
Parallel Computing using MPI & OpenMP

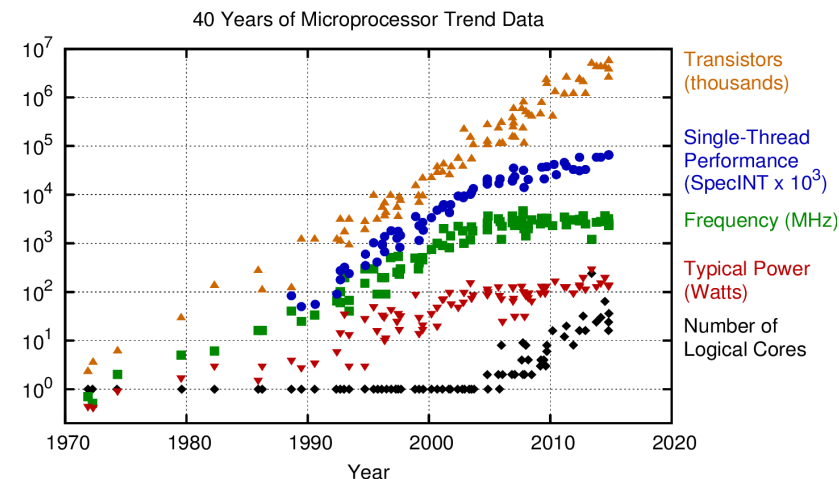
Pietro Cicotti

Director, Advanced Technology Lab

Summer Institute 2017

Why Parallel Computing?

- **Why we need high performance computing?**
 - Science!
 - Qualitative improvement in simulation resolution
 - Explosion of data to be analyzed
 - Both in industry and academic community
 - Competitive advantage
 - National Security



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

What is parallel computing?

- **Executing instructions concurrently on physical resources (not time slicing)**
 - Multiple tightly coupled resources (e.g. cores) collaboratively solving a single problem
- **Benefits**
 - Capacity
 - Memory, storage
 - Performance
 - More instructions per unit of time (FLOPS)
 - Data streaming capability
- **Cost and Complexity**
 - Coordinate tasks and resources
 - Use resources efficiently

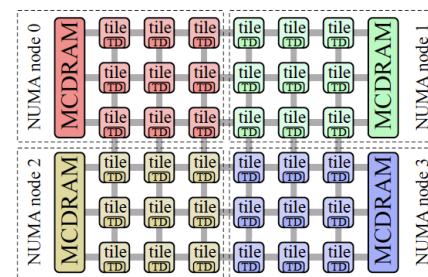
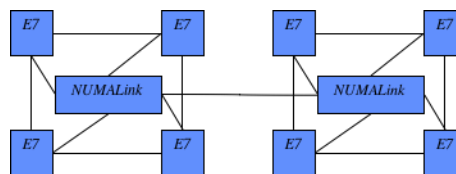
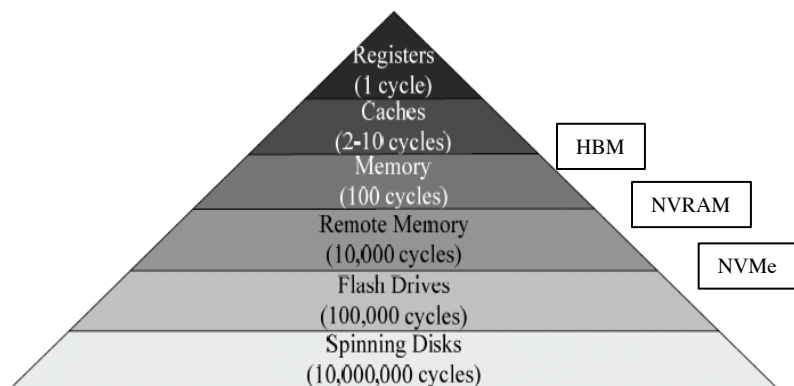
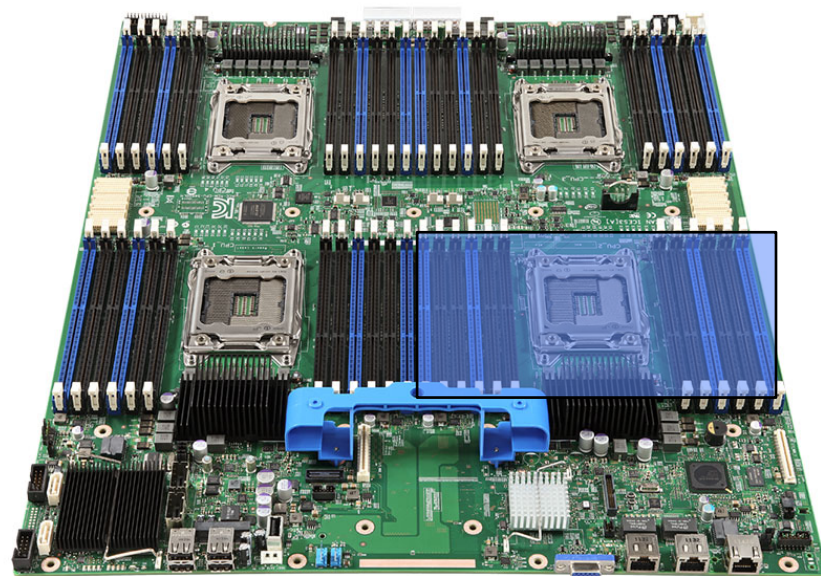
Flynn's Taxonomy

Single Instruction Single Data	Single Instruction Multiple Data
Multiple Instructions Single Data	Multiple Instructions Multiple Data

- **Single Program Multiple Data (SPMD)**
- **Multiple Program Multiple Data (MPMD)**

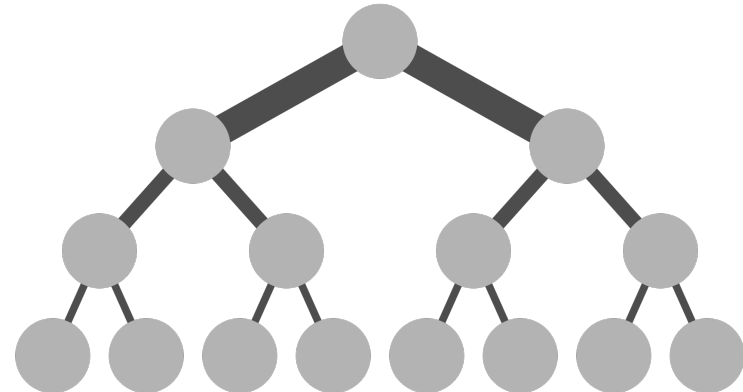
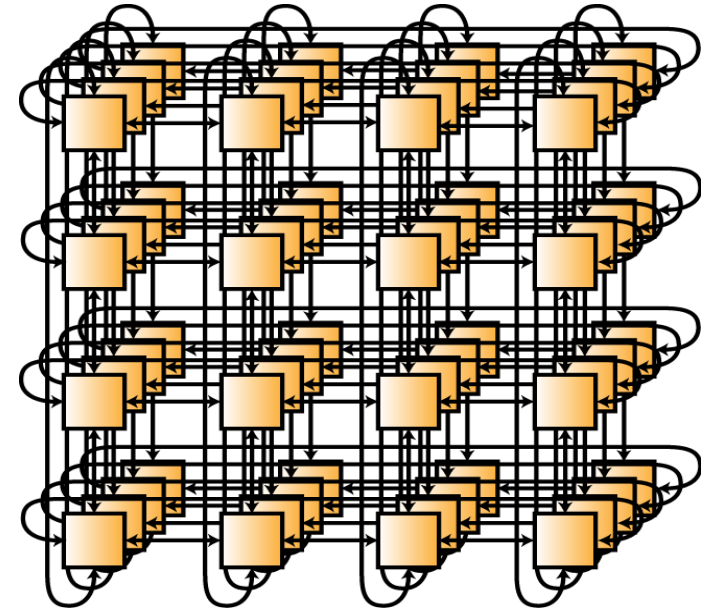
Today's clusters

- **Multi-socket server nodes**
 - NUMA, accelerators
- **High performance interconnect**
 - E.g. Infiniband, OmniPath



Distributed Memory Clusters Topologies

- **Mesh, Torus, Hypercube**
- **Tree based**
 - Fat-tree
 - Clos
- **Dragonfly**
- **Metrics**
 - Bandwidth
 - Diameter, Connectivity
 - Bisection bandwidth



Memory and Execution Models

- **Shared**
 - Communication model: share memory
- **Distributed**
 - Communication model: exchange messages
- **Execution**
 - Fork-Join (e.g. Thread Level Parallelism)
 - SPMD
- **Parallelism enabled by decomposing work**
 - Tasks can be executed concurrently
 - Some tasks can have dependencies

What is Multi-Threading?

- **A process is an instance of a program in execution**
- **A thread is a scheduling and execution unit**
 - AKA lightweight process
 - Has its own *context*
 - Shares virtual address space within the process
 - Communication based on a shared memory model
 - Share other resources (e.g. file descriptors, buffers)
 - (in HPC) Run on a core or hardware thread
- **Pthreads (POSIX), C++11, Java, etc.**
- **Hardware threads support OS threads**
 - Hyper-threading (Intel)

What is OpenMP

- **High level parallelism abstraction based on threads**
 - Easy to use
 - Suitable to an incremental approach
- **A specification and evolving standard**
 - “a portable, scalable model ... for developing portable parallel programs”
 - <http://openmp.org>
 - GNU, Intel, PGI, etc.
- **A set of**
 - Compiler directives
 - Library routines
 - Environment variables
- **Supports C/C++ and Fortran**

```
#pragma omp parallel  
{  
  ....  
}
```

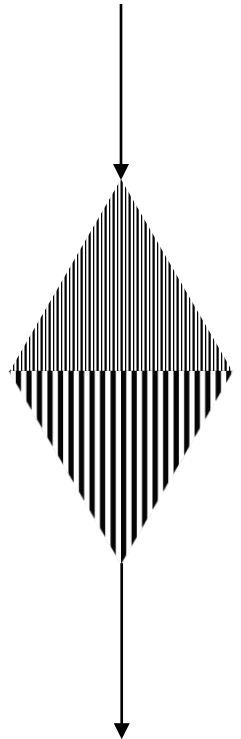
OpenMP Models

- **Fork/Join Execution**

- Process starts single threaded (master thread)
- Forks child threads activated in parallel regions (team)
 - The team synchronizes and threads are disbanded
 - barrier
 - Overhead is mitigated by reusing threads
- Master thread continues execution of serial phases

- **Work decomposition**

- Programming constructs
 - Scope and compound statements
- Declarative in loops
 - Mapping to threads can be static or dynamic
 - Barriers and synchronization automatically inserted



Directives

- **Compiler directives apply to the succeeding structured block**
 - `#pragma omp`
 - Single statement or compound statement `{ }`
 - Clauses modify properties of the directive
 - E.g. *private, if, nowait*
 - Compiler generate code
 - Instructions, functions, function calls
 - Transparent to the user
- **Main mechanism for declaring parallel regions of execution**
 - E.g. loops, sections

Regions, Loops, Sections, etc.

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line  
structured-block  
clause:  
if(scalar-expression)  
num_threads(integer-expression)  
default(shared | none)  
private(list)  
firstprivate(list)  
shared(list)  
copyin(list)  
reduction(operator: list)
```

- **#pragma omp single/master**
- **simd**
- **tasks**

```
#pragma omp for [clause[[,] clause] ... ] new-line for-loops  
clause:  
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction(operator: list)  
schedule(kind[, chunk_size])  
collapse(n)  
ordered  
nowait
```

```
#pragma omp sections [clause[[,] clause] ...] new-line  
{  
  #pragma omp section  
  structured-block  
  ...  
}  
clause:  
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction(operator: list)  
nowait
```

Scope of Variables

- **Clauses determine the scope of variables**
 - Default: shared (external)
 - private
 - Also if declared inside region
 - firstprivate
 - shared
 - lastprivate
 - reductions
 - default
 -
- **Avoid race conditions!**

Decomposition and mapping of the iteration space

schedule(kind[,chunk_size])

kind:

static: Iterations are divided into chunks of size *chunk_size* and assigned to threads in the team in round-robin fashion in order of thread number.

dynamic: Each thread executes a chunk of iterations then requests another chunk until none remain.

guided: Each thread executes a chunk of iterations then requests another chunk until no chunks remain to be assigned.

auto: The decision regarding scheduling is delegated to the compiler and/or run me system.

runtime: The schedule and chunk size are taken from the *run-sched-var* ICV.

More Synchronization

- **#pragma omp critical**
 - Executed by one thread at a time
- **#pragma omp barrier**
 - Explicit barrier
- **#pragma omp atomic**
 - Atomic instruction
 - Storage is accessed atomically

Controlling and Querying the environment

- **Env vars**
 - e.g. OMP_NUM_THREADS
- **Routines**
 - Execution
 - omp_[get | set]_num_threads
 - omp_get_thread_num
 - Locking
 - omp_init_lock, omp_set_lock, omp_unset_lock
 - Timing
 - omp_get_wtime()
 - omp_get_wtick

Compute PI with openmp

- <https://github.com/sdsc/sdsc-summer-institute-2017>
- `hpc2_mpi_openmp/examples/openmp`
- `salloc -N 1 -t 00:30:00 --reservation SI2017DAY3`
- On Comet...

Correctness Considerations

- **Sharing data**

- Dependencies must be enforced
- Operations are not atomic unless specified

Thread 0	Thread 1
x=0	
++X	++X
x==2?	

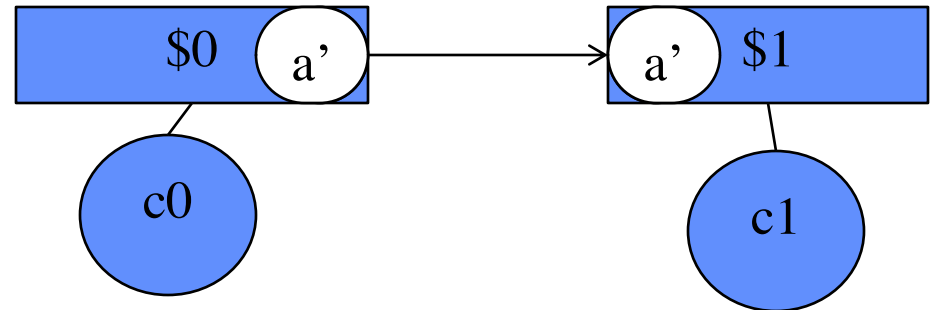
- Caches are coherent, registers are not
- **Loops may carry dependencies across iterations**
 - for i=0 to 9 do A[i]+=B[i]
 - for i=1 to 9 do A[i]+=A[i-1]

Performance Considerations

- **Synchronizations and serialization hurt performance**
 - barriers, locks, critical sections, single thread blocks
 - nowait close
- **Coarse parallelization reduces overhead**
- **Preserve locality**
 - NUMA
 - Bind threads to cores
 - Avoid false sharing
- **Use optimal scheduling**

Sharing and ...

- **Threads share the address space**
 - Great for sharing data and synchronization
 - Dangerous because of sharing data and synchronization
- **Variables and data scoping**
 - Automatic
 - Thread local
 - Heap
- **Sharing**
 - Memory consistency model governs visibility of changes
 - Coherency protocol
 - invalidates and moves data between caches



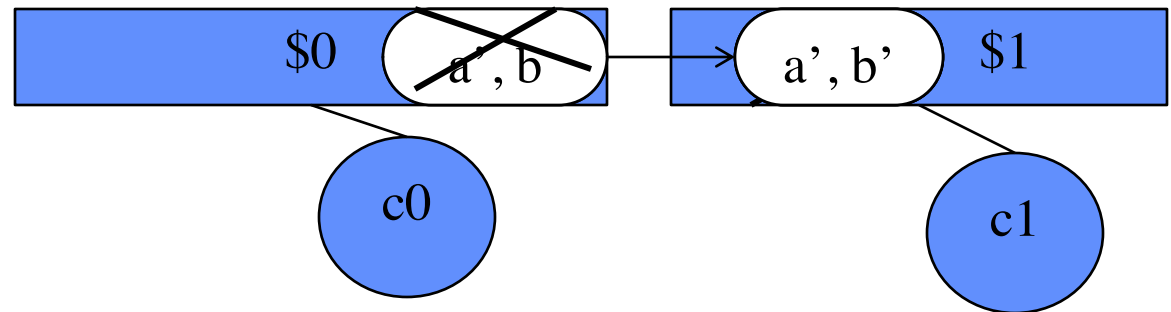
... False Sharing

- **False**

- Threads write different variables

- **Sharing**

- The variables are on the same cache line



- **Solution**

- Align to cache lines and separate variables
 - padding
 - Use pre-allocated blocks

False sharing: example

```
#pragma omp parallel for schedule(static,1)
for(int i=0; i<N; ++i)
    ++x[i];
```

- **Solutions**

- align data and partition boundaries to cache line size
 - `int x __attribute__((aligned (16))) = 0;`
- pad arrays when needed
 - Element are cache line aligned
 - Boundaries are cache line aligned
- Use local copies whenever possible

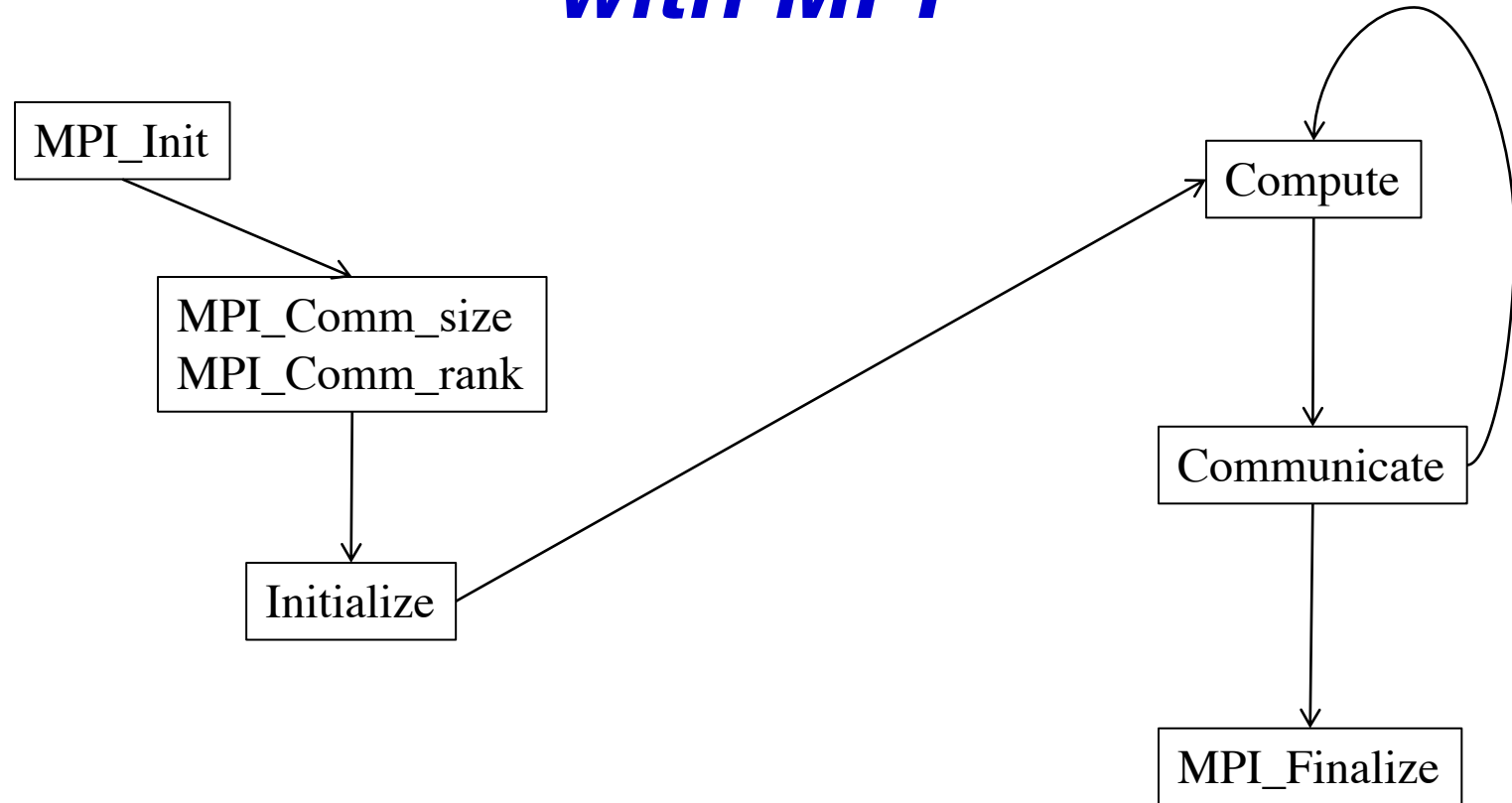
Practice!

- **Check the specification**
 - <http://www.openmp.org>
- **Try to write a program to do a parallel sort**
 - d&c: quick sort (man qsort), serial merge
 - Can you improve on serial merge?
 - Amdhal's law?

Message Passing Interface (MPI)

- **Low level message passing abstraction**
 - SPMD execution model + messages
 - Designed for distributed memory
 - send-recv basic primitives
- **MPI: API specification**
 - Portable: de-fact standard for parallel computing
 - <http://www.mpi-forum.org>
 - E.g. openMPI, mpich, mvapich, LAM
 - High performance implementations available virtually on any interconnect and system
 - Point-to-point communication, datatypes, collective operations
 - One-sided communication, Parallel file I/O, Tool support, ...

Bulk Synchronous Programming with MPI



- ~Locally execute a serial program?

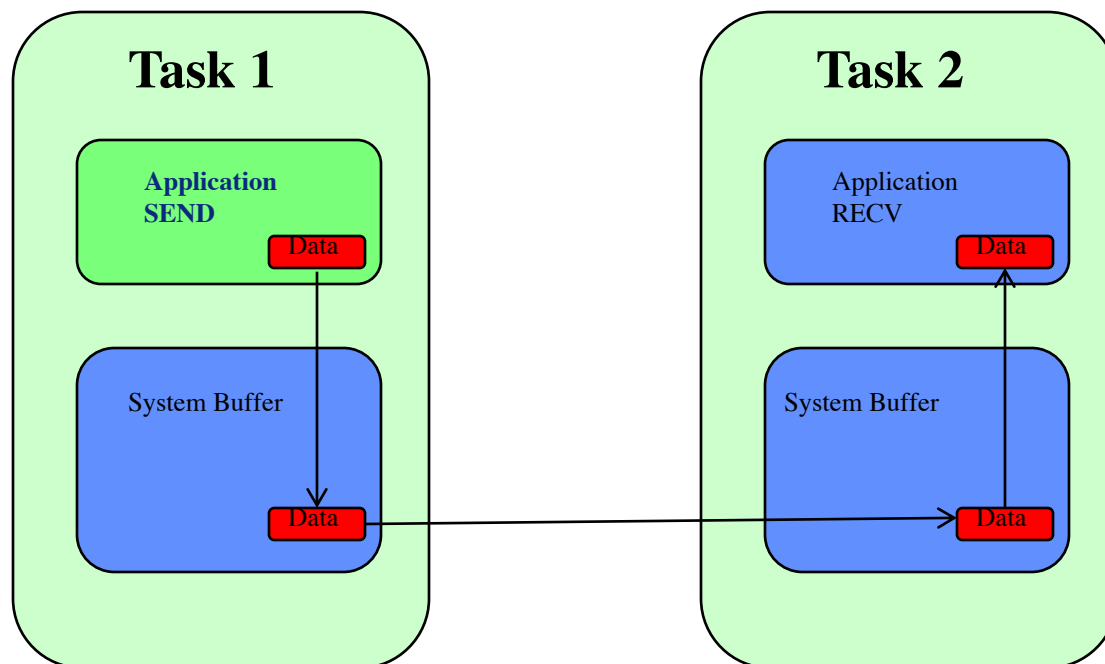
Communicators

- **Define a communication domain**
 - Set of processes that communicate with each other
 - Required for message transfer routines
- **MPI_COMM_WORLD**
 - Default communicator
 - Includes all the processes
- **Useful for library developers**
 - Libraries define own communicators
 - For encapsulation
 - Avoid interference with main program and other libraries
- **Logically partition the data/processes**
 - Match data and work decomposition
- **MPI_Comm_size, MPI_Comm_rank**

Point-to-Point Communication

- **MPI_[I][?]Send, MPI_[I]Recv**
- **Message = data + envelop (src,dst,comm,tag)**
- **[?] communication mode modifies the semantics of the send**
 - Standard
 - Buffered
 - Synchronous
 - Ready
- **[I] Immediate routines**
 - Blocking vs. non-blocking
 - Start, wait, test
- **Example**
 - `MPI_Send(buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD)`
 - `MPI_Recv(buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status)`

Buffering and message transfer



Avoiding deadlocks

- **Deadlocks are common mistakes**
 - Circular dependencies
 - Control error or unexpected behavior/semantics
- **Example**

```
if(myrank==1) {  
    MPI_Ssend(buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);  
    MPI_Recv(buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);  
} else { // myrank == 0  
    MPI_Ssend(buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);  
    MPI_Recv(buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);  
}
```

Fixes to deadlock example?

```
if(myrank==1) {  
    MPI_Ssend(buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);  
    MPI_Recv(buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);  
} else {  
    MPI_Recv(buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);  
    MPI_Ssend(buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);  
}
```

```
MPI_Send(buf, 1, MPI_INT, myrank?0:1, 0, MPI_COMM_WORLD);  
MPI_Recv(buf, 1, MPI_INT, myrank?0:1, 0, MPI_COMM_WORLD, &status);
```

```
MPI_Isend(buf, 1, MPI_INT, myrank?0:1, 0, MPI_COMM_WORLD, &req);  
MPI_Recv(buf, 1, MPI_INT, myrank?0:1, 0, MPI_COMM_WORLD, &status);  
MPI_Wait(&req, &status);
```

Collective Communication

- **All ranks in a communicator participate**
 - Potential optimizations with respect to point-to-point
 - Broadcast: $n-1$ messages vs. $\log(n)$ messages
- **Barriers**
 - Synchronize all ranks
- **Broadcast**
- **Reduction**
- **Scan**
- **Gather/Scatter, Alltoall**
- ***All* and *vector* variants**

Data Types

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

MPI Reduction Operations

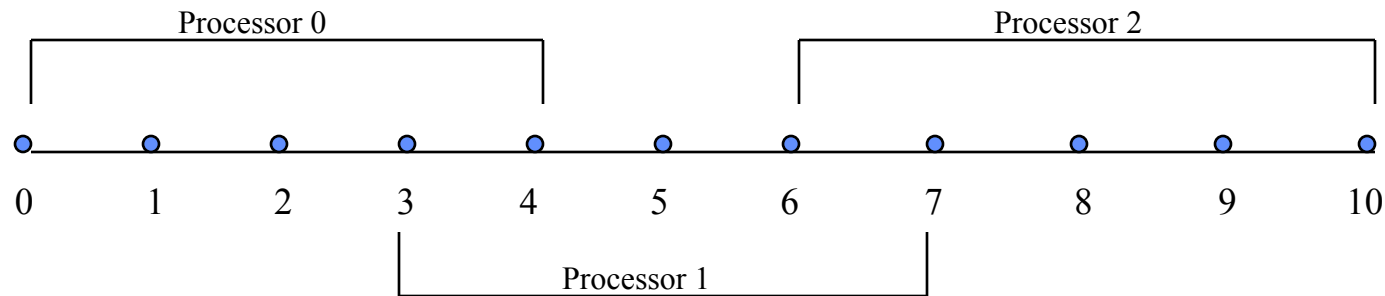
NAME	OPERATION
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bit-wise AND
MPI_LOR	Logical OR
MPI_BOR	Bit-wise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bit-wise XOR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

Decomposition and Mapping

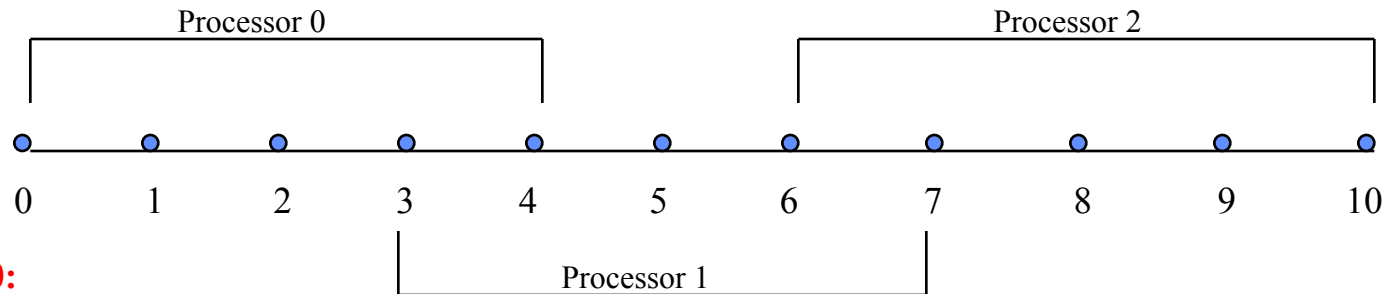
- **Data and work decomposition**
 - Map partitioned domain to processes
- **Mapping**
 - Processes/ranks topology
 - System/Domain/Data
- **How to share data?**
 - Exchange messages and replicate data
- **Load imbalance**
 - What is the system is not regular?
 - Work is not proportional to data?

1D heat equation with MPI

- $\partial T / \partial t = \alpha (\partial^2 T / \partial x^2)$; $T(0) = 0$; $T(1) = 0$; $(0 \leq x \leq 1)$
- $T(x, 0)$ initial condition
- Discretization
 - $T(x_i, n+1) - T(x_i, n) = (\alpha \Delta t / \Delta x^2) (T(x_{i-1}, n) - 2T(x_i, n) + T(x_{i+1}, n))$
- Partitioning
 - Ghost cells



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:

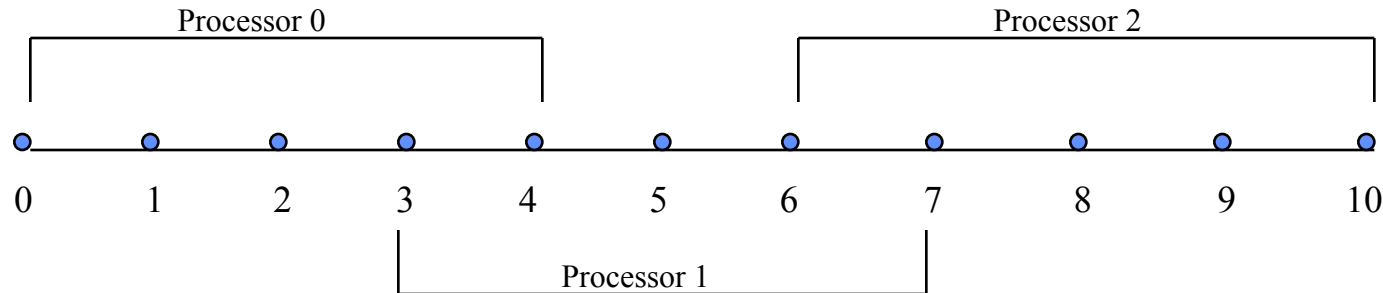
Local Data Index : $i_{\text{local}} = 0, 1, 2, 3, 4$

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

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
```

Examples

- <https://github.com/sdsc/sdsc-summer-institute-2017>
- `hpc2_mpi_openmp/examples/mpi`
- `salloc -N 1 -t 00:30:00 --reservation SI2017DAY3`
- `mpirun`
- On Comet...

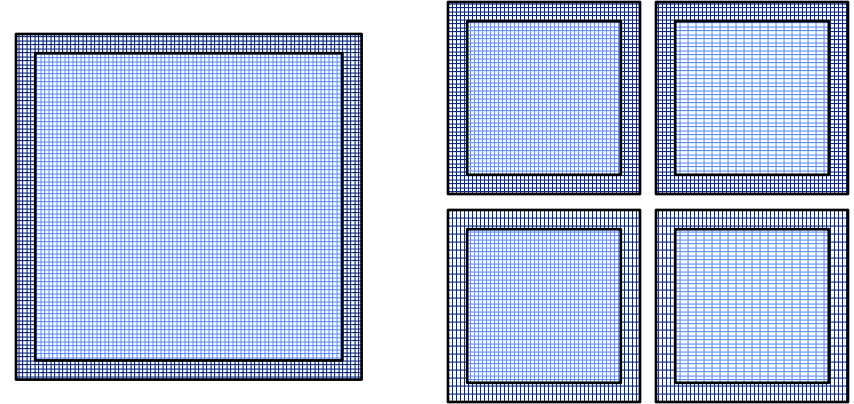
Performance Considerations

- **Overlap communication with computation**
 - Use non-blocking primitives
 - Hide communication cost
 - Split-phase programming
- **Minimize surface-to-volume ratio**
 - Ghost cell exchange
- **Avoid communication**
 - Even at the cost of some more computation
 - Example: double size of ghost cell and communicate every other time step
 - Communication avoiding algorithms

Surface to Volume Ratio

- **Effect of dimensionality**

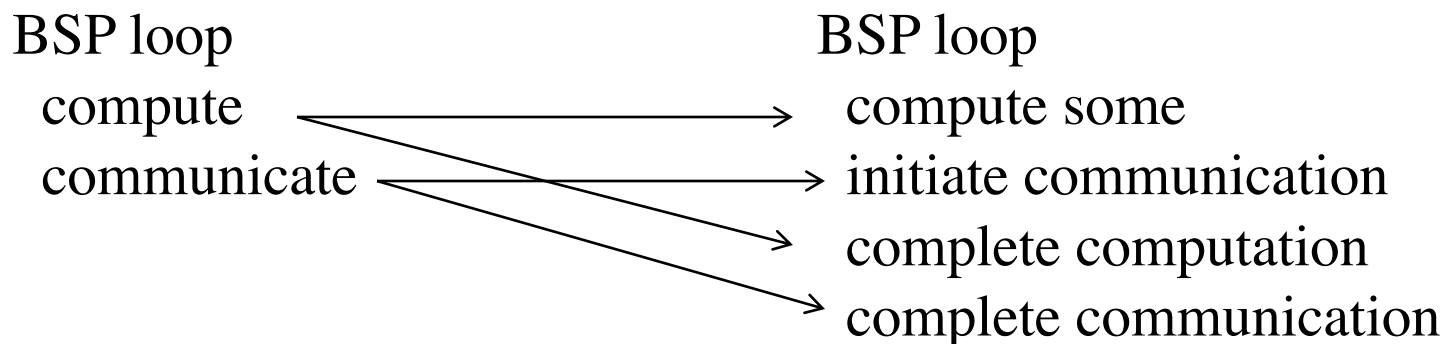
- 1D $2:n$
- 2D $4n:n^2$
- 3D $6n^2:n^3$



- **Computation is proportional to volume**
- **Communication is proportional to surface**
- **Strong scaling**
 - Surface to Volume ratio increases
 - Higher communication overhead

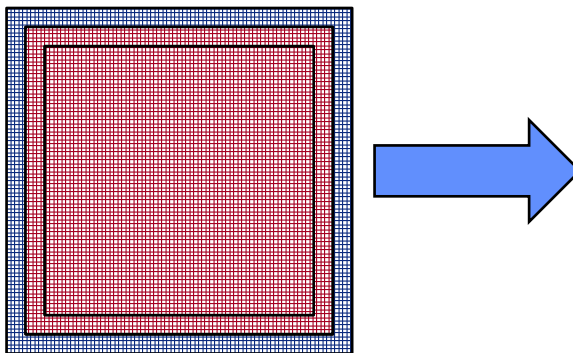
Asynchronous Communication

- **Overlap communication w/ computation**
 - High performance interconnects can offload communication tasks from CPU to adapter
- **Condition**
 - No data dependencies on transfer
- **Split-phase programming**



Hiding Communication

- **Domain decomposition**
 - Ghost cells to apply stencil on boundaries of local domain
- **Split phase**
 - Isend, Irecv, Wait
 - Thin ghost layer effect on locality



Debugging, Profiling, Tracing

- **Command line debuggers**
 - gdb, idb
- **Allinea DDT is installed on Comet**
 - GUI
- **Profiling**
 - mpiP, fpmpi, *ipm*, *TAU*
 - runtime breakdown: communication vs. computation
- **Identify bottlenecks and scaling issues**

mpiP sample 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	24	
Send	1	1	6	8	8	48	
Send	1	2	3	8	8	24	
Send	1	*	12	8	8	96	

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

fpmpi

Date: Wed Aug 2 17:31:50 2017

Processes: 3

Execute time: 0.3295

Timing Stats: [seconds] [min/max] [min rank/max rank]

wall-clock: 0.3295 sec 0.329253 / 0.329837 1 / 0

user: 0.01033 sec 0.008998 / 0.012998 1 / 0

sys: 0.07099 sec 0.067989 / 0.073988 1 / 0

Memory Usage Stats (RSS) [min/max KB]: 9348/13472

Average of sums over all processes

Routine	Calls	Time	Msg Length	%Time by message length
			0.....1.....1.....	
			K M	
MPI_Recv	: 4	2.03e-05	32 00*000000000000000000000000	
MPI_Send	: 4	1.1e-05	32 00*000000000000000000000000	

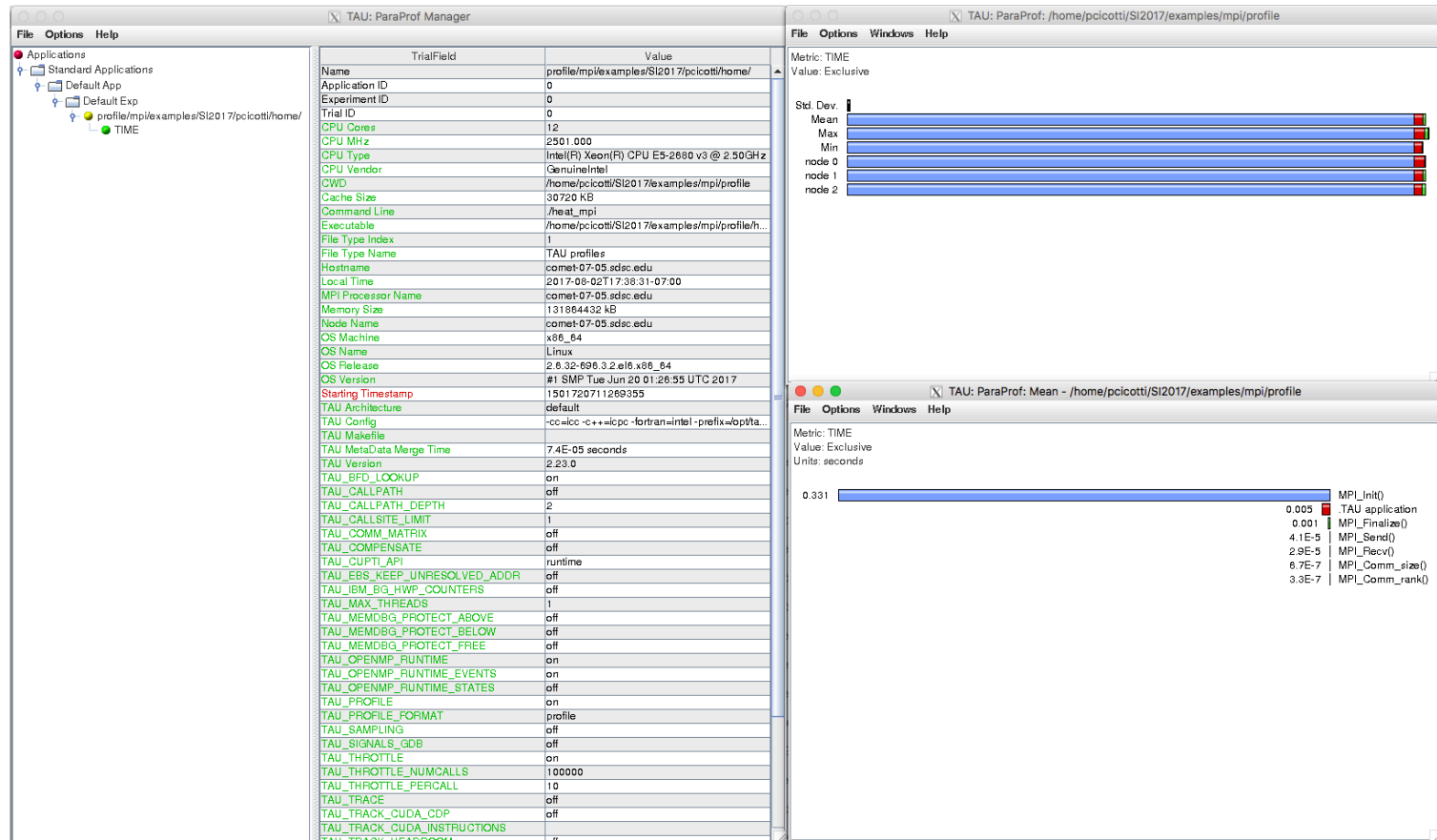
Details for each MPI routine

Average of sums over all processes

			% by message length
	(max over	0.....1.....1.....	
	processes [rank])	K M	
MPI_Recv:			
Calls	: 4	6 [1]	00*000000000000000000000000
Time	: 2.03e-05	2.41e-05 [0]	00*000000000000000000000000
Data Sent	: 32	48 [1]	
SyncTime	: 0		
By bin	: 5-8 [3,6]	[1.79e-05, 2.41e-05]	
MPI_Send:			
Calls	: 4	6 [1]	00*000000000000000000000000
Time	: 1.1e-05	1.19e-05 [1]	00*000000000000000000000000
Data Sent	: 32	48 [1]	
SyncTime	: 0		
By bin	: 5-8 [3,6]	[1e-05, 1.19e-05]	
Partners	: 1.33 max 2(at 1) min 1(at 0)		

TAU

- `mpirun -n 3 tau_exec -T MPI ./heat_mpi`



Advanced MPI

- **One-sided communication**
- **Derived data types**
- **Groups, topologies, and communicators management**
- **Parallel I/O**
- **Dynamic process creation and management**
- **Tools support**

Practice

- **<https://github.com/sdsc/sdsc-summer-institute-2017>**
- **hpc2_mpi_openmp/examples/mpi**
- **Find bugs in code sample.f**
 - Compile mpif90 -o sample sample.f
- **Questions?**
- **Implement your code?**

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/>