ Application Interface*, version 1.0, http://www.openmp.org/mp-documents/cspec.pdf, OpenMP Architecture Review Board, 1998.
- *OpenMP Fortran Application Program Interface*, version 1.1, http://www.openmp.org/mp-documents/fspec11.pdf, OpenMP Architecture Review Board, 1999.