Homework #5 Hints

- Smoothing in 2-dimensions
  - 2-d array with ghostcells on all sides
  - Assume dx = dy
  - You can use my 1-d code as a starting point if you wish

Parallel Computing

- Individual processors themselves are not necessarily getting much faster on their own (the GHz-wars are over)
  - Chips are packing more processing cores into the same package
  - Even your phone is likely a multicore chip
- If you don't use the other cores, then they are just “space heaters”
- Some techniques for parallelism require only simple modifications of your codes and can provide great gains on the single workstation
- There are lots of references online
  - Great book: High Performance Computing by Dowd and Severance—freely available (linked to from our webpage).
  - We'll use this for some background

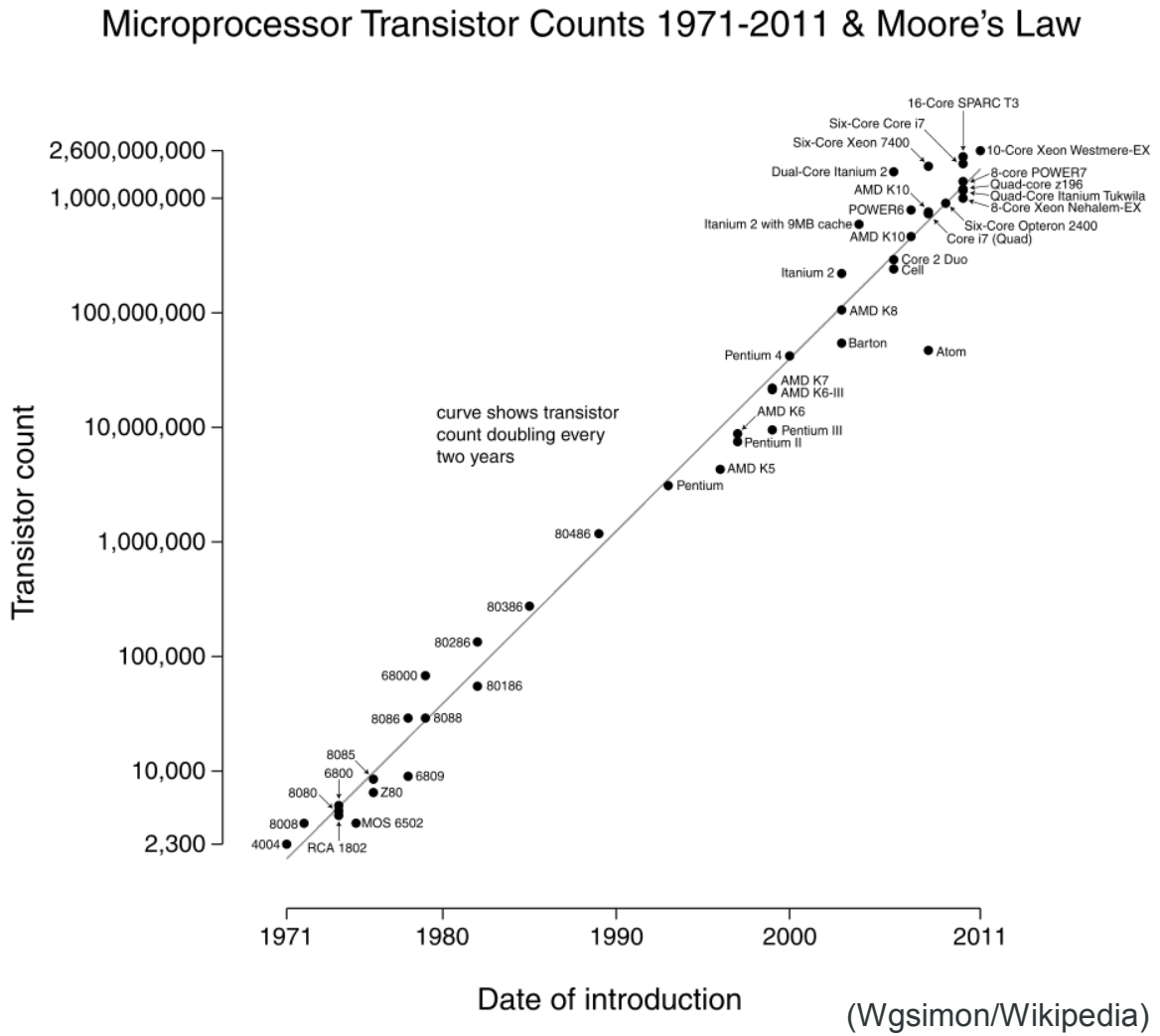
Types of Machines

- Modern computers have multiple cores that all access the same pool of memory directly—this is a shared-memory architecture
- Supercomputers are built by connecting LOTS of nodes (each a shared memory machine with ~4-32 cores) together with a high-speed network—this is a distributed-memory architecture
- Different parallel techniques and libraries are used for each of these paradigms:
  - Shared-memory: OpenMP
  - Distributed-memory: message-passing interface (MPI)

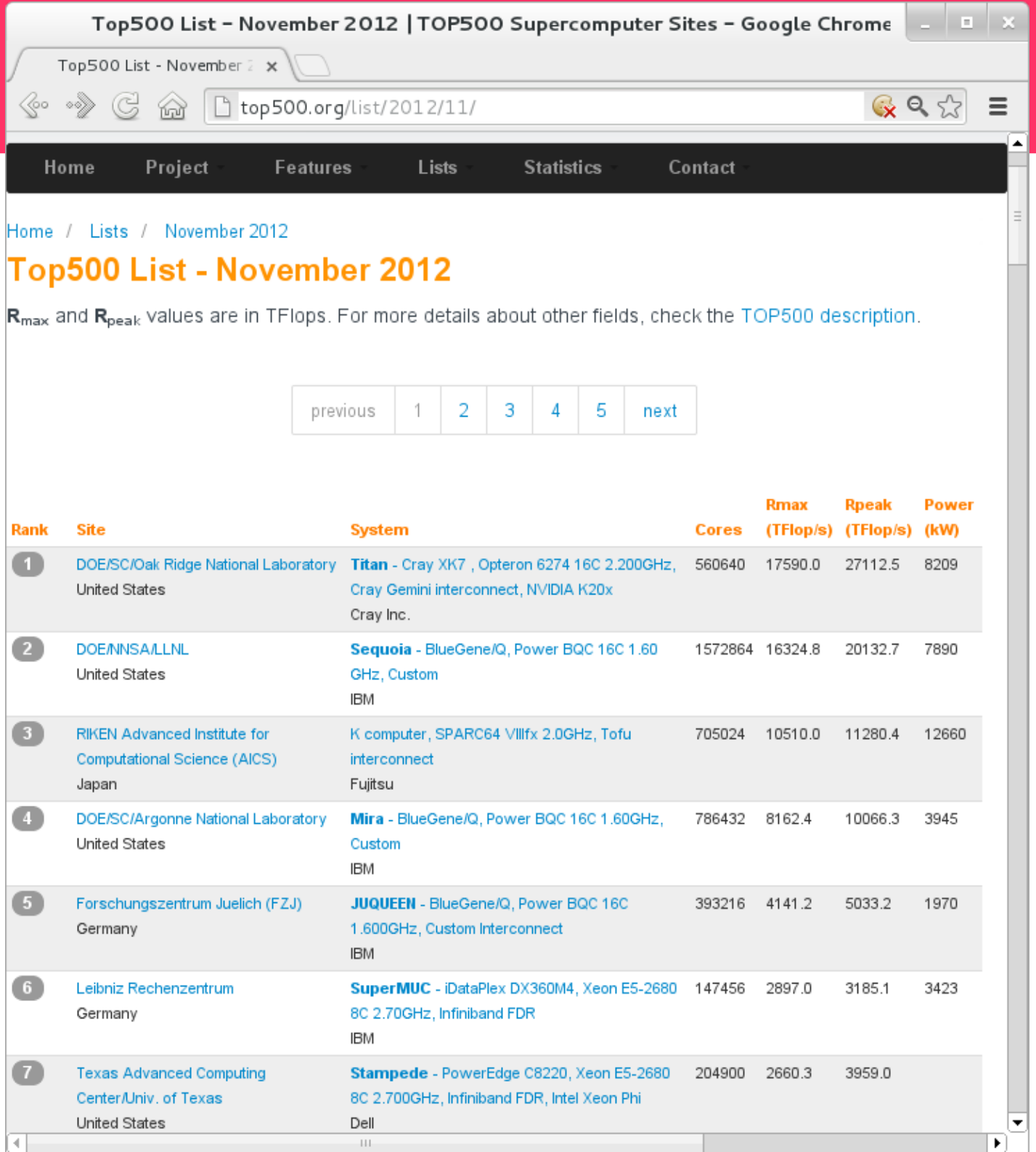
Moore's Law

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year... Certainly over the short term this rate can be expected to continue, if not to increase."

—Gordon Moore, Electronics Magazine, 1965



(Wgsimon/Wikipedia)

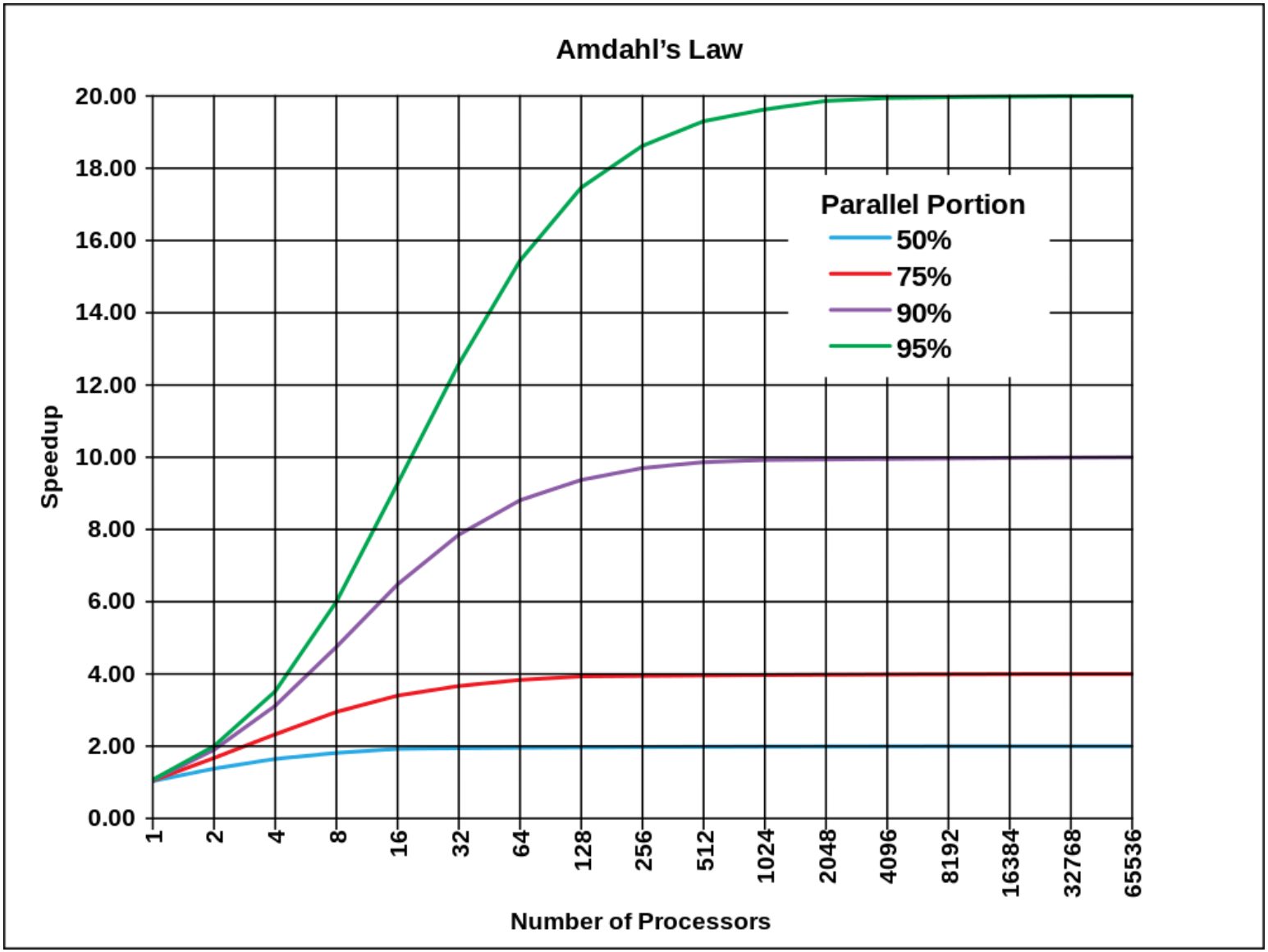

Top500 List – November 2012 | TOP500 Supercomputer Sites – Google Chrome
Top500 List - November
top500.org/list/2012/11/
Home
Project
Features
Lists
Statistics
Contact
Home / Lists / November 2012
Top500 List - November 2012
Rmax and Rpeak values are in TFlops. For more details about other fields, check the TOP500 description.
previous
1
2
3
4
5
next
Rank
Site
System
Cores
Rmax (TFlop/s)
Rpeak (TFlop/s)
Power (kW)
1
DOE/SC/Oak Ridge National Laboratory
United States
Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x
Cray Inc.
560640
17590.0
27112.5
8209
2
DOE/NNSA/LLNL
United States
Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom
IBM
1572864
16324.8
20132.7
7890
3
RIKEN Advanced Institute for Computational Science (AICS)
Japan
K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect
Fujitsu
705024
10510.0
11280.4
12660
4
DOE/SC/Argonne National Laboratory
United States
Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom
IBM
786432
8162.4
10066.3
3945
5
Forschungszentrum Juelich (FZJ)
Germany
JUQUEEN - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect
IBM
393216
4141.2
5033.2
1970
6
Leibniz Rechenzentrum
Germany
SuperMUC - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR
IBM
147456
2897.0
3185.1
3423
7
Texas Advanced Computing Center/Univ. of Texas
United States
Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi
Dell
204900
2660.3
3959.0

Amdahl's Law

- In a typical program, you will have sections of code that adapt easily to parallelism, and stuff that remains serial
  - For instance: initialization may be serial and the resulting computation parallel
- Amdahl's law: speedup attained from increasing the number of processors, N, given the fraction of the code that is parallel, P:

$$S = \frac{1}{(1-P) + (P/N)}$$

Amdahl's Law



(Daniels220 at English Wikipedia)

Amdahl's Law

- This seems to argue that we'd never be able to use 100,000s of processors
- However (Dowd & Severance):
  - New algorithms have been designed to exploit massive parallelism
  - Larger computers mean bigger problems are possible—as you increase the problem size, the fraction of the code that is serial likely descreases

Types of Parallelism
(Wikipedia)

- Flynn's taxonomy classifies computer architectures
- 4 classifications: single/multiple data; single/multiple instruction
  - Single instruction, single data (SISD)
    - Think typical application on your computer—no parallelism
  - Single instruction, multiple data (SIMD)
    - The same instruction set is done to multiple pieces of data all at once
    - Old days: vector computers; today: GPUs
  - Multiple instructions, single data (MISD)
    - Not very interesting...
  - Multiple instructions, multiple data (MIMD)
    - What we typically think of as parallel computing. The machines on the top 500 list fall into this category

Types of Parallelism
(Wikipedia)

- We can do MIMD different ways:
  - Single program, multiple data
    - This is what we normally do. MPI allows this
    - Differs from SIMD in that general CPUs can be used, doesn't require direct synchronization for all tasks

Trivially Parallel

- Sometimes our tasks are trivially parallel
  - No communication is needed between processes
- Ex: ray tracing or Monte Carlo
  - Each realization can do its work independently
  - At the end, maybe, we need to do some simple processing of all the results
- Large data analysis
  - You have a bunch of datasets and a reduction pipeline to work on them.
  - Use multiple processors to work on the different data files as resources become available.
  - Each file is processed on a single core

Trivially Parallel via Shell Script

- Ex: data analysis—launch independent jobs
- This can be done via a shell script—no libraries necessary
  - Loop over files
    - Run jobs until all of the processors are full
    - Use lockfiles to indicate a job is running
    - When resources become free, start up the next job
- Let's look at the code...

How Do We Make Our Code Parallel?

- Despite your best wishes, there is no simple compiler flag "--make-this-parallel"
  - You need to understand your algorithm and determine what parts are amenable to parallelism
- However... if the bulk of your work is on one specific piece (say, solving a linear system), you may get all that you need by using a library that is already parallel
  - This will require minimal changes to your code

Shared Memory vs. Distributed

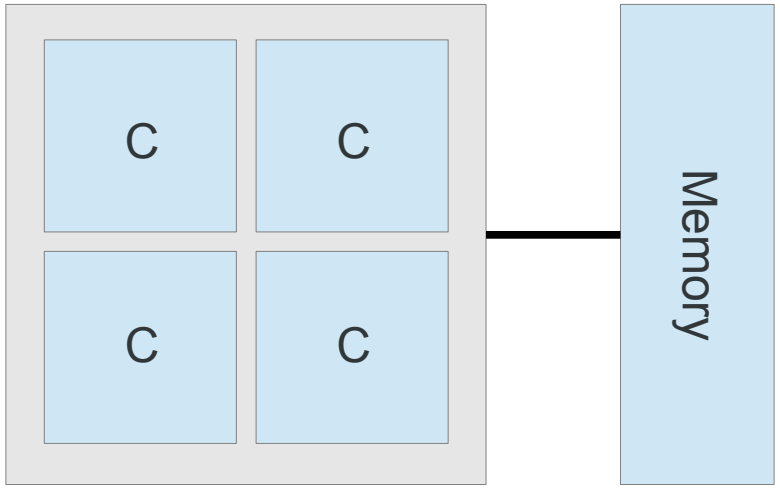
- Imagine that you have a single problem to solve and you want to divide the work on that problem across available processors
- If all the core see the same pool of memory (shared-memory), then parallelism is straightforward
  - Allocate a single big array for your problem
  - Spawn threads: separate instance of a sequence of instructions operating
    - Multiple threads operate simultaneously
  - Each core/thread operates on a smaller portion of the same array, writing to the same memory
  - Some intermediate variables may need to be duplicated on each thread—thread-private data
  - OpenMP is the standard here

Shared Memory vs. Distributed

- Distributed computing: running on a collection of separate computers (CPU + memory, etc.) connected by a high-speed network
  - Each task cannot directly see the memory for the other tasks
  - Need to explicitly send messages from one machine to another over the network exchanging the needed data
  - MPI is the standard here

Shared Memory

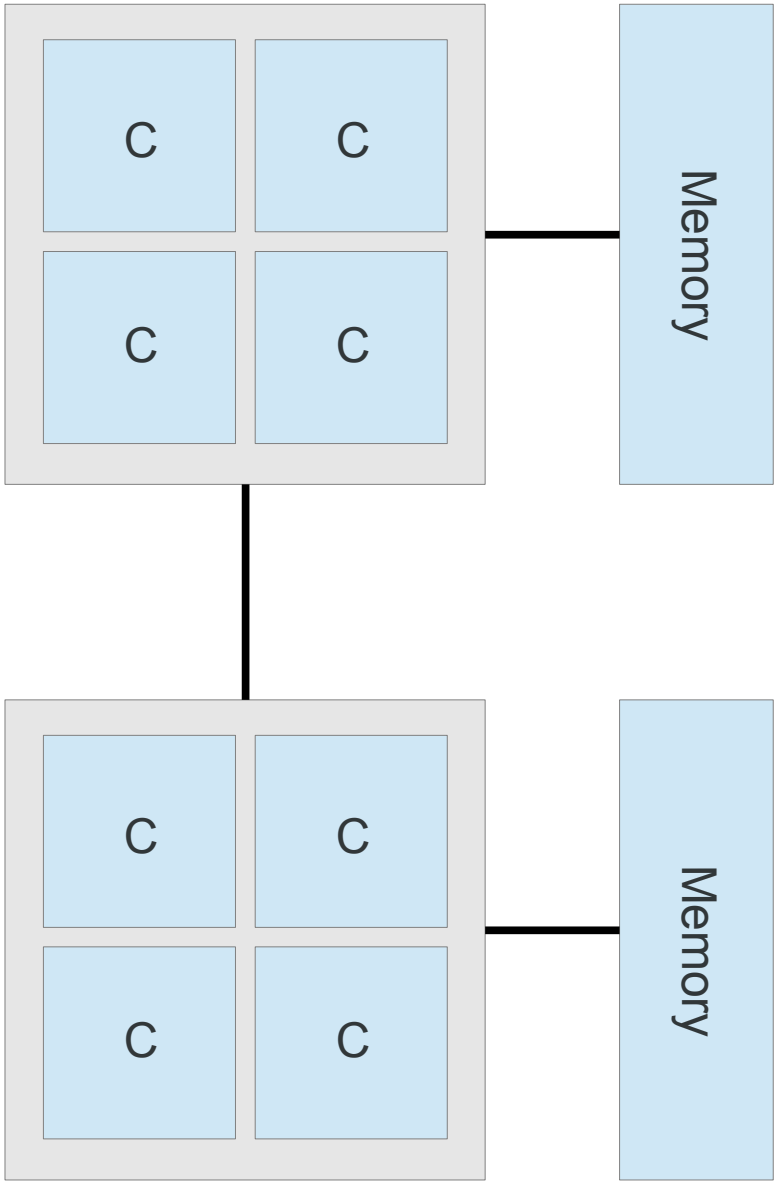
- Nodes consist of one or more chips each with many cores (2-16 typically)
  - Everything can access the same pool of memory



Single 4-core chip and its pool of memory

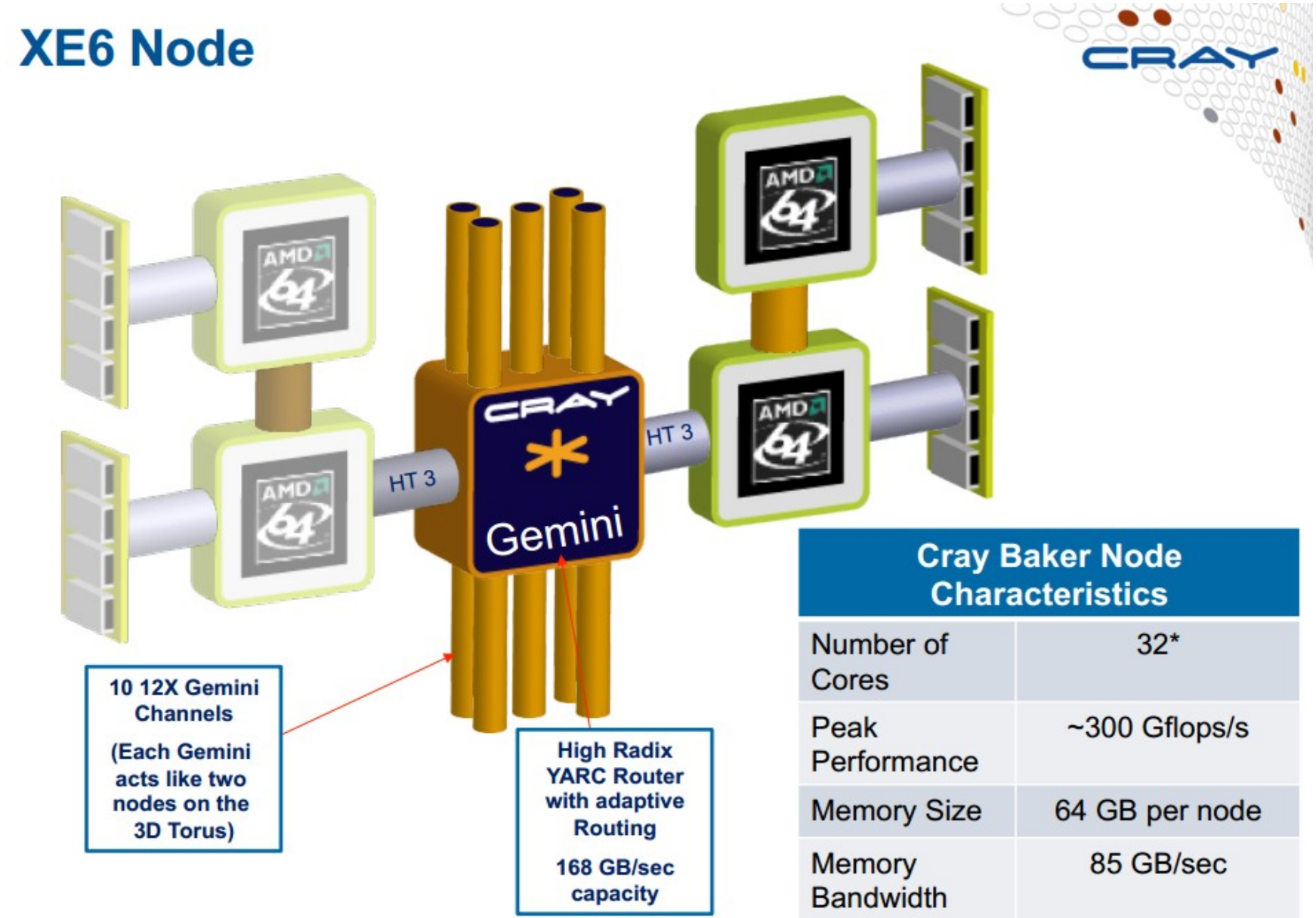
Shared Memory

- Some machines are more complex—multiple chips each with their own pool of local memory can talk to on another on the node
  - Latency may be higher when going “off-chip”
- Best performance will require knowning your machine's architecture



Two 4-core chips comprising a single node—each has their own pool of memory

Ex: Blue Waters Machine

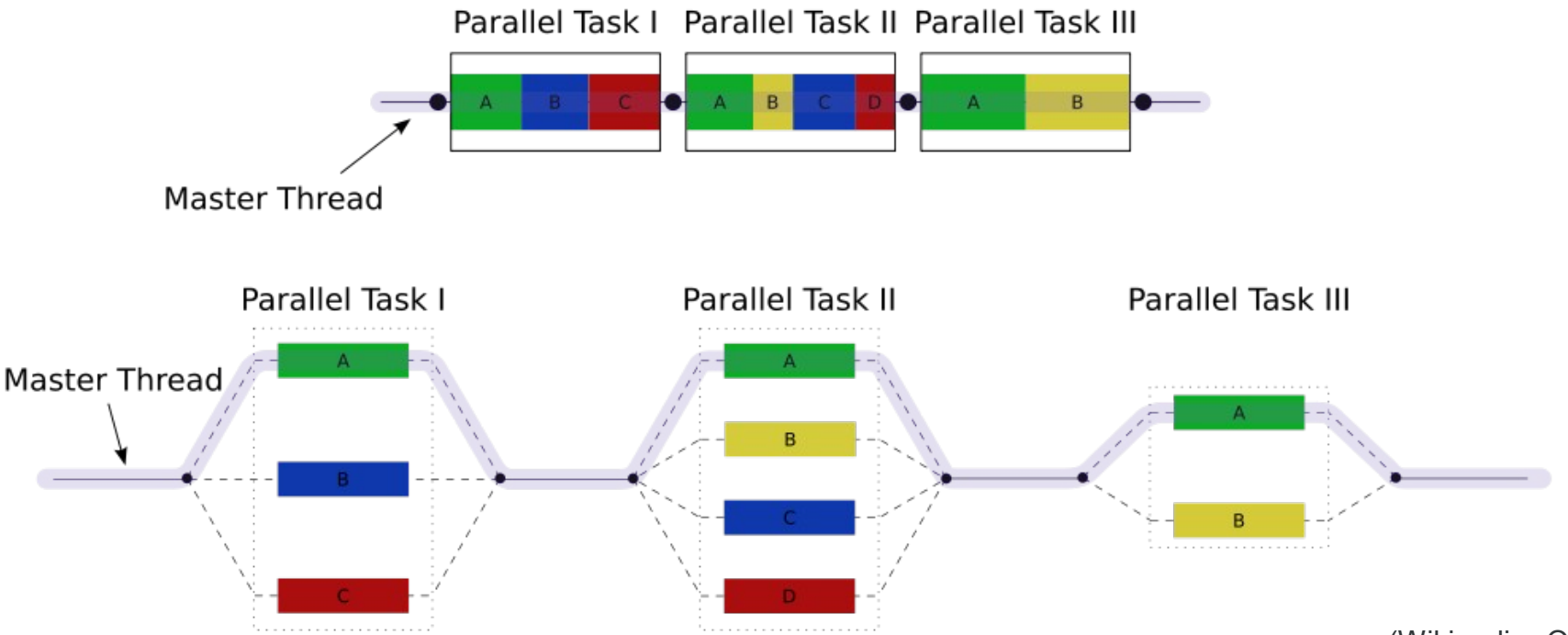


Cray Baker Node Characteristics	
Number of Cores	32*
Peak Performance	~300 Gflops/s
Memory Size	64 GB per node
Memory Bandwidth	85 GB/sec

(Cray, Inc.)

Open MP

- Threads are spawned as needed
- When you run the program, there is one thread—the master thread
  - When you enter a parallel region, multiple threads run concurrently



(Wikipedia--OpenMP)

OpenMP “Hello World”

- OpenMP is done via directives or pragmas
  - Look like comments unless you tell the compiler to interpret them
  - Environment variable `OMP_NUM_THREADS` sets the number of threads
  - Support for C, C++, and Fortran
- Hello world:

```
program hello

  !$OMP parallel
  print *, "Hello world"
  !$OMP end parallel

end program hello
```

  - Compile with : `gfortran -o hello -fopenmp hello.f90`

C Hello World

- In C, the preprocessor is used for the pragmas

```
#include <stdio.h>

void main() {
  #pragma omp parallel
  printf("Hello world\n");
}
```

OMP Functions

- In addition to using pragmas, there are a few functions that OpenMP provides to get the number of threads, the current thread, etc.

```
program hello

  use omp_lib

  print *, "outside parallel region, num threads = ", &
      omp_get_num_threads()

  !$OMP parallel
  print *, "Hello world", omp_get_thread_num()
  !$OMP end parallel

end program hello
```

OpenMP

- Most modern compilers support OpenMP
  - However, the performance across them can vary greatly
  - GCC does a reasonable job. Intel is the fastest
- There is an overhead associated with spawning threads
  - You may need to experiment
  - Some regions of your code may not have enough work to offset the overhead

Number of Threads

- There will be a systemwide default for `OMP_NUM_THREADS`
- Things will still run if you use more threads than cores available on your machine—but don't!
- Scaling: if you double the number of cores does the code take 1/2 the time?

Parallel Loops

- Splitting loops across cores
- Ex: matrix multiplication:

```
program matmul

  ! matrix multiply

  integer, parameter :: N = 1000

  double precision a(N,N)
  double precision x(N)
  double precision b(N)

  integer :: i, j

  ! initialize the matrix and vector

  !$omp parallel do private(i, j)
  do j = 1, N
     do i = 1, N
        a(i,j) = dble(i + j)
     enddo
  enddo
  !$omp end parallel do
```

Parallel Loops

Continued...

```
  do i = 1, N
     x(i) = i
  enddo

  ! multiply

  !$omp parallel do private(i, j)
  do i = 1, N
     b(i) = 0.0
     do j = 1, N
        b(i) = b(i) + a(i,j)*x(j)
     enddo
  enddo
  !$end parallel do

end program matmul
```

Loop Parallel

- We want to parallelize all loops possible
  - Instead of `f(:,:) = 0.d0`, we write out loops and thread
- Private data
  - Inside the loop, all threads will have access to all the variables declared in the main program
  - For some things, we will want a private copy on each thread. These are put in the `private()` clause

Reduction

- Suppose you are finding the minimum value of something, or summing
  - Loop spread across threads
  - How do we get the data from each thread back to a single variable that all threads see?
- `reduction()` clause
  - Has both shared and private behaviors
  - Compiler ensures that the data is synchronized at the end

Reduction

- Example of a reduction

```
program reduce

  implicit none

  integer :: i

  double precision :: sum

  sum = 0.0d0

!$omp parallel do private (i) reduction(+:sum)
  do i = 1, 10000
     sum = sum + exp((mod(dble(i), 5.0d0) - 2*mod(dble(i),7.0d0)))
  end do
!$omp end parallel do

  print *, sum

end program reduce
```

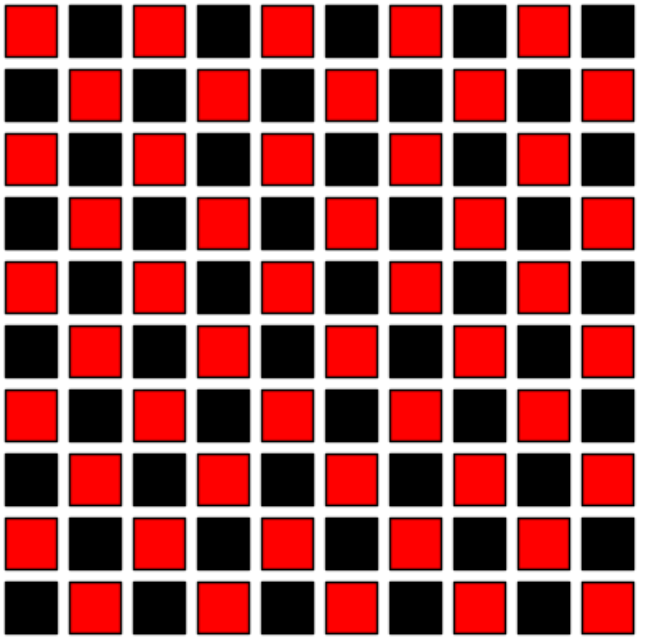
Do we get the same answer when run with differing number of threads?

Example: Relaxation

- In two-dimensions, with $\Delta x = \Delta y$, we have:

$$\phi_{i,j} = \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - \Delta x^2 f_{i,j}\right)$$

  - Red-black Gauss-Seidel:
    - Update in-place
    - First update the red cells (black cells are unchanged)
    - Then update black cells (red cells are unchanged)



Example Relaxation

- Let's look at the code
- All two-dimensional loops are wrapped with OpenMP directives
- We can measure the performance
  - Fortran 95 has a `cpu_time()` intrinsic
    - Be careful though—it returns the CPU time summed across all threads
  - OpenMP has the `omp_get_wtime()` function
    - This returns wallclock time
  - Looking at wallclock: if we double the number of processors, we want the code to take 1/2 the wallclock time

Example Relaxation

- Performance:

This is an example of a strong scaling test—the amount of work is held fixed as the number of cores is increased

256x256 bender (-Ofast)

threads	wallclock time (2 runs)	
1	0.5014	0.4960
2	0.2809	0.2873
4	0.1683	0.1710

512x512

threads	wallclock time (2 runs)	
1	2.163	2.157
2	1.153	1.156
4	0.6142	0.6018
8	0.3823	0.3601
12	0.3543	0.5133

1024x1024

threads	wallclock time (2 runs)	
1	9.431	9.475
2	4.145	4.109
4	2.235	3.410
8	1.355	1.350
12	2.116	1.346

Threadsafe

- When sharing memory you need to make sure you have private copies of any data that you are changing directly
- Applies to functions that you call in the parallel regions too!
- What if your answer changes when running with multiple threads?
  - Some roundoff-level error is to be expected if sums are done in different order
  - Large differences indicate a bug—most likely something needs to be private that is not
- Unit testing
  - Run with 1 and multiple threads an compare the output

Threadsafe

- Fortran:
  - Common blocks are simply a list of memory spaces where data can be found. This is shared across multiple routines
    - Very dangerous—if one thread updates something in a common block, every other thread sees that update
    - Much safer to use arguments to share data between functions
  - Save statement: the value of the data persists from one call to the next
    - What if a different thread is the next to call that function—is the saved quantity the correct value?

Legacy Code

- Sometimes you inherit code that works really well, but was written in a time before threadsafety was a concern
- Common blocks: use threadprivate directive
  - Ex: VODE...

Critical Sections

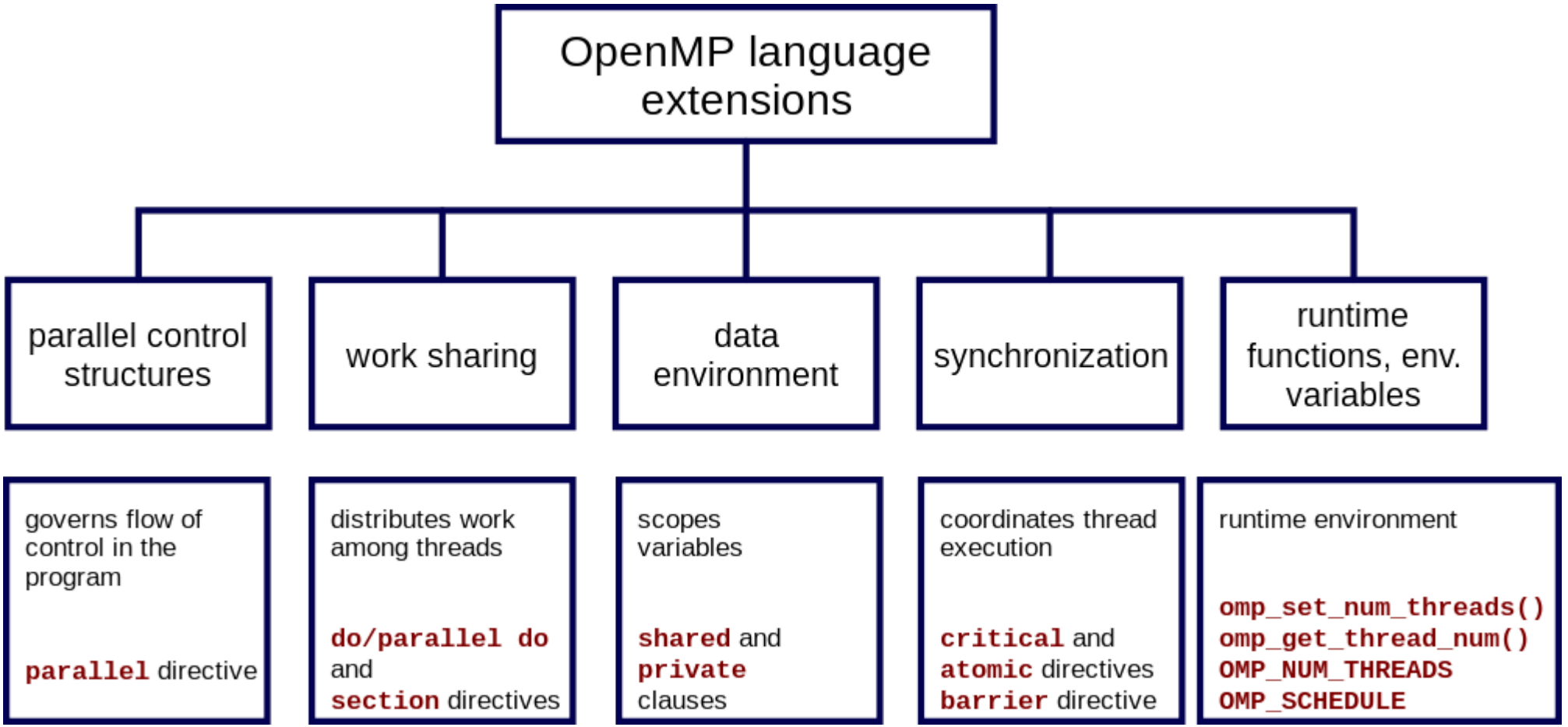
- Within a parallel region, sometimes you need to ensure that only one thread at a time can write to a variable
- Consider the following:

```
if ( a(i,j) > maxa ) then
   maxa = a(i,j)
   imax = i
   jmax = j
endif
```

  - If this is in the middle of a loop, what happens if 2 different threads meet the criteria?
  - Marking this section as critical will ensure only one thread changes things at a time
- Warning: critical sections can be VERY slow

OpenMP

- OpenMP is relatively big



Porting to OpenMP

- You can parallelize your code piece-by-piece
- Since OpenMP directives look like comments to the compiler, your old version is still there
- Generally, you are not changing any of your original code—just adding directives

More Advanced OpenMP

- `if` clause tells OpenMP only to parallelize if a certain condition is met (e.g. a test of the size of an array)
- `firstprivate`: like private except each copy is initialized to the value from the original value
- `schedule`: affects the balance of the work distributed to threads

OpenMP in Python

- Python enforces a “global interpreter lock” that means only one thread can talk to the interpreter at any one time
  - OpenMP within pure python is not possible
- However, C (or Fortran) extensions called from python can do shared-memory parallelism
  - Underlying code can do parallel OpenMP

MPI

- The Message Passing Library (MPI) is the standard library for distributed parallel computing
  - Now each core cannot directly see each other's memory
  - You need to manage how the work is divided and explicitly send messages from one process to the other as needed.

MPI Hello World

- No longer do we simply use comments—now we call subroutines in the library:

```
program hello

  use mpi

  implicit none

  integer :: ierr, mype, nprocs

  call MPI_Init(ierr)

  call MPI_Comm_Rank(MPI_COMM_WORLD, mype, ierr)
  call MPI_Comm_Size(MPI_COMM_WORLD, nprocs, ierr)

  if (mype == 0) then
     print *, "Running Hello, World on ", nprocs, " processors"
  endif

  print *, "Hello World", mype

  call MPI_Finalize(ierr)
end program hello
```

MPI Hello World

- MPI jobs are run using a commandline tool
  - usually `mpirun` or `mpiexec`
  - Eg: `mpiexec -n 4 ./hello`
- You need to install the MPI libraries on your machine to build and run MPI jobs
  - MPICH-2 is the most popular
  - Fedora: `yum install mpich2 mpich2-devel mpich2-autoload`

MPI Concepts

(based on *Using MPI*)

- A separate instance of your program is run on each processor—these are the MPI processes
  - Threadsafety is not an issue here, since each instance of the program is isolated from the others
- You need to tell the library the datatype of the variable you are communicating and how big it is (the buffer size).
  - Together with the address of the buffer specify what is being sent
- Processors can be grouped together
  - Communicators label different groups
  - `MPI_COMM_WORLD` is the default communicator (all processes)
- Many types of operations:
  - Send/receive, collective communications (broadcast, gather/scatter)

MPI Concepts

(based on *Using MPI*)

- There are > 100 functions
  - But you can do any messaging passing algorithm with only 6:
    - `MPI_Init`
    - `MPI_Comm_Size`
    - `MPI_Comm_Rank`
    - `MPI_Send`
    - `MPI_Recv`
    - `MPI_Finalize`
  - More efficient communication can be done by using some of the more advanced functions
  - System vendors will usually provide their own MPI implementation that is well-matched to their hardware

Ex: Computing Pi

(based on *Using MPI*)

- This is an example from *Using MPI*
  - Compute π by doing the integral:

$$\int_0^1 \frac{1}{1+x^2} dx = \arctan(x)|_0^1 = \frac{\pi}{4}$$

  - We will divide the interval up, so that each processor sees only a small portion of [0,1]
  - Each processor computes the sum for its intervals
  - Add all the integrals together at the end to get the value of the total integral
  - We'll pick one processor as the I/O processor—it will communicate with us
  - Let's look at the code...

Send/Receive Example

- The main idea in MPI is sending messages between processes.
- MPI_Send() and MPI_Recv() pairs provide this functionality
  - This is a blocking send/receive
    - For the sending code, the program resumes when it is safe to reuse the buffer
    - For the receiving code, the program resumes when the message was received
  - May cause network contention if the destination process is busy doing its own communication
  - See *Using MPI* for some diagnostics on this
- There are non-blocking send, sends where you explicitly attach a buffer

Send/Receive Example

- Simple example (mimics ghost cell filling)
  - On each processor allocate an integer array of 5 elements
  - Fill the middle 3 with a sequence (proc 0 gets 1,2,3, proc 1 get 4,5,6, ...)
  - Send messages to fill the left and right element with the corresponding element from the neighboring processors
- Let's look at the code...

Send/Receive

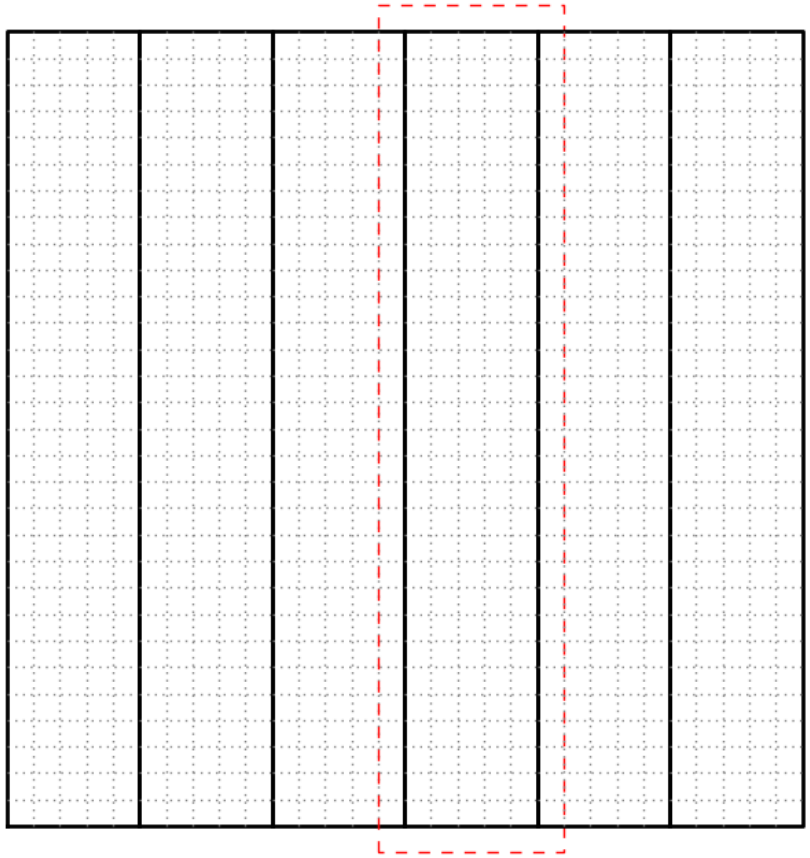
- Good communication performance often requires staggering the communication
- A combined `sendreceive()` call can help avoid deadlocking
- Let's look at the same task with `sendreceive()`

Relaxation

- Let's do the same relaxation problem, but now using MPI instead of OpenMP
  - In the OpenMP version, we allocated a single array covering the entire domain, and all processors saw the whole array
  - In the MPI version, each processor will allocate a smaller array, covering only a portion of the entire domain, and they will only see their part directly.

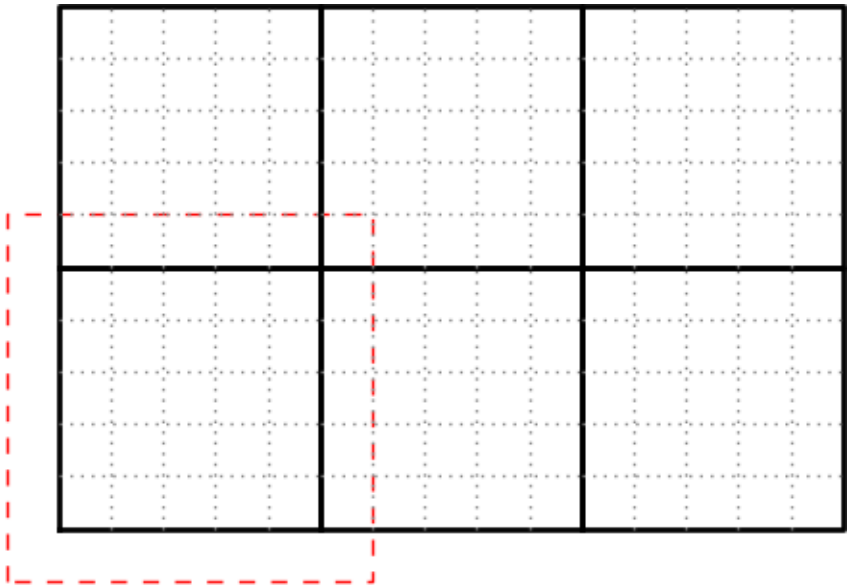
Relaxation

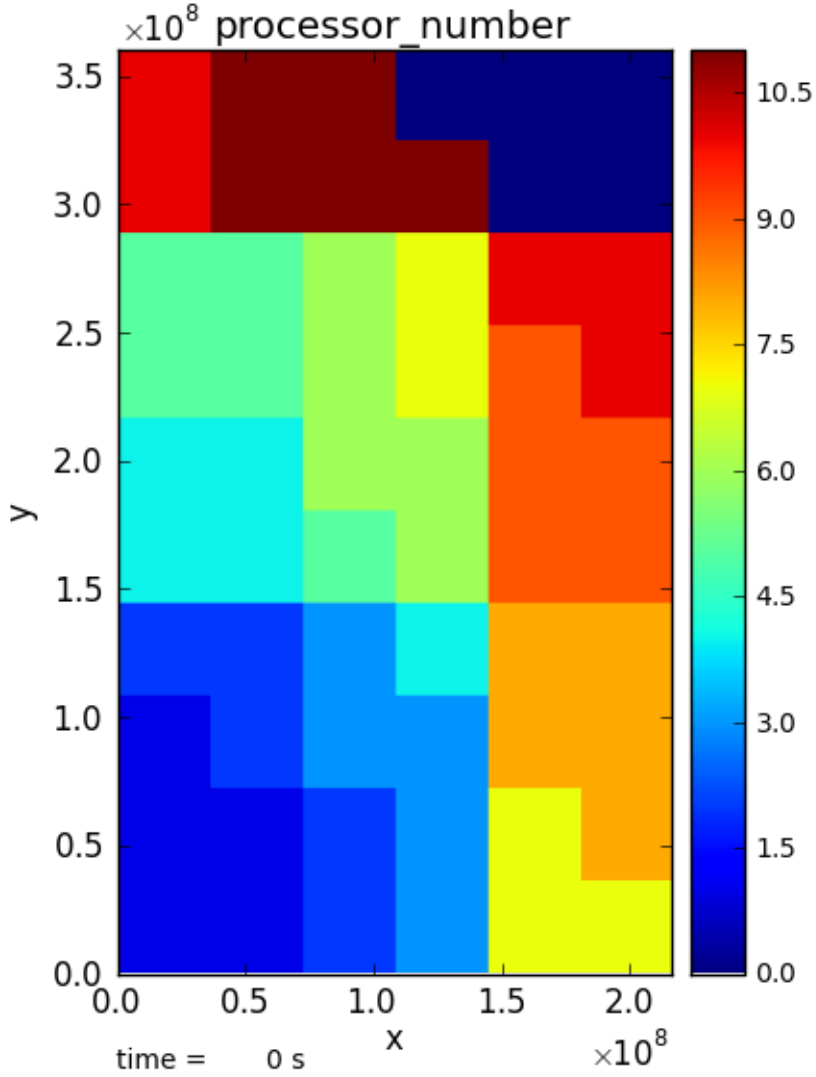
- We will do 1-d domain decomposition
  - Each processor allocates a slab that covers the full y-extent of the domain
  - Width in x is `nx/nprocs`
    - if not evenly divisible, then some slabs have a width of 1 more cell
  - Perimeter of 1 ghost cell surrounding each subgrid
- We will refer to a global index space `[0:nx-1]` × `[0:ny-1]`
  - Arrays allocated as: `f(ilo-ng:ihi+ng,jlo-ng:jhi+ng)`



Domain Decomposition

- Generally speaking, you want to minimize the surface-to-volume (this reduces communication)





Relaxation

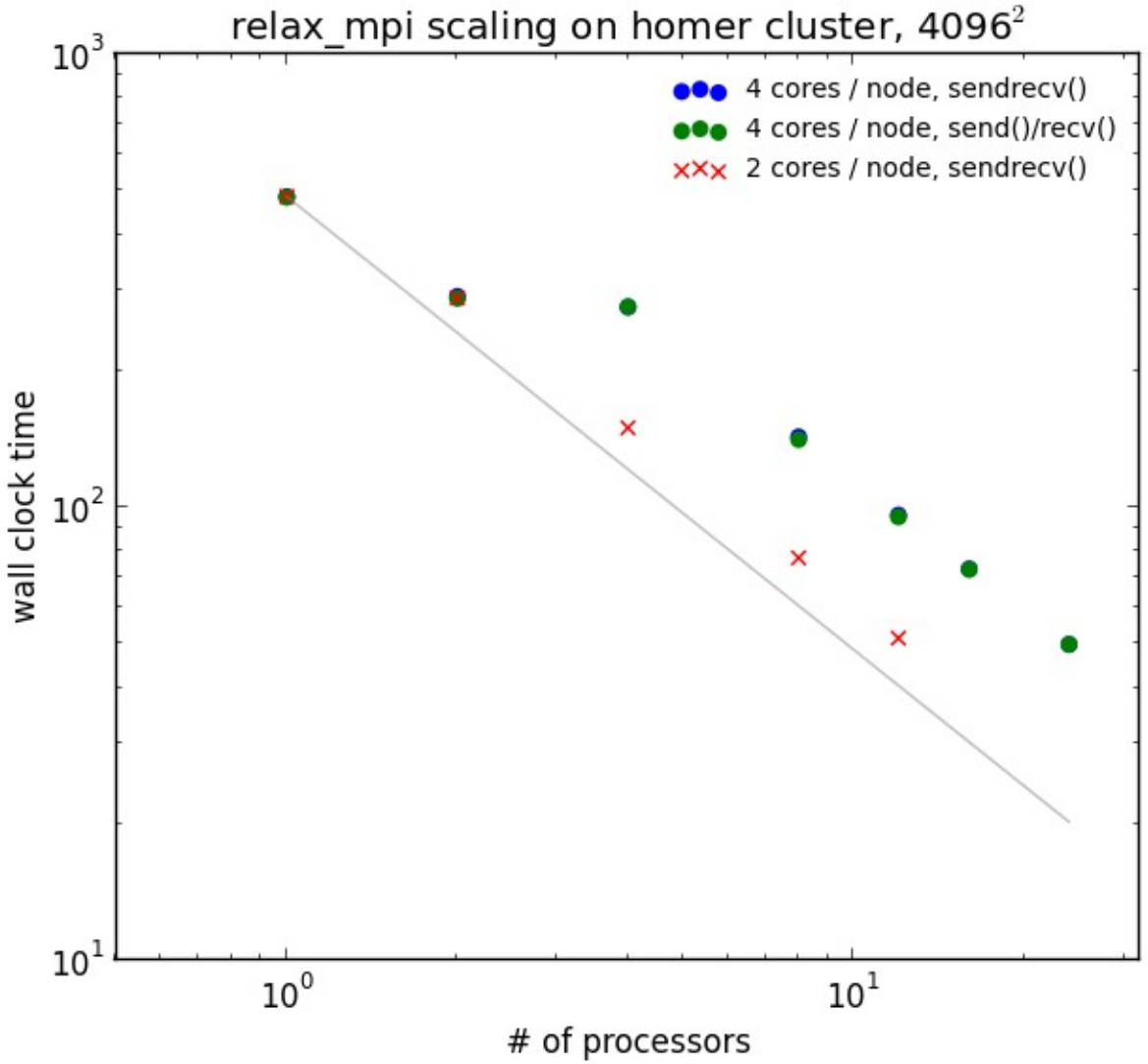
- Most of the parallelism comes in the ghost cell filling
  - Fill left GCs by receiving data from processor to the left
  - Fill right GCs by receiving data from processor to the right
  - Send/receive pairs—we want to try to avoid contention (this can be very tricky, and people spend a lot of time worrying about this...)
- On the physical boundaries, we simply fill as usual
- For computing a norm, we will need to reduce the local sums across processors
- Let's look at the code...

MPI Relaxation Results

- Run on my cluster
  - 6 nodes with 2 dual-core processors, circa 2006
  - Connected with gigabit ethernet
  - Test with send/recv and sendrecv, and using 4 vs. 2 cores per node

MPI Relaxation Results

- Notice that there seems to be a penalty on this machine when using all 4 cores on a node



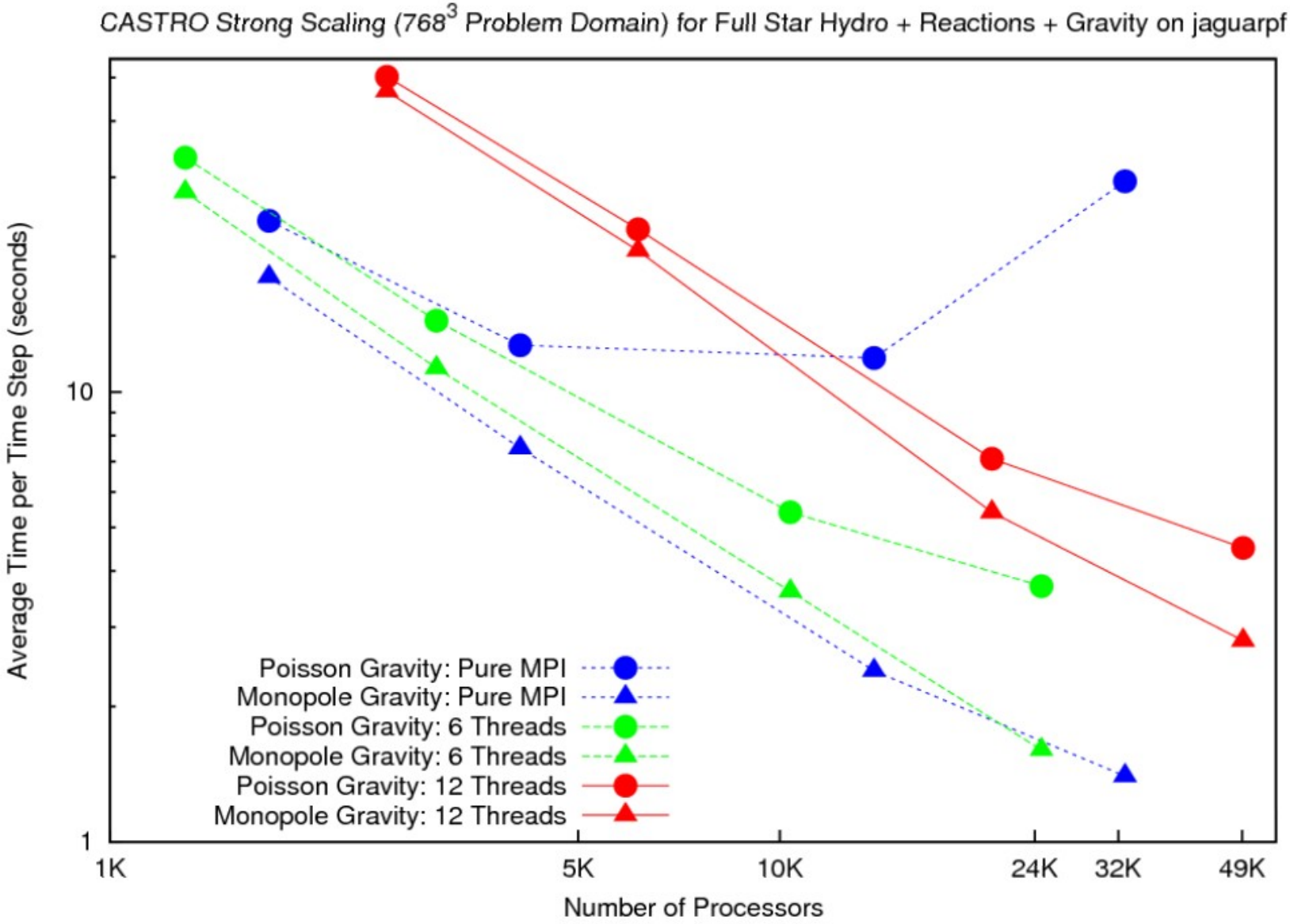
Weak vs. Strong Scaling

- In assessing the parallel performance of your code there are two methods that are commonly used
  - Strong scaling: keep the problem size fixed and increase the number of processors
    - Eventually you will become work-starved, and your scaling will stop (communication dominates)
  - Weak scaling: increase the amount of work in proportion to the number of processors
    - In this case, perfect scaling will result in the same wallclock time for all processor counts

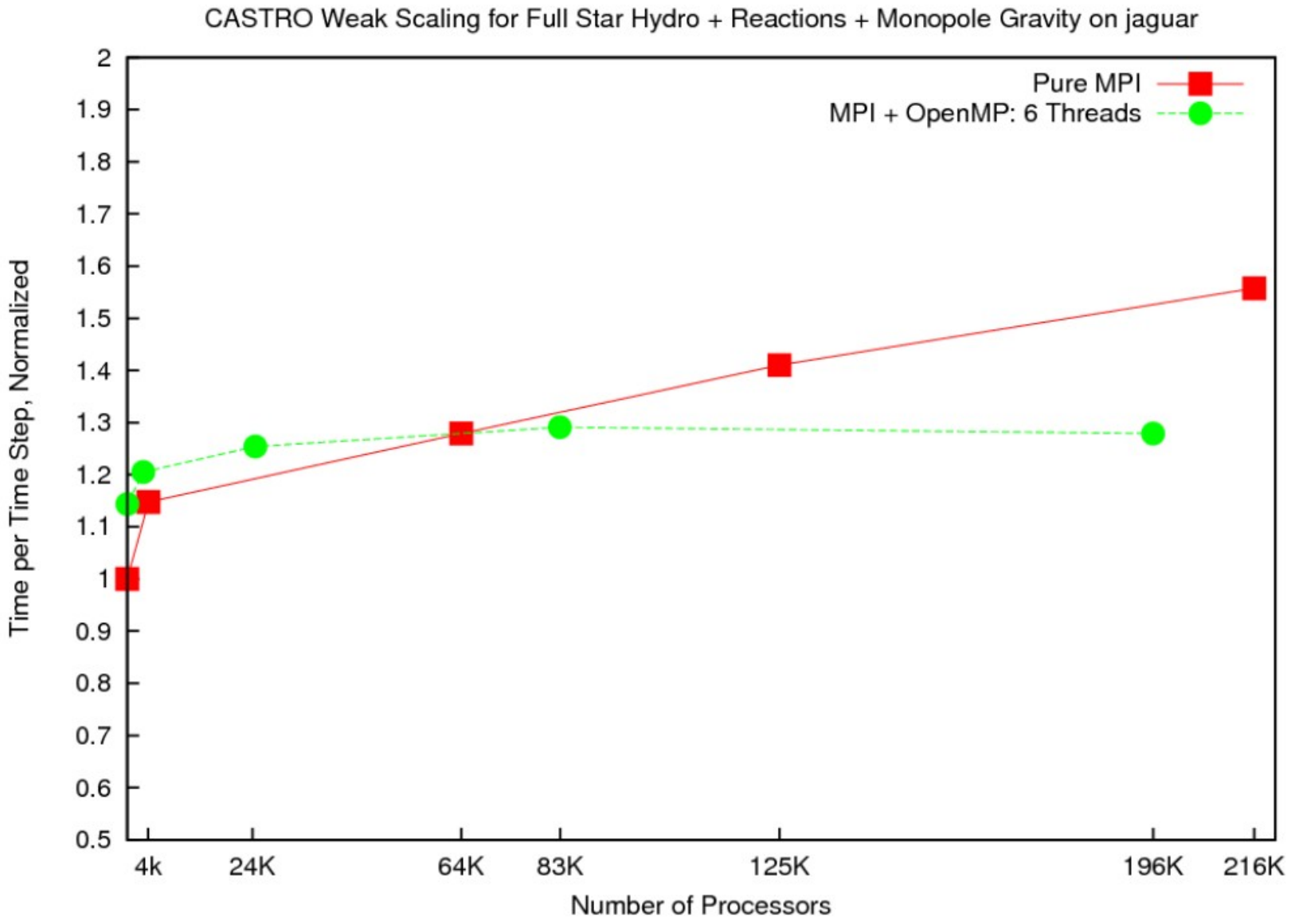
Ex: Castro Scaling

- Castro is a publicly available adaptive mesh refinement compressible hydrodynamics code
  - Models astrophysical flows
  - General equation of state, reactions, explicit diffusion
  - Radiation transport
  - Self-gravity (multigrid Poisson solve)

Ex: Castro Scaling



Ex: Castro Scaling



Debugging

- There are parallel debuggers (but these are pricey)
- Print is still your friend
  - Run as small of a problem as possible on as few processors as necessary
- Some roundoff-level differences are to be expected from sums (different order of operations)

Hybrid Parallelism

- To get good performance on current supercomputers, you need to do hybrid parallelism:
  - OpenMP within a node, MPI across nodes
- For example, in our MPI relaxation code, we could split the loops over each subdomain over multiple cores on a node using OpenMP.
  - Then we have MPI to communicate across nodes and OpenMP within nodes
  - This hybrid approach is often needed to get the best performance on big machines

Parallel Python

- MPI has interfaces for Fortran and C/C++
- There are several python modules for MPI
  - mpi4py: module that can be imported into python
    - Fedora:
      - yum install mpi4py-mpich2
      - Add /usr/lib64/python2.7/site-packages/mpich2/ to PYTHONPATH
  - pyMPI: changes the python interpreter itself

Parallel Python

- Hello world:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

print "Hello, world", rank
```

- Run with `mpiexec -n 4 python hello.py`

Coarray Fortran

- Part of the Fortran 2008 standard
  - Parallel version of Fortran
  - Separate image (instance of the program) is run on each processor
  - [ ] on arrays is used to refer to different processors
  - Not yet widely available

Supercomputing Centers

- Supercomputing centers
  - National centers run by NSF (through XSEDE program) and DOE (NERSC, OLCF, ALCF)
  - You can apply for time—starter accounts available at most centers to get up to speed
  - To get lots of time, you need to demonstrate that your codes can scale to $O(10^4)$ processors or more
- Queues
  - You submit your job to a queue, specifying the number of processors (MPI + OpenMP threads) and length of time
  - Typical queue windows are 2-24 hours
  - Job waits until resources are available

Supercomputing Centers

- Checkpoint/restart
  - Long jobs won't be able to finish in the limited queue window
  - You need to write your code so that it saves all of the data necessary to restart where it left off
- Archiving
  - Mass storage at centers is provided (usually through HPSS)
  - Typically you generate far more data than is reasonable to bring back locally—remote analysis and visualization necessary

Future...

- The big thing in supercomputing these days is accelerators
  - GPUs or Intel Phi boards
  - Adds a SIMD-like capability to the more general CPU
- Originally with GPUs, there were proprietary languages for interacting with them (e.g. CUDA)
- Currently, OpenACC is an OpenMP-like way of dealing with GPUs/accelerators
  - Still maturing
  - Portable
  - Will merge with OpenMP in the near future
- Data transfer to the accelerators moves across the slow system bus
  - Future processors may move these capabilities on-die