Hybrid Programming

Alice Koniges, Berkeley Lab/NERSC

Rusty Lusk, Argonne National Laboratory (ANL)

Rolf Rabenseifner, HLRS, University of Stuttgart, Germany

Gabriele Jost, Texas Advanced Computing Center

This short talk is a conglomeration of larger presentations given by the above authors at a variety of tutorials including SC08, SC09 (upcoming) ParCFD 2009, SciDAC 2009

We are grateful for the use of these slides at the NUG User Group Meeting

Despite continued "packing" of transistors, performance is flatlining

- **New Constraints**
  - 15 years of *exponential* clock rate growth has ended
- **But Moore's Law continues!**
  - How do we use all of those transistors to keep performance increasing at historical rates?
  - Industry Response: #cores per chip doubles every 18 months *instead* of clock frequency!

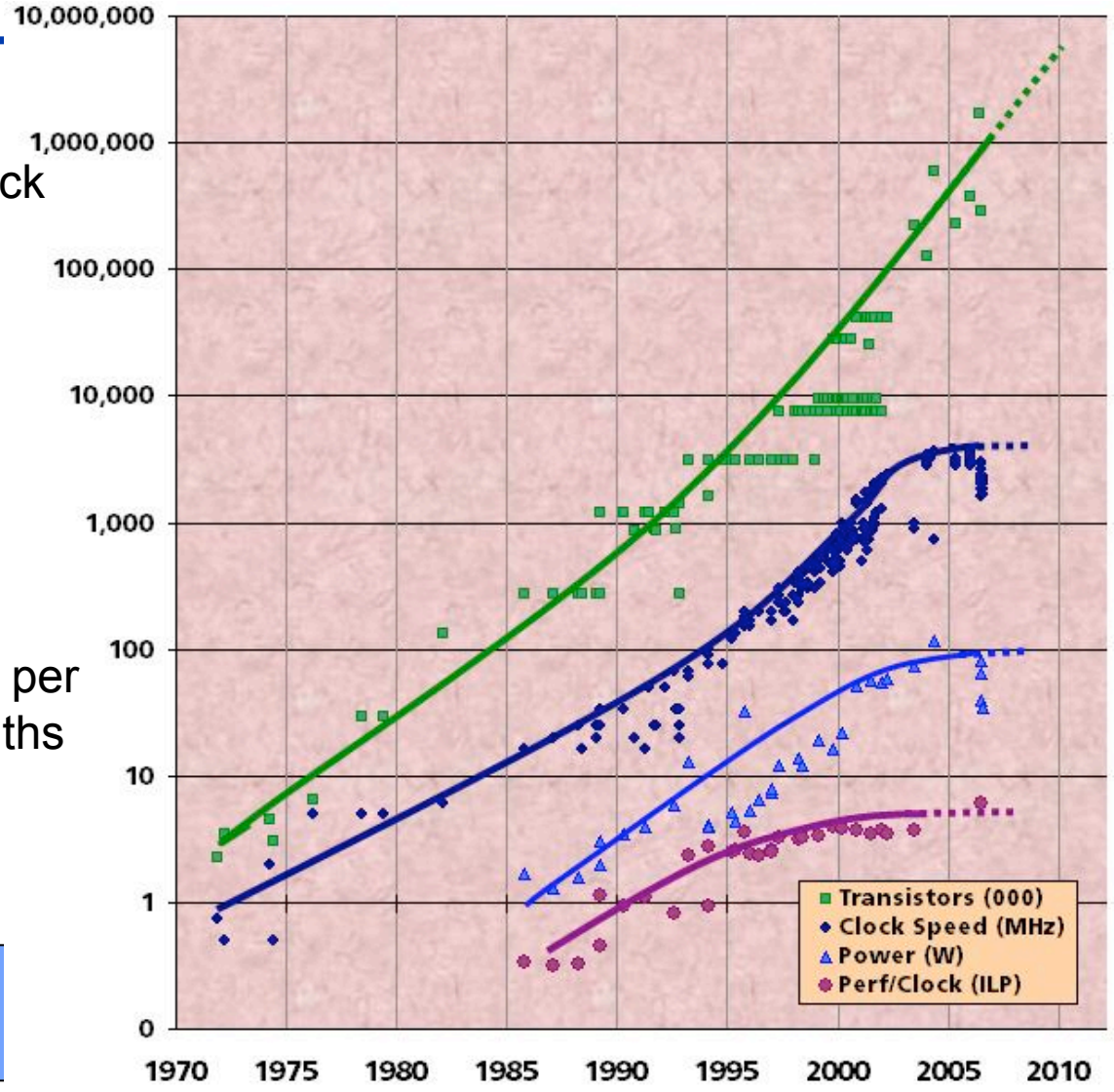
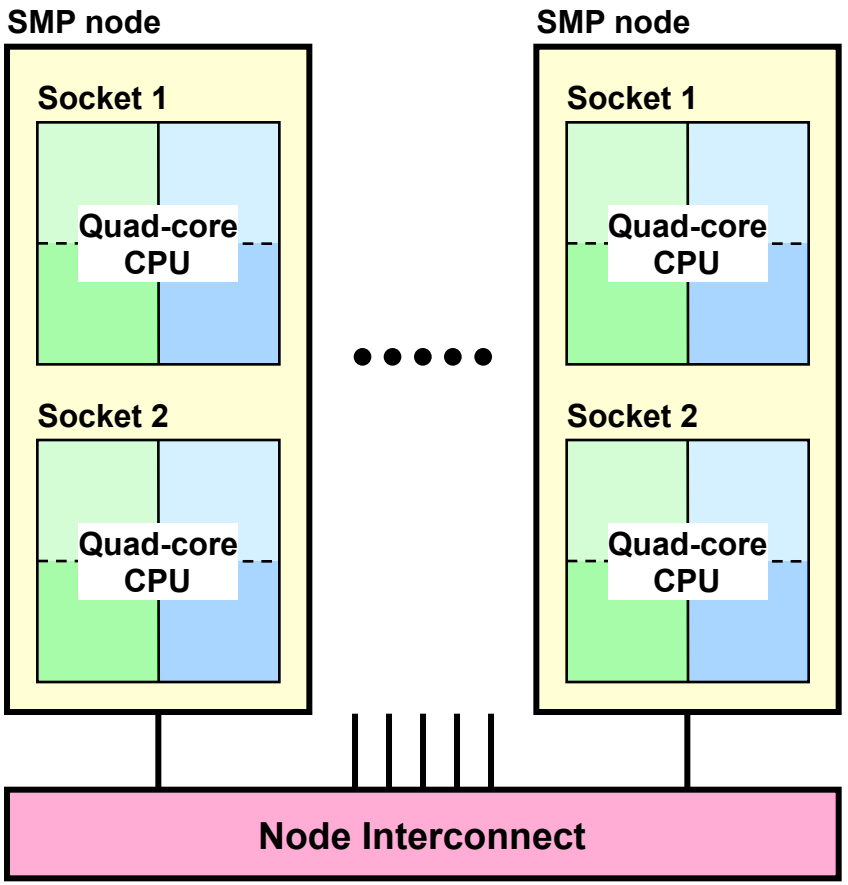


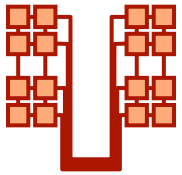
Figure courtesy of Kunle Olukotun, Lance Hammond, Herb Sutter, and Burton Smith

Supercomputers are Hierarchical

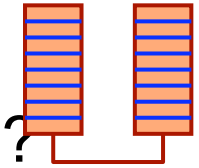


- **Which programming model is fastest?**

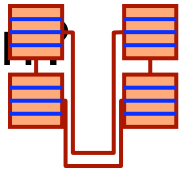
MPI everywhere?



Fully hybrid MPI & OpenMP?



Something between? (Mixed model)



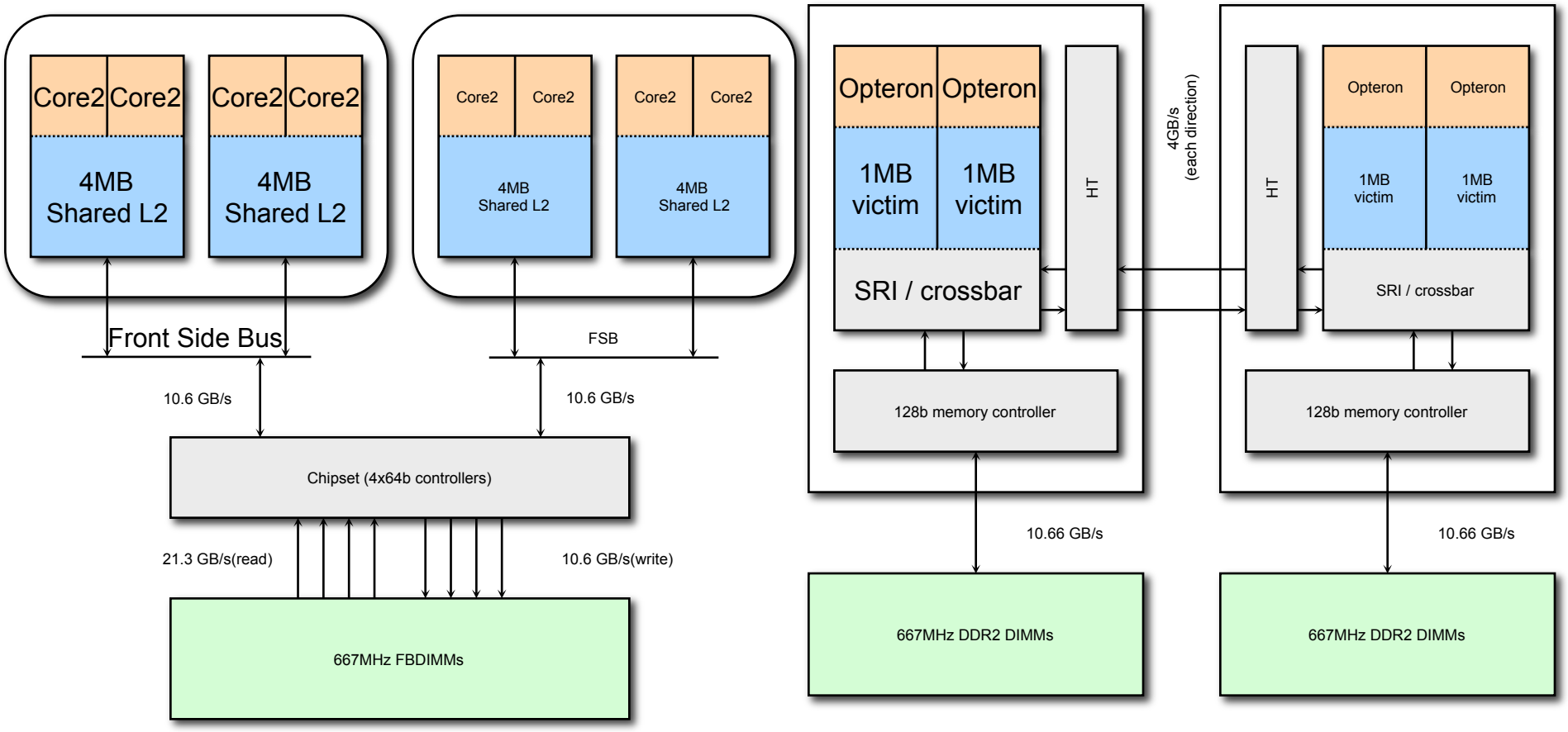
Often hybrid programming can be **slower** than pure MPI

?

Current Multicore SMP Systems can have different memory access and cache use patterns

Intel Clovertown

AMD Opteron



Uniform Memory Access

Non-uniform Memory Access

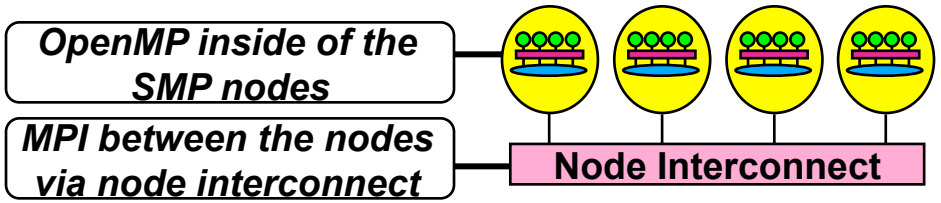
Adapted from Sam Williams, John Shalf, LBL/NERSC et al.

MPI and Threads

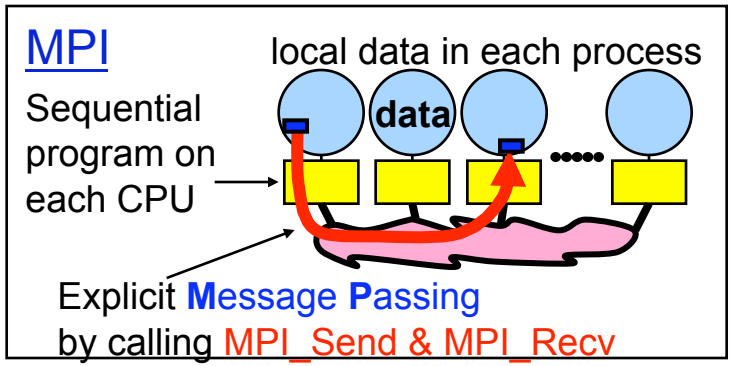
- **MPI describes parallelism between *processes* (with separate address spaces)**
- ***Thread* parallelism provides a shared-memory model within a process, commonly Pthreads and OpenMP**
- **In the threads model of parallel programming, a single process can have multiple, concurrent execution paths**
  - Pthreads (Posix Threads) is a standard library implementation that can be used for parallel programming
  - Pthreads generally provides more complicated and dynamic approaches
  - OpenMP is a set of compiler directives, callable runtime library routines, and environment variables that extend Fortran, C and C++
  - OpenMP provides convenient features for loop-level parallelism
  - OpenMP 3.0 adds task parallelism (released May 2008)

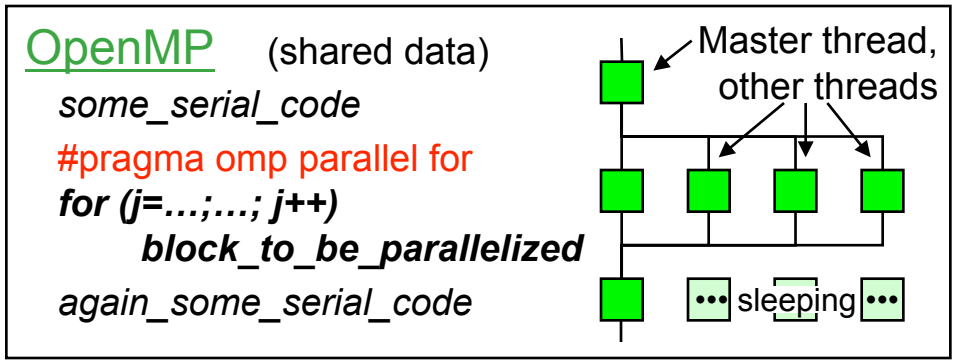
Programming models can be designed for hybrid systems

- **Pure MPI (one MPI process on each CPU) "MPI-Everywhere"**
- **Hybrid MPI+OpenMP**
  - shared memory OpenMP
  - distributed memory MPI



- **Other: Virtual shared memory systems, PGAS, HPF, …**
- **New Models combine MPI and UPC or CAF (see Lusk, et al. SC09, SIAM Feb. 2009.)**





Hybrid Programming generally combines message passing with shared memory

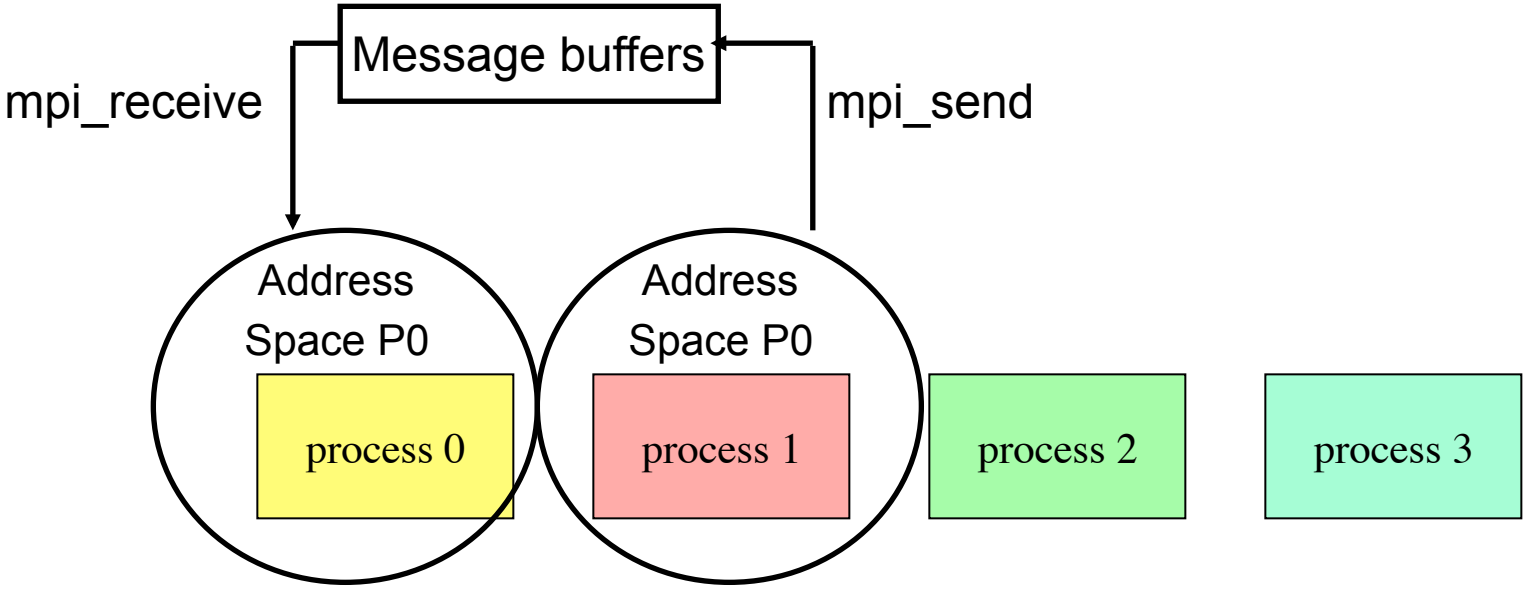
- Other choices for a shared memory model include an implementations by Microsoft and others, as well as Pthreads
- In this talk, we concentrate on combining MPI with OpenMP
- OpenMP consists mostly of directives, where as MPI, related message passing libraries, and Pthreads methods consist of library routines
- Data movement in message passing libraries must be explicitly programmed, in contrast to OpenMP, where it happens automatically as data is read and written
- One goal of OpenMP is to make parallel programming easy
- It is also designed to be implemented incrementally
- We first explain some basic concepts of OpenMP, since we assume you are starting with an MPI program, or knowledge of MPI

Basic Concepts of OpenMP

- OpenMP is an explicit programming model, namely the programmer specifies the parallelism. The compiler and run time system translate this into the parallel execution model.
- The task of the programmer is to correctly identify the parallelism and the dependencies
- OpenMP can have both implicit and explicit synchronization points
- Although OpenMP can be used beyond loop parallelism, we recommend studying loop parallelism first as a way to understand certain concepts such as private and shared variables, false sharing, race conditions, …

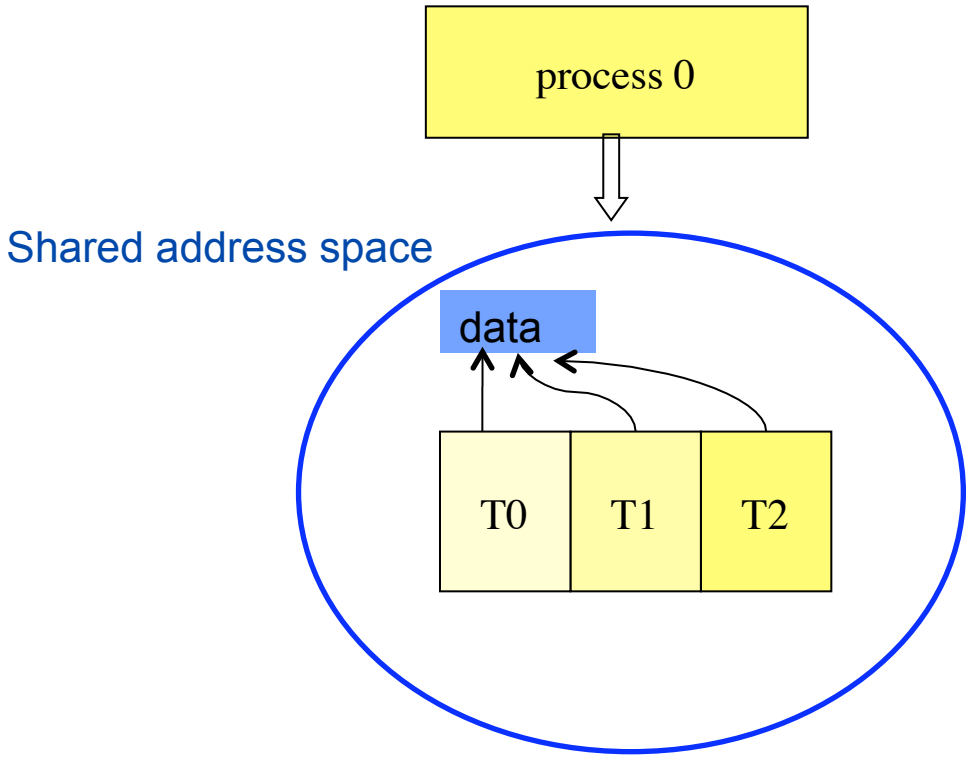
MPI Memory Model

- **Message Passing Interface**
- **Memory Model:**
  - MPI assumes a private address space
  - Private address space for each MPI Process
  - Data needs to be explicitly communicated
- **Applies to distributed and shared memory computer architectures**



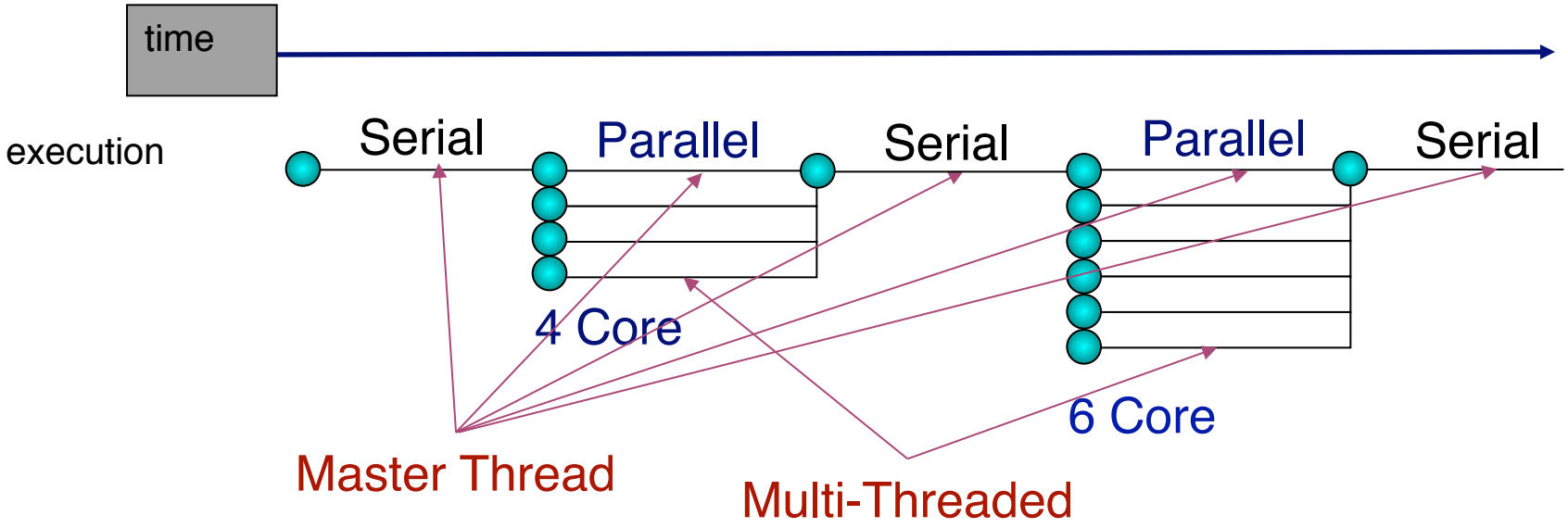
OpenMP Memory Model

- OpenMP assumes a shared address space
- No communication is required between threads
- Thread Synchronization is required when accessing shared data



OpenMP Code General Structure

- Fork-Join Model:
- Execution begins with a single “Master Thread”
- A team of threads is created at each parallel region
- Threads are joined at the end of parallel regions
- Execution is continued after parallel region by the Master Thread until the beginning of the next parallel region



The ParLab at Berkeley sponsored a Parallel Computing Bootcamp

- **Note: All Bootcamp taped lectures, slides can be downloaded**
  - See http://parlab.eecs.berkeley.edu/bootcampagenda
  - Including:
    - OpenMP
    - PGAS Languages
    - OpenCL
    - And several other interesting lectures
- **From this website, you can also download the following:**
  - makefile that should work on all NERSC clusters if you uncomment appropriate lines,
  - job-franklin-serial, job-franklin-pthreads4, job-franklin-openmp4, job-franklin-mpi4,
  - job-bassi-serial, job-bassi-pthreads8, job-bassi-openmp8, job-bassi-mpi8
  - sample batch files to launch jobs on Franklin and Bassi. Use qsub to submit on Franklin and llsubmit to submit on Bassi.
- **We are also preparing sample codes to be available from the NERSC Website**
- **We recommend if you don't know OpenMP, you watch the ParLab BootCamp talk by Tim Mattson**

Comments on Example from ParLAB Talk by Tim Mattson, Intel Corp.

Program to compute PI by quadrature, Tim starts with a simple serial program:

```
static long num_steps = 100000;
double step;
void main()
{       int i; double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;
        x = 0.5 * step;
        for (i=0;i<=num_steps;i++){
                x+=step;
                sum+=4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Next he shows
Steps of Parallelization by OpenMP

- **Identify concurrency in the loop, iterations may be executed concurrently:**

```
for (i=0;i<=num_steps;i++){
                    x+=step;
                    sum+=4.0/(1.0+x*x);
          }
```

- **Steps: isolate data tha must be shared from data that will be local to a task**
- **Redefine x to remove loop carried dependence**
- **Look at how to rewrite the “reduction,” where results from each iteration are accumulated into a single global sum**
- **Trick to promote scalar “sum” to an array indexed by the number of threads to create thread local copies of shared data**

Examines differences between explicitly specifying the OpenMP other methods

- **Final loop with explicit safe update of shared data**

```
#include <omp.h>
static long num_steps = 100000;
#define NUM 4
double step;
void main()
{               double pi, sum = 0.0;
                step = 1.0/(double) num_steps;
#pragma omp parallel num_threads(NUM)
                int i, ID;      double x, psum=0.0;
                ID = omp_get_thread_num();
                for (i=ID;i<=num_steps;i+=nthreads){
                                x=(i+0.5)*step;
                                psum+=4.0/(1.0+x*x);
                }
                #pragma omp critical
                                sum += psum
}
                pi = step * sum;
}
```

- **Along the way, example exposes concepts of private and shared data, false sharing, and other concepts. Refer to ParLab talk/slides.**

Another version of the same PI program uses common OpenMP constructs

- **Private clause for creating data local to a thread**
- **Reduction clause for managing data dependencies**

```
#include <omp.h>
static long num_steps = 100000; double step;
void main()
{       int i; double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;
#pragma omp parallel for private(i,x) reduction (+:sum)
        for (i=0;i<=num_steps;i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

- **Most people would write the OpenMP in this fashion, but the example serves to illustrate some of the steps behind the directives and the thought process for shared/private variables**

In order to do hybrid programming, models must be combined via Standards

- **Hybrid programming (two programming models) requires that the standards make commitments to each other on semantics.**
- **OpenMP's commitment: if a thread is blocked by an operating system call (e.g. file or network I/O), the other threads remain runnable.**
  - This is a major commitment; it involves the thread scheduler in the OpenMP compiler's runtime system and interaction with the OS.
  - What this means in the MPI context: An MPI call like MPI_Recv or MPI_Wait only blocks the calling thread.
- **MPI's commitments are more complex…**

The MPI Standard Defines 4 Levels of Thread Safety that affect Hybrid

- **Note that these are not specific to Hybrid OpenMP Models**
- **The are in the form of commitments that the multithreaded application makes to the MPI implementation**
  - MPI_THREAD_SINGLE: only one thread in the application
  - MPI_THREAD_FUNNELED: only one thread makes MPI calls, the Master Thread in the OpenMP context (next slide)
  - MPI_THREAD_SERIALIZED: Multiple threads make MPI calls, but only one at a time (not concurrently)
  - MPI_THREAD_MULTIPLE: Any thread may make MPI calls at any time, no restrictions
- **MPI-2 defines an alternative to MPI_Init**
  - MPI_Init_thread(requested, provided)
    - Allows applications to say what level it needs, and the MPI implementation to say what it provides

What This Means in the OpenMP Context

- **MPI_THREAD_SINGLE**
  - There is no OpenMP multithreading in the program.
- **MPI_THREAD_FUNNELED**
  - All of the MPI calls are made by the master thread i.e., all MPI calls are
    - Outside OpenMP parallel regions, or
    - Inside OpenMP master regions, or
    - Guarded by call to MPI_Is_thread_main MPI call.
      - (same thread that called MPI_Init_thread)
- **MPI_THREAD_SERIALIZED**

```
#pragma omp parallel
…
#pragma omp atomic
{
  …MPI calls allowed here…
}
```

- **MPI_THREAD_MULTIPLE**
  - Anything goes; any thread may make an MPI call at any time

Threads and MPI in MPI-2

- An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is not required to be thread safe in order to be standard-conforming
- A fully thread-compliant implementation will support MPI_THREAD_MULTIPLE
- A portable program that does not call MPI_Init_thread should assume that only MPI_THREAD_SINGLE is supported

For MPI_THREAD_MULTIPLE

- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- It is the user's responsibility to prevent races when threads in the same process post conflicting MPI calls
- User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads

The Current Situation

- **All MPI implementations support MPI_THREAD_SINGLE (duh).**
- **They probably support MPI_THREAD_FUNNELED even if they don't admit it.**
  - Does require thread-safe malloc
  - Probably OK in OpenMP programs
- **"Thread-safe" usually means MPI_THREAD_MULTIPLE.**
- **This is hard for MPI implementations that are sensitive to performance, like MPICH2.**
  - Lock granularity issue
  - Working on lock-free MPICH2 implementation
- **"Easy" OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED.**
  - So don't need "thread-safe" MPI for many hybrid programs
  - But watch out for Amdahl's Law!

How to determine thread support

```
MPI_Init_thread(&argc,&argv, MPI_THREAD_MULTIPLE,&provided);
Printf("Supports level %d of %d %d %d %d\n",
 provided,
MPI_THREAD_SINGLE,
MPI_THREAD_FUNNELED,
MPI_THREAD_SERIALIZED,
MPI_THREAD_MULTIPLE);
```

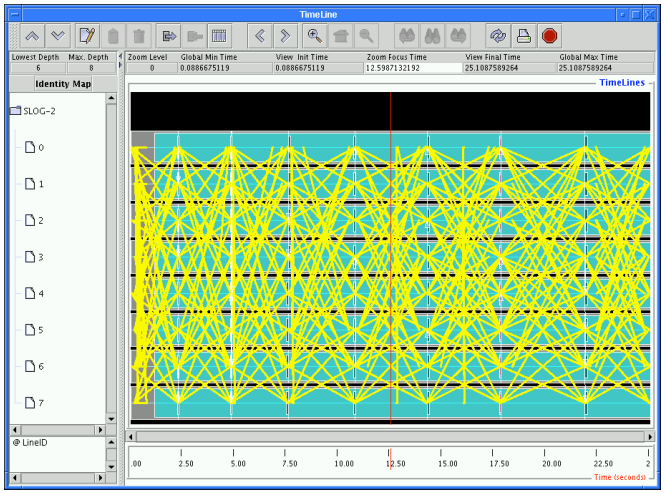
Example output:

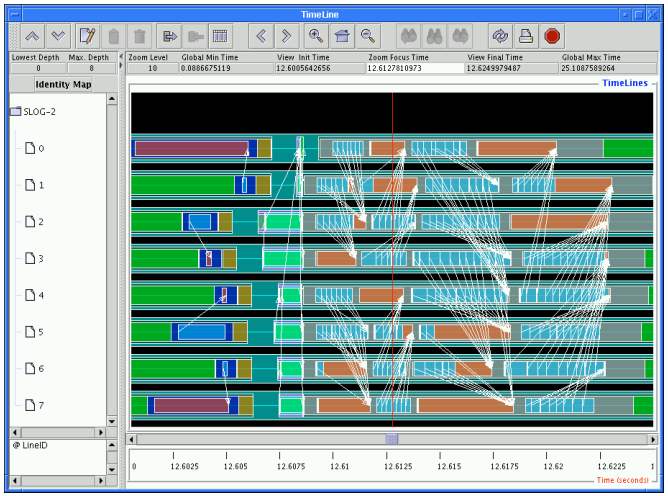
>Supports level 1 of 0 1 2 3

Support may vary depending on chosen compiler and MPI library.

Argonne has developed tools to Visualize the Behavior of Hybrid Programs

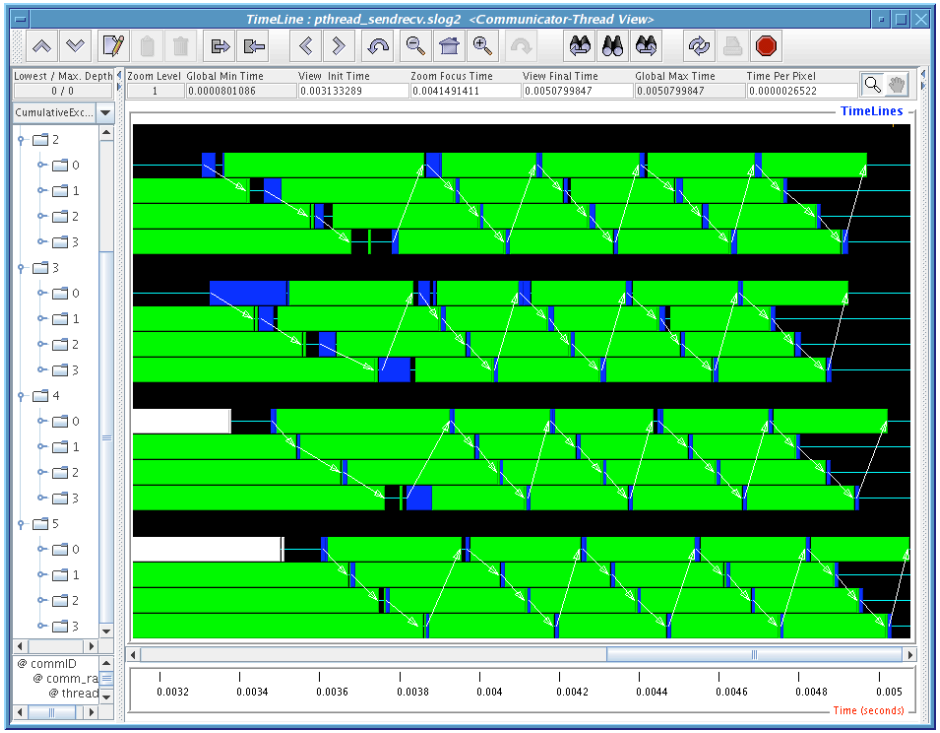
- **Jumpshot is a logfile-based parallel program visualizer of the "standard" type. Uses MPI profiling interface.**
- **Recently it has been augmented in two ways to improve scalability.**
  - Summary states and messages are shown as well as individual states and messages.
    - Provides a high-level view of a long run.
    - SLOG2 logfile structure allows fast interactive access (jumping, scrolling, and zooming) for large logfiles.





Jumpshot and Multithreading

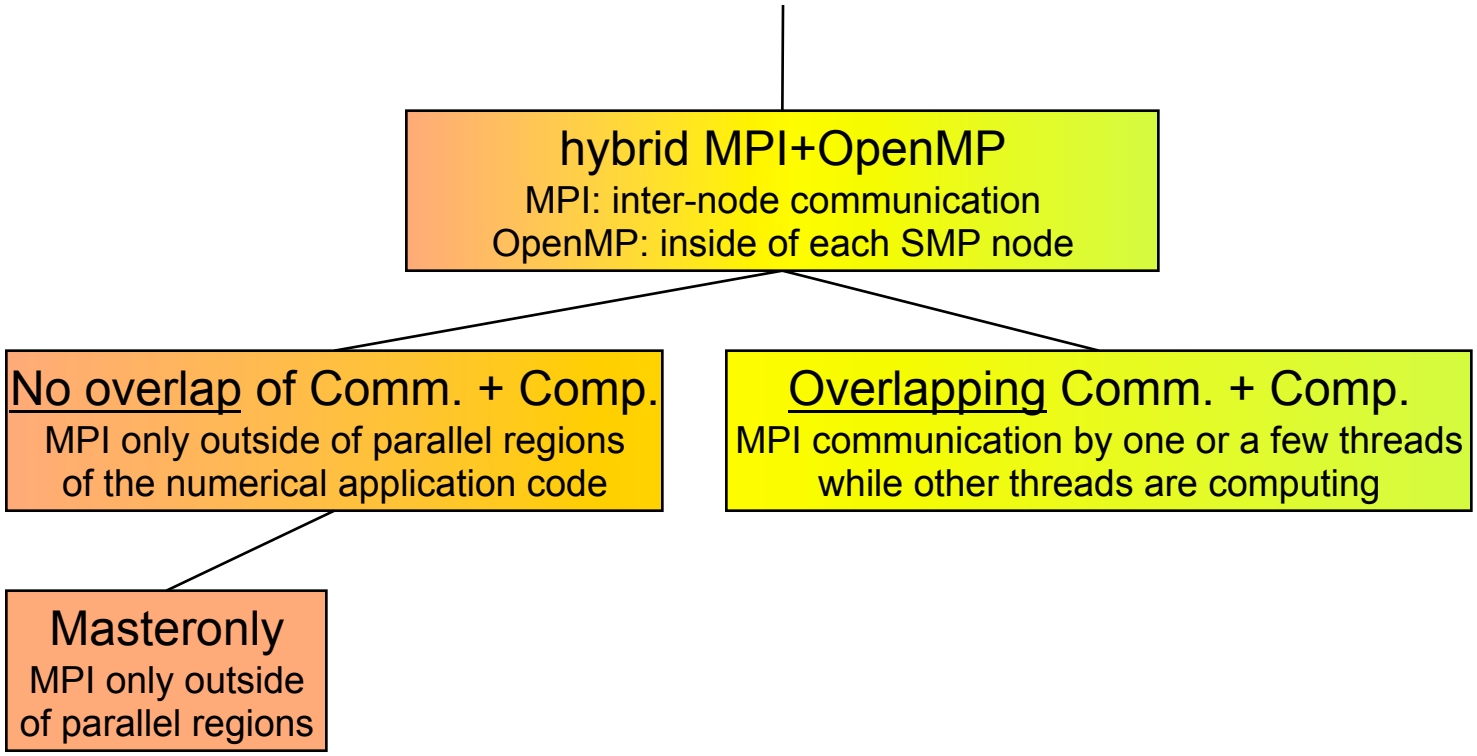
- **Newest additions are for multithreaded and hybrid programs that use pthreads.**
  - Separate timelines for each thread id
  - Support for grouping threads by communicator as well as by process



Using Jumpshot with Hybrid Programs

- **SLOG2/Jumpshot needs two properties of the OpenMP implementation that are not guaranteed by the OpenMP standard**
  - OpenMP threads must be pthreads
    - Otherwise, the locking in the logging library necessary to preserve exclusive access to the logging buffers would need to be modified.
  - These pthread ids must be reused (threads are “parked” when not in use)
    - Otherwise Jumpshot would need zillions of time lines.
- **At NERSC, we are currently examining the use of Jumpshot and other tools for the analysis of hybrid programs**
  - Part of the Cray Center of Excellence (COE) Program

Rabensiefners Models for Hybrid



Hybrid Masteronly

Masteronly
MPI only outside of parallel regions

Advantages
- No message passing inside of the SMP nodes
- No topology problem

```
for (iteration ….)
{
  #pragma omp parallel
    numerical code
  /*end omp parallel */

  /* on master thread only */
    MPI_Send (original data
      to halo areas
      in other SMP nodes)
    MPI_Recv (halo data
      from the neighbors)
} /*end for loop
```

Major Problems

- All other threads are sleeping while master thread communicates!
- What is inter-node bandwidth?
- MPI-lib must support at least MPI_THREAD_FUNNELED

Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

```
if (my_thread_rank < …) {

   MPI_Send/Recv….
     i.e., communicate all halo data

} else {

   Execute those parts of the application
     that do not need halo data
     (on non-communicating threads)
}


Execute those parts of the application
  that  need halo data
  (on all threads)
```

This can get very complicated.
Looks a little bit more like MPI Programming.

Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

Three problems:

- **the application problem:**
  - one must separate application into:
    - code that can run before the halo data is received
    - code that needs halo data
  - ➔ **very hard to do !!!**

- **the thread-rank problem:**
  - comm. / comp. via thread-rank
  - cannot use work-sharing directives
  - ➔ **loss of major OpenMP support (see next slide)**

- **the load balancing problem**

```
if (my_thread_rank < 1) {
   MPI_Send/Recv….
} else {
   my_range = (high-low-1) / (num_threads-1) + 1;
   my_low = low + (my_thread_rank+1)*my_range;
   my_high=high+ (my_thread_rank+1+1)*my_range;
   my_high = max(high, my_high)
   for (i=my_low; i<my_high; i++) {
            ….
   }
}
```

Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

Subteams

- **Important proposal for OpenMP 3.x or OpenMP 4.x**

Barbara Chapman et al.: Toward Enhancing OpenMP's Work-Sharing Directives. In proceedings, W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 645-654, 2006.

```
#pragma omp parallel
{
#pragma omp single onthreads( 0 )
  {
    MPI_Send/Recv….
  }
#pragma omp for onthreads( 1 : omp_get_numthreads()-1 )
    for (……..)
    { /* work without halo information */
    }  /* barrier at the end is only inside of the subteam */
   …
#pragma omp barrier
#pragma omp for
    for (……..)
    { /* work based on halo information */
    }
} /*end omp parallel */
```

Multi-zone NAS Parallel Benchmarks – Characteristics

- **Aggregate sizes and zones:**
  - Class B: 304 x 208 x 17 grid points, 64 zones
  - Class C: 480 x 320 x 28 grid points, 256 zones
  - Class D: 1632 x 1216 x 34 grid points, 1024 zones
  - Class E: 4224 x 3456 x 92 grid points, 4096 zones
- BT-MZ:
  Block tridiagonal simulated CFD application
  - Size of the zones varies widely:
    - large/small about 20
    - requires multi-level parallelism to achieve a good load-balance
- SP-MZ:
  Scalar Pentadiagonal simulated CFD application
  - Size of zones identical
    - no load-balancing required

Expectations:

Pure MPI: Load-balancing problems!

Good candidate for MPI+OpenMP

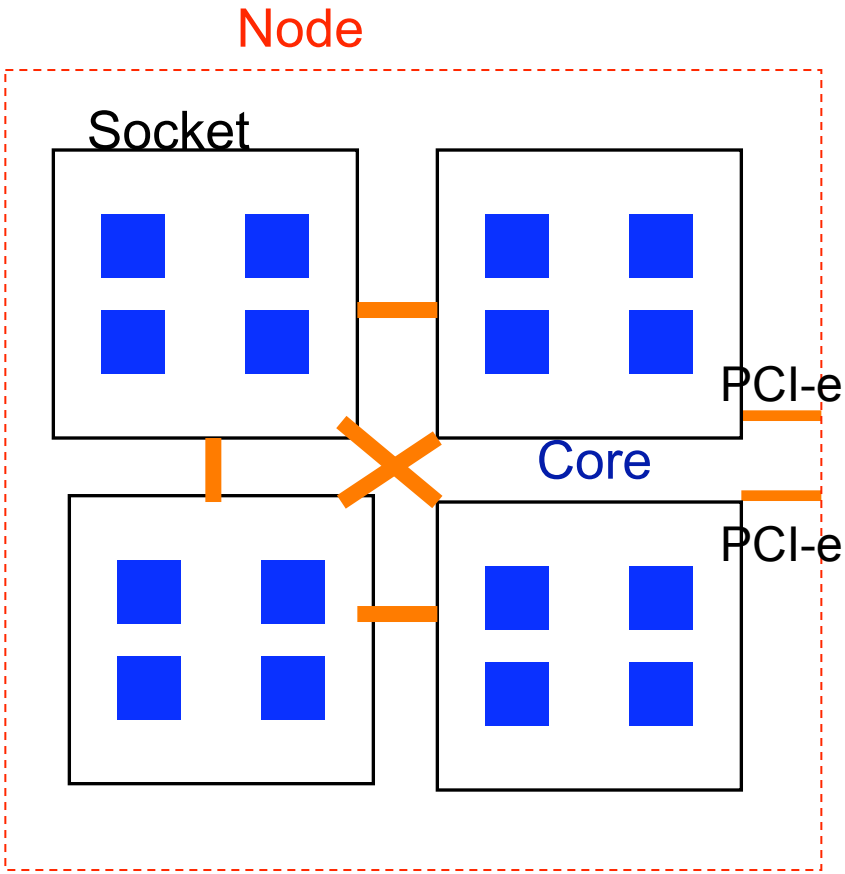
Load-balanced on MPI level: Pure MPI should perform best

Sun Constellation Cluster Ranger (1)

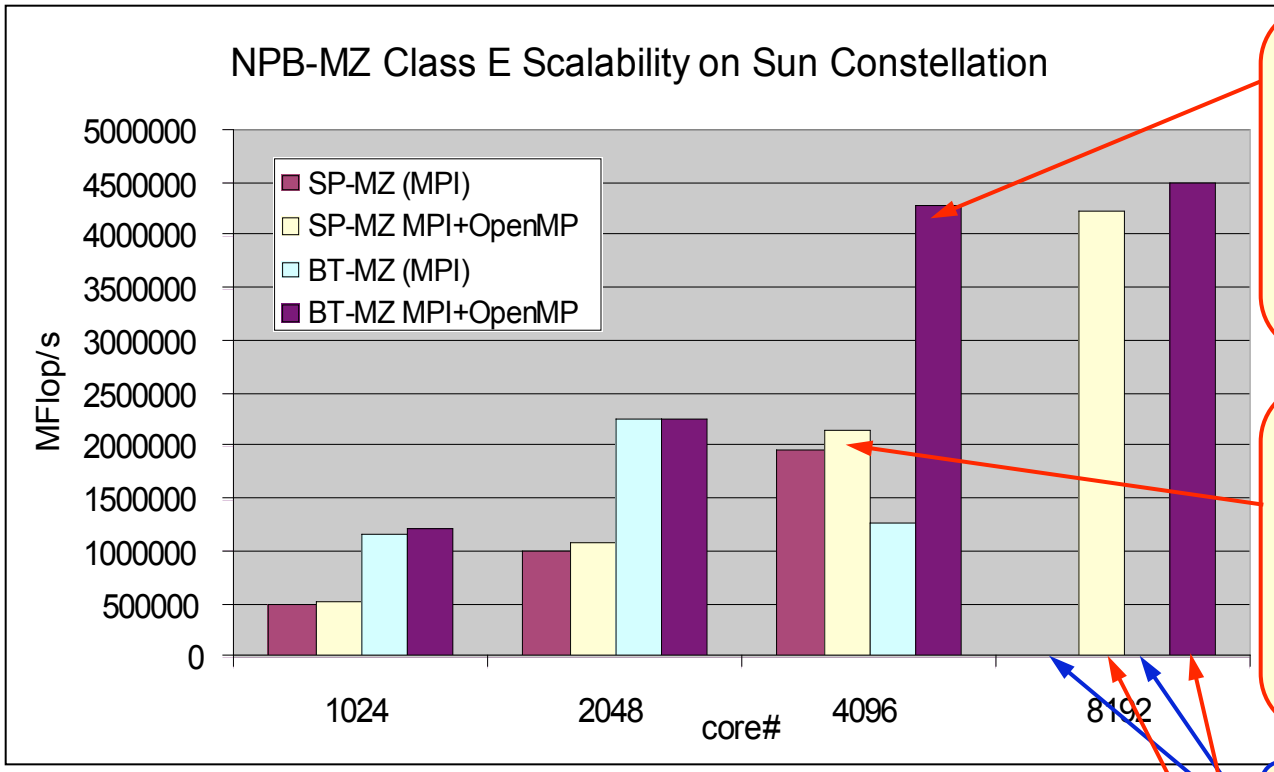
- Located at the Texas Advanced Computing Center (TACC), University of Texas at Austin (http://www.tacc.utexas.edu)
- 3936 Sun Blades, 4 AMD Quad-core 64bit 2.3GHz processors per node (blade), 62976 cores total
- 123TB aggregrate memory
- Peak Performance 579 Tflops
- InfiniBand Switch interconnect
- Sun Blade x6420 Compute Node:
  - 4 Sockets per node
  - 4 cores per socket
  - HyperTransport System Bus
  - 32GB memory

Sun Constellation Cluster Ranger (2)

- **Compilation:**
  - PGI pgf90 7.1
  - mpif90 -tp barcelona-64 -r8
- **Cache optimized benchmarks Execution:**
  - MPI MVAPICH
  - setenv OMP_NUM_THREAD NTHREAD
  - Ibrun numactl bt-mz.exe
- **numactl** controls
  - Socket affinity: select sockets to run
  - Core affinity: select cores within socket
  - Memory policy: where to allocate memory
  - http://www.halobates.de/numaapi3.pdf



NPB-MZ Class E Scalability on Ranger



BT

Significant improvement (235%):

Load-balancing issues solved with MPI+OpenMP

SP

Pure MPI is already load-balanced.

But hybrid programming 9.6% faster

Cannot be build for 8192 processes!

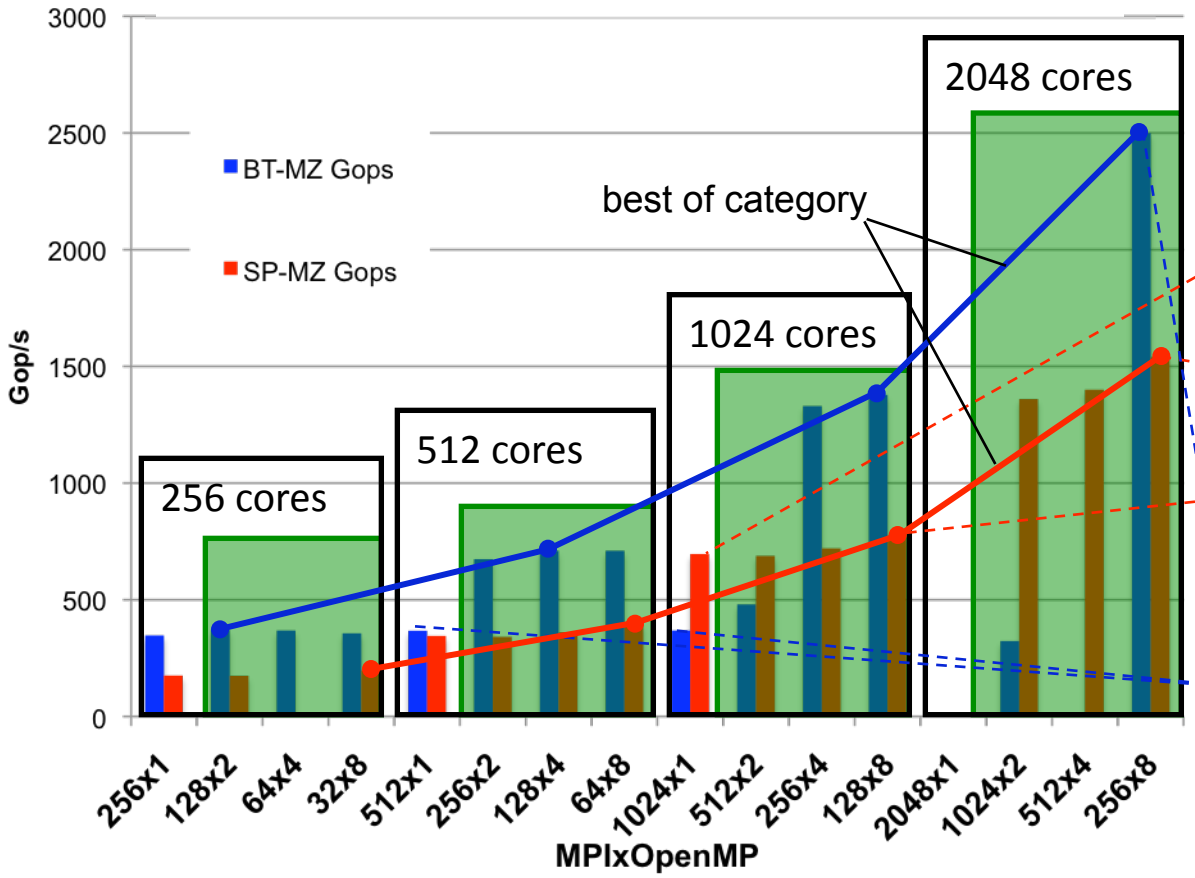
Hybrid:

SP: still scales

BT: does not scale

- **Scalability in Mflops**
- **MPI/OpenMP outperforms pure MPI**
- **Use of numactl essential to achieve scalability**

Cray XT5: NPB-MZ Class D Scalability



Results reported for Class D on 256-2048 cores

Expected: #MPI processes limited

- SP-MZ pure MPI scales up to 1024 cores
- SP-MZ MPI/OpenMP scales to 2048 cores
- SP-MZ MPI/OpenMP outperforms pure MPI for 1024 cores

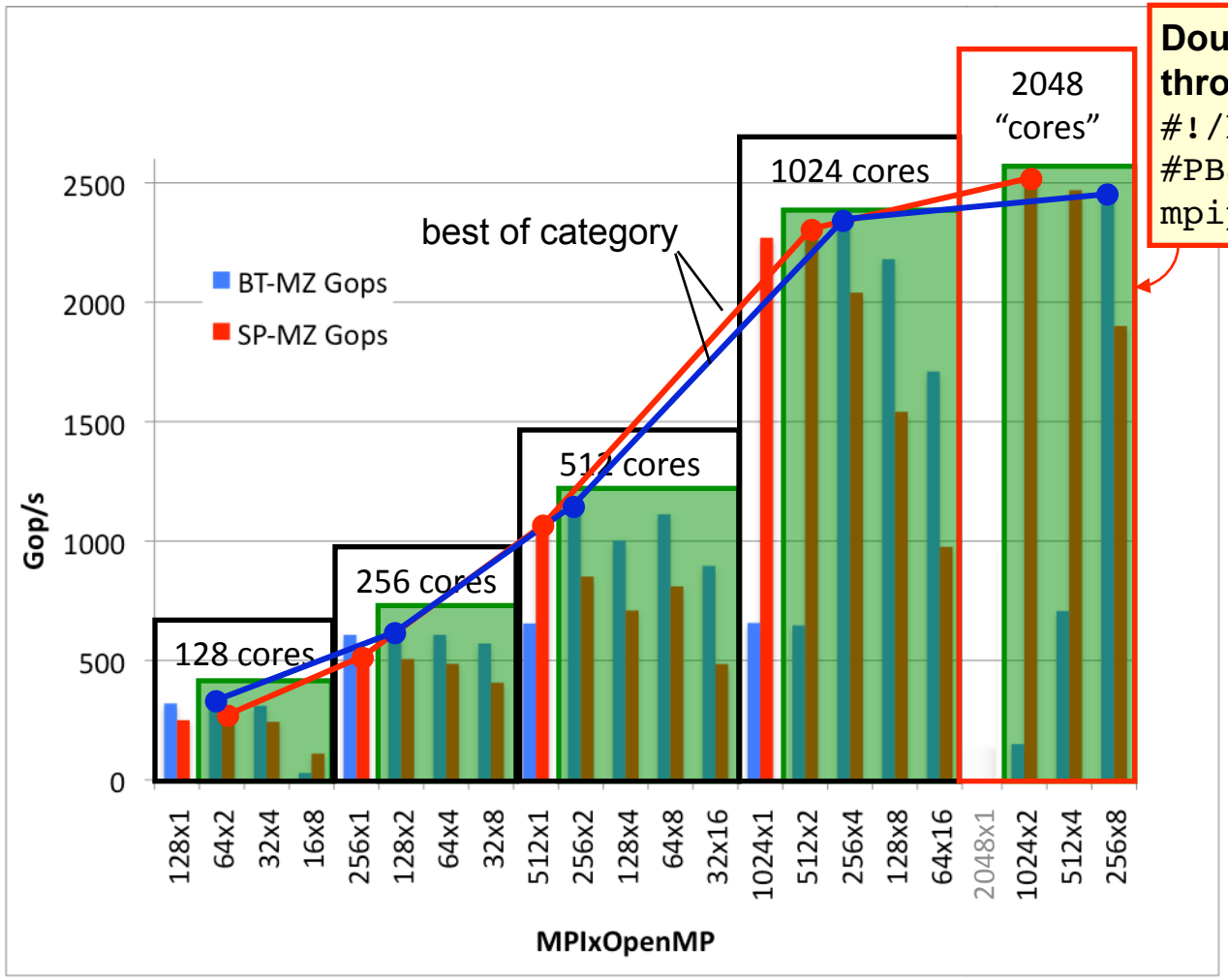
Unexpected!

- BT-MZ MPI does not scale
- BT-MZ MPI/OpenMP scales to 2048 cores, outperforms pure MPI

Expected: Load-Imbalance for pure MPI

Courtesy of Gabriele Jost (TACC/NPS)

NPB-MZ Class D on IBM Power 6: Exploiting SMT for 2048 Core Results



Doubling the number of threads through hyperthreading (SMT):

```
#!/bin/csh
#PBS -l select=32:ncpus=64:
mpiprocs=NP:ompthreads=NT
```

- Results for 128-2048 cores
- Only 1024 cores were available for the experiments
- BT-MZ and SP-MZ show benefit from **Simultaneous Multithreading** (**SMT**): 2048 threads on 1024 cores

Courtesy of Gabriele Jost (TACC/NPS)

Hybrid parallelization affords new opportunities

- Nested Parallelism
- Load-Balancing
- Memory consumption
- Opportunities, if MPI speedup is limited due to "*algorithmic*" problem
- MPI scaling problems

Load-Balancing (on same or different level of parallelism)

- **OpenMP enables**
  - Cheap ***dynamic*** and ***guided*** load-balancing
  - Just a parallelization option (clause on omp for / do directive)
  - Without additional software effort
  - Without explicit data movement
- **On MPI level**
  - **Dynamic load balancing** requires moving of parts of the data structure through the network
  - Significant runtime overhead
  - Complicated software / therefore not implemented
- MPI & OpenMP
  - Simple static load-balancing on MPI level, dynamic or guided on OpenMP level } **medium quality cheap implementation**

Memory consumption

- **Shared nothing**
  - Heroic theory
  - In practice: Some data is duplicated

- MPI & OpenMP
  With n threads per MPI process:
  - Duplicated data is reduced by factor n

Memory consumption (continued)

- **Future:**
  With 100+ cores per chip the memory per core is limited.
  - Data reduction through usage of shared memory may be a key issue
  - Domain decomposition on each hardware level
    - **Maximizes**
      - Data locality
      - Cache reuse
    - **Minimizes**
      - CCnuma accesses
      - Message passing
  - No halos between domains inside of SMP node
    - **Minimizes**
      - Memory consumption

How many multi-threaded MPI processes per SMP node

- SMP node = with **m sockets** and **n cores/socket**
- How many threads (i.e., cores) per MPI process?
  - Too many threads per MPI process
    → overlapping of MPI and computation may be necessary,
    → some NICs unused?
  - Too few threads
    → too much memory consumption (see previous slides)
- Optimum
  - somewhere between 1 and m x n

Opportunities, if MPI speedup is limited due to "*algorithmic*" problems

- **Algorithmic opportunities due to larger physical domains inside of each MPI process**
  - → If multigrid algorithm only inside of MPI processes
  - → If separate preconditioning inside of MPI nodes and between MPI nodes
  - → If MPI domain decomposition is based on physical zones

To overcome MPI scaling problems

- **Reduced number of MPI messages, reduced aggregated message size** } compared to pure MPI
- **MPI has a few scaling problems**
  - Handling of more than 10,000 processes
  - Irregular Collectives: MPI_....v(), e.g. MPI_Gatherv()
    - **Scaling applications should not use MPI_....v() routines**
  - MPI-2.1 Graph topology (MPI_Graph_create)
    - **MPI-2.2 MPI_Dist_graph_create_adjacent**
  - Creation of sub-communicators with MPI_Comm_create
    - **MPI-2.2 introduces a new scaling meaning of MPI_Comm_create**
- **Hybrid programming reduces all these problems (due to a smaller number of processes)**

Summary of Hybrid Programming

MPI + OpenMP

- **Significant opportunity → higher performance on fixed number of cores**
- **NPB-MZ examples**
  - BT-MZ → strong improvement (as expected)
  - SP-MZ → small improvement (none was expected)
- **Often however, no speedup is obtained, especially for naïve implementations**
- **Hybrid MPI + OpenMP can solve certain problems with MPI-Everywhere**
  - Load balancing
  - Memory consumption
  - Two levels of parallelism
    - Outer → distributed memory → halo data transfer → MPI
    - Inner → shared memory → ease of SMP parallelization → OpenMP

Summary (cont)

MPI+OpenMP – Pitfalls:

- **Problems with OpenMP performance remain**
  - On ccNUMA → e.g., first touch
  - Pinning of threads on cores
- **Problems with combination of MPI & OpenMP**
  - topology and mapping problems
  - mismatch problems
- **Most hybrid programs → Masteronly style**
- **Overlapping communication and computation with several threads**
  - Requires thread-safety quality of MPI library
  - Loss of OpenMP support → future OpenMP subteam concept

Hybrid Programming: more information

"***Hybrid MPI and OpenMP Parallel Programming***"
SC09 half-day tutorial M09, Monday, 8:30am – 12:00pm

Reminder, these slides are excerpted various tutorials given by
Rolf Rabenseifner, Gabriele Jost, Rusty Lusk, Alice Koniges, et al.
including
SC08, SC09 (upcoming) ParCFD 2009, SciDAC 2009
We are grateful for the use of these slides at the NUG User Group Meeting