Introduction to Distributed-Memory Computing

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

More Concurrency

- So far we have talked about concurrency “within a box”
  - Within a processor
    - Pipelining
    - Multiple functional units
    - Instruction Level Parallelism
    - Hyper-Threading
  - Across processors
    - Multi-proc systems
    - Multi-core systems
    - Multi-proc/core systems

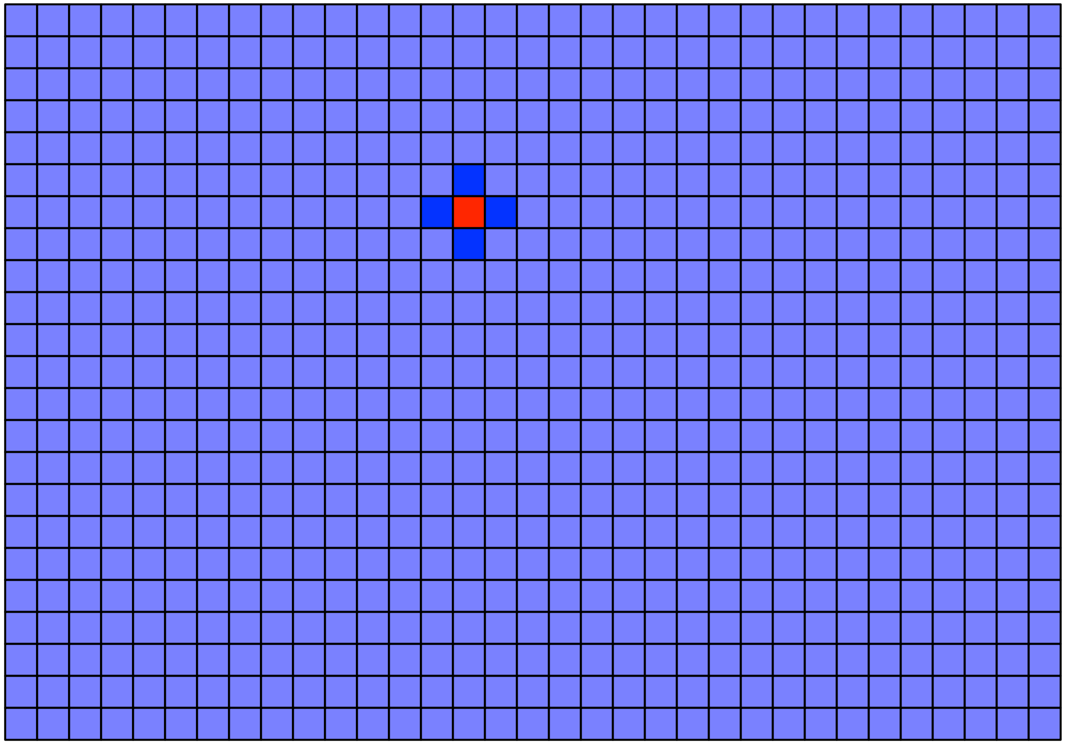
- But this can only get us so far for many applications...

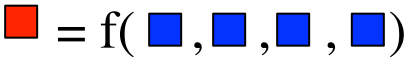
Toward Distributed Memory

- We saw that we go to concurrency for need of more CPU cycles (i.e., we want to use all cores)
- But that's often not enough and we can't use a single system anymore
- Reason #1: We need way more cycles than that in a single machine
- Reason #2: We need way more RAM than that in a single machine
- Solution: Use more than one machine

Example: Image Processing Filter

- Say you want to apply a simple filter to a domain (image, computational fluid dynamics, etc.)





Sample Stencil App Code

```
int a[N][N], a_new[N][N];
for (i=1; i<N-1; i++) {
     #pragma omp parallel for private(j)
     for (j=1; j<N-1; j++) {
          a_new[i][j] = f(a[i][j],
                      a[i-1][j],a[i+1][j],
                     a[i][j-1],a[i][j+1]);
     }
}
```

Too Large?

- This is all well and good, but what if my array requires 8GB of memory and I only have 1GB of RAM?
- I could think of just relying on virtual memory
  - This is bound to be very slow
- I could manage the reads and writes to disk myself
  - Could be a bit faster than virtual memory if I am really clever, but would be complicated and still slow
  - Called an “out of core” implementation
- Or, I could use 8 different machines with 1GB RAMs and run fast without really ever swapping between the memory and the disk!

Distributed Memory Programming

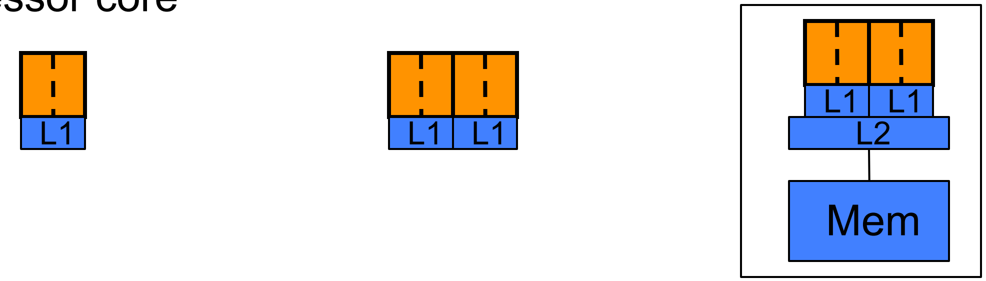
- So, I give you a bunch of individual hosts, all connected via a network
- The big question is: How do we write code for something like this?
- The application now consists of multiple processes running on different machines
  - Each process can consist of multiple threads!
- Let’s look at this on a picture

Distributed Memory Platform

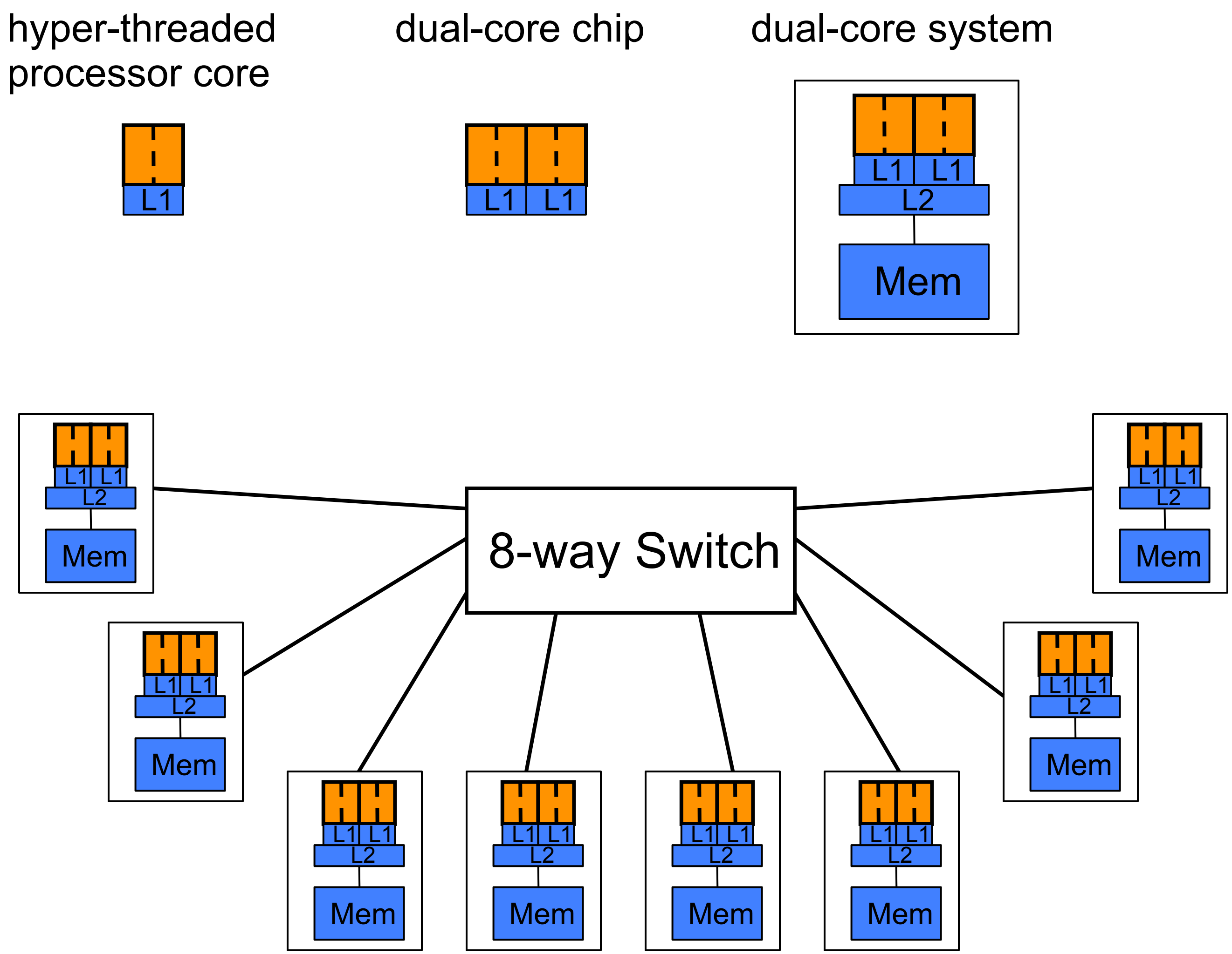
hyper-threaded processor core

dual-core chip

dual-core system

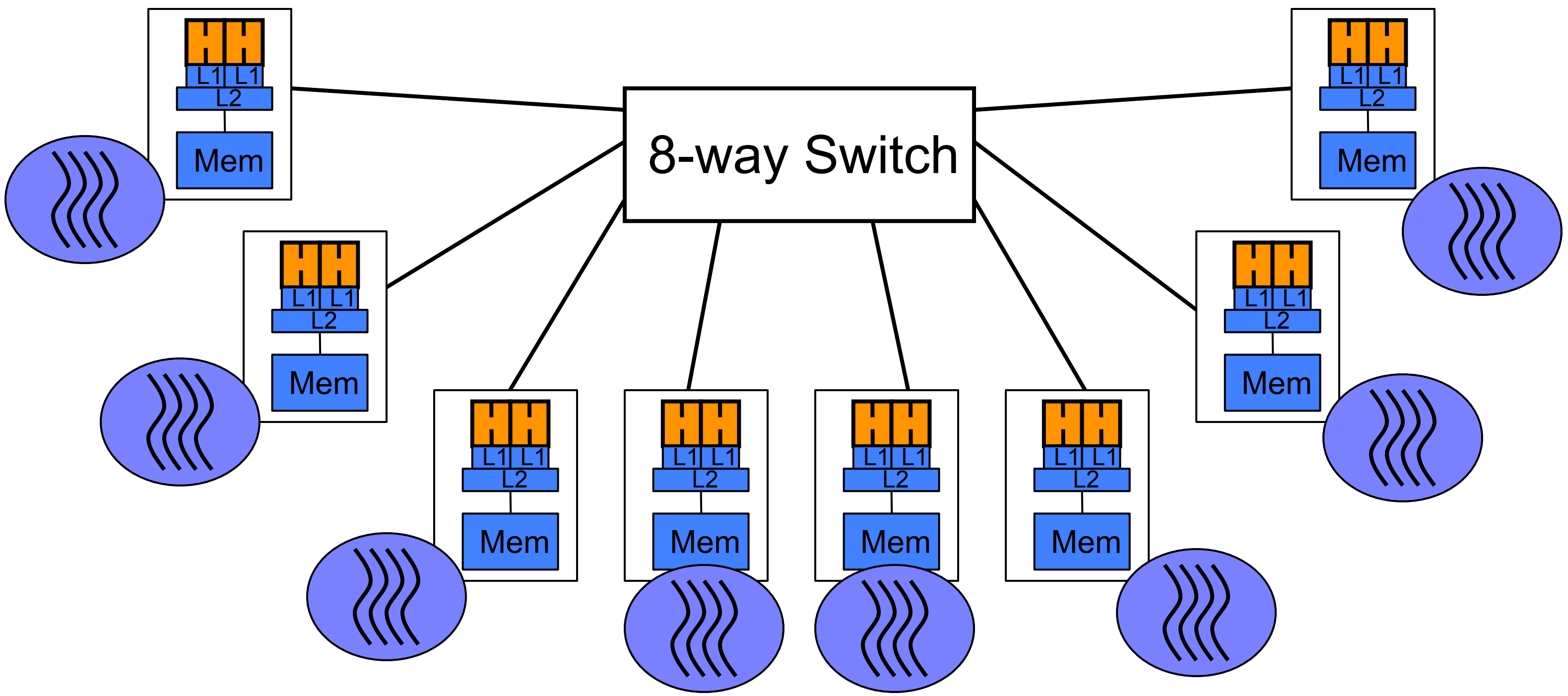


Distributed Memory Platform



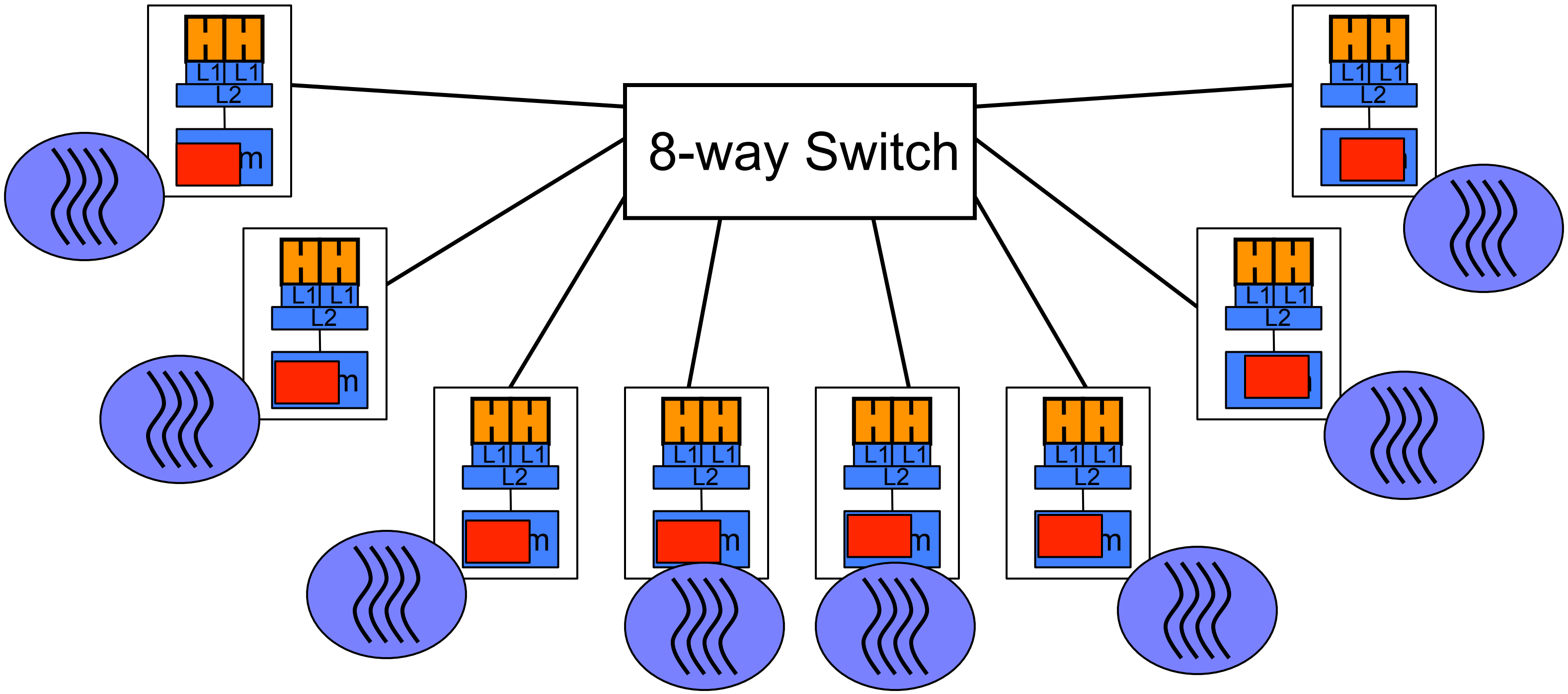
Cluster of dual-core systems

Distributed Memory Program



- 8 processes
- Each process contains, for example, 4 threads
  - 2 threads are running on each core using hyper threading

Distributed Memory Program



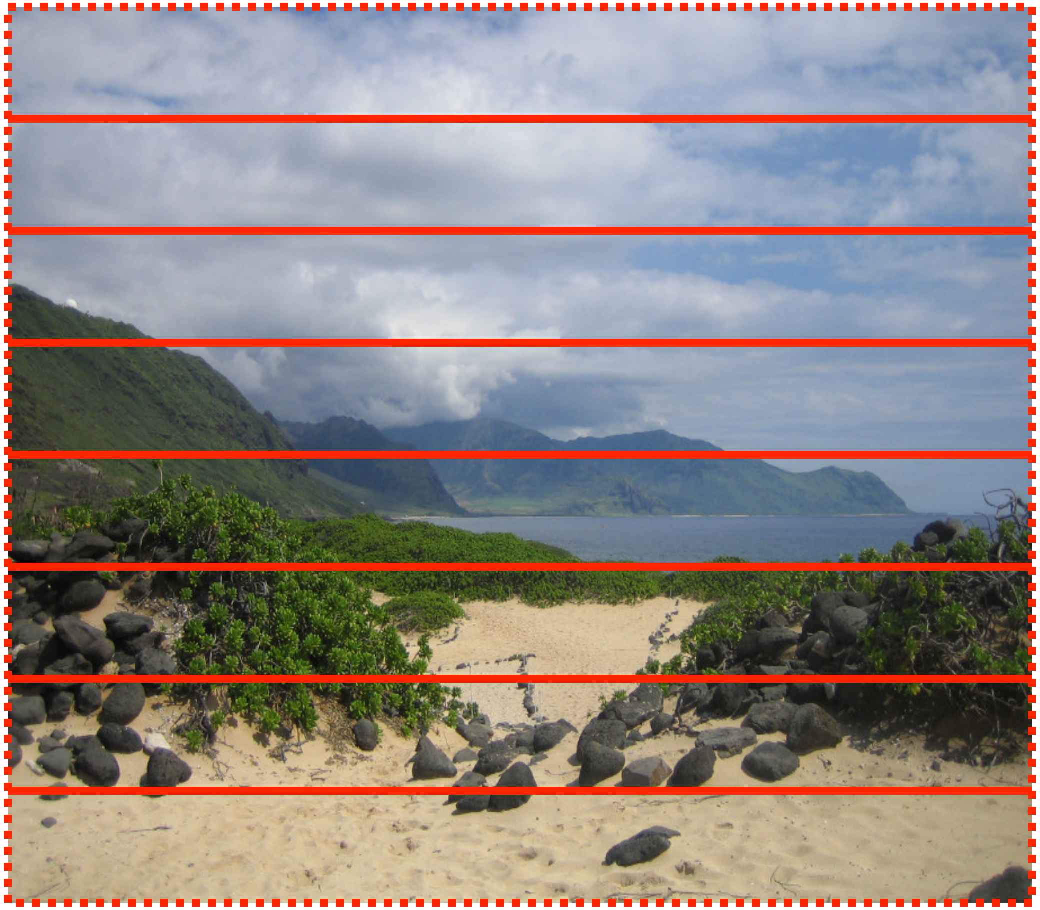
- Each process stores some data in the memory of its box

How do we even declare arrays?

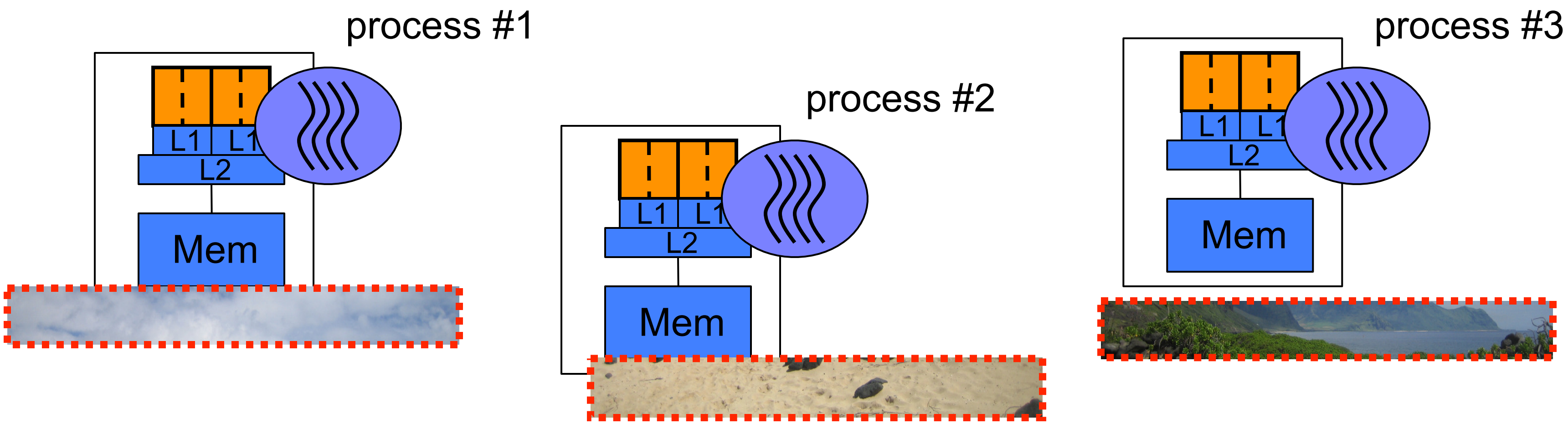
- We cannot have a declaration of an NxN array any more, because that would not fit in memory
- Each process (running on a different system) must handle an array of size N x N/8
  - Each process allocates memory for 1/8 of the overall array
- This is the same kind of “cutting the image into slabs” approach as we would used for a shared-memory implementation...

Data Distribution

Data Distribution



Data Distribution

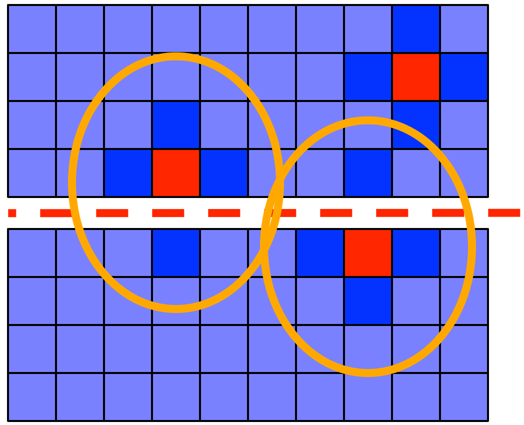


- Each piece of the image is stored in the memory of a different system
- A process running on one system can only “see” (i.e., address) the local image piece, and has no way to address other pieces: NO SHARED MEMORY
- This is what makes distributed memory programming MUCH harder than shared-memory programming

Boundaries!

- One of the problems now is: what happens at the boundaries/edges of the image tiles?

process #1



process #2

- Process #1 needs pixels from process #2
- Process #2 needs pixels from process #1
- But processes cannot share memory because they're on different systems:
  - With multiple threads all on the same system, there is no notion that a thread can't see some data!
  - In fact, we use threads because we want them to see the data
  - But now we're forced to use processes, and on different machines to boot

Message-Passing

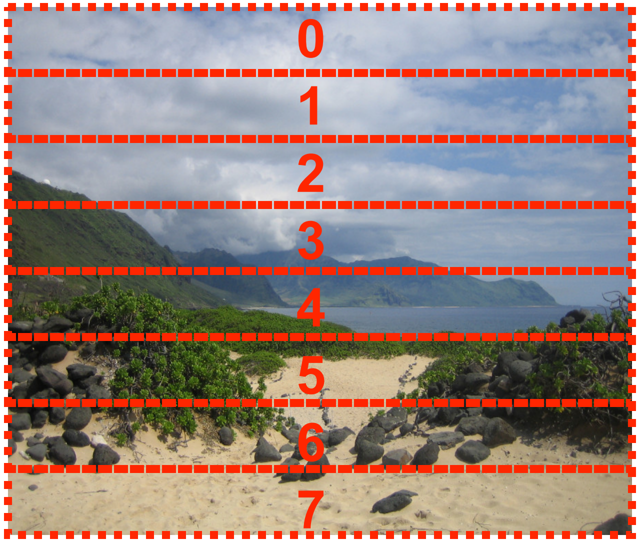
- Since processes cannot share memory, they have to exchange messages
  - “here are the pixels you need from me, give me the ones I need from you”
- This type of programming is called “message-passing”
- Uses network communication
  - e.g., socket and TCP
- So your code will have special function calls:
  - Send(...)
  - Receive(...)
- We’re getting further away from “simple” shared-memory programming

SPMD Program

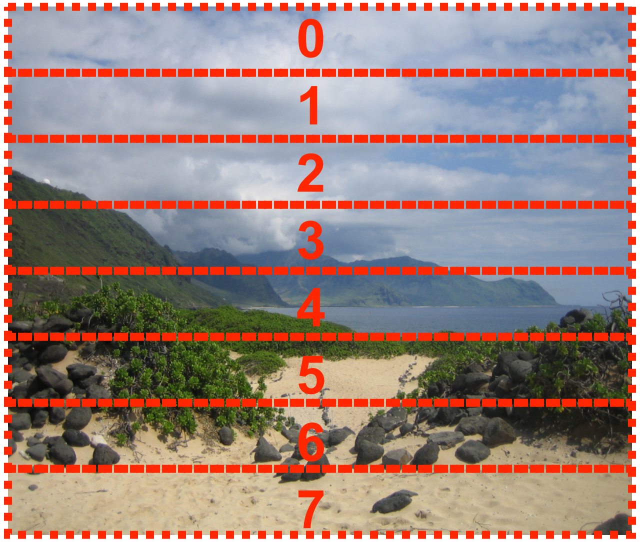
- So at this point, we could
  - implement 8 different programs
  - start them up somehow on different nodes of our cluster (for instance)
  - have them all somehow identify their left and right neighbors, if any
- Turns out that this is really cumbersome
  - And if I want to use 1000 processes, I have to write 1000 programs?
- Typically one uses/implements the notion of a process' rank

Process Ranks

- To identify the processes participating in the computation, each process is assigned an index from 0 to N-1
- Each process can find out what its rank is and how many processes there are in total



Communication Patterns



- Process 0 will send to 1 and receive from 1
- Process 1 will send to 0, receive from 0, send to 2, and receive from 2
- ...
- Process 7 will receive from 6 and send to 6

SPMD Programming

- If every process can find out its rank and the total number of processes, then one can write a Single Program to operate on Multiple pieces of Data simultaneously (SPMD):

```
int main() {

  if (my_rank() == 0) {
     // talk to my below neighbor
  } else if (my_rank() == num_processes() -1) {
    // talk to my above neighbor
  } else {
    // talk to my  above and below neighbors
  }
}
```

Ranks and Number of Processes

- For now we're going to assume we have the my_rank() and the num_processes() functions, and the all the logistics of starting up the processes is taken care of
  - The same assumption that we can make with OpenMP within a single machine
- But this can also be implemented by hand if necessary
- The way to write distributed memory programs is to rely on process ranks

Writing the SPMD Program

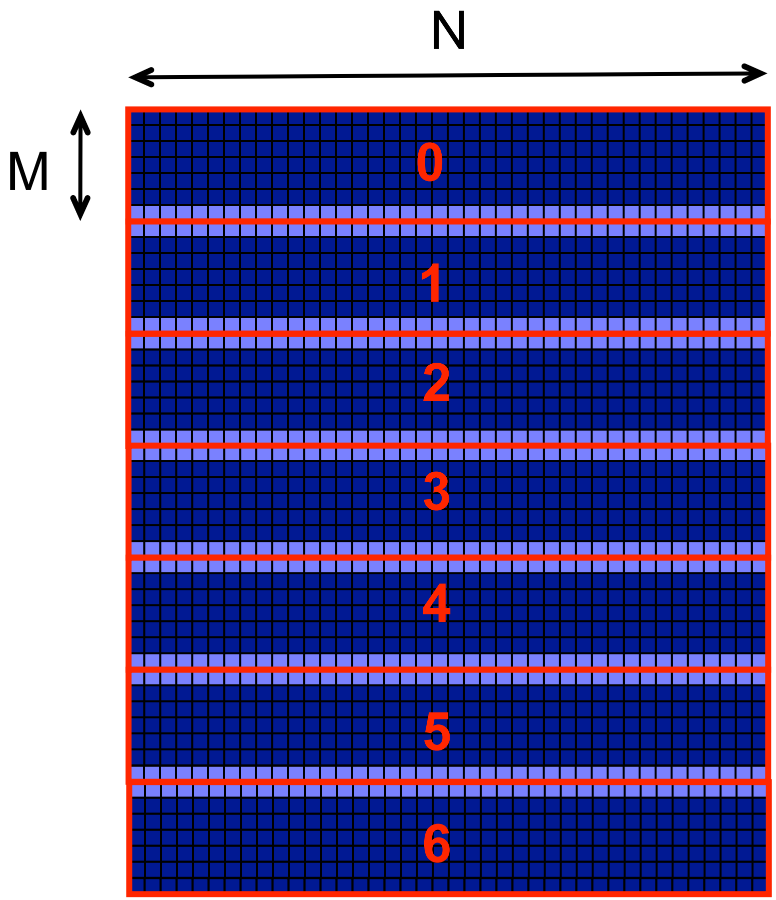
- The pseudo-code of the SPMD program could then look like

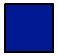
```
int main() {
  int M = N/num_processes();  // assumed to be integer!
  int original_image[M][N];
  int new_image[M][N];

  // load my part of the image from disk
  // compute all the pixels that do not require communication
  // send border pixels to my neighbor(s)
  // receive border pixels from my neighbors()
  // compute the remaining pixels
  // save the new image to file in orderly fashion
}
```

Writing the SPMD Program

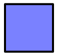
- For now, let's ignore the issue of loading/ writing files to disk
  - There are a lot of options here, simple/slow ones, and complex/fast ones
- Let's focus on computation and communication

Computing the "easy" pixels





Can be computed without communication



Requires pixels from neighbors

(note that process 0 and process N-1 can compute one more row than the others without any communication

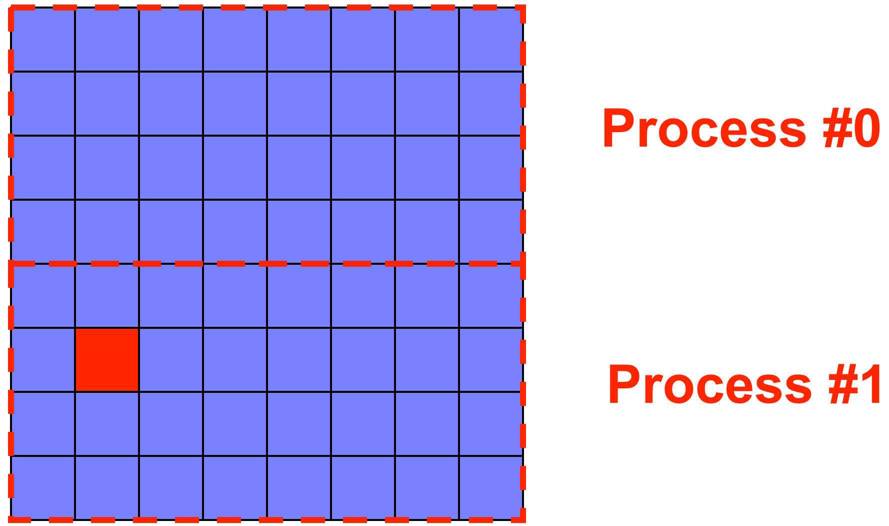
Computing the "easy" pixels

```
for (j=0; j<N; j++) {
   if (my_rank() == 0) { // top process can compute an extra row
     new_image[0][j] = f ( original_image[0][j],
                                      original_image[0][j-1], original_image[0][j+1],
                                      original_image[1][j] );
   }
   if (my_rank() == num_processes()-1) { // bottom process can compute
                                                             // an extra row
     new_image[M-1][j] = f ( original_image[M-1][j],
                                         original_image[M-1][j-1], original_image[M-1][j+1],
                                         original_image[M-2][j] );
   }
   for (i=1; i<M-1; i++) // Everybody computes the "middle" M-2 rows
      new_image[i][j] =  f ( original_image[i][j],
                                      original_image[i+1][j], original_image[i-1][j],
                                      original_image[i][j-1], original_image[i][j+1] );
}
```

Global/Local Index

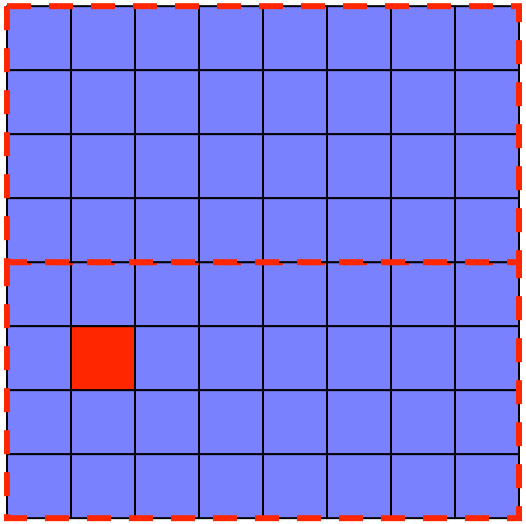
- One of the reason why distributed memory programming is difficult is because of the discrepancy between “global” and “local” indices
- When I think “globally” of the whole image, I know where pixel at coordinates (100,100) is
- But when I write the code, I will not reference the pixel as image[100][100]!
- Let’s look at this on an example

Global/Local Index



- The red pixel's global coordinates are (5,1)
  - The pixel on the 6th row and the 2nd column of the big array
- But when Process #1 references it, it must use coordinates (1,1)
  - The pixel on the 2nd row and the 2nd column of the tile that's stored in Process #1

Global/Local Index



Process #0

Process #1

```
// Shared-Memory
double array[8][8];
array[5][1] = 12;
```

```
// Distributed-Memory
double array[4][8];
array[1][1] = 12;
```

Message Passing

- Let’s assume that we have a send() function that takes as argument
  - The rank of the destination process
  - An address in local memory
  - A size (in bytes)
- Let’s assume that we have a recv() function that takes as argument
  - An address in local memory
  - A size (in bytes)

A Process' Memory

Sending/Receiving Pixels

```
double buffer_top[N], buffer_bottom[N];


if (my_rank() != 0) {      // receive from above neighbor
  send(my_rank()-1,&(original_image[0][0]),sizeof(double)*N);
  recv(buffer_top, sizeof(double)*N);
}
if (my_rank() != num_processes()-1) {      // receive from below neighbor
  send(my_rank()+1, &(original_image[M-1][0]), sizeof(double)*N);
  recv(buffer_bottom, sizeof(double)*N);
}

// assumes "non-blocking" sending
```

Computing Remaining Pixels

```
if (my_rank() != 0) {     // update top pixels
     for (j=0; j<N; j++) {
         new_image[i][j] =  f (  original_image[i][j],
                                 original_image[i+1][j], buffer_top[0][j],
                                 original_image[i][j-1], original_image[i][j+1] );
    }
}


if (my_rank() != N-1) {     // update bottom pixels
     for (j=0; j<N; j++) {
         new_image[i][j] =  f (  original_image[i][j],
                                 buffer_bottom[0][j], original_image[i+1][j],
                                 original_image[i][j-1], original_image[i][j+1] );
    }
}
```

We’re done!

- At this point, we have written the whole code
- What’s missing is I/O:
  - Read the image in
  - Write the image out
- Dealing with I/O (efficiently) is a difficult problem, and we won’t really talk about it in depth
- And of course we need to use a tool that provides the my_rank(), the num_processors(), the send() and the recv() functions
- Each process allocates 1xN + 1xN + 2(M/P)xN = (2M/P+2)N pixels, where P is the number of processors
- Therefore, the total number of pixels allocated is: 2MN + 2NP
  - 2NP extra pixels allocated than in the sequential version
  - But it’s insignificant when spread across multiple systems

The Full Code

Distributed Memory

```
int main() {
  int i, j, M = N/num_processes();  // assumed to be integer!
  int original_image[M][N], new_image[M][N];
  double buffer_top[M], buffer_bottom[M];
  for (j=0; i<N; j++) {
     if (my_rank() == 0) { // top process can compute an extra row
       new_image[0][j] = f ( original_image[0][j], original_image[0][j-1],
original_image[0][j+1], original_image[1][j] );
     }
     if (my_rank() == num_processes()-1) { // bottom process can compute an
extra row
       new_image[M-1][j] = f ( original_image[M-1][j], original_image[M-1][j-1],
original_image[M-1][j+1], original_image[M-2][j] );
     }
     for (i=1; i<M-1; i++) // Everybody computes the “middle” M-2 rows
        new_image[i][j] =  f ( original_image[i][j], original_image[i+1][j],
original_image[i-1][j], original_image[i][j-1], original_image[i][j+1] );
  }
  if (my_rank() != 0) {      // receive from above neighbor
    send(my_rank()-1,&(original_image[0][0]),sizeof(double)*N);
    recv(buffer_top, sizeof(double)*N);
  }
  if (my_rank() != num_processes()-1) {      // receive from below neighbor
    send(my_rank()+1, &(original_image[M-1][0]), sizeof(double)*N);
    recv(buffer_bottom, sizeof(double)*N);
  }
  if (my_rank() != 0) {     // update top pixels
    for (j=0; j<N; j++) {
       new_image[i][j] =  f (  original_image[i][j], original_image[i+1][j], buffer_top[0]
[j], original_image[i][j-1], original_image[i][j+1] );
     }
  }
  if (my_rank() != N-1) {     // update bottom pixels
    for (j=0; j<N; j++) {
       new_image[i][j] =  f (  original_image[i][j], buffer_bottom[0][j],
original_image[i+1][j], original_image[i][j-1], original_image[i][j+1] );
    }
  }
}
```

Sequential

```
int main() {
  int i, j;
  int original_image[N][N], new_image[N][N];
  for (i=1; I<M-1; i++)
     for (j=1; j < M-1; j++)
        new_image[i][j] =  f ( original_image[i][j], original_image[i+1]
[j], original_image[i-1][j], original_image[i][j-1], original_image[i]
[j+1] );
}
```

The Full Code

Distributed Memory

```
int main() {
  int i, j, M = N/num_processes();  // assumed to be integer!
  int original_image[M][N], new_image[M][N];
  double buffer_top[M], buffer_bottom[M];
  for (j=0; i<N; j++) {
     if (my_rank() == 0) { // top proc      an compute an extra row
       new_image[0][j] = f ( origin          0][j], original_image[0][j-1],
original_image[0][j+1], origin
     }
     if (my_rank() == num                          m process can compute an
extra row
       new_image[M                                 original_image[M-1][j-1],
original_image[M
     }
     for (i=1;                                 s the "middle" M-2 rows
        new                                   [j], original_image[i+1][j],
origina                                      1], original_image[i][j+1] );
  }
                                     rom above neighbor
                                mage[0][0]),sizeof(double)*N);
                               ble)*N);

                              ocesses()-1) {      // receive from below neighbor
                         &(original_image[M-1][0]), sizeof(double)*N);
                         n, sizeof(double)*N);

                    = 0) {      // update top pixels
                  N; j++) {
               mage[i][j] =  f (  original_image[i][j], original_image[i+1][j], buffer_top[0]
             nal_image[i][j-1], original_image[i][j+1] );


  if (my_rank() != N-1) {     // update bottom pixels
    for (j=0; j<N; j++) {
      new_image[i][j] =  f (  original_image[i][j], buffer_bottom[0][j],
original_image[i+1][j], original_image[i][j-1], original_image[i][j+1] );
    }
  }
}
```

Sequential

```
int main() {
  int i, j;
  int original_image[N][N], new_image[N][N]
  for (i=1; i<M-1; i++)
     for (j=1; j < M-1; j++)
        new_image[i][j] =  f ( original_image[i][j], origi
[j], original_image[i-1][j], original_image[i][j-1], origina
[j+1] );
}
```

Plus #pragma omp around for loops everywhere to use multiple cores

Too hard?

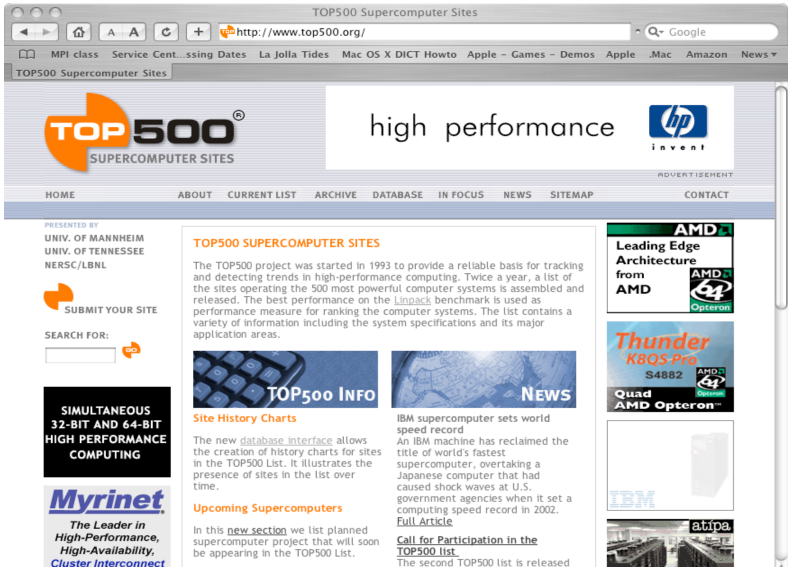
- Clearly the previous example is a bit scary
- Many researchers in academia and industry are trying to make this better
  - Tons of libraries written by smart people so that you don't have to be
  - New languages / compilers
  - New programming models
    - Map-Reduce anyone?
  - New ways to think of applications

Distributed-Memory Computing

- Bottom-line: Distributed-Memory computing is not easy, but it's the only way to scale many applications
- As a result"parallel computing platforms" have been built for many decades
  - So-called "supercomputers"
- The main idea:
  - Get a bunch of individual systems (commodity computers, or cool custom computers)
  - Get a network (commodity switches, cool custom interconnects)
  - Install software to make it possible to write/run program
  - and off we go....

A host of parallel machines

- There are (have been) many kinds of parallel machines
- For the last 12 years their performance has been measured and recorded with the LINPACK benchmark, as part of Top500
- It is a good source of information about what machines are and how they have evolved
- Note that it's really about "supercomputers"



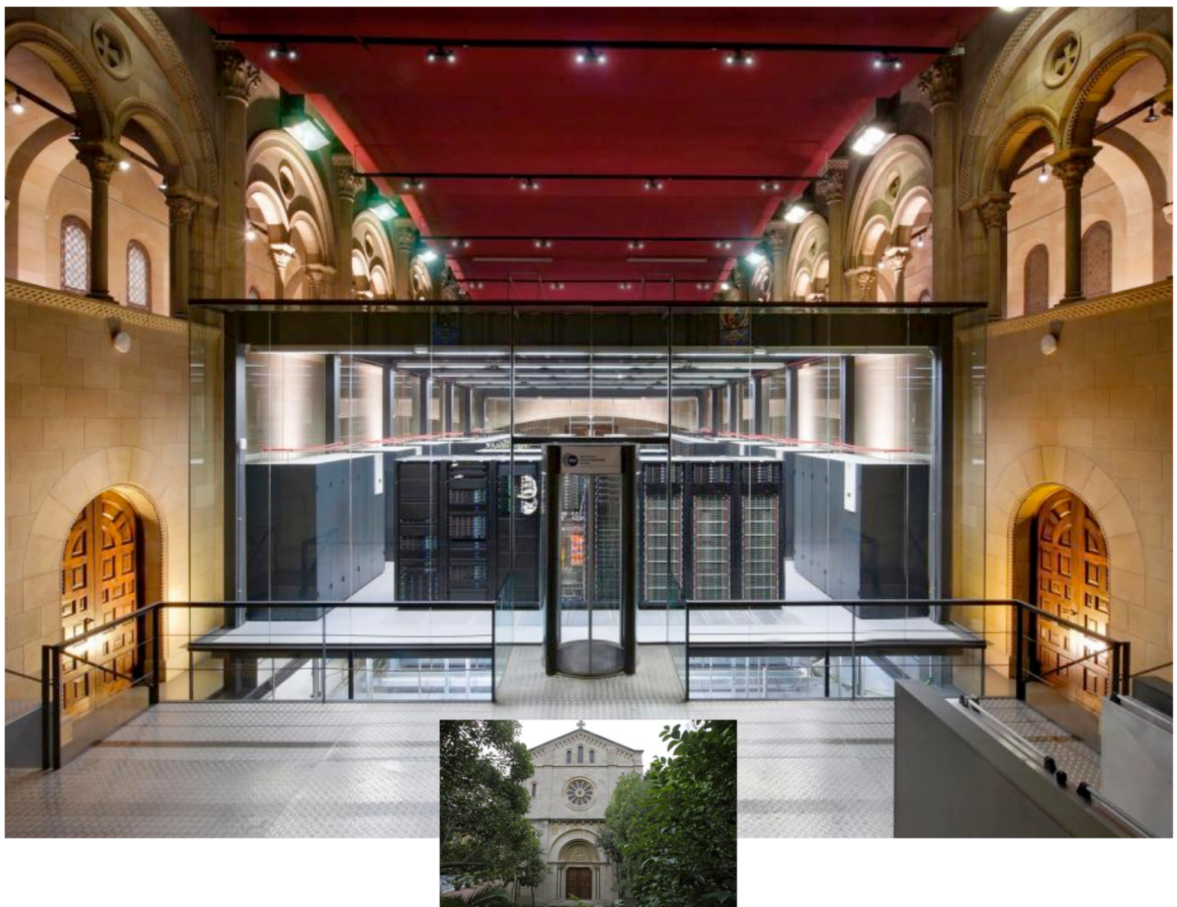
http://www.top500.org

What is Beowulf?

- An experiment in parallel computing systems
- Established <u>vision</u> of low cost, high end computing, with public domain software (and led to software development)
- Tutorials and book for best practice on how to build such platforms
- Today by Beowulf cluster one means a commodity cluster that runs Linux and GNU-type software
- Project initiated by T. Sterling and D. Becker at NASA in 1994

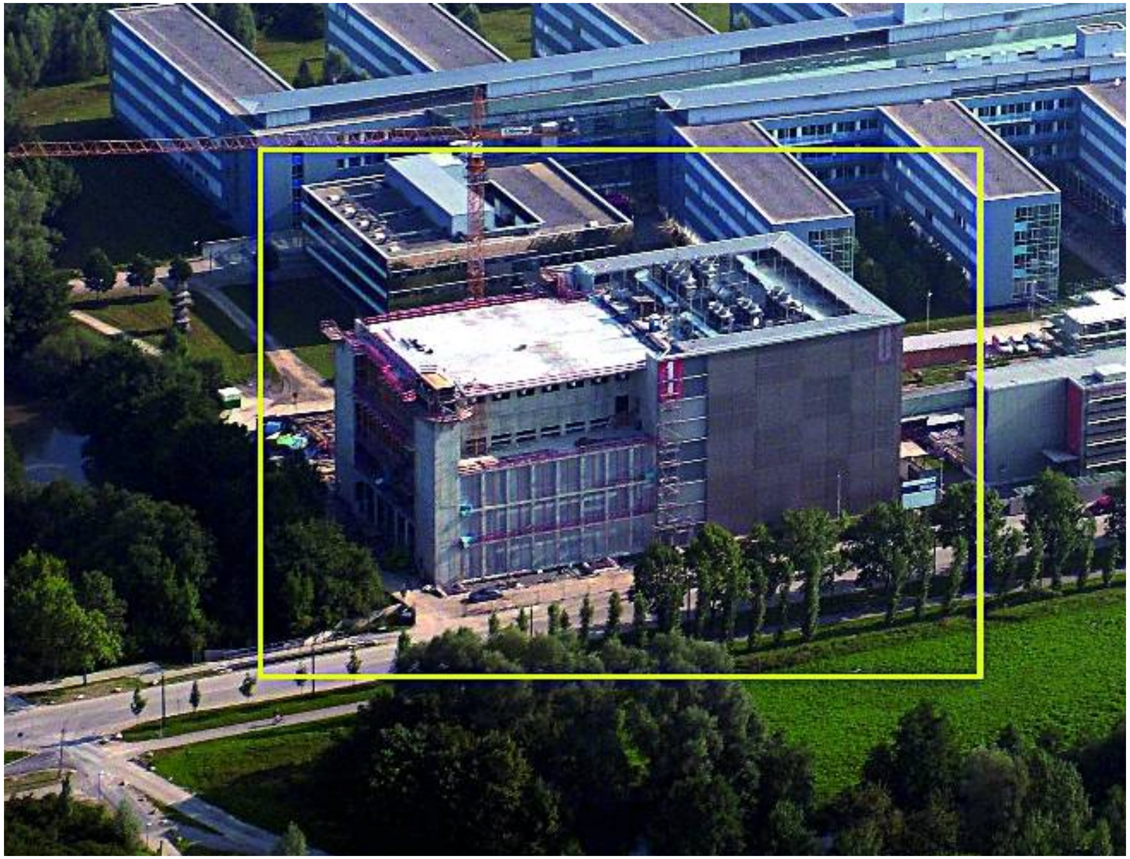
The Prettiest Supercomputer?

http://degiorgi.math.hr/~vsego/phun/beautiful_supercomputer/



River-Water Cooled Supercomputer

- http://www.research.ibm.com/articles/superMUC.shtml



Conclusion

- Writing distributed memory code is much more complex than shared memory code
  - One must identify what must be communicated
  - One must keep a mental picture of the memory across systems
  - In addition to all the concerns we have mentioned in class
    - e.g., cache reuse, synchronization among threads
  - And the typical problems of shared memory are still there
    - There can be “communication” deadlocks, race conditions, etc.
- Big “supercomputers” are amazing and expensive machines with a long and politically/economically-charged history
- Almost all of you will write some type of distributed-memory application (not necessarily High-Performance Computing, but using the same concepts)
- If you’re into all this, take ICS632