Parallel Computing Using MPI & OpenMP

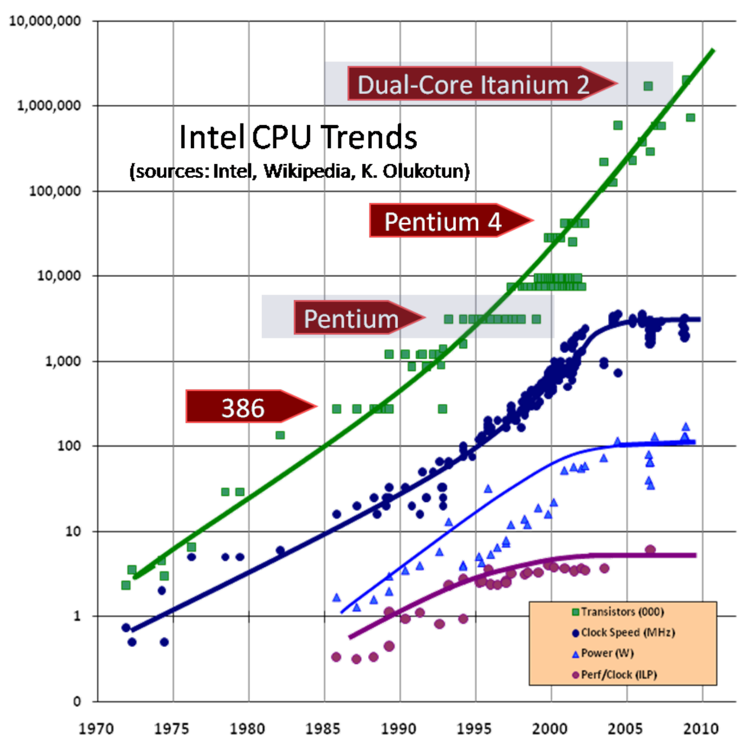
Pietro Cicotti

(pcicotti@sdsc.edu)

08/04/2015

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
- **Why is parallel computing necessary?**



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)
- **Cost and Complexity**
  - Coordinate tasks and resources
  - Use resources efficiently

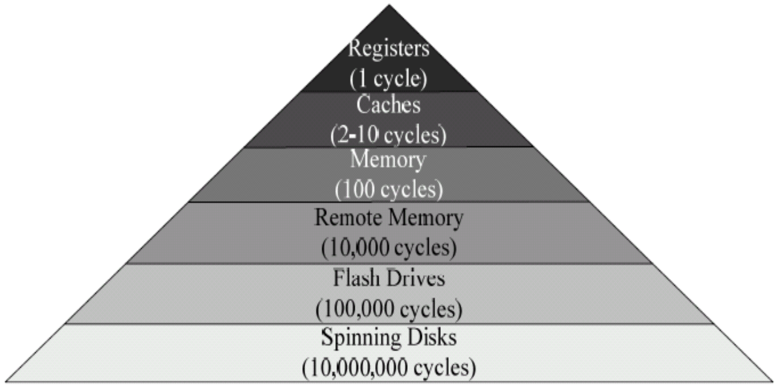
Flynn's Taxonomy

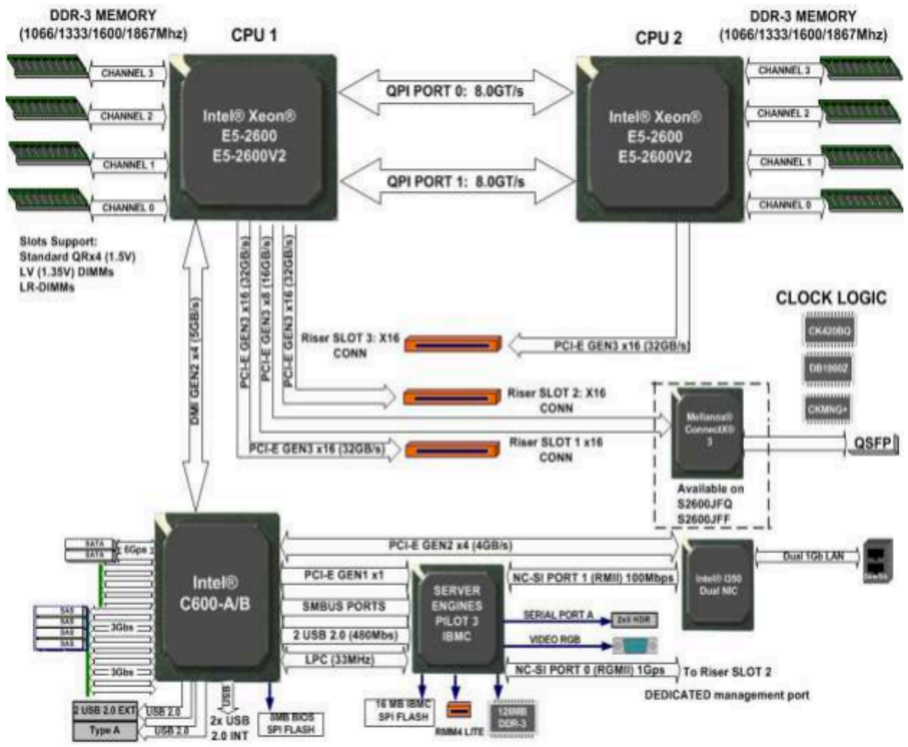
Single Instruction Single Data	Single Instruction Multiple Data
Multiple Instructions Single Data	Multiple Instructions Multiple Data

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

Today's clusters

- **Multi-socket server nodes**
  - NUMA, accelerators
- **High performance interconnect**
  - E.g. IB
- **Parallel File System and Storage System**
  - E.g. Lustre, GPFS

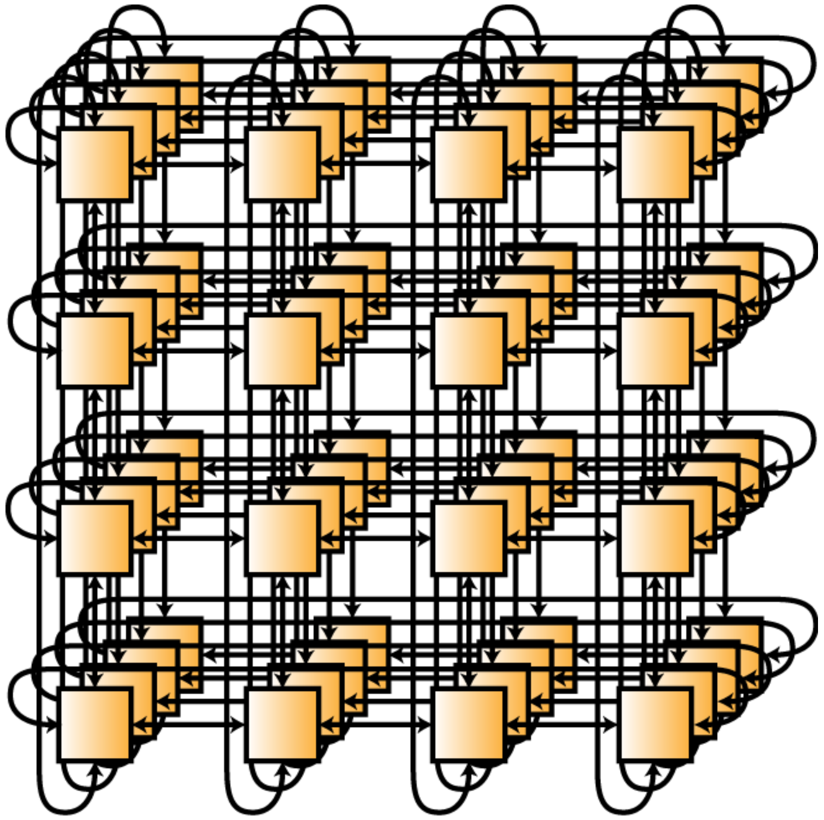




SDSC SAN DIEGO SUPERCOMPUTER CENTER

Distributed Memory Clusters Topologies

- **Mesh, Torus, Hypercube**
- **Tree based**
  - Fat-tree
  - Clos
- **Dragonfly**
- **Metrics**
  - Bandwidth
  - Diameter, Connectivity
  - Bisection bandwidth
- **Example**
  - Gordon
    - dual-rail
    - 3D torus of switches



Architecture, Memory, 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 thread is a lightweight process**
  - OS entity
  - Shares virtual address space within the process
  - Share other resources (e.g. file descriptors, buffers)
  - Does not share a *context*
- **Fork and join are faster than for processes**
- **Fork and join are not free!**
- **Hardware threads support OS threads**
  - Hyperthreading

What is OpenMP

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

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
    - Overhead is mitigated by reusing threads
  - Master thread continues execution of serial phases
- **Work decomposition**
  - Explicit constructs
  - Declarative in loops
    - 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 the properties of the directive
  - 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
dlause:
if(*scalar-expression*)
num_threads(*integer-expression*)
default(shared | none)
private(*list*)
firstprivate(*list*)
shared(*list*)
copyin(*list*)
reduction(*operator*: *list*)

#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

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

Scope of Variables

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

Decomposition 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

- **Examples on Gordon**
  - /home/diag/opt/SI2016/openmp/
  - qsub -I -lnodes=1:ppn=16:native,walltime=00:60:00

1D heat equation with OpenMP

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=0 to 9 do A[i]+=A[i+1] (try with a unit array)

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**

False Sharing

- **Modern processors have SRAM caches**
  - Low capacity
  - High performance
    - e.g. 1 cycle, 20 cycle, 100 cycles, 300 cycles
- **Caches**
  - size, associativity, line (IA64 uses 64B cache lines)
  - x[0]+=1; x[7]+=1; // may be same line
- **Shared memory**
  - Coherency preserved by coherency protocol
  - Invalidate copies when writing
    - Writes cause coherency traffic and serialization
- **Severe impact on performance!**

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

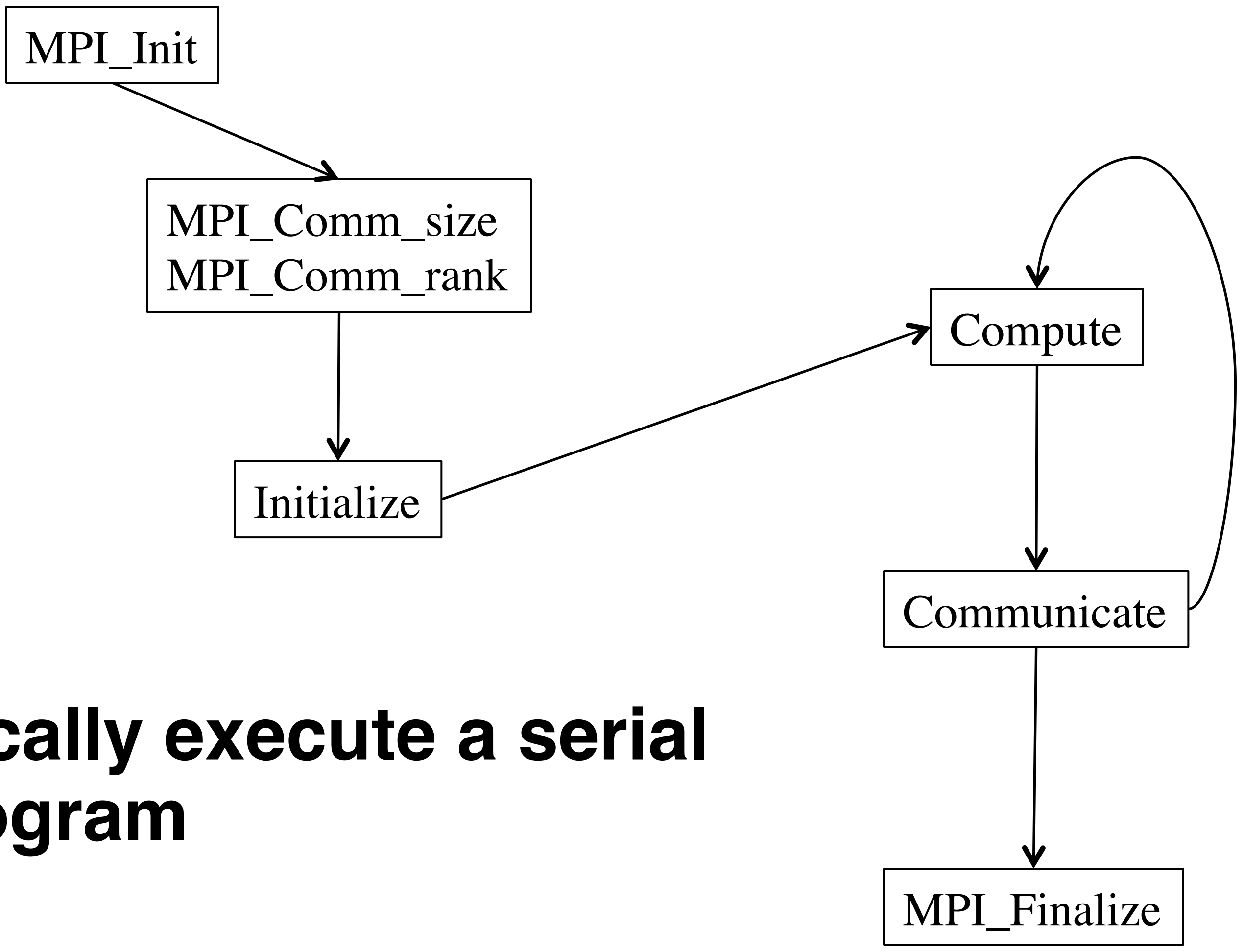
Practice!

- **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?
- **Code in /home/diag/opt/SI2016**

Message Passing Interface (MPI)

- **Low level message passing abstraction**
  - SPMD execution model + messages
  - Designed for distributed memory
  - send-recv basic primites
- **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**

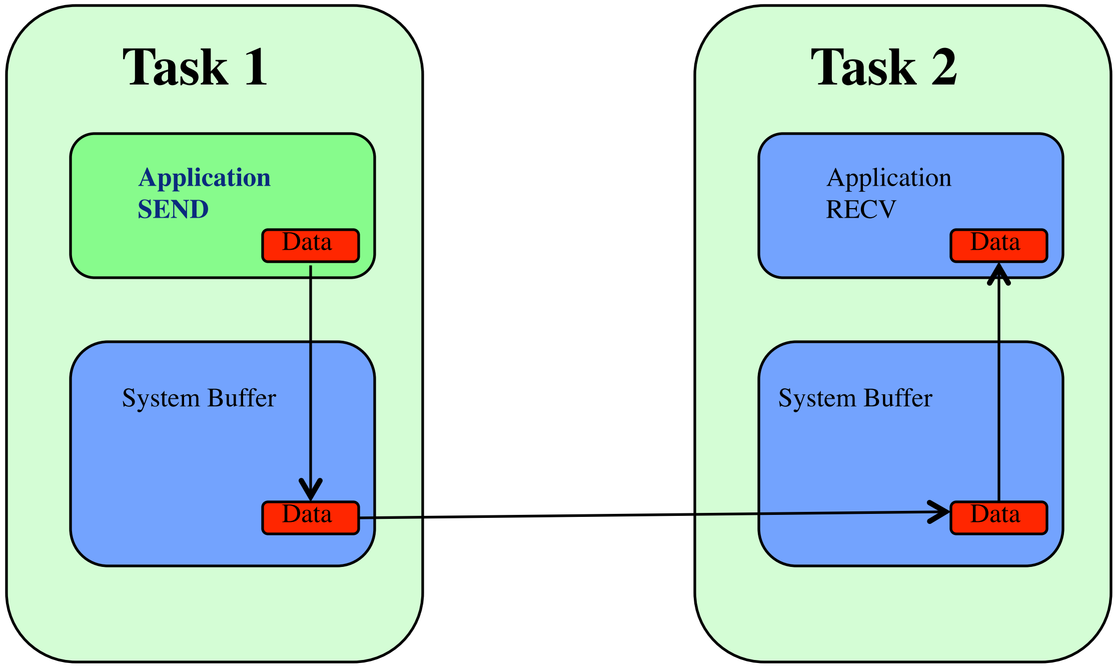
Communicators

- **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**
- **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



Blocking Send-Recv

Avoiding deadlocks

- **Deadlocks are common mistakes**
  - Unexpected behavior/semantics
  - Circular dependencies

- **Example (try different modes!):**

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

Fixes to deadlock example?

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

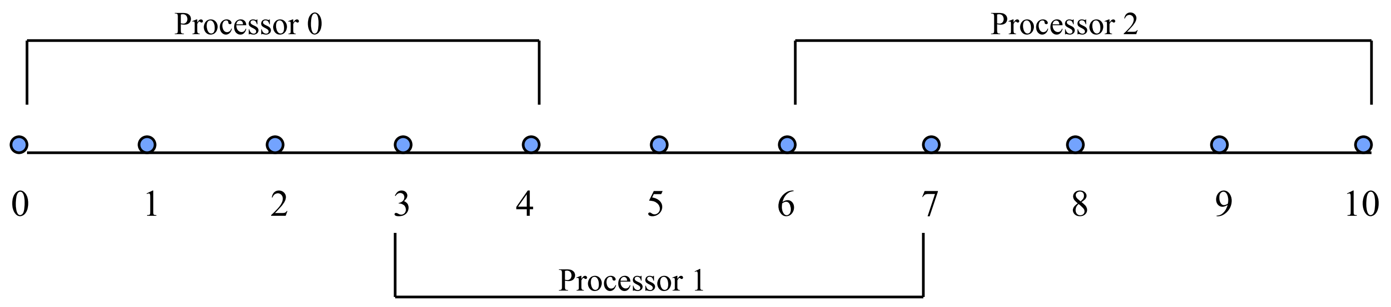
Compute PI with MPI

Collective Communication

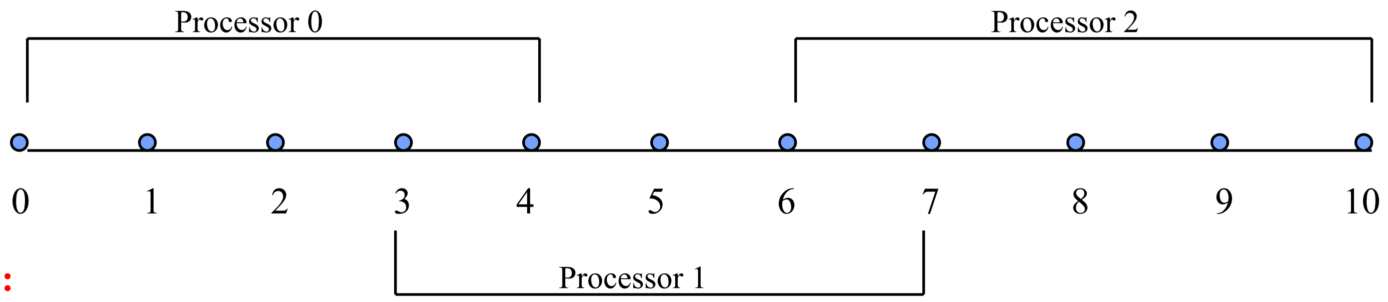
- **All ranks in a communicator participate**
  - Potential optimizations with respect to point-to-point
  - Broadcast: n-1 messages vs. log(n) messages
- **Barriers**
  - Synchronize all ranks
- **Broadcast**
- **Reduction**
- **Gather/Scatter, Alltoall**
- **Scan**
- ***All* and *vector* variants**

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
  - $T(x_i,n+1) - T(x_i,n) = (\alpha\Delta t/\Delta x^2)(T(x_{i-1},n)-2T(x_i,n)+T(x_i,n+1))$
- Partitioning
  - Ghost cells



Simple Application using MPI: 1-D Heat Equation



Processor 0:

Local Data Index : ilocal = 0 , 1, 2, 3, 4

Global Data Index: iglobal = 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 : ilocal = 0, 1, 2, 3, 4

Global Data Index : iglobal = 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:

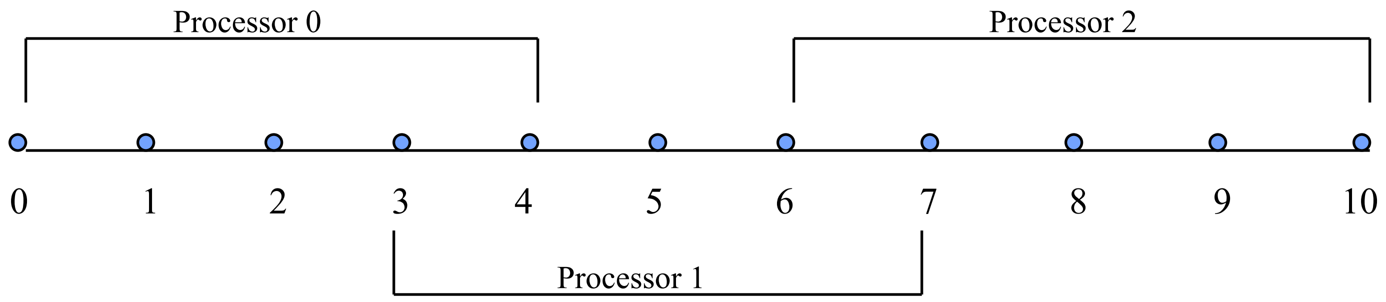
Local Data Index : ilocal = 0, 1, 2, 3, 4

Global Data Index : iglobal = 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



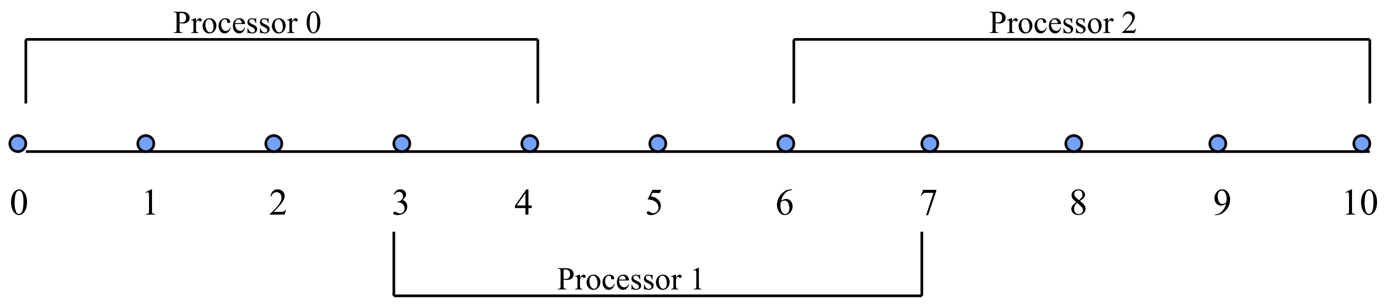
```
% more data0.dat
 Processor  0
 ilocal= 0 ;iglobal= 0 ;T= 0.000000000000000000E+00
 ilocal= 1 ;iglobal= 1 ;T= 0.307205621017284991
 ilocal= 2 ;iglobal= 2 ;T= 0.584339815421976549
 ilocal= 3 ;iglobal= 3 ;T= 0.804274757358271253
 ilocal= 4 ;iglobal= 4 ;T= 0.945481682332597884
```

```
% 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
```

Fortran MPI Code: 1-D Heat Equation

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
```

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

Debugging, Profiling, Tracing

- **Hard to use command line debuggers**
  - gdb, idb
- **Allinea DDT is installed on Gordon and Comet**
  - GUI
- **Profiling**
  - mpiP, TAU, IPM installed on Gordon and Comet
- **Useful information**
  - runtime breakdown: communication vs. computation
- **Identify bottlenecks and scaling issues**

mpiP sample output

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
```

Run

```
mpirun_rsh -hostfile $PBS_NODEFILE -np 3 ./heat_mpi_profile.exe
```

```
@ 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

Advanced MPI

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

Practice

- **Code in /home/diag/opt/SI2016**
- **Find bugs in code sample.f**
  - Compile mpif90 –o sample sample.f
- **Questions?**
- **Try your favorite computation**

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/