

---

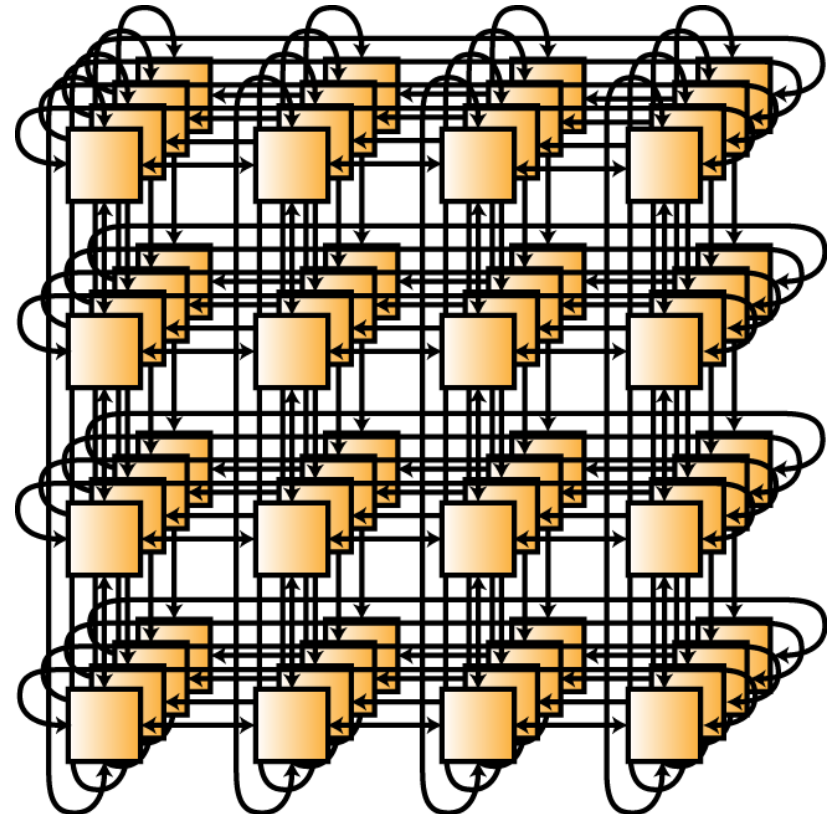
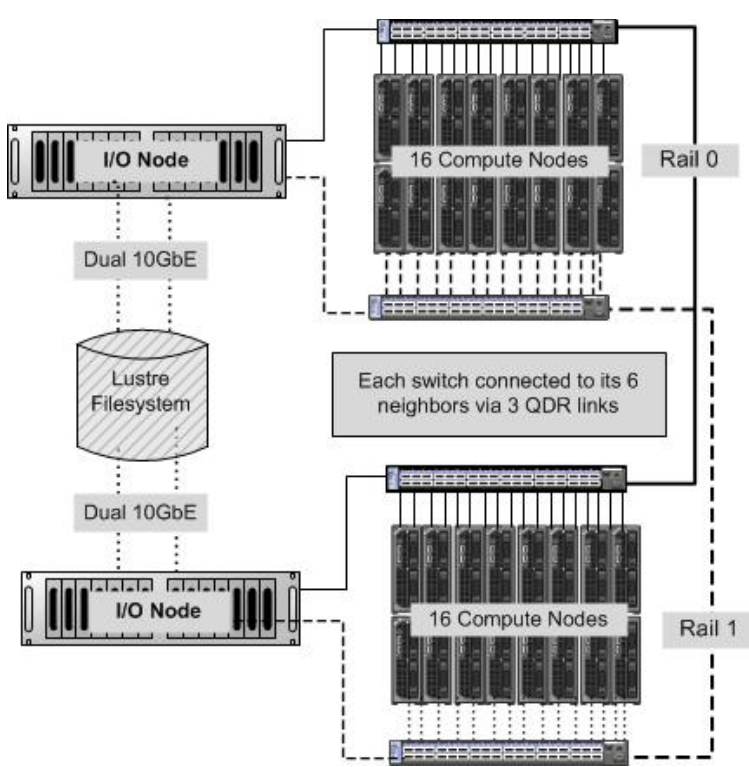
# *Parallel Computing Using MPI & OpenMP*

**Mahidhar Tatineni**  
**(mahidhar@sdsc.edu)**

**08/06/2014**

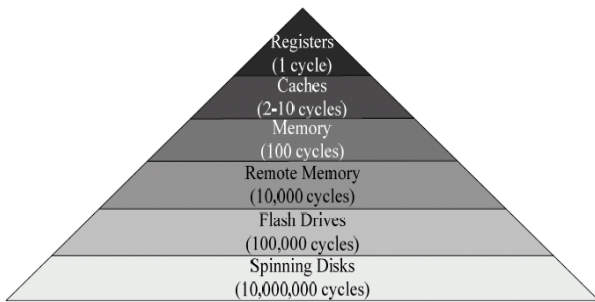
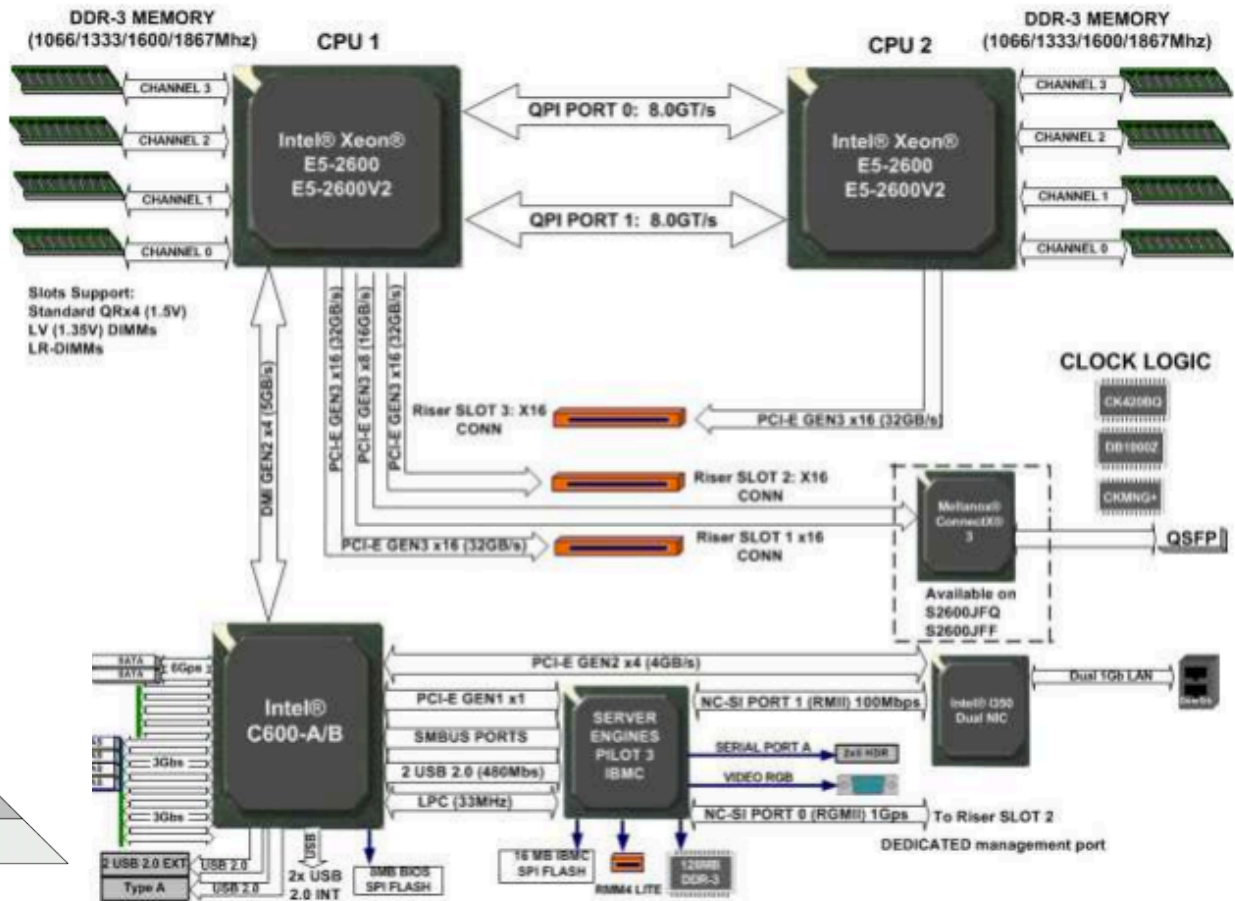
# General System Architecture Overview

- Today's supercomputers are massively parallel with thousands of nodes interconnected with a fast low latency network.
- The nodes typically feature multiple processors, each of which has multiple cores. Gordon system level architecture shown below:



# Typical Node Architecture

- Typical compute node features a multi-socket (processor) architecture. Each processor has multiple cores.
- Memory hierarchy – multiple levels of cache, main memory, disk.



---

# *Parallel Computing*

- Bulk of the speed up on modern supercomputers comes from scaling up the number of processing units available => programmers need to parallelize their codes to see this benefit.
- Parallelism needed both at the node level (given the availability of multiple cores), and at the system level (to utilize all the available nodes).
- In addition to the performance benefit, parallel approach is needed to make large/complex simulations feasible – often times the memory and storage resource on a single node are not sufficient to handle large problems.
- Lot of real world problems are inherently parallel and conducive to using massively parallel resources.

---

# *Classification*

- **Single Instruction, Single Data [Serial codes]**
- **Single Instruction, Multiple Data**
  - Processors run the same instructions, each operates on different data
  - Technically, Hadoop MapReduce fits this mode
  - GPUs
- **Multiple Instruction, Single Data**
  - Multiple instructions acting on single data stream. E.g. Different analysis on same set of data.
- **Multiple Instruction, Multiple Data**
  - Every processor may execute different instructions
  - Every processor may work on different parts of data
  - Execution can be synchronous or asynchronous, deterministic or non-deterministic

---

## *Message Passing Interface (MPI)*

- MPI is a standard; Several implementations available. On Gordon & Trestles we use mvapich2 and openmpi.
- Originally designed for distributed memory architectures.
- Present implementations work on hybrid distributed memory/shared memory systems.
- Implementations adapted/developed for handling most network interconnects and protocols – for example InfiniBand, and Ethernet (TCP).
- MPI standard allows for easy portability, standard functionality, and implementations can optimize functions to exploit native hardware. User does not need to be aware of the hardware level optimizations.

# Typical MPI Code Structure

MPI Include File

Variable declarations, etc  
Begin Program

...

Serial code

....

MPI Initialization

Parallel Code begins

MPI Rank (process identification)

...

Parallel code based on rank

...

MPI Communications between processes

...

Parallel code based on rank

...

MPI Communications between processes

MPI Finalize (terminate)

Parallel Code ends

Serial Code

# *Simple MPI Program – Compute PI*

- Initialize MPI (MPI\_Init function)
- Find the number of tasks and taskids (MPI\_Comm\_size, MPI\_Comm\_rank)
- PI is calculated using an integral. The number of intervals used for the integration is fixed at 128000.
- Computes the sums for a different sections of the intervals in each MPI task.
- At the end of the code, the sums from all the tasks are added together to evaluate the final integral. This is accomplished through a reduction operation (MPI\_Reduce function).
- Simple code illustrates decomposition of problem into parallel components.



# MPI Program to Compute Pi

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

int main(int argc, char *argv[])
{
    int numprocs, rank;
    int i, iglob, INTERVALS, INTLOC;
    double n_1, x;
    double pi, piloc;

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

    INTERVALS=128000;
    printf("Hello from MPI task= %d\n", rank);
    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0)
    {
        printf("Number of MPI tasks = %d\n", numprocs);
    }

    INTLOC=INTERVALS/numprocs;
    piloc=0.0;
    n_1=1.0/(double)INTERVALS;
    for (i = 0; i < INTLOC; i++)
    {
        iglob = INTLOC*rank+i;
        x = n_1 * ((double)iglob - 0.5);
        piloc += 4.0 / (1.0 + x * x);
    }
    MPI_Reduce(&piloc,&pi,1,MPI_DOUBLE,MPI_SUM,
0,MPI_COMM_WORLD);
    if (rank == 0)
    {
        pi *= n_1;
        printf ("Pi = %.12lf\n", pi);
    }

    MPI_Finalize();
}
```

# *PI Code : MPI Environment Functions*

**MPI\_Init(&argc, &argv);**

Initializes MPI, \*must\* be called (only once) in every MPI program before any MPI functions.

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);**

Returns the total number of tasks in the communicator. MPI uses communicators to define which collections of processes can communicate with each other. The default MPI\_COMM\_WORLD includes all the processes. User defined communicators are an option.

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

Returns the rank (ID) of the calling MPI process within the communicator.

**MPI\_Finalize();**

Ends the MPI execution environment. No MPI calls after this.!

The other routines in the code are collectives and we will discuss them later in the talk.

# Compiling and Running PI Example

Location: \$HOME/SI2014/PARALLEL/SIMPLE

Compile: **mpicc -o pi\_mpi.exe pi\_mpi.c**

Submit Job: **qsub pi\_mpi.cmd**

## Output:

Hello from MPI task= 12

Hello from MPI task= 14

Hello from MPI task= 8

Hello from MPI task= 3

Hello from MPI task= 2

Hello from MPI task= 4

Hello from MPI task= 13

Hello from MPI task= 9

Hello from MPI task= 5

Hello from MPI task= 1

Hello from MPI task= 11

Hello from MPI task= 10

Hello from MPI task= 15

Hello from MPI task= 7

Hello from MPI task= 6

Hello from MPI task= 0

Number of MPI tasks = 16

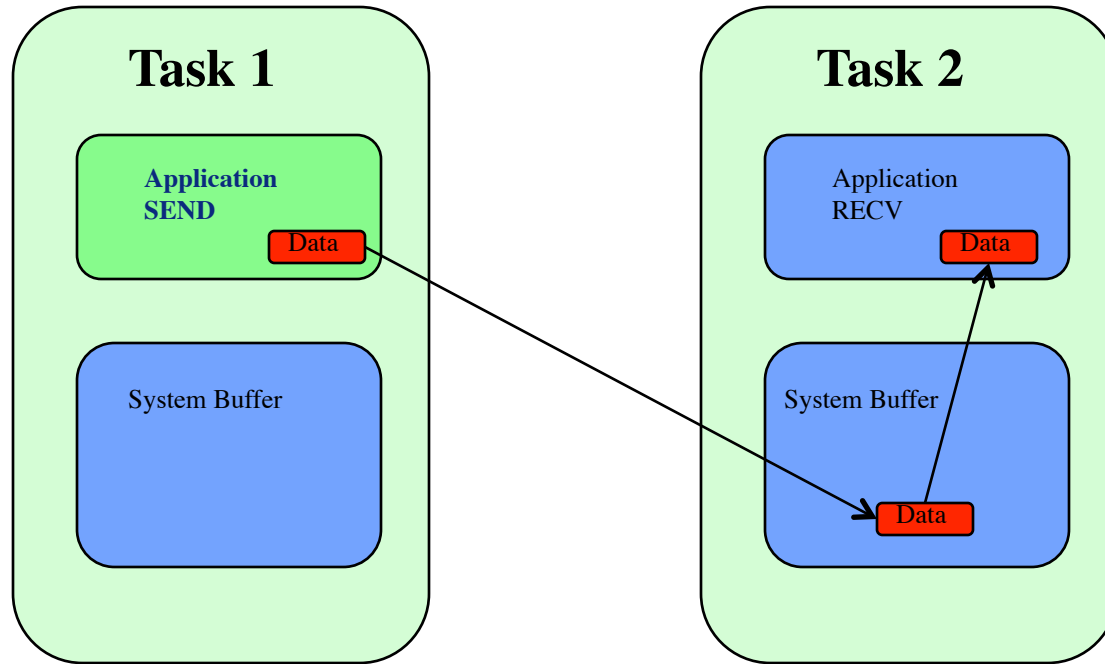
Pi = 3.141594606714

---

## *Point to Point Communication*

- Passing data between two, and only two different MPI tasks.
- Typically one task performs a send operation and the other task performs a matching receive.
- MPI Send operations have choices with different synchronization (when does a send complete) and different buffering (where the data resides till it is received) modes.
- Any type of send routine can be paired with any type of receive routine.
- MPI also provides routines to probe status of messages, and “wait” routines.

# Buffers



- Buffer space is used for data in transit – whether its waiting for a receive to be ready or if there are multiple sends arriving at the same receiving tasks.
- Typically a system buffer area managed by the MPI library (opaque to the user) is used. Can exist on both sending & receiving side.
- MPI also provides for user managed send buffer.

---

# *Blocking MPI Send, Receive Routines*

- Blocking send call will return once it is safe for the application buffer (send data) to be reused.
- This can happen as soon as the data is copied into the system (MPI) buffer on receiving process.
- Synchronous if there is confirmation of safe send, and asynchronous otherwise.
- Blocking receive returns once the data is in the application buffer (receive data) and can be used by the application.

## *Blocking Send, Recv Example (Code Snippet)*

```
if(myid == 0) {
    for(i = 0; i < 10; i++) {
        s_buf[i] = i*4.0;
    }
    MPI_Send(s_buf, size, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
}
else if(myid == 1) {
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag, MPI_COMM_WORLD,
&reqstat);
    for (i = 0; i < 10; i++) {
        printf("r_buf[%d] = %f\n", i, r_buf[i] );
    }
}
```

---

# *Blocking Send, Recv Example*

Location: \$HOME/SI2014/PARALLEL/PTOP

Compile: **mpicc -o blocking.exe blocking.c**

Submit Job: **qsub blocking.cmd**

Output: **more blocking.out**

**r\_buf[0] = 0.000000**

**r\_buf[1] = 4.000000**

**r\_buf[2] = 8.000000**

**r\_buf[3] = 12.000000**

**r\_buf[4] = 16.000000**

**r\_buf[5] = 20.000000**

**r\_buf[6] = 24.000000**

**r\_buf[7] = 28.000000**

**r\_buf[8] = 32.000000**

**r\_buf[9] = 36.000000**



# Deadlocking MPI Tasks

- Take care to sequence blocking send/recvs. Easy to deadlock processes waiting on each other.
- For example, take the following code snippet:

```
if(myid == 0) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD, &reqstat);
}
else if(myid == 1) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &reqstat);
    for (i = 0; i < 10; i++){
        printf("r_buf[%d] = %f\n", i, r_buf[i] );
    }
}
```

- The MPI\_Ssend on both tasks will not complete till the MPI\_Recv is posted (which will never happen given the order).

# *Deadlock Example*

- Location: `$HOME/SI2014/PARALLEL/PTOP`
- Compile: `mpicc -o deadlock.exe deadlock.c`
- Submit Job: `qsub deadlock.cmd`
- It should technically finish in less than a second since the data transferred is a few bytes. However, the code deadlocks and hits the wallclock limit (2 minutes in the script). Error info:

```
[gcn-9-73.sdsc.edu:mpirun_rsh][signal_processor] Caught signal 15, killing job  
[gcn-9-73.sdsc.edu:mpirun_rsh][signal_processor] Caught signal 15, killing job  
=>> PBS: job killed: walltime 131 exceeded limit 120
```

# Deadlock Example – Simple Fix

- Change the order on one of processes!
- For example, take the following code snippet:

```
if(myid == 0) {
    MPI_Ssend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD);
    MPI_Recv(r_buf, size, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD, &reqstat);
}
else if(myid == 1) {
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &reqstat);
    MPI_Ssend(s_buf, size, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD);
    for (i = 0; i < 10; i++) {
        printf("r_buf[%d] = %f\n", i, r_buf[i] );
    }
}
```

- Now the MPI\_Ssend on task 0 will complete since the corresponding MPI\_Recv is posted first on task 1. (qsub deadlock-fix1.cmd)
- We will look at **Non-Blocking** options next.

# *Deadlock Example (Fix 1)*

- **Location:** \$HOME/SI2014/PARALLEL/PTOP
- **Compile:** **mpicc -o deadlock-fix1.exe deadlock-fix1.c**
- **Submit Job:** **qsub deadlock-fix1.cmd**
- **Fix works!**

```
$ more deadlock-fix1.out
```

```
r_buf[0] = 0.000000
```

```
r_buf[1] = 4.000000
```

```
r_buf[2] = 8.000000
```

```
r_buf[3] = 12.000000
```

```
r_buf[4] = 16.000000
```

```
r_buf[5] = 20.000000
```

```
r_buf[6] = 24.000000
```

```
r_buf[7] = 28.000000
```

```
r_buf[8] = 32.000000
```

```
r_buf[9] = 36.000000
```

---

# *Non-Blocking MPI Send, Receive Routines*

- Non-Blocking MPI Send, Receive routines return before there is any confirmation of receives or completion of the actual message copying operation.
- The routines simply put in the request to perform the operation.
- MPI wait routines can be used to check status and block till the operation is complete and it is safe to modify/use the information in the application buffer.
- This non-blocking approaches allows computations (that don't depend on this data in transit) to continue while the communication operations are in progress. This allows for hiding the communication time with useful work and hence improves parallel efficiency.

# Non-Blocking Send, Recv Example

- **Example uses MPI\_Isend, MPI\_Irecv, MPI\_Wait**

- **Code snippet:**

```
if(myid == source){
    s_buf=1024;
    MPI_Isend(&s_buf,count,MPI_INT,destination,tag,MPI_COMM_WORLD,&request);
}
if(myid == destination {
    MPI_Irecv(&r_buf,count,MPI_INT,source,tag,MPI_COMM_WORLD,&request);
}
MPI_Wait(&request,&status);
```

- **Compile & Run:**

```
mpicc -o nonblocking.exe nonblocking.c
```

```
qsub nonblocking.cmd
```

```
more nonblocking.out
```

```
processor 0 sent 1024
```

```
processor 1 got 1024
```

# Deadlock Example – Non-Blocking Option

- Change the order on one of processes!
- For example, take the following code snippet:

```
if(myid == 0) {  
    MPI_Isend(s_buf, size, MPI_FLOAT, 1, tag1, MPI_COMM_WORLD, &request);  
    MPI_Recv(r_buf, size, MPI_FLOAT, 1, tag2, MPI_COMM_WORLD, &reqstat);  
}  
else if(myid == 1) {  
    MPI_Ssend(s_buf, size, MPI_FLOAT, 0, tag2, MPI_COMM_WORLD);  
    MPI_Recv(r_buf, size, MPI_FLOAT, 0, tag1, MPI_COMM_WORLD, &reqstat);  
    for (i = 0; i < 10; i++){  
        printf("r_buf[%d] = %f\n", i, r_buf[i] );  
    }  
}
```

- Now the MPI\_Ssend on task 0 will complete since the corresponding MPI\_Recv is posted first on task 1. (qsub deadlock-fix1.cmd)
- We will look at **Non-Blocking** options next.

## *Deadlock Example (Fix 2)*

- **Location:** \$HOME/SI2014/PARALLEL/PTOP
- **Compile:** `mpicc -o deadlock-fix2-nb.exe deadlock-fix2-nb.c`
- **Submit Job:** `qsub deadlock-fix2-nb.cmd`
- **Fix works!**

```
$ more deadlock-fix2-nb.out
```

```
r_buf[0] = 0.000000
```

```
r_buf[1] = 4.000000
```

```
r_buf[2] = 8.000000
```

```
r_buf[3] = 12.000000
```

```
r_buf[4] = 16.000000
```

```
r_buf[5] = 20.000000
```

```
r_buf[6] = 24.000000
```

```
r_buf[7] = 28.000000
```

```
r_buf[8] = 32.000000
```

```
r_buf[9] = 36.000000
```



---

## *Collective MPI Routines*

- **Synchronization Routines:** All processes in group/communicator wait till they get synchronized.
- **Data Movement: Send/Receive data from all processes. E.g. Broadcast, Scatter, Gather, AlltoAll.**
- **Collective Computation (reductions):** Perform reduction operations (min, max, add, multiply, etc.) on data obtained from all processes.
- **Collective Computation and Data Movement combined (Hybrid).**

---

# *Synchronization Example*

- **Our simple PI program had a synchronization example.**

- **Code Snippet:**

```
printf("Hello from MPI task= %d\n", rank);  
MPI_Barrier(MPI_COMM_WORLD);  
if (rank == 0)  
{  
    printf("Number of MPI tasks = %d\n", numprocs);  
}
```

- **All tasks will wait till they are synchronized at this point.**

# Broadcast Example

- **Code Snippet** (All collectives examples in \$HOME/SI2014/PARALLEL/COLLECTIVES):

- if(myid .eq. source)then
- do i=1,count
- buffer(i)=i
- enddo
- endif
- **Call MPI\_Bcast(buffer, count, MPI\_INTEGER,source,&**
- **MPI\_COMM\_WORLD,ierr)**

- **Compile:**

- **mpif90 -o bcast.exe bcast.f90**

- **Run:**

- **qsub bcast.cmd**

- **Output:**

```
processor    1 got      1      2      3      4
processor    0 got      1      2      3      4
processor    2 got      1      2      3      4
processor    3 got      1      2      3      4
```

# Reduction Example

- **Code Snippet:**

```
myidp1 = myid+1
```

```
call MPI_Reduce(myidp1,ifactorial,1,MPI_INTEGER,MPI_PROD,root,MPI_COMM_WORLD,ierr)
```

```
if (myid.eq.root) then
```

```
    write(*,*)numprocs,"! = ",ifactorial
```

```
endif
```

- **Compile:**

```
mpif90 -o factorial.exe factorial.f90
```

- **Run:**

```
qsub factorial.cmd
```

- **Output:**

```
8 ! = 40320
```

# *MPI\_Allreduce example*

- **Code Snippet:**

```
imaxloc=IRAND(myid)
```

```
call MPI_ALLREDUCE(imaxloc,imax,1,MPI_INTEGER,MPI_MAX,MPI_COMM_WORLD,  
mpi_err)
```

```
if (imax.eq.imaxloc) then
```

```
    write(*,*)"Max=",imax,"on task",myid
```

```
endif
```

- **Compile:**

```
mpif90 -o allreduce.exe allreduce.f90
```

- **Run:**

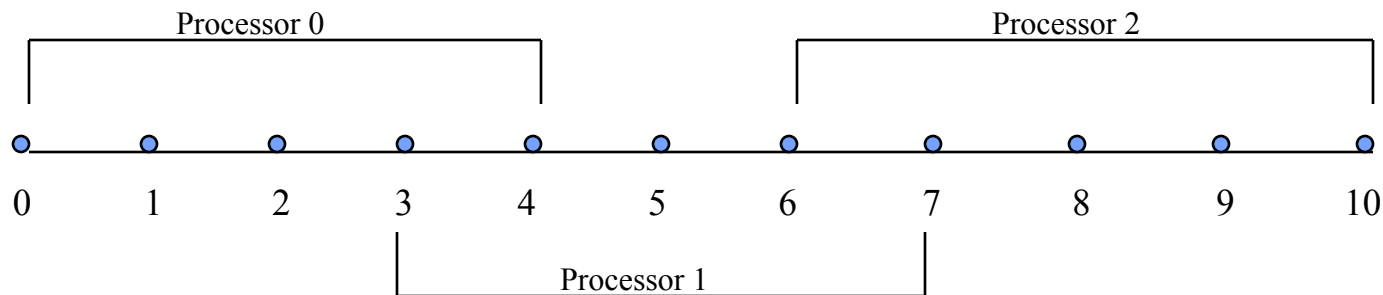
```
qsub allreduce.cmd
```

- **Output:**

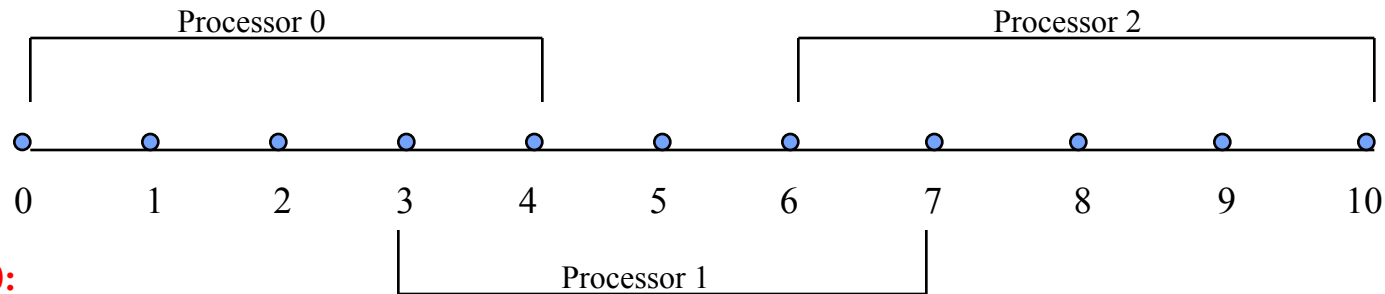
```
Max= 337897 on task 7
```

## Simple Application using MPI: 1-D Heat Equation

- $\partial T/\partial t = \alpha(\partial^2 T/\partial x^2)$ ;  $T(0) = 0$ ;  $T(1) = 0$ ; ( $0 \leq x \leq 1$ )  
 $T(x,0)$  is known as an initial condition.
- Discretizing for numerical solution we get:  
$$T^{(n+1)}_i - T^{(n)}_i = (\alpha\Delta t/\Delta x^2)(T^{(n)}_{i-1} - 2T^{(n)}_i + T^{(n)}_{i+1})$$
  
( $n$  is the index in time and  $i$  is the index in space)
- In this example we solve the problem using 11 points and we distribute this problem over exactly 3 processors (for easy demo) shown graphically below:



# Simple Application using MPI: 1-D Heat Equation



## Processor 0:

Local Data Index :  $i_{\text{local}} = 0, 1, 2, 3, 4$

Global Data Index:  $i_{\text{global}} = 0, 1, 2, 3, 4$

Solve the equation at (1,2,3)

**Data Exchange:** Get 4 from processor 1; Send 3 to processor 1

## Processor 1:

Local Data Index :  $i_{\text{local}} = 0, 1, 2, 3, 4$

Global Data Index :  $i_{\text{global}} = 3, 4, 5, 6, 7$

Solve the equation at (4,5,6)

**Data Exchange:** Get 3 from processor 0; Get 7 from processor 2; Send 4 to processor 0; Send 6 to processor 2

## Processor 2:

Local Data Index :  $i_{\text{local}} = 0, 1, 2, 3, 4$

Global Data Index :  $i_{\text{global}} = 6, 7, 8, 9, 10$

Solve the equation at (7,8,9)

**Data Exchange:** Get 6 from processor 1; Send 7 to processor 1

# ***FORTRAN MPI CODE: 1-D Heat Equation***

```
PROGRAM HEATEQN
```

```
implicit none
```

```
include "mpif.h"
```

```
integer :: iglobal, ilocal, itime
```

```
integer :: ierr, nnodes, my_id
```

```
integer :: dest, from, status(MPI_STATUS_SIZE), tag
```

```
integer :: msg_size
```

```
real*8 :: xalp, delx, delt, pi
```

```
real*8 :: T(0:100,0:5), TG(0:10)
```

```
CHARACTER(20) :: FILEN
```

```
delx = 0.1d0
```

```
delt = 1d-4
```

```
xalp = 2.0d0
```

```
call MPI_INIT(ierr)
```

```
call MPI_COMM_SIZE(MPI_COMM_WORLD,  
nnodes, ierr)
```

```
call MPI_COMM_RANK(MPI_COMM_WORLD,  
my_id, ierr)
```

```
if (nnodes.ne.3) then
```

```
if (my_id.eq.0) then
```

```
print *, "This test needs exactly 3 tasks"
```

```
endif
```

```
print *, "Process ", my_id, "of", nnodes, "has started"
```

```
!***** Initial Conditions
```

```
*****
```

```
pi = 4d0*datan(1d0)
```

```
do ilocal = 0, 4
```

```
iglobal = 3*my_id+ilocal
```

```
T(0,ilocal) = dsin(pi*delx*dfloat(iglobal))
```

```
enddo
```

```
write(*,*) "Processor", my_id, "has finished setting  
+ initial conditions"
```

```
!***** Iterations
```

```
*****
```

```
do itime = 1, 3
```

```
if (my_id.eq.0) then
```

```
write(*,*) "Running Iteration Number ", itime
```

```
endif
```

```
do ilocal = 1, 3
```

```
T(itime,ilocal)=T(itime-1,ilocal)+
```

```
+ xalp*delt/delx/delx*
```

```
+ (T(itime-1,ilocal-1)-2*T(itime-1,ilocal)
```

```
+T(itime-1,ilocal+1))
```

```
enddo
```

```
if (my_id.eq.0) then
```

```
write(*,*) "Sending and receiving overlap points"
```

```
dest = 1
```



# Fortran MPI Code: 1-D Heat Equation (Contd.)

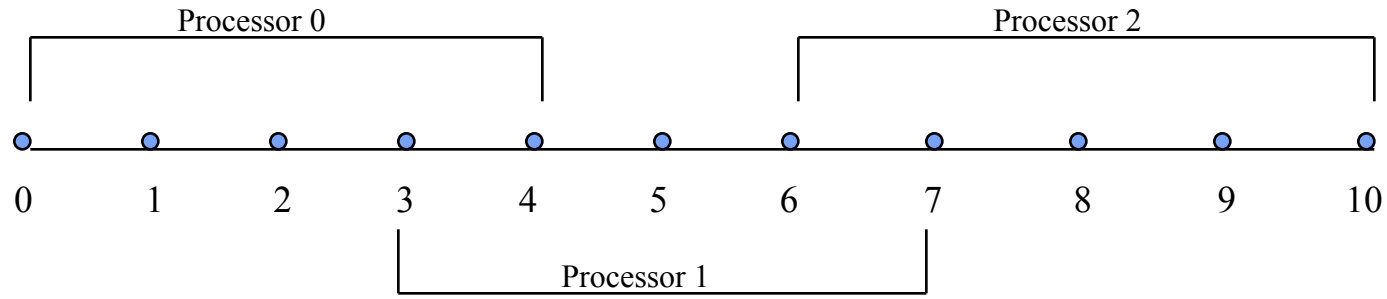
```
msg_size = 1
  call MPI_SEND(T(itime,
3),msg_size,MPI_DOUBLE_PRECISION,dest,
+   tag,MPI_COMM_WORLD,ierr)
endif
if (my_id.eq.1) then
  from = 0
  dest = 2
  msg_size = 1
  call MPI_SEND(T(itime,
3),msg_size,MPI_DOUBLE_PRECISION,dest,
+   tag,MPI_COMM_WORLD,ierr)
  call MPI_RECV(T(itime,
0),msg_size,MPI_DOUBLE_PRECISION,from,
+   tag,MPI_COMM_WORLD,status,ierr)
endif
if (my_id.eq.2) then
  from = 1
  dest = 1
  msg_size = 1
  call MPI_SEND(T(itime,
1),msg_size,MPI_DOUBLE_PRECISION,dest,
+   tag,MPI_COMM_WORLD,ierr)
  call MPI_RECV(T(itime,
0),msg_size,MPI_DOUBLE_PRECISION,from,
+   tag,MPI_COMM_WORLD,status,ierr)
endif
if (my_id.eq.1) then
```

```
  from = 2
  dest = 0
  msg_size = 1
  call MPI_RECV(T(itime,4),msg_size,MPI_DOUBLE_PRECISION,from,
+   tag,MPI_COMM_WORLD,status,ierr)
  call MPI_SEND(T(itime,1),msg_size,MPI_DOUBLE_PRECISION,dest,
+   tag,MPI_COMM_WORLD,ierr)
endif
if (my_id.eq.0) then
  from = 1
  msg_size = 1
  call MPI_RECV(T(itime,4),msg_size,MPI_DOUBLE_PRECISION,from,
+   tag,MPI_COMM_WORLD,status,ierr)
endif
enddo

if (my_id.eq.0) then
  write(*,*)"SOLUTION SENT TO FILE AFTER 3 TIMESTEPS:"
endif
FILEN = 'data//char(my_id+48)//'.dat'
open (5, file=FILEN)
write(5,*)"Processor ",my_id
do ilocal = 0 , 4
  iglobal = 3*my_id + ilocal
  write(5,*)"ilocal=",ilocal,";iglobal=",iglobal,";T=",T(3,ilocal)
enddo
close(5)
call MPI_FINALIZE(ierr)

END
```

## Simple Application using MPI: 1-D Heat Equation



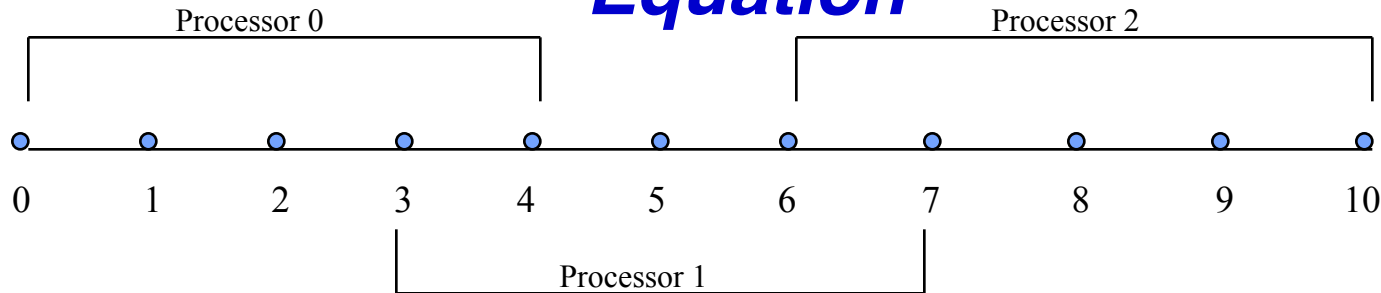
- Compilation

Fortran: `mpif90 -nofree -o heat_mpi.exe heat_mpi.f90`

- Run Job:

`qsub heat_mpi.cmd`

# Simple Application using MPI: 1-D Heat Equation



## OUTPUT FROM SAMPLE PROGRAM

Process 0 of 3 has started

Processor 0 has finished                      setting initial conditions

Process 1 of 3 has started

Processor 1 has finished                      setting initial conditions

Process 2 of 3 has started

Processor 2 has finished                      setting initial conditions

Running Iteration Number 1

Sending and receiving overlap points

Running Iteration Number 2

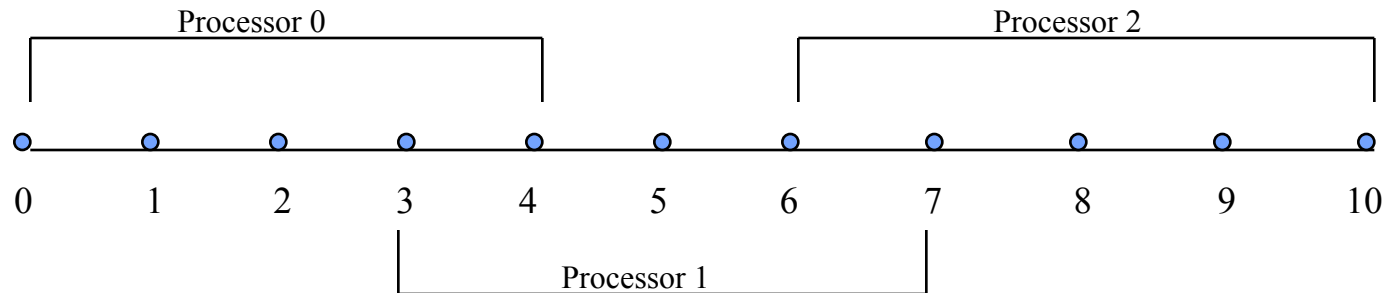
Sending and receiving overlap points

Running Iteration Number 3

Sending and receiving overlap points

SOLUTION SENT TO FILE AFTER 3 TIMESTEPS:

# Simple Application using MPI: 1-D Heat Equation



```
% more data0.dat
Processor 0
ilocal= 0 ;iglobal= 0 ;T= 0.000000000000000000E+00
ilocal= 1 ;iglobal= 1 ;T= 0.307205621017284991
ilocal= 2 ;iglobal= 2 ;T= 0.584339815421976549
ilocal= 3 ;iglobal= 3 ;T= 0.804274757358271253
ilocal= 4 ;iglobal= 4 ;T= 0.945481682332597884
```

```
% more data2.dat
Processor 2
ilocal= 0 ;iglobal= 6 ;T= 0.945481682332597995
ilocal= 1 ;iglobal= 7 ;T= 0.804274757358271253
ilocal= 2 ;iglobal= 8 ;T= 0.584339815421976660
ilocal= 3 ;iglobal= 9 ;T= 0.307205621017285102
ilocal= 4 ;iglobal= 10 ;T= 0.000000000000000000E+00
```

```
% more data1.dat
Processor 1
ilocal= 0 ;iglobal= 3 ;T= 0.804274757358271253
ilocal= 1 ;iglobal= 4 ;T= 0.945481682332597884
ilocal= 2 ;iglobal= 5 ;T= 0.994138272681972301
ilocal= 3 ;iglobal= 6 ;T= 0.945481682332597995
ilocal= 4 ;iglobal= 7 ;T= 0.804274757358271253
```

---

# ***MPI – Profiling, Tracing Tools***

- Several options available. On Gordon we have mpiP, TAU, and IPM installed.
- Useful when you are trying to isolate performance issues.
- Tools can give you info on how much time is being spent in communication. The levels of detail vary with each tool.
- In general identify scaling bottlenecks and try to overlap communication with computation where possible.

---

# *mpiP example*

- **Location: \$HOME/SI2014/PARALLEL/MISC**
- **Compile:**  
`mpif90 -nofree -g -o heat_mpi_profile.exe heat_mpi.f90 -L/  
home/diag/opt/mpiP/v3.4.1/lib -lmpiP -L/opt/gnu/lib -lbfd -lz  
-liberty`
- **Executable already exists. Just submit  
heat\_mpi\_profile.cmd.**
- **Once the job runs you get a .mpiP file.**

# *mpiP output*

**@ mpiP**  
**@ Command** : ./heat\_mpi\_profile.exe  
**@ Version** : 3.4.1  
**@ MPIP Build date** : Aug 3 2014, 19:18:28  
**@ Start time** : 2014 08 06 08:50:44  
**@ Stop time** : 2014 08 06 08:50:44  
**@ Timer Used** : PMPI\_Wtime  
**@ MPIP env var** : [null]  
**@ Collector Rank** : 0  
**@ Collector PID** : 53941  
**@ Final Output Dir** : .  
**@ Report generation** : Single collector task  
**@ MPI Task Assignment** : 0 gcn-13-35.sdsc.edu  
**@ MPI Task Assignment** : 1 gcn-13-35.sdsc.edu  
**@ MPI Task Assignment** : 2 gcn-13-35.sdsc.edu

# *mpiP Output*

-----  
@--- MPI Time (seconds) -----  
-----

Task	AppTime	MPITime	MPI%
0	0.0702	0.00513	7.30
1	0.0728	0.00516	7.09
2	0.0732	0.00519	7.08
*	0.216	0.0155	7.16

-----

@--- Callsites: 1 -----  
-----

ID	Lev	File/Address	Line	Parent_Funct	MPI_Call
1	0	0x40dbf4	main		Send

-----

@--- Aggregate Time (top twenty, descending, milliseconds) -----  
-----

Call	Site	Time	App%	MPI%	COV
Send	1	15.2	7.04	98.34	0.00
Recv	1	0.257	0.12	1.66	0.23



# *mpiP output*

-----  
@--- Aggregate Sent Message Size (top twenty, descending, bytes) -----  
-----

Call	Site	Count	Total	Avrg Sent%
Send	1	12	96	8 100.00

-----

-----  
@--- Callsite Time statistics (all, milliseconds): 6 -----  
-----

Name	Site	Rank	Count	Max	Mean	Min	App%	MPI%
Recv	1	0	3	0.052	0.0233	0.008	0.10	1.37
Recv	1	1	6	0.026	0.0132	0.004	0.11	1.53
Recv	1	2	3	0.057	0.036	0.02	0.15	2.08
Send	1	0	3	5.05	1.69	0.004	7.20	98.63
Send	1	1	6	5.06	0.847	0.003	6.98	98.47
Send	1	2	3	5.07	1.69	0.004	6.93	97.92
Send	1	*	24	5.07	0.645	0.003	7.16	100.00

-----

-----  
@--- Callsite Message Sent statistics (all, sent bytes) -----  
-----

Name	Site	Rank	Count	Max	Mean	Min	Sum
Send	1	0	3	8	8	8	24
Send	1	1	6	8	8	8	48
Send	1	2	3	8	8	8	24
Send	1	*	12	8	8	8	96

-----

-----  
@--- End of Report -----  
-----

# Data Types

C Data Types		FORTRAN Data Types
MPI_CHAR	MPI_C_DOUBLE_COMPLEX	MPI_CHARACTER
MPI_WCHAR	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER
MPI_SHORT	MPI_C_BOOL	MPI_INTEGER1
MPI_INT	MPI_LOGICAL	MPI_INTEGER2
MPI_LONG	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER4
MPI_LONG_LONG_INT	MPI_INT8_T	MPI_REAL
MPI_LONG_LONG	MPI_INT16_T	MPI_REAL2
MPI_SIGNED_CHAR	MPI_INT32_T	MPI_REAL4
MPI_UNSIGNED_CHAR	MPI_INT64_T	MPI_REAL8
MPI_UNSIGNED_SHORT	MPI_UINT8_T	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_LONG	MPI_UINT16_T	MPI_COMPLEX
MPI_UNSIGNED	MPI_UINT32_T	MPI_DOUBLE_COMPLEX
MPI_FLOAT	MPI_UINT64_T	MPI_LOGICAL
MPI_DOUBLE	MPI_BYTE	MPI_BYTE
MPI_LONG_DOUBLE	MPI_PACKED	MPI_PACKED
MPI_C_COMPLEX		
MPI_C_FLOAT_COMPLEX		

# *MPI Reduction Operations*

NAME	OPERATION
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bit-wise AND
MPI_LOR	Logical OR
MPI_BOR	Bit-wise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bit-wise XOR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

---

# *More Complex routines*

- **Derived Data Types**
- **User defined reduction functions**
- **Groups/communicator management**
- **Parallel I/O**
- **One Sided Communication Routines (RDMA)**
- **MPI-3 Standard has over 400 routines(!).**

---

# *Homework!*

- **Change directory to \$HOME/SI2014/PARALLEL/MISC**
- **Code “sample.f” has two bugs.**

- **Compile:**

```
mpif90 -o sample.exe sample.f
```

```
Run: qsub sample.cmd
```

See if you can identify the two bugs!

---

# *What is OpenMP?*

- An API to direct multi-threaded, shared memory parallelism. Provides a Standard for a variety of shared memory architectures.
- **Components – Compiler Directives, Runtime Library Routines, Environment Variables**
- Can achieve parallelism with very few simple directives.
- **C/C++ and Fortran support.**
- Easy to use, can be incrementally implemented, at both coarse and fine grain levels.

---

# *Fork-Join Model*

- OpenMP uses the fork-join model of execution.
- Programs start with a single process (master thread).
- Master thread **forks** parallel threads.
- Once the parallel work is done the threads synchronize and terminate leaving the master thread (**JOIN**).
- This can be repeated arbitrary number of times.
- **No explicit parallel I/O options.** Can be programmed.

---

# *Compiler Directives*

- **Compiler directives is the main mechanism for introducing parallelism. Functionality enabled includes:**
  - Spawning a parallel region
  - Diving code among threads
  - Distributing loop iterations over threads
  - Serialization of parts of the code
  - Synchronization of work
- **Example:**  
`#pragma omp parallel default(shared) private(beta,pi)`



---

# *Parallel Region Construct*

```
!$OMP PARALLEL [clause ...]  
    IF (scalar_logical_expression)  
    PRIVATE (list)  
    SHARED (list)  
    DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)  
    FIRSTPRIVATE (list)  
    REDUCTION (operator: list)  
    COPYIN (list)  
    NUM_THREADS (scalar-integer-expression)
```

code block

```
!$OMP END PARALLEL
```

---

# *Number of Threads*

- **Number of threads will be determined in the following order of precedence:**
  - Evaluation of the IF clause
  - Setting of NUM\_THREADS clause
  - omp\_set\_num\_threads() library function
  - OMP\_NUM\_THREADS environment variable
  - Default – usually ends up being the **\*number of cores on the node\*** (!)
- **The last factor can accidentally lead to oversubscription of nodes in hybrid MPI/OpenMP codes.**

---

# *Work – Sharing Constructs*

Directive format (C version):

```
#pragma omp for [clause ...] newline
    schedule (type [,chunk])
    ordered
    private (list)
    firstprivate (list)
    lastprivate (list)
    shared (list)
    reduction (operator: list)
    collapse (n)
    nowait
```

for\_loop

---

# *Work-Sharing*

- **Schedule:**

- Static – Loop iterations are statically divided (chunk or as close to even as possible)
- Dynamic – Loop iterations are divided in size chunk, and dynamically scheduled among threads. When a thread finishes one chunk it is dynamically assigned another
- Guided – Similar to dynamic but chunk size is proportionally reduced based on work remaining.
- Runtime – set at runtime by environment variables
- Auto – set by compiler or runtime system.

---

# ***Simple OpenMP Program – Compute PI***

- Find the number of tasks and taskids (`omp_get_num_threads`, `omp_get_thread_num`)
- PI is calculated using an integral. The number of intervals used for the integration is fixed at 128000.
- Use OpenMP loop parallelization to divide up the compute work.
- Introduce concept of private and shared variables.
- OpenMP reduction operation used to compute the sum for the final integral.

# OpenMP Program to Compute PI

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
    int nthreads, tid;
    int i, INTERVALS;
    double n_1, x, pi = 0.0;

    INTERVALS=128000;
    /* Fork a team of threads giving them their own
       copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtain thread number */
        tid = omp_get_thread_num();
        printf("Hello from thread = %d\n", tid);
```

```
        /* Only master thread does this */
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }

        /* All threads join master thread and disband */

        n_1 = 1.0 / (double)INTERVALS;

        /* Parallel loop with reduction for calculating PI */
        #pragma omp parallel for private(i,x)
        shared(n_1,INTERVALS) reduction(+:pi)
        for (i = 0; i < INTERVALS; i++)
        {
            x = n_1 * ((double)i - 0.5);
            pi += 4.0 / (1.0 + x * x);
        }
        pi *= n_1;
        printf ("Pi = %.12lf\n", pi);
    }
```

---

## *More Work-Share Constructs*

- **SECTIONS** directive – enclosed sections are divided among the threads.
- **WORKSHARE** directive – divides execution of block into units of work, each of which is executed once.
- **SINGLE** directive – Enclosed code is executed by only one thread.

---

# *Synchronization Constructs*

- **MASTER** directive – Specifies region is executed only by the master thread.
- **CRITICAL** directive – Region of the code that is executed one thread at a time.
- **BARRIER** directive – synchronize all threads
- **TASKWAIT** directive – wait for all child tasks to complete
- **ATOMIC** directive – specific memory location updated atomically (not let all threads write at the same time)



# Fortran OpenMP Code: 1-D Heat Equation

```
PROGRAM HEATEQN
  implicit none
  integer :: iglobal, itime, nthreads
  real*8 :: xalp,delx,delt,pi
  real*8 :: T(0:100,0:10)
  integer:: id
  integer:: OMP_GET_THREAD_NUM,
            OMP_GET_NUM_THREADS

!$OMP PARALLEL SHARED(nthreads)
!$OMP MASTER
  nthreads = omp_get_num_threads()
  write (*,*) 'There are', nthreads, 'threads'
!$OMP END MASTER
!$OMP END PARALLEL
  if (nthreads.ne.3) then
    write(*,*)"Use exactly 3 threads for this case"
    stop
  endif
  delx = 0.1d0
  delt = 1d-4
  xalp = 2.0d0

***** Initial Conditions
*****
  pi = 4d0*datan(1d0)
  do iglobal = 0, 10
    T(0,iglobal) = dsin(pi*delx*dfloat(iglobal))
  enddo
***** Iterations
*****
  do itime = 1, 3
    write(*,*)"Running Iteration Number ", itime
!$OMP PARALLEL DO PRIVATE(iglobal)
    SHARED(T,xalp,delx,delt,itime)
    do iglobal = 1, 9
      T(itime,iglobal)=T(itime-1,iglobal)+
      + xalp*delt/delx/delx*
      + (T(itime-1,iglobal-1)-2*T(itime-1,iglobal)+T(itime-1,iglobal
+1))
    enddo
!$OMP BARRIER
  enddo
  do iglobal = 0, 10
    write(*,*)iglobal,T(3, iglobal)
  enddo
END
```

---

# *OpenMP result: 1-D Heat Equation*

```
$ export OMP_NUM_THREADS=3
```

```
$ ./a.out
```

```
There are      3 threads
```

```
Running Iteration Number      1
```

```
Running Iteration Number      2
```

```
Running Iteration Number      3
```

```
0 0.0000000000000000E+000
```

```
1 0.307205621017285
```

```
2 0.584339815421976
```

```
3 0.804274757358271
```

```
4 0.945481682332598
```

```
5 0.994138272681972
```

```
6 0.945481682332598
```

```
7 0.804274757358271
```

```
8 0.584339815421977
```

```
9 0.307205621017285
```

```
10 0.0000000000000000E+000
```

---

# *Data Scope Attribute Clauses*

- **PRIVATE** – variables in the list are private to each thread.
- **SHARED** – variables in the list are shared between all threads.
- **DEFAULT** – default scope for all variables in a parallel region.
- **FIRSTPRIVATE** – variables are private and initialized according to value prior to entry into parallel or work sharing construct.
- **LASTPRIVATE** – variables are private, the value from the last iteration or section is copied to original variable object.
- **Others** – **COPYIN**, **COPYPRIVATE**
- **REDUCTION** – reduction on variables in the list

---

# *Run Time Library Routines*

- **Setting and querying number of threads**
- **Querying thread identifier, team size**
- **Setting and querying dynamic threads feature**
- **Querying if in parallel region and at what level**
- **Setting and querying nested parallelism**
- **Setting, initializing and terminating locks, nested locks.**
- **Querying wall clock time and resolution.**

---

# *Environment Variables*

- **OMP\_SCHEDULE** – e.g set to “dynamic”
- **OMP\_NUM\_THREADS**
- **OMP\_DYNAMIC** (TRUE or FALSE)
- **OMP\_PROC\_BIND** (TRUE or FALSE)
- **OMP\_NESTED** (TRUE or FALSE)
- **OMP\_STACKSIZE** – size of stack for created threads
- **OMP\_THREAD\_LIMIT**

---

## ***General OpenMP Performance Considerations***

- **Avoid or minimize use of BARRIER, CRITICAL (complete serialization here!), ORDERED regions, and locks. Can use NOWAIT clause to avoid redundant barriers.**
- **Parallelize at a high level, i.e. maximize the work in the parallel regions to reduce parallelization overhead.**
- **Use appropriate loop scheduling – static has low synchronization overhead but can be unbalanced, dynamic (and guided) have higher synchronization overheads but can improve load balancing.**
- **Avoid false sharing (more about it in following slide)!**

---

# *What is False Sharing?*

- Most modern processors have a cache buffer between slow memory and high speed registers of the CPU.
- Accessing a memory location causes a “cache line” to be copied into the cache.
- In an OpenMP code two processors may be accessing two different elements in the same cache line. On writes this will lead to “cache line” being marked invalid (because cache coherency is being maintained).
- This will lead to an increase in memory traffic even though the write is to different elements (hence the term **false sharing**).
- This can have a drastic performance impact if such updates are occurring frequently in a loop.

---

# *False Sharing Example*

## Code snippet:

```
double global=0.0, local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
  int tid = omp_get_thread_num();
  local[tid] = 0.0;
  #pragma omp for
  for (i = 0; i < N; i++)
    local[tid] += x[i];
  #pragma omp atomic
  global += global_local[me];
}
```



---

# *False Sharing - Solutions*

- **Three options**

- Compiler directives to align individual variables on cache line boundaries

```
__declspec (align(64)) int thread1_global_variable;
```

```
__declspec (align(64)) int thread2_global_variable;
```

- Pad arrays/data structures to make sure array elements begin on cache line boundary.
- Use thread local copies of data (assuming the copy overhead is small compared to overall run time).

# *Homework!*

- **Download matrix multiply example from LLNL site:**
  - [https://computing.llnl.gov/tutorials/openMP/samples/Fortran/omp\\_mm.f](https://computing.llnl.gov/tutorials/openMP/samples/Fortran/omp_mm.f)  
(wget [https://computing.llnl.gov/tutorials/openMP/samples/Fortran/omp\\_mm.f](https://computing.llnl.gov/tutorials/openMP/samples/Fortran/omp_mm.f) on Gordon)
- **Compile (ifort -fopenmp omp\_mm.f) and run the example. See if you can vary the environmental variables, scheduling to get better performance!**

---

# *References*

- **Excellent tutorials from LLNL:**
  - <https://computing.llnl.gov/tutorials/mpi/>
  - <https://computing.llnl.gov/tutorials/openMP/>
- **MPI for Python:**
  - <http://mpi4py.scipy.org/docs/usrman/tutorial.html>
- **MVAPICH2 User Guide:**
  - <http://mvapich.cse.ohio-state.edu/userguide/>