
Introduction to Cluster Ohio

**Science & Technology Support
High Performance Computing**

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

Table of Contents

Introduction to MPI (continued)

Collective Communications

Numerical Libraries

Solving Laplace's Equation in 2D

Problem Description

Serial Solution

Parallelize with OpenMP

Parallelize with MPI

Collective Communication

[Collective Communication](#)

[Barrier Synchronization](#)

[Broadcast*](#)

[Scatter*](#)

[Gather](#)

[Gather/Scatter Variations](#)

[Summary Illustration](#)

[Global Reduction Operations](#)

[Predefined Reduction
Operations](#)

[MPI Reduce](#)

[Minloc and Maxloc*](#)

[User-defined Reduction Operators](#)

[Reduction Operator Functions](#)

[Registering a User-defined
Reduction Operator](#)

[Variants of MPI Reduce](#)

*includes sample C and Fortran
programs

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 non-blocking collective communication**
- **No tags**
- **Receive buffers must be exactly the right size**

Barrier Synchronization

- **Red** light for each processor: turns **green** when all processors have arrived
- **Some systems offer hardware support for faster execution**
 - Cray T3E supports barriers in hardware
 - Clusters support software barrier

C:

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

Fortran:

```
INTEGER COMM, IERROR
```

```
CALL MPI_BARRIER (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:

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

- **All processes must specify same root rank and communicator**

Sample Program #2 - 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();
}
```

```
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 #2 - Fortran

```
PROGRAM broadcast
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param
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.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
```

```
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.
```

Scatter

- **One-to-all communication: different data sent to each process in the communicator (in rank order)**

C:

```
int MPI_Scatter(void* sendbuf, int sendcount,
               MPI_Datatype sendtype, void* recvbuf,
               int recvcount, MPI_Datatype recvtype, int root,
               MPI_Comm comm)
```

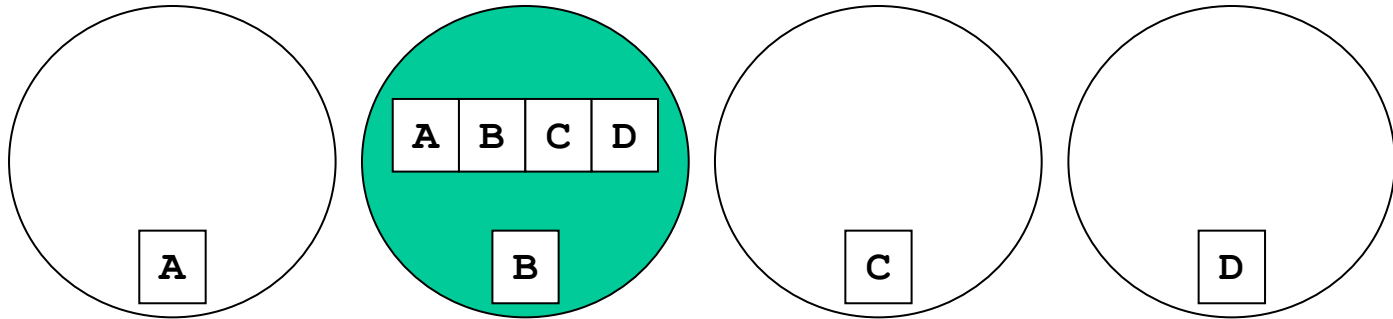
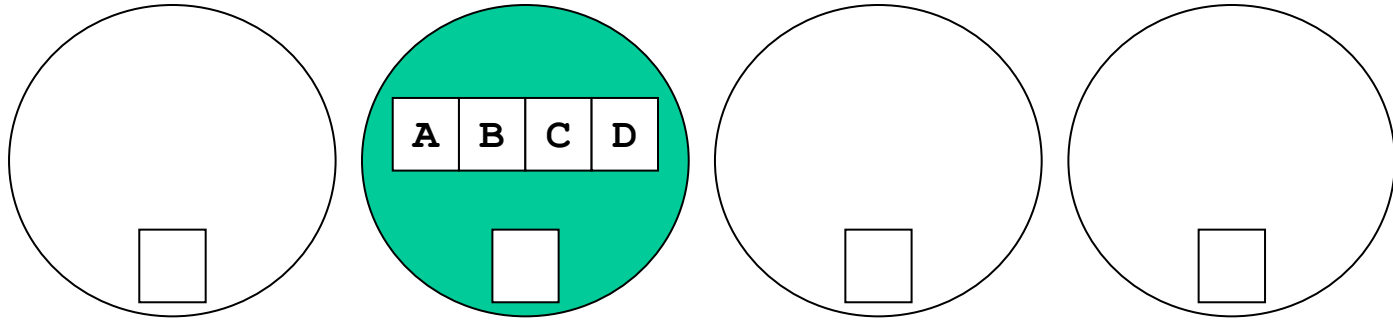
Fortran:

```
<type> SENDBUF(*), RECVBUF(*)
```

```
CALL MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
```

- **sendcount is the number of elements sent to each process, not the “total” number sent**
 - send arguments are significant only at the root process
-

Scatter Example



rank

0

1

2

3

Sample Program #3 - C

```
#include <mpi.h>
void main (int argc, char *argv[]) {
    int rank,size,i,j;
    double param[4],mine;
    int sndcnt,revcnt;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    revcnt=1;
    if(rank==3){
        for(i=0;i<4;i++) param[i]=23.0+i;
        sndcnt=1;
    }

    MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,revcnt,MPI_DOUBLE,3,MPI_COMM_
WORLD);
    printf("P:%d mine is %f\n",rank,mine);
    MPI_Finalize();
}
```

```
P:0 mine is 23.000000
P:1 mine is 24.000000
P:2 mine is 25.000000
P:3 mine is 26.000000
```

Sample Program #3 - Fortran

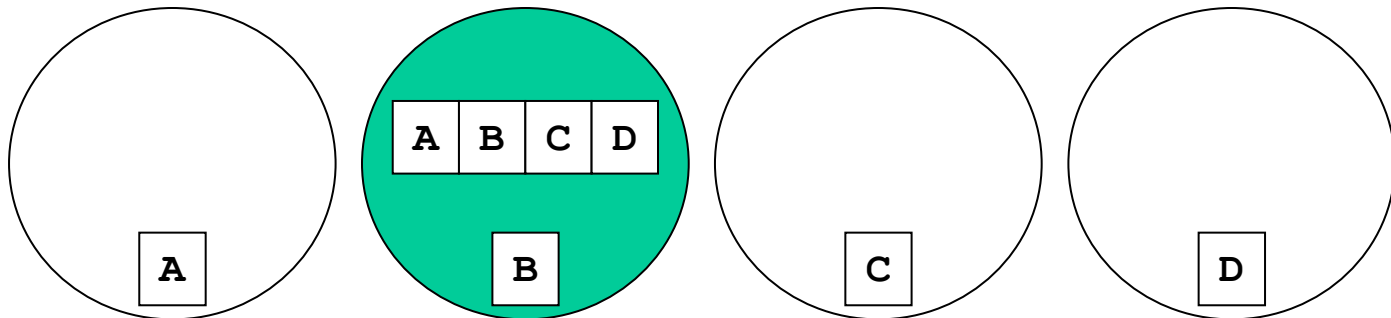
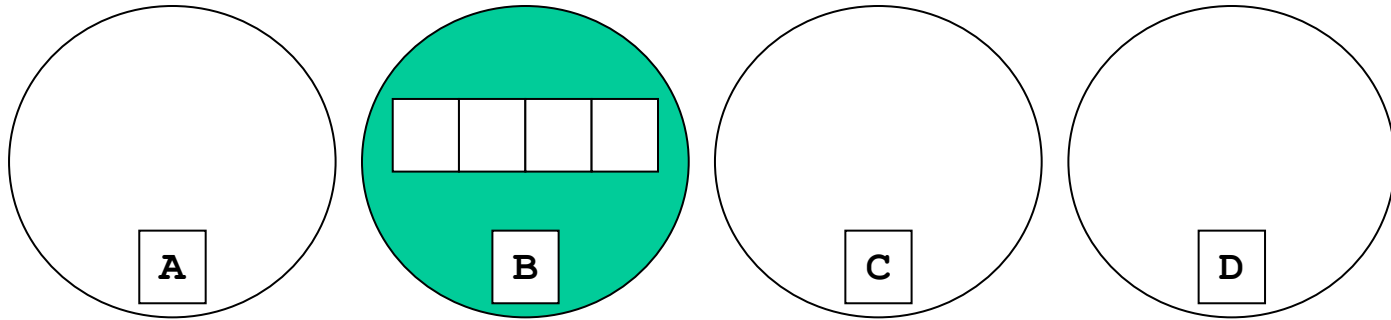
```
PROGRAM scatter
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param(4), mine
integer sndcnt,rcvcnt
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
rcvcnt=1
if(rank.eq.3) then
  do i=1,4
    param(i)=23.0+i
  end do
  sndcnt=1
end if
call MPI_SCATTER(param,sndcnt,MPI_REAL,mine,rcvcnt,MPI_REAL,
&                3,MPI_COMM_WORLD,err)
print *,"P:",rank," mine is ",mine
CALL MPI_FINALIZE(err)
END
```

```
P:1 mine is 25.
P:3 mine is 27.
P:0 mine is 24.
P:2 mine is 26.
```

Gather

- **All-to-one communication: different data collected by root process**
 - Collection done in rank order
- **MPI_GATHER (C and Fortran variants) have same arguments as matching scatter routines**
- **Receive arguments only meaningful at the root process**

Gather Example



rank

0

1

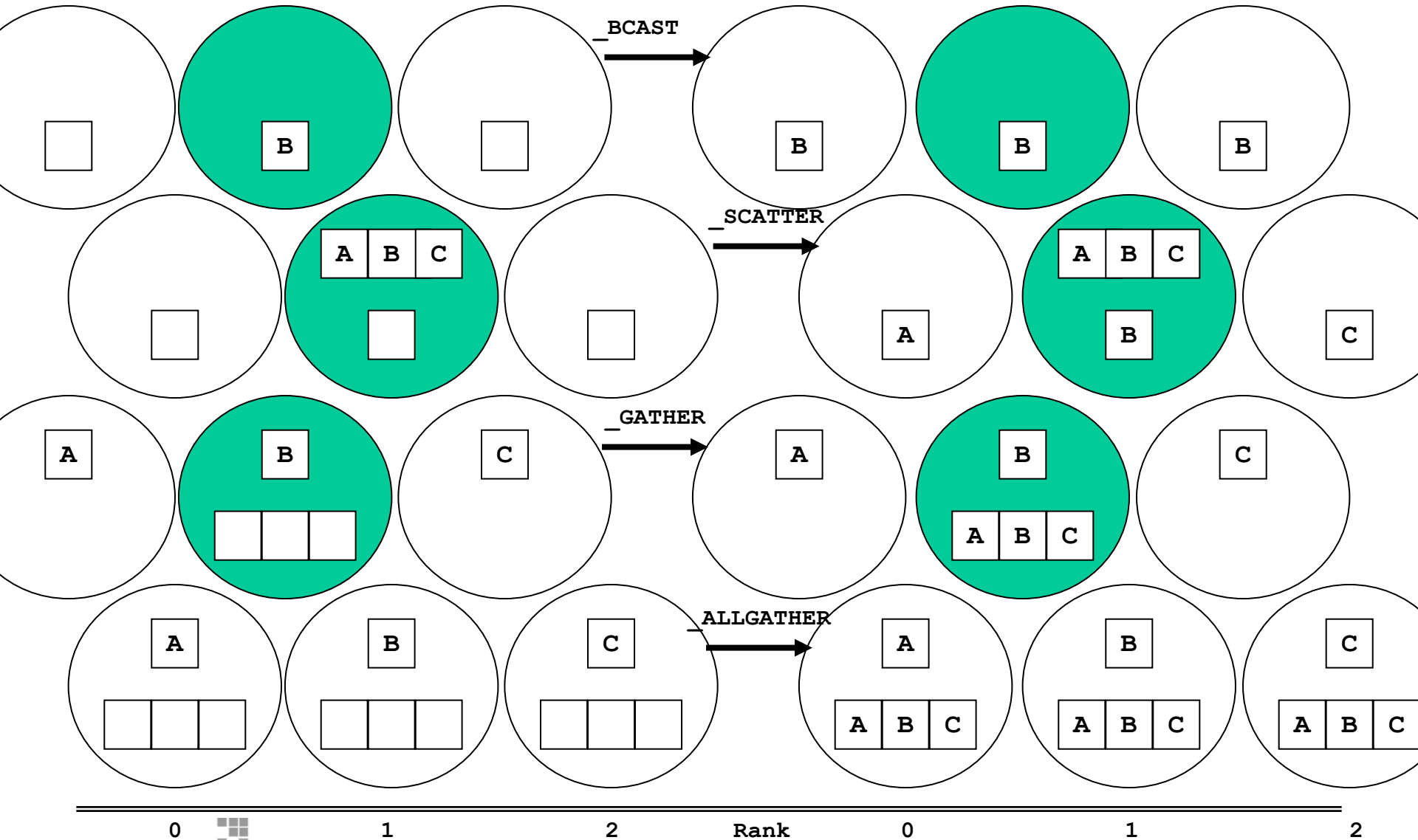
2

3

Gather/Scatter Variations

- `MPI_Allgather`
- `MPI_Alltoall`
- **No root process specified: all processes get gathered or scattered data**
- **Send and receive arguments significant for all processes**

Summary



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 a Global Sum

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

C:

```
MPI_Reduce (&x, &result, 1, MPI_INTEGER, 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

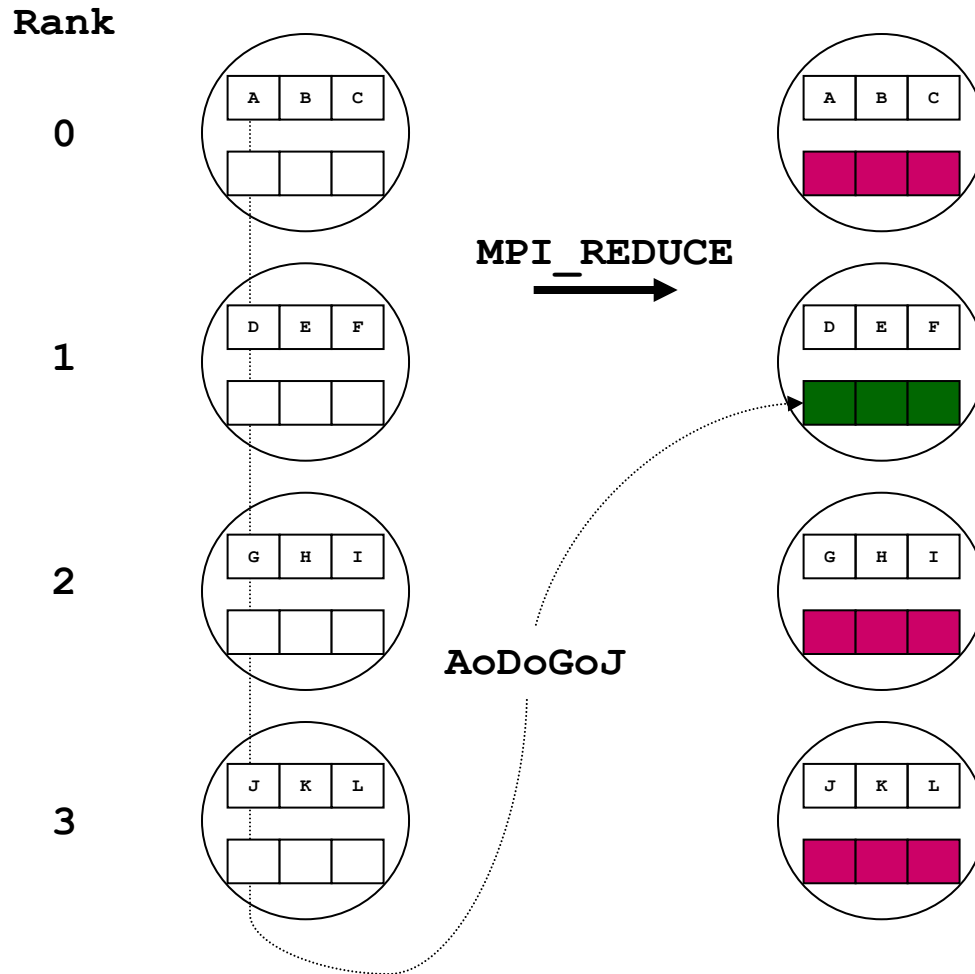
C:

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

Fortran:

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

MPI_Reduce



Numerical Libraries

Introduction to Numerical Libraries

BLAS

PBLAS

LAPACK

BLACS

ScaLAPACK

First: The Acronyms

BLAS: Basic Linear Algebra Subprograms

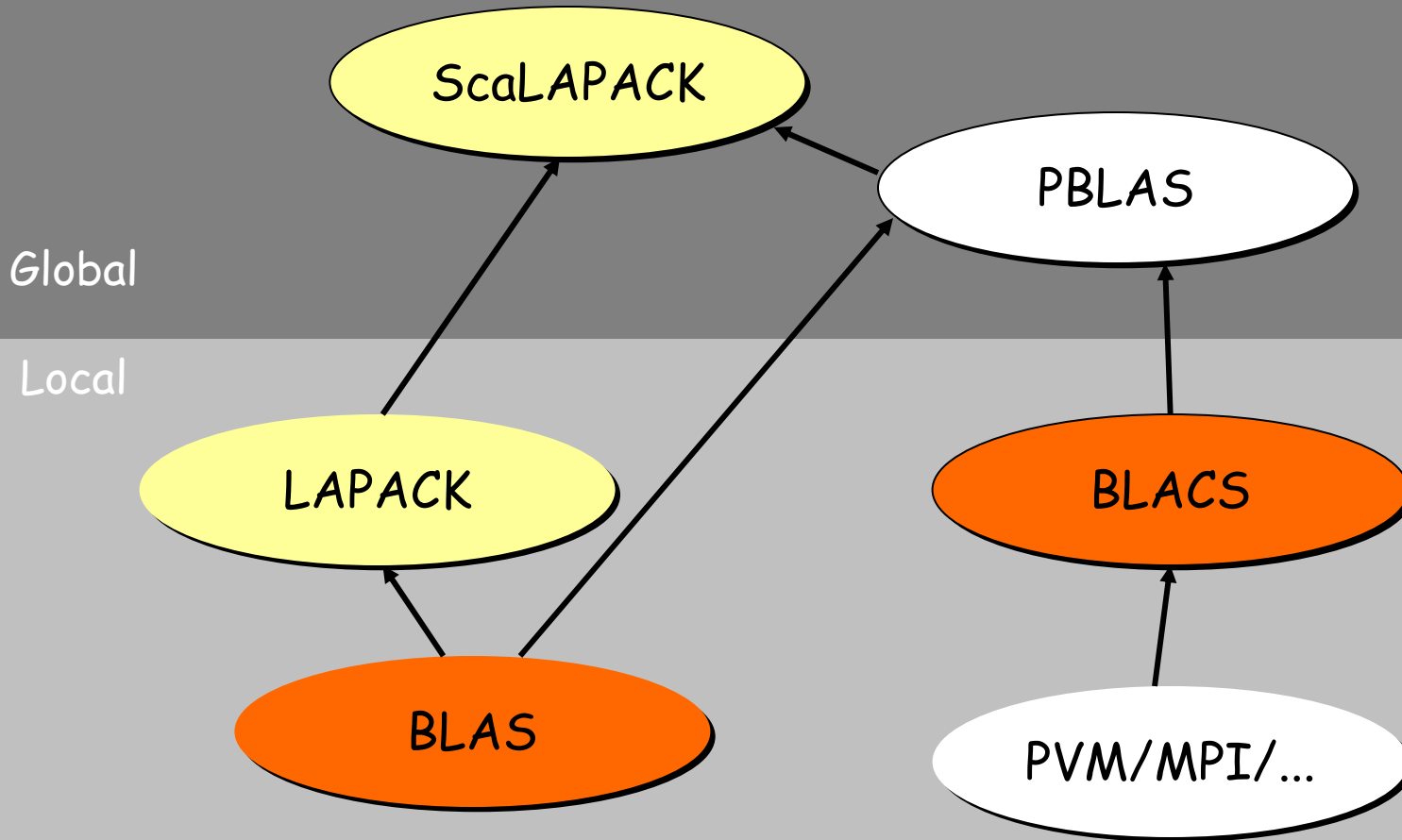
PBLAS: Parallel Basic Linear Algebra Subprograms

BLACS: Basic Linear Algebra Communication Subprograms

LAPACK: Linear Algebra PACKage

ScaLAPACK: Scalable Linear Algebra PACKage

Mathematical Library Hierarchy



Library Interconnections

The libraries lying below the “Local” keyword are serial libraries

- Optimized and only work on single processors: vector supercomputers, workstations, or one processor of an MPP system
- Use blocking algorithms to keep and reuse data in the lowest levels of the memory hierarchy: registers -> primary cache -> secondary cache -> local memory

The libraries lying above the “Global” keyword are parallel libraries

- NAG Parallel contains [and](#) is built on BLACS

The BLACS library is built on top of a message-passing library

- Install the version of BLACS that matches the message-passing library present on a given machine
- MPI is the industry standard and, therefore the most common library BLACS interfaces

BLAS

Basic Linear Algebra Subprograms

Low-level routines that perform elemental vector and matrix operations

Routines are organized in three levels

- BLAS Level 1: routines compute vector-vector operations
- BLAS Level 2: routines compute vector-matrix operations
- BLAS Level 3: routines compute matrix-matrix operations

Highly efficient versions

- Hand tuned assembly by hardware vendors
- Auto Tuning Linear Algebra Library (ATLAS)

Provide near peak performance

Additional information:

<http://www.netlib.org/blas>

BLAS/PBLAS

Serial and parallel versions of basic linear algebra procedures

Three Levels

- **Level 1:** Vector-Vector Operations
 - Examples: swap, copy, addition, dot product, ...
- **Level 2:** Matrix-Vector Operations
 - Examples: multiply, rank-updates, ...
- **Level 3:** Matrix-Matrix Operations
 - Examples: multiply, transpose, rank-updates, ...

Work with general, symmetric, complex, Hermitian, & triangular matrices

- Supercomputer dependent

Additional information:

<http://www.netlib.org/scalapack/pblas-qref.html>

LAPACK

LAPACK is a library of subroutines for solving dense linear algebra problems

- Solve systems of linear equations
- Linear least squares problems
- Eigenvalue problems
- Singular value problems

Variety of matrices supported

- bidiagonal
- diagonal
- general tridiagonal
- Many more.....

Successor to LINPACK and EISPACK

LAPACK Naming Scheme

The name of each LAPACK routine is a coded specification of its function

Most routines have names of the form $XYZZZ$

The first letter, X , indicates the data type as follows:

- S REAL
- D DOUBLE PRECISION
- C COMPLEX
- Z COMPLEX*16 or DOUBLE COMPLEX

(Remember that CRAY systems are 64 bit by default, so only single precision LAPACK functions are provided on these platforms)

The next two letters, YY , indicate the type of matrix

The last three letters ZZZ indicate the computation performed

Additional information:

<http://www.netlib.org/lapack>

<http://www.netlib.org/lapack/lug> (LAPACK Users Guide)

BLACS

BLACS (Basic Linear Algebra Communications Subprograms)

Used primarily for transferring data between processors

- **Point-Point** routines for data sent from one processor and received by another
- **Broadcast** routines for data broadcast from one processor and received by all the other processors being used
- All communication routines are **array-based**: data that is transferred is entire arrays or subarrays

Also has “**combining**” routines for doing global calculations with pieces of data in each local processors memory

- Operations: summation, maximum, & minimum

All NAG Parallel routines based on a 2-D processor grid whose size and shape is controlled by the user

Processor is identified **not** by its traditional ID number, but rather by its row and column number in the processor grid

ScaLAPACK

Parallel routines for more advanced linear algebra calculations

Three major classes

- Solution of a set of simultaneous linear equations
- Eigenvalue/Eigenvector problems
- Linear Least squares fit

Factorization algorithms used:

- LU, Cholesky, QR, LQ, Orthogonal, ...

Matrix types allowed:

- General, complex, symmetric, banded, tridiagonal, Hermitian, orthogonal, triangular, ...

OSC Class - Using Parallel Numerical Libraries (focus on ScaLAPACK)

<http://oscinfo.osc.edu/training/parlib>

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. Δx and Δy 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:

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

- 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 .

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:

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

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

Initial and Boundary Conditions

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

$$0 \leq x \leq L$$

$$0 \leq y \leq L$$

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

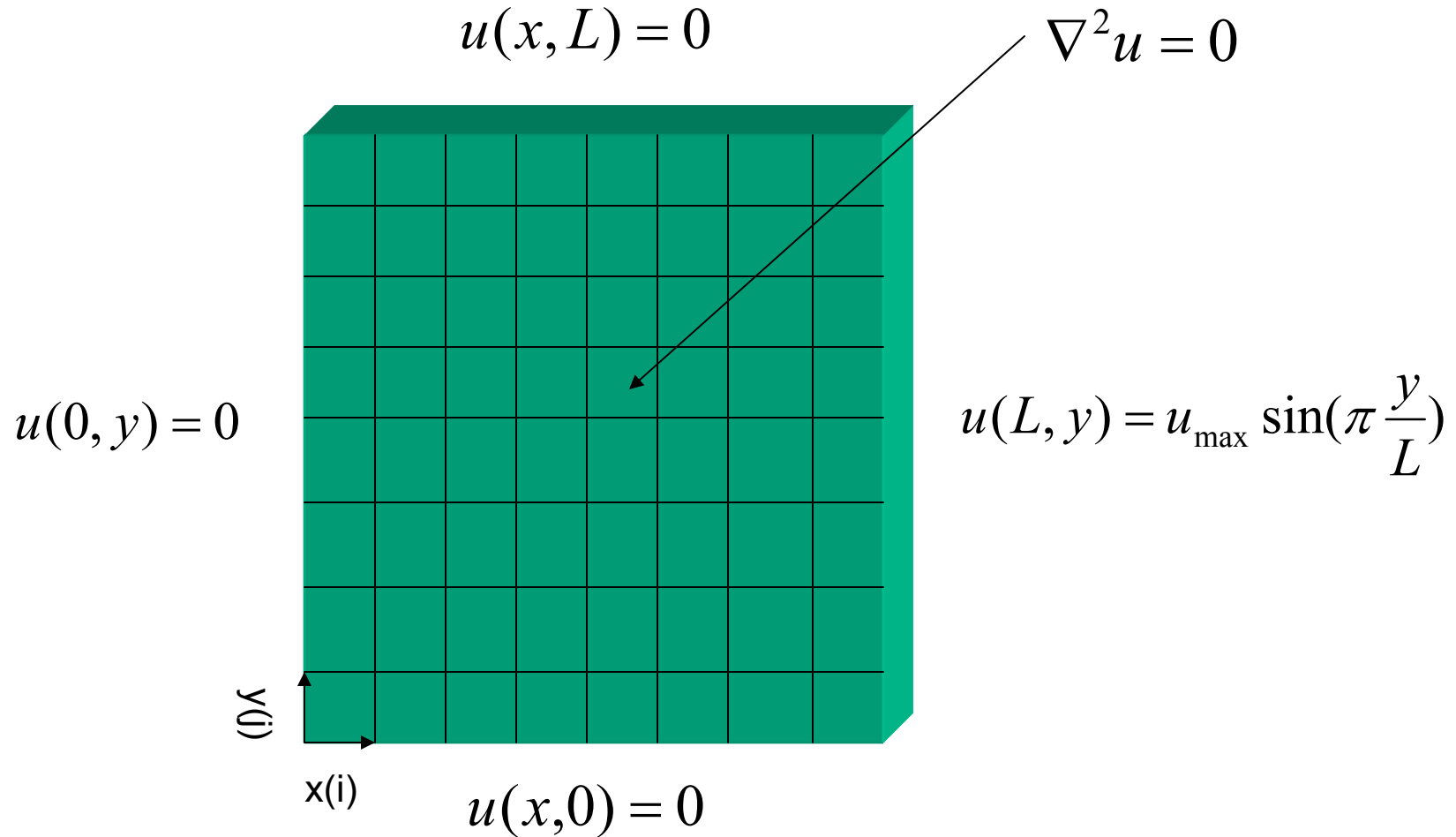
$$u(L, y) = u_{\max} \sin\left(\pi \frac{y}{L}\right)$$

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

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

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

Domain Layout



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 recurrence 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 -- Cache-Friendly

```
program lpcache
integer imax,jmax,im1,im2,jm1,jm2,it,itmax
parameter (imax=4001,jmax=4001)
parameter (im1=imax-1,im2=imax-2,jm1=jmax-1,jm2=jmax-2)
parameter (itmax=200)
real*8 u(imax,jmax),du(imax,jmax),umax,dumax,tol
parameter (umax=10.0,tol=1.0e-6)
```

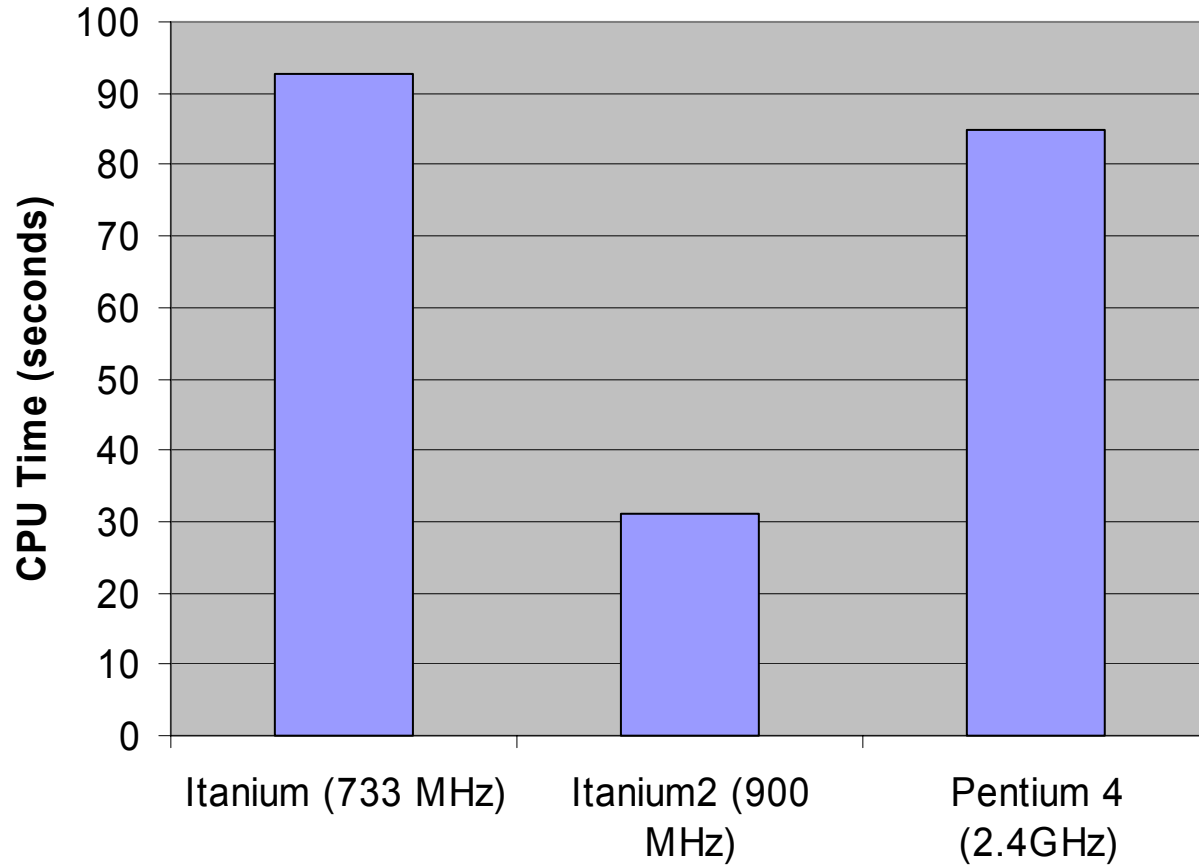
```
! Initialize
```

```
do j=1,jmax
do i=1,imax-1
u(i,j)=0.0
du(i,j)=0.0
enddo
u(imax,j)=umax
enddo
```

Serial Implementation -- Cache-Friendly

```
! 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)))
      enddo
    enddo
    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 Performance Characteristics



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 not 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 recurrence 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()` intrinsic.

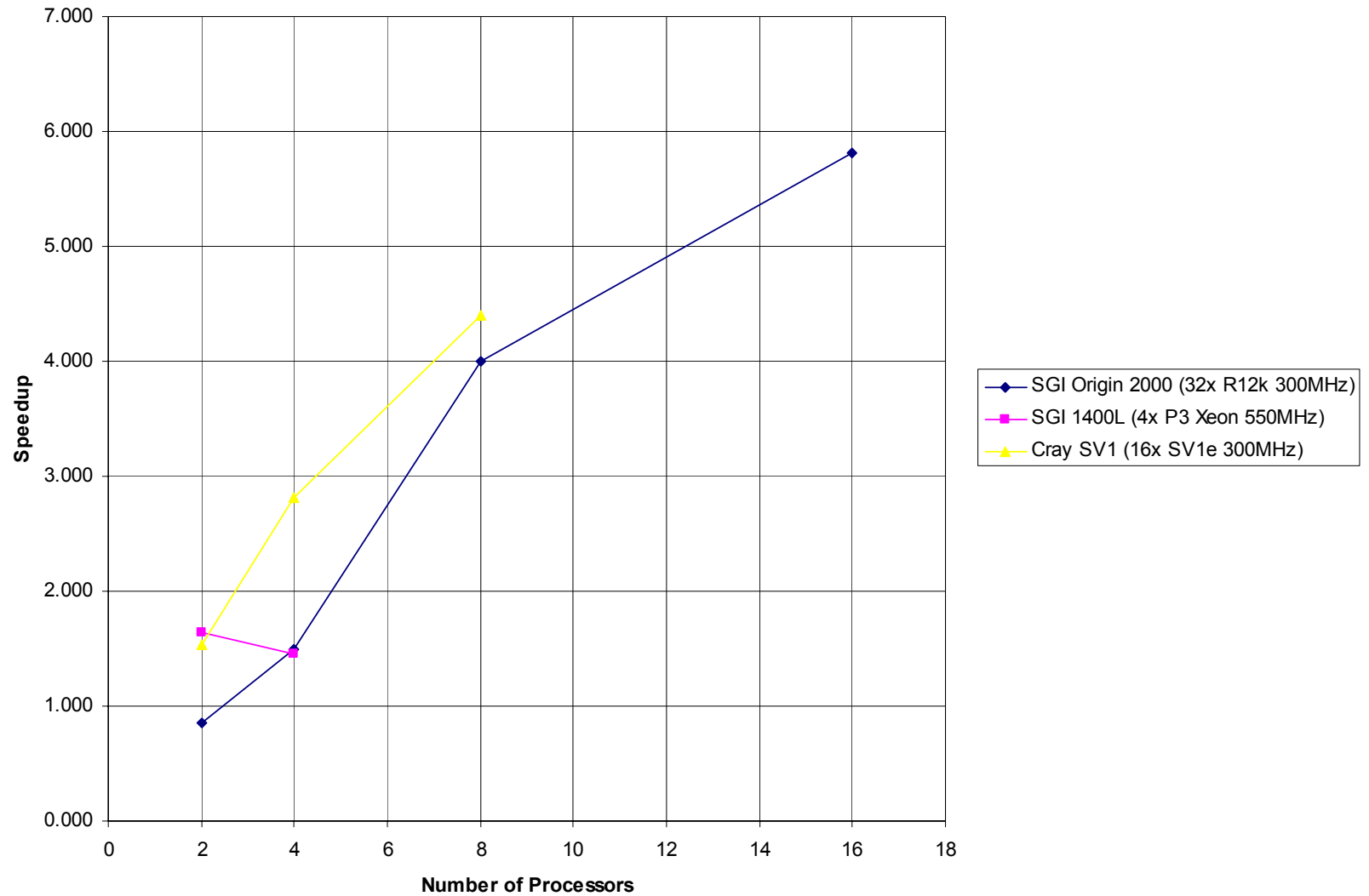
OpenMP Implementation of a Laplace Solver

```
program lpomp
integer imax,jmax,im1,im2,jm1,jm2,it,itmax
parameter (imax=4001,jmax=4001)
parameter (im1=imax-1,im2=imax-2,jm1=jmax-1,jm2=jmax-2)
parameter (itmax=200)
real*8 u(imax,jmax),du(imax,jmax),umax,dumax,tol
parameter (umax=10.0,tol=1.0e-6)
call omp_set_num_threads(2)
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,j)
!$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
  enddo
!$OMP END DO
!$OMP END PARALLEL
  it=1
  dumax=1.0
```

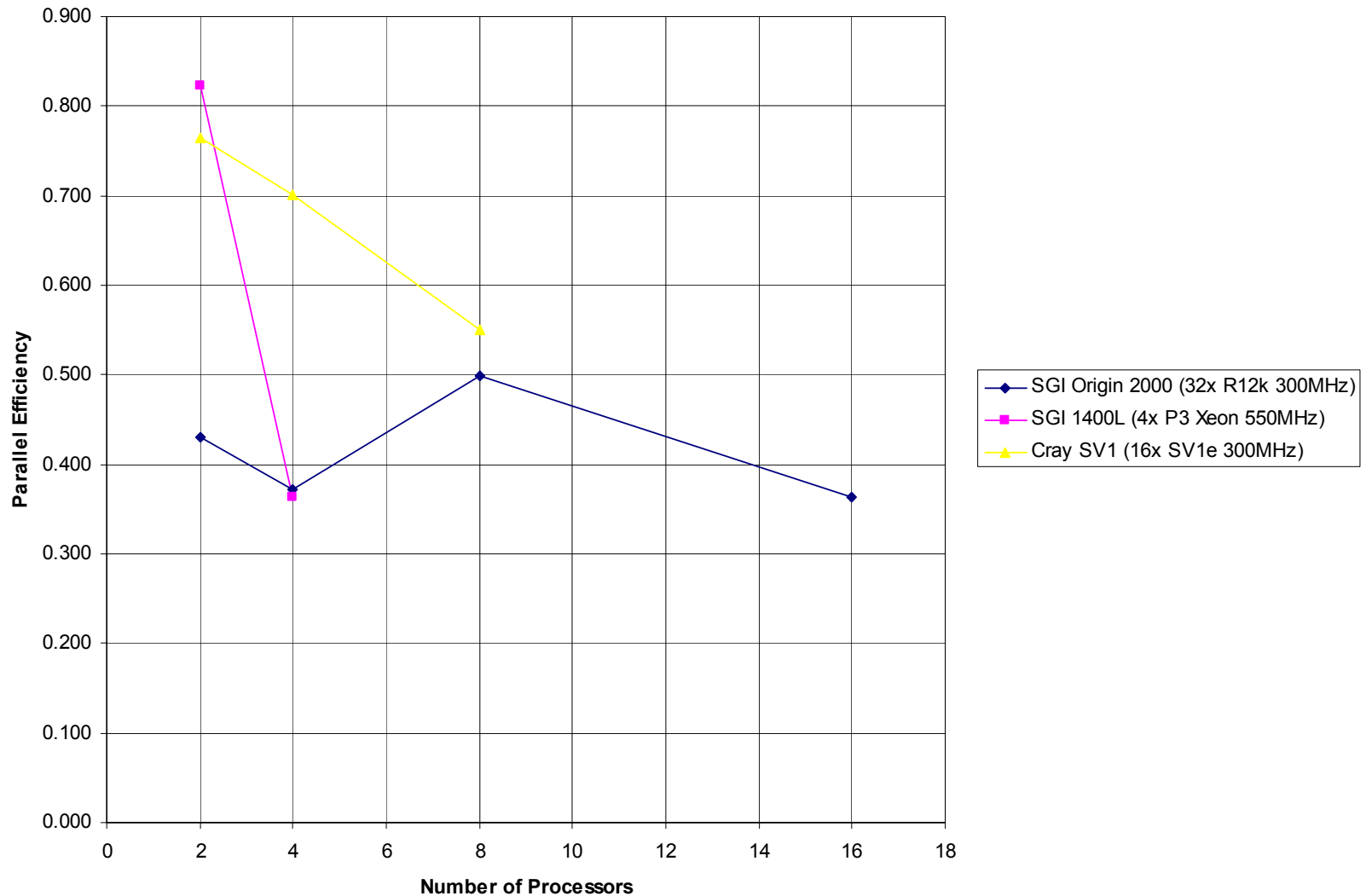
OpenMP Implementation of a Laplace Solver

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,j)
! Main computation loop
  do it=1,itmax
    dumax=0.0
!$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)))
      enddo
    enddo
!$OMP END DO
!$OMP DO
    do j=2,jm1
      do i=2,im1
        u(i,j)=u(i,j)+du(i,j)
      enddo
    enddo
!$OMP END DO
  enddo
!$OMP END PARALLEL
  stop
end
```

Performance Characteristics of OpenMP Implementation



Performance Characteristics of OpenMP Implementation (con't)



Laplace Solver Using MPI

Topologies for Domain Decomposition

Applying MPI at the Boundaries

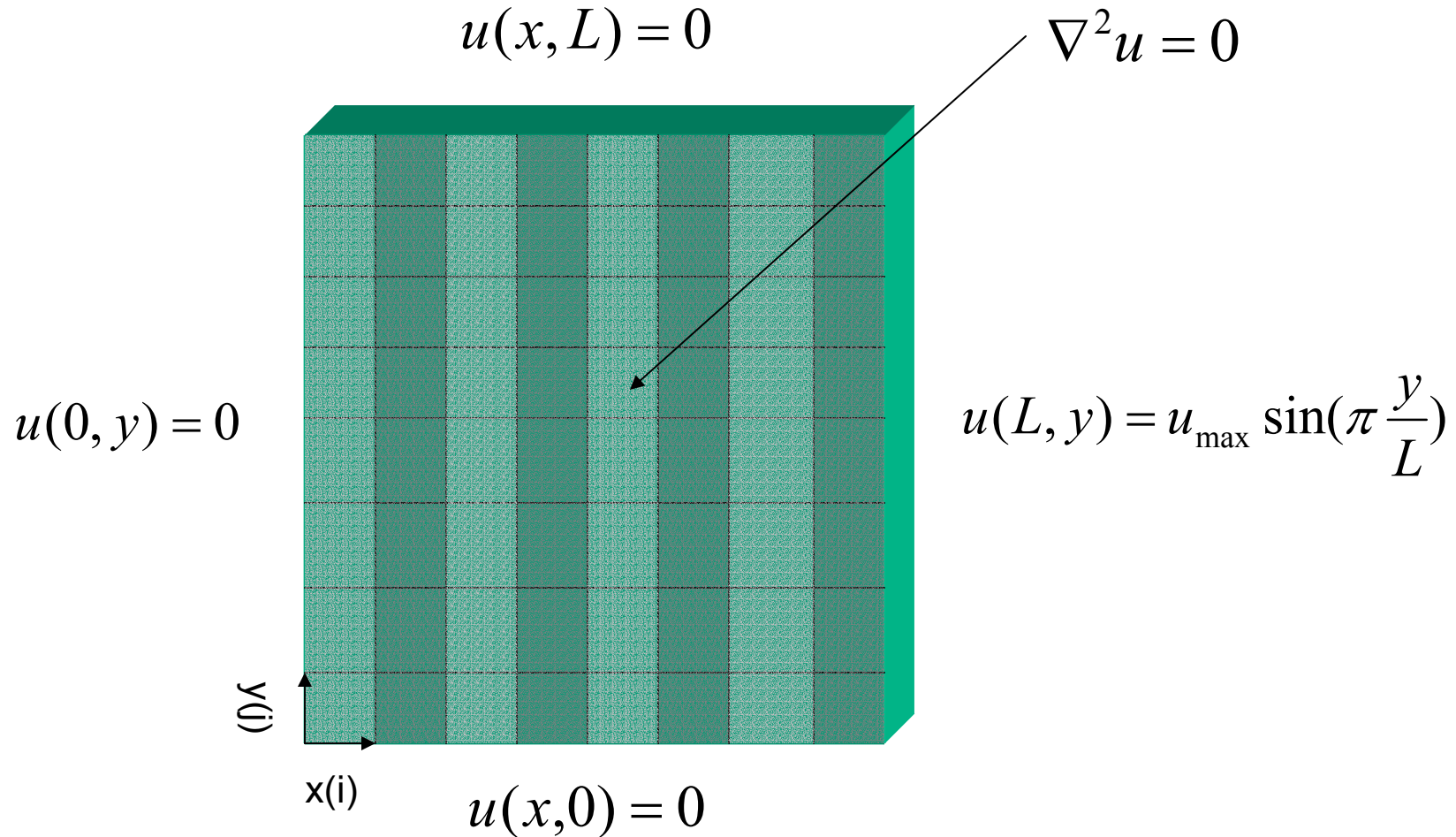
MPI Implementation of Laplace Solver

Performance Characteristics of MPI Implementation

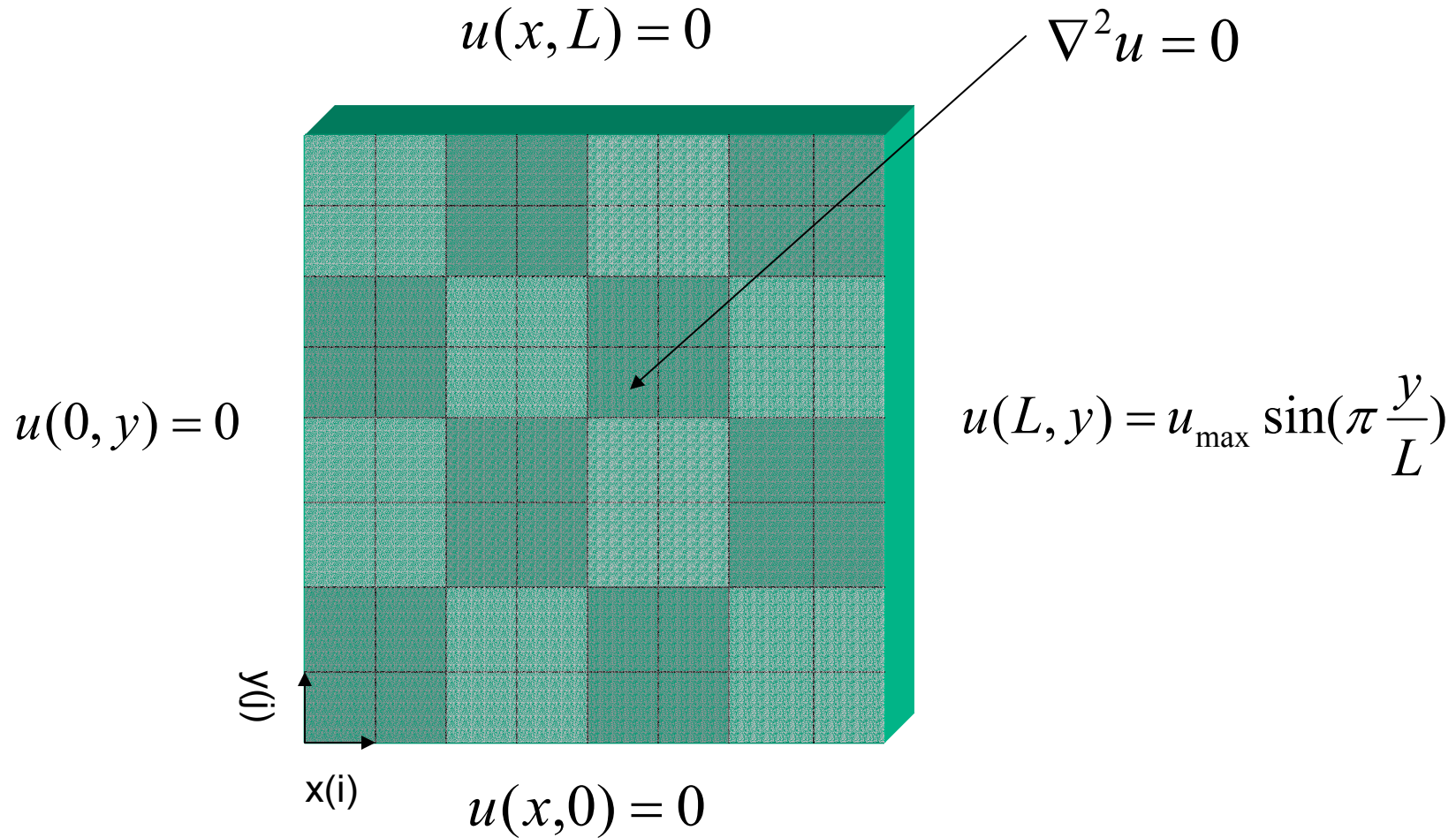
Topologies for Domain Decomposition

- 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



2D Domain Decomposition



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=4001,jmax=4001)
parameter (im1=imax-1,im2=imax-2,jm1=jmax-1,jm2=jmax-2)
parameter (itmax=200)
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(MPI_STATUS_SIZE),length
integer left,right,left_end,right_end
integer ureq,dreq
integer ustat(MPI_STATUS_SIZE),dstat(MPI_STATUS_SIZE)
real*8 tstart,tend,gdumax
real*8 uibuf(imax),dibuf(imax)

! Initialize
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)
write(6,*) 'Hello from processor',rank,' out of',size
```

MPI Implementation (cont.)

```
istart=2
iend=imax-1
jsize=jmax/size
jstart=rank*jsize+1
if (jstart.LE.1) jstart=2
right=rank+1
left=rank-1
right_end=size
left_end=-1
jend=(rank+1)*jsize
if (jend.GE.jmax) jend=jmax-1
write(6,*) 'Process',rank,' has range:',jstart,jend
write(6,*) 'Process',rank,' right=',right,' left=',left
call flush()

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
```

MPI Implementation (cont.)

```
! Main computation loop
  call MPI_BARRIER(MPI_COMM_WORLD,ierr)
  tstart=MPI_WTIME()
  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)))
      enddo
    enddo
    do j=jstart,jend
      do i=istart,iend
        u(i,j)=u(i,j)+du(i,j)
      enddo
    enddo
  ! Compute the overall residual
  call MPI_REDUCE(dumax,gdumax,1,MPI_DOUBLE_PRECISION,
+               MPI_MAX,0,MPI_COMM_WORLD,ierr)
```

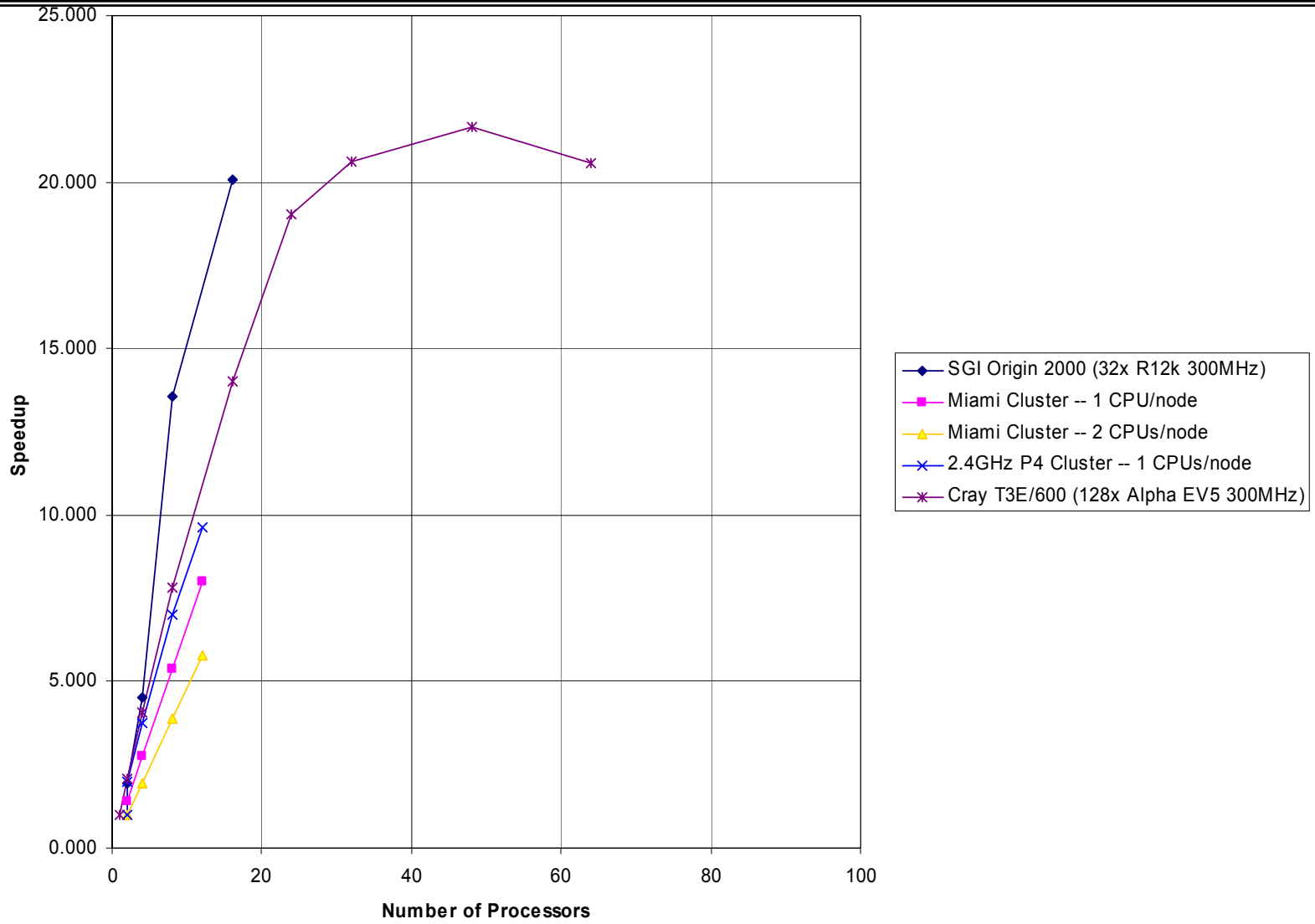
MPI Implementation (cont.)

```
! Send phase
    if (right.NE.right_end) then
        length=iend-istart+1
        call MPI_ISSEND(u(istart,jend),length,
&                      MPI_DOUBLE_PRECISION,
&                      right,1,mpi_comm_world,dreq,ierr)
    endif
    if (left.NE.left_end) then
        length=iend-istart+1
        call MPI_ISSEND(u(istart,jstart),length,
&                      MPI_DOUBLE_PRECISION,
&                      left,2,mpi_comm_world,ureq,ierr)
    endif
```

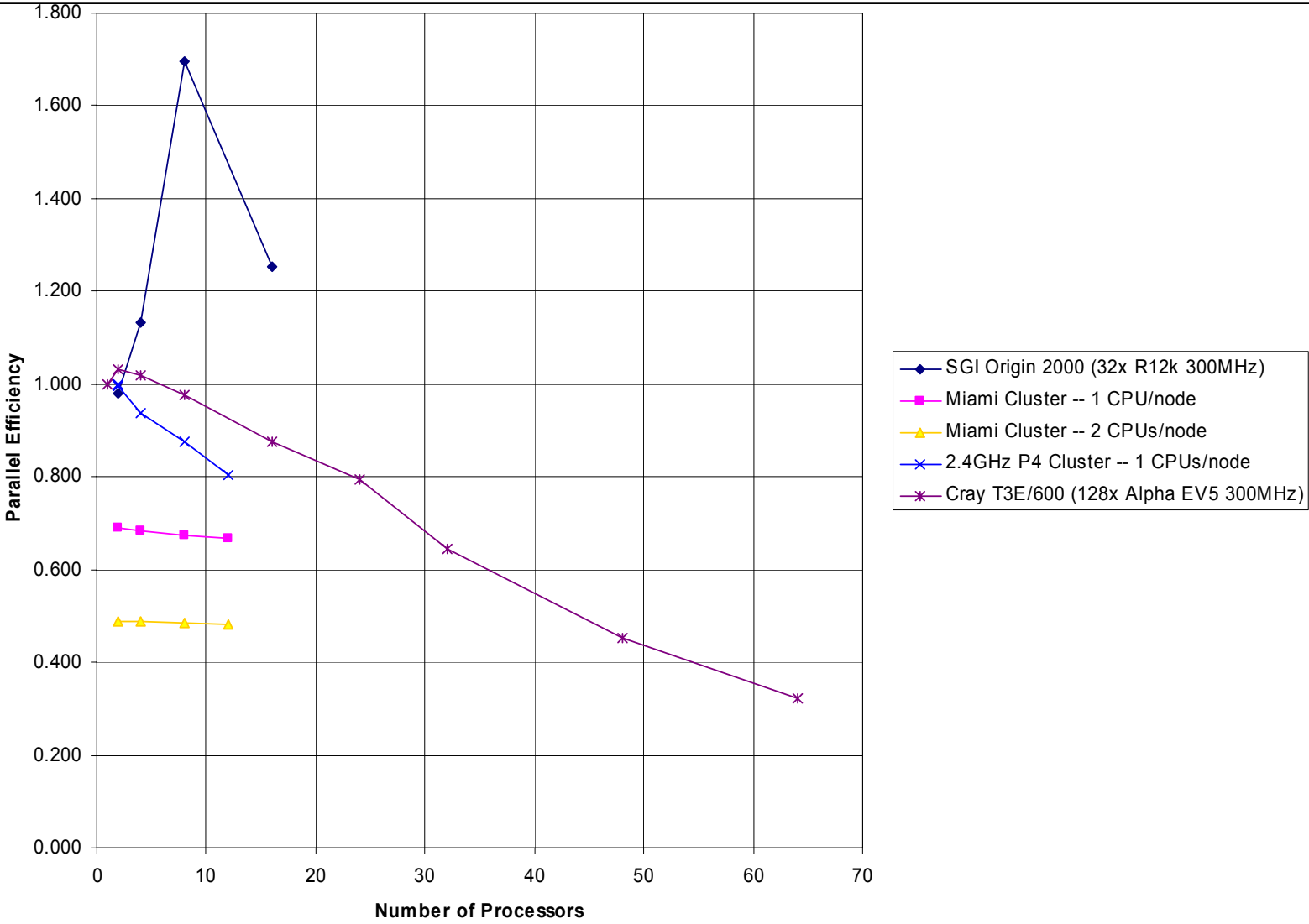
MPI Implementation (cont.)

```
! Receive phase
  if (right.NE.right_end) then
    length=iend-istart+1
    call MPI_RECV(u(istart,jend+1),length,MPI_DOUBLE_PRECISION,
&                right,2,mpi_comm_world,istat,ierr)
    call MPI_WAIT(dreq,dstat,ierr)
  endif
  if (left.NE.left_end) then
    length=iend-istart+1
    call MPI_RECV(u(istart,jstart-1),length,MPI_DOUBLE_PRECISION
&                ,left,1,mpi_comm_world,istat,ierr)
    call MPI_WAIT(ureq,ustat,ierr)
  endif
c  write (rank+10,*) rank,it,dumax,gdumax
  call flush()
  if (rank.eq.0) write (6,*) it,gdumax
enddo
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



Performance Characteristics of MPI Implementation(cont.)



Shell Script Writing

What is a shell script?

- A shell script is a collection of Unix commands entered into a file. When the file (script) is executed, the commands are executed in order.
- Running a shell script is EXACTLY equivalent to interactively entering each command
- For example, say a user **always** does the same four steps before printing out a file: making the file owner- writable, using vi to check/change the file, printing it out, and then making the file unwritable (for safety reasons)
- An interactive session of a user following this procedure for the file comp.txt is shown on the next page

Script Programming Language

The real power in writing shell scripts comes from using special shell commands and variables that essentially make your script act like a program

Programming statements which can be used in scripts include:

- Initializing and using script variables, both single valued and arrays
- Performing operations with variables
- Decision-making statements (if-then-else, etc)
- Looping statements (for, etc)
- Defining and using functions in scripts
- Script I/O commands and procedures

Script programming has less capabilities and commands than formal programming languages such as Fortran or C

Why use shell scripts?

- Convenience: Run a number of Unix commands by typing a single script name
- Command log: Script can permanently contain a detailed set of commands with numerous options. (If interactive, complex command is gone)
- Usefulness: Write a script with programming commands to give the script a number of capabilities/options
- Take “full” use of what the shell has built-in to offer you
- Considering a script as a program, no compiler, loader, or libraries needed to make it executable
- Portability: If a script works in the shell on your machine it will work on the same shell on a different machine
- Easy to debug: Small number of programming statements which are relatively simple
- Easy to modify, expand, and customize: Just type in the new commands
- Run-type interaction: Can send signals to running scripts

Korn shell scripts

Several types of Unix shells exist, each with their own script programming languages:

- Bourne (sh), Born Again shell (bash), Korn shell
- C shell (csh), TENEX C shell (tcsh)

Features you would expect from a well-written shell:

- Convenient interactive features: command history, command-line editing, and filename completion
- Powerful scripting capabilities including reliability and efficiency
- Examples in this material will use Korn shell scripting
 - It has all the convenient interactive features
 - It is backward compatible with the Bourne shell (the first shell)
 - Most scripts are sh
 - Start-up sequence for ksh same as sh only enhanced
 - Unix utilities and administration commands depend on sh
 - It does not have the scripting bugs that the C shell has

References

Internet URLs

- nacphy.physics.orst.edu/rubin/melanie/node144.html (**Tutorial**)
- gonzo.tamu.edu/csh.whynot.html (**Bugs in C shell scripting**)

Usenet Newsgroups

- comp.unix.shell
- comp.unix.questions

Anonymous ftp

- athos.rutgers.edu/~patra/vi.html (**shell-101.BetaA.Z**)
- rtfm.mit.edu:/pub/usenet/news.answers/unix-faq/faq (**Unix FAQ**)

OSC Unix-Series workshops (oscinfo.osc.edu)

- Basic Unix
- Intermediate Unix
- Introduction to Perl
- Unix Korn Shell Scripting

A Simple Script

In the script called `triangle`, a C program is compiled and then executed. The program calculates the area of the triangle and reads in the lengths of the sides from a file and outputs the result to another file.

Here is an example of a user running the script:

```
$ cat triangle
cc -n32 -cckr -O3 -LNO -Ofast -TARG:madd=on -o area area.c
area < sides.in > area.dat
cat area.dat
rm area
$ ls -l triangle
-rwxr--r--  1 dje      appl      108 May  3 12:09 triangle
$ cat sides.in
8 3 7
$ triangle
Area of Triangle is 10.392305
```

A Simple Script

Limitation of the simple triangle script

- Script only works for the one area program. **What if you want to run other triangle-related programs with this same script ?**

A More Versatile Script

Scripts can take arguments on the command line when they are run. The first argument is passed to a variable with a special name: **\$1**. We can use this feature to make the triangle script work for **any** program.

Here is the resulting new script in action:

```
$ cat triangle
cc -n32 -cckr -O3 -LNO -Ofast -TARG:madd=on -o $1 $1.c
$1 < sides.in > $1.dat
cat $1.dat
rm $1

$ triangle area
Area of Triangle is 10.392305

$ triangle perim
Perimeter of Triangle is 18.000000
```

A More Versatile Script

Limitation of versatile script:

- Scripts should be as independent of possible of “outside” existing files (for portability and ease of use). Since the sides.in file is so short, we should create the file within the script

Creating a Shell Script

A shell script is an executable file which is executed by the shell line-by-line. It can contain the following:

- UNIX commands
- shell programming statements
- comments

Create using editor of choice

Can include a `#!` construct in first line of script to override login shell

- `#!/bin/ksh` uses Korn shell to execute script
- `#!/bin/csh` uses C shell to execute script
- Recommend always put this in every Korn shell script. Makes it clear what the file is. "file" Unix command will even identify it as a Korn Shell script

Name should be short and descriptive

Executing a Shell Script

There are 3 ways to execute a shell script:

1."dot" method

```
$ . scriptname
```

2."just the name" method

```
$ scriptname
```

3.in the background

```
$ scriptname &
```

Executing a Shell Script

Method 1 runs the command as if you typed them in on the command line

- No ksh subprocess created

Note that methods 2 and 3 require:

- execute permission for scriptname

```
chmod +x scriptname
```

- current directory (.) must be in PATH or else must use `./scriptname`

Repeating Program Execution

It is often desired to run a program several times. Even a serial programs may have to run repeatedly to debug intermittent problems, find logical errors, etc.

In HPC parallel processing arena, there are even more reasons to write a script that will execute code many times:

- Each run is with a different number of processors so that the traditional speedup graph data can be collected (and compared with Ahmdal's Law)
- To perform **repeated timing** of the entire or partial code.
 - Several parallel machines do not allow dedicated use of processors
 - A severe load imbalance may come and go depending on how concurrently the various processors actually work
 - Over synchronized program's times can vary due to the order of arrival of the processors at barriers
- To gather data from program output for statistical analysis. One can also gather timing data for similar reasons.
- To confirm that a parallel program is preventing race conditions which can arise when one processor is interrupted by another when the former is trying to read, update, and write back to a shared memory location

Repeated runs is easily done in our introductory triangle script by using a while loop. Here the script will run the timearea code which times how long the program takes to calculate the area

Timing Script

```
$ cat triangle
cc -n32 -cckr -O3 -LNO -Ofast -TARG:madd=on -o $1 $1.c
cat << EOF > sides.in
8 3 7
EOF
let i=0
while (( i < $2 )); do
    $1 < sides.in >> $1.dat
    let i=i+1
done
cat $1.dat
rm $1 sides.in
```

Timing Script Usage & Output

```
$ triangle timearea 17
```

```
1.215377
```

```
1.227504
```

```
1.197454
```

```
1.193690
```

```
1.220033
```

```
1.190733
```

```
1.194406
```

```
1.221519
```

```
1.201336
```

```
1.187614
```

```
1.197026
```

```
1.220640
```

```
1.217389
```

```
1.218919
```

```
1.192417
```

```
1.218107
```

```
1.228289
```