Parallel Computing
Optimization

- Getting performance out of your code means
  - Picking the right algorithm
  - Implementing the algorithm efficiently
- We talked a lot about picking the proper algorithm, and saw some examples of speed-ups you can get
- For performance in the implementation:
  - You need to understand a bit about how the computer's CPU works
  - You may need to consider parallel methods
Modern CPU + Memory System

- Memory hierarchy
  - Data is stored in main memory
  - Multiple levels of cache (L3, L2, L1)
    - A line of memory is moved into cache—you amortize the costs if you use all the data in the line
  - Data gets to the registers in the CPU—this is where the computation takes place
- It is expensive to move data from main memory to the registers
  - You need to exploit cache
  - For arrays, loop over data such that you operate on elements that are adjacent in memory
Modern CPU + Memory System

- Some numbers (http://www.7-cpu.com/cpu/Haswell.html)
- Intel i7-4770 (Haswell), 3.4 GHz (Turbo Boost off), 22 nm. RAM: 32 GB (PC3-12800 cl11 cr2).
  - L1 Data cache = 32 KB, 64 B/line, 8-WAY.
  - L1 Instruction cache = 32 KB, 64 B/line, 8-WAY.
  - L2 cache = 256 KB, 64 B/line, 8-WAY
  - L3 cache = 8 MB, 64 B/line
- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L1 Data Cache Latency = 5 cycles for access with complex address calculation (size_t n, *p; n = p[n]).
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 36 cycles
- RAM Latency = 36 cycles + 57 ns
Arrays

- Row vs. Column major: $A(m, n)$
  - First index is called the row
  - Second index is called the column
  - Multi-dimensional arrays are flattened into a one-dimensional sequence for storage
  - Row-major (C, python): rows are stored one after the other
  - Column-major (Fortran, matlab): columns are stored one after the other

- Ordering matters for:
  - Passing arrays between languages
  - Deciding which index to loop over first
Arrays

- This is why in Fortran, you want to loop as:

```fortran
double precision :: A(M,N)

do j = 1, N
  do i = 1, M
    A(i,j) = ...
  enddo
enddo
```

- And in C:

```c
double A[M][N];

for (i = 0; i < M; i++) {
  for (j = 0; j < N; j++) {
    A[i][j] = ...
  }
}
```
Arrays

- Floating point unit uses pipelining to perform operations
- Most efficient if you can keep the pipe full—again, taking advantage of nearby data in cache
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**)
  - Offloading to accelerators: **OpenACC, OpenMP,** or **CUDA**
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
35 YEARS OF MICROPROCESSOR TREND DATA

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
### Top500 List - November 2017

$R_{\text{max}}$ and $R_{\text{peak}}$ values are in TFlops. For more details about other fields, check the TOP500 description.

$R_{\text{peak}}$ values are calculated using the advertised clock rate of the CPU. For the efficiency of the systems you should take into account the Turbo CPU clock rate where it applies.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>$R_{\text{max}}$ (TFlop/s)</th>
<th>$R_{\text{peak}}$ (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Supercomputing Center in Wuxi, China</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.495GHz, Sunway NRPC</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>2</td>
<td>National Super Computer Center in Guangzhou, China</td>
<td>Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5 2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3,120,000</td>
<td>33,852.7</td>
<td>54,902.4</td>
<td>17,808</td>
</tr>
<tr>
<td>3</td>
<td>Swiss National Supercomputing Centre (CSCS), Switzerland</td>
<td>Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect, NVIDIA Tesla P100</td>
<td>361,780</td>
<td>19,590.0</td>
<td>25,326.3</td>
<td>2,272</td>
</tr>
<tr>
<td>4</td>
<td>Japan Agency for Marine-Earth Science and Technology, Japan</td>
<td>Gyoukou - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700MHz ExaScaler</td>
<td>19,860,000</td>
<td>19,135.8</td>
<td>28,192.0</td>
<td>1,350</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SC/Oak Ridge National Laboratory, United States</td>
<td>Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.</td>
<td>560,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>6</td>
<td>DOE/NNSA/LLNL, United States</td>
<td>Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM</td>
<td>1,572,864</td>
<td>17,173.2</td>
<td>20,132.7</td>
<td>7,890</td>
</tr>
<tr>
<td>7</td>
<td>DOE/NNSA/LANL/SNL, United States</td>
<td>Trinity - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.</td>
<td>979,988</td>
<td>14,137.3</td>
<td>43,902.6</td>
<td>3,844</td>
</tr>
<tr>
<td>8</td>
<td>DOE/SC/LBNL/NERSC, United States</td>
<td>Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.</td>
<td>622,336</td>
<td>14,014.7</td>
<td>27,880.7</td>
<td>3,939</td>
</tr>
<tr>
<td>9</td>
<td>Joint Center for Advanced High Performance Computing, Japan</td>
<td>Oakforest-PACS - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path</td>
<td>556,104</td>
<td>13,554.6</td>
<td>24,913.5</td>
<td>2,719</td>
</tr>
</tbody>
</table>
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 decreases
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...
- Also see GNU parallel
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 cores 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
• **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
Shared Memory

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

Two 4-core chips comprising a single node—each has their own pool of memory
Ex: Blue Waters Machine

XE6 Node

10 12X Gemini Channels
(Each Gemini acts like two nodes on the 3D Torus)

High Radix YARC Router with adaptive Routing
168 GB/sec capacity

Cray Baker Node Characteristics

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>32*</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>~300 Gflops/s</td>
</tr>
<tr>
<td>Memory Size</td>
<td>64 GB per node</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>85 GB/sec</td>
</tr>
</tbody>
</table>

(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
Parallel Computing
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:**

```fortran
program hello

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

end program hello
```

- Compile with: `gfortran -o hello -fopenmp hello.f90`
In C, the preprocessor is used for the pragmas

```c
#include <stdio.h>

void main() {
    #pragma omp parallel
    printf("Hello world\n");
}
```
In addition to using pragmas, there are a few functions that OpenMP provides to get the number of threads, the current thread, etc.

```fortran
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
```

code: hello-omp.f90
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?
Aside: Stack vs. Heap

- Memory allocated at compile time is put on the stack, e.g.:
  - Fortran: double precision a(1000)
  - C: double a[1000]

- Stack memory has a fixed (somewhat small size)
  - It's managed by the operating system
  - You don't need to clean up this memory

- Dynamic allocation puts the memory on the heap
  - Much bigger pool
  - You are responsible for deallocating
Parallel Loops

- Ex: matrix multiplication:

```fortran
program matmul

  use omp_lib

  implicit none
  integer, parameter :: N = 50000
  double precision, allocatable :: a(:, :)
  double precision :: x(N), b(N)
  double precision :: start_omp, finish_omp
  integer :: i, j

  start_omp = omp_get_wtime()
  allocate(a(N,N))

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

  finish_omp = omp_get_wtime()
  write(*,*) 'Parallel time: ', finish_omp - start_omp
end program matmul
```
Parallel Loops

Continued...

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

finish_omp = omp_get_wtime()

print *, "execution time: ", finish_omp - start_omp

end program matmul
Timing

- We can use the `omp_get_wtime()` command to get the current wallclock time (in seconds)
  - This is better than, e.g., the Fortran `cpu_time()` intrinsic, which measures time for all threads summed together

<table>
<thead>
<tr>
<th>OMP_NUM_THREADS</th>
<th>run 1 time (s)</th>
<th>run 2 time (s)</th>
<th>run 3 time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26.276</td>
<td>26.294</td>
<td>26.696</td>
</tr>
<tr>
<td>2</td>
<td>18.696</td>
<td>17.514</td>
<td>16.287</td>
</tr>
<tr>
<td>4</td>
<td>8.628</td>
<td>9.072</td>
<td>10.680</td>
</tr>
<tr>
<td>8</td>
<td>4.744</td>
<td>6.582</td>
<td>4.923</td>
</tr>
<tr>
<td>16</td>
<td>3.066</td>
<td>3.146</td>
<td>3.111</td>
</tr>
</tbody>
</table>

Timings on 2x Intel Xeon Gold 5115 CPU using gfortran, N = 50000
Loop Ordering

- This is a great example to see the effects of loop ordering—what happens if you switch the order of the loops?
Loop Parallel

- We want to parallelize all loops possible
  - Instead of $f(\ : \ , \ :) = 0 \cdot d\theta$, 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

```fortran
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?

code: reduce.f90
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.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Wallclock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.583</td>
</tr>
<tr>
<td>2</td>
<td>0.8413</td>
</tr>
<tr>
<td>4</td>
<td>0.3979</td>
</tr>
<tr>
<td>8</td>
<td>0.2253</td>
</tr>
<tr>
<td>16</td>
<td>0.1634</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads</th>
<th>Wallclock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.211</td>
</tr>
<tr>
<td>2</td>
<td>3.179</td>
</tr>
<tr>
<td>4</td>
<td>1.717</td>
</tr>
<tr>
<td>8</td>
<td>0.8832</td>
</tr>
<tr>
<td>16</td>
<td>0.5076</td>
</tr>
</tbody>
</table>

code: relax.f90

groot w/ gfortran -Ofast

512x512

<table>
<thead>
<tr>
<th>Threads</th>
<th>Wallclock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.583</td>
</tr>
<tr>
<td>2</td>
<td>0.8413</td>
</tr>
<tr>
<td>4</td>
<td>0.3979</td>
</tr>
<tr>
<td>8</td>
<td>0.2253</td>
</tr>
<tr>
<td>16</td>
<td>0.1634</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads</th>
<th>Wallclock Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.211</td>
</tr>
<tr>
<td>2</td>
<td>3.179</td>
</tr>
<tr>
<td>4</td>
<td>1.717</td>
</tr>
<tr>
<td>8</td>
<td>0.8832</td>
</tr>
<tr>
<td>16</td>
<td>0.5076</td>
</tr>
</tbody>
</table>
Threadsafes

- 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?
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:

```plaintext
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

OpenMP language extensions

- parallel control structures
- work sharing
- data environment
- synchronization
- runtime functions, env. variables

<table>
<thead>
<tr>
<th>parallel control structures</th>
<th>work sharing</th>
<th>data environment</th>
<th>synchronization</th>
<th>runtime functions, env. variables</th>
</tr>
</thead>
</table>
| governs flow of control in the program
  **parallel** directive | distributes work among threads
  **do/parallel do** and **section** directives | scopes variables
  **shared** and **private** clauses | coordinates thread execution
  **critical** and **atomic** directives
  **barrier** directive | runtime environment
  **omp_set_num_threads()**
  **omp_get_thread_num()**
  **OMP_NUM_THREADS**
  **OMP_SCHEDULE**
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.
• No longer do we simply use comments—now we call subroutines in the library:

```fortran
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 is the most popular
  - Fedora: `dnf install mpich mpich-devel mpich-autoload`

---

code: hello_mpi.f90
MPI Concepts
(based on Using MPI)

- A separate instance of your program is run on each processor—these are the MPI processes
  - Thread safety 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 \( \pi \) by doing the integral:
    \[
    \int_0^1 \frac{1}{1 + x^2} \, dx = \arctan(x) \bigg|_0^1 = \frac{\pi}{4}
    \]
  - 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...

code: pi.f90
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: 0,1,2; proc 1: 3,4,5, ...)
  - Send messages to fill the left and right element with the corresponding element from the neighboring processors

![Diagram of array allocation and message passing]

code: send_recv.f90
Send/Receive

- Good communication performance often requires staggering the communication
- A combined sendrecv() call can help avoid deadlocking
- Let's look at the same task with MPI_Sendrecv()
Parallel Computing
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.
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 $[0: nx-1] \times [0: ny-1]$

- Memory needs spread across all processors
- Arrays allocated as:
  
  $f(ilo-\text{ng} : \text{ihi}+\text{ng}, jlo-\text{ng} : \text{jhi}+\text{ng})$
Relaxation

- Left set of ghost cells are filled by receiving a message from processor (slab) to left
Relaxation

- Right set of ghost cells are filled by receiving a message from processor (slab) to right
- Top and bottom ghost cells are physical boundaries
• 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
- The way this is written, our relaxation routine doesn't need to do any parallelism itself—it just operates on the domain it is given.
- For computing a norm, we will need to reduce the local sums across processors
- Let's look at the code...

code: relax_mpi.f90
Note that the smaller problem sizes become work starved more easily.
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: Maestro Scaling**

- **Maestro** is a publicly available adaptive mesh refinement low Mach number hydrodynamics code
  - Models astrophysical flows
  - General equation of state, reactions, implicit diffusion
  - Elliptic constraint enforced via multigrid
  - [https://github.com/AMReX-Astro/MAESTRO](https://github.com/AMReX-Astro/MAESTRO)
Ex: Maestro Scaling

NERSC Edison Scaling for Maestro 3-d XRB

- Intel compilers (15.0.1)
- MPI + OpenMP
- Cray compilers (8.4.0)
- pure MPI

average time to advance timestep

number of cores

384 x 384 x 768 zones; 48^3 domain decomposition

2015-09-10
Ex: Maestro Scaling

OLCF Titan Scaling for Maestro 3-d XRB

- 384x384x768
- 768x768x768

MPI + OpenMP
pure MPI

48^3 domain decomposition
Cray 8.4.0 compilers; 2015-08-31
Ex: Castro Scaling

- Castro is a publicly available adaptive mesh refinement compressible radiation hydrodynamics code
  - Used to model stellar explosions
  - Self-gravity solved via multigrid
Ex: Castro Scaling
Debugging

- There are parallel debuggers (but these are pricey)
- It's possible to spawn multiple gdb sessions, but this gets out of hand quickly
- 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
  - pyMPI: changes the python interpreter itself
• **Hello world:**

```python
from mpi4py import MPI

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

print "Hello, world", rank
```

• **Run with mpiexec -n 4 python hello.py**
Parallel Python

- We can easily parallelize our Monte Carlo poker odds code
  - Each processor considers hands independently
  - Do a reduction at the end
There are lots of libraries that provide parallel frameworks for writing your application.

Some examples:

- Linear Algebra / PDEs
  - PETSc: linear and nonlinear system solvers, parallel matrix/vector routines
  - hypre: sparse linear system solver
- I/O
  - HDF5: platform independent parallel I/O built on MPI-IO
- Adaptive mesh refinement (grids)
  - BoxLib: logically Cartesian AMR with elliptic solvers
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
GPUs

- GPU offloading can greatly accelerate computing
- Main issue: data needs to transfer from the CPU across the (relatively slow) PCIe bus to GPU
  - Good performance requires that lots of work is done on the data to “pay” the cost of the transfer
- GPUs work as SIMD parallel machines
  - The same instructions operate on all the data in lockstep
  - Branching (if-tests) is slower
- Best performance requires that you structure your code to be vectorized
OpenACC

- OpenACC is a directives-based method for offloading computing to GPUs
  - Looks like OpenMP
  - A big difference is that you need to explicitly add directives that control data movement
- There's a big cost in moving data from the CPU to the GPU
  - You need to do a lot of computing on the GPU to cover that expense
  - We can separately control what is copied to and from the GPU
- We can do our same relaxation example using OpenACC
  - Note: we need to explicitly write out the separate red-black updates to ensure that a loop doesn't access adjacent elements

code: relax-openacc.f90
```fortran
!$acc data copyin(f, dx, imin, imax, jmin, jmax, bc_lo_x, bc_hi_x, bc_lo_y, bc_hi_y)
copy(v)
do m = 1, nsmooth

!$acc parallel
!$acc loop
!$acc loop
do j = jmin, jmax
  v(imin-1,j) = 2*bc_lo_x - v(imin,j)
  v(imax+1,j) = 2*bc_hi_x - v(imax,j)
enddo

!$acc loop
do i = imin, imax
  v(i,jmin-1) = 2*bc_lo_y - v(i,jmin)
  v(i,jmax+1) = 2*bc_hi_y - v(i,jmax)
enddo

!$acc wait
!$acc loop collapse(2)
do j = jmin, jmax, 2
  do i = imin, imax, 2
    v(i,j) = 0.25d0*(v(i-1,j) + v(i+1,j) +
                     v(i,j-1) + v(i,j+1) - dx*dx*f(i,j))
  enddo
enddo
...
```

This is part of the smoother function marked up with OpenACC
OpenACC

- This relaxation code runs about $30 \times$ faster on the GPU vs. CPU (single core) on a local machine
  - Note: when comparing CPU to GPU, a fair comparison would include all of the CPU cores, so for a 12 core machine, it is about $2.5 \times$ faster on the GPU.
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
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