Today's Outline

- Motivation for Inter-machine Parallelism
- Inter-machine parallelism hardware
  - Supercomputing
  - Distributed computing
  - Grid computing
  - Cluster computing
- Inter-machine parallelism examples
  - Message Passing Interface (MPI)
  - Google's MapReduce paradigm
  - Programming Challenges

Example Applications

- Science & Medicine
  - Global climate modeling
    - Biogeochemical process models
  - Astrophysical modeling
  - Computational Chemistry, Material Sciences and Nanosciences
  - Computational Genomics
- Engineering
  - Nanodevice design
  - Earthquake and structural modeling
  - Fluid dynamics, heat transfer
  - Combustion engine design
  - Crash simulation
  - Computational Game Theory (e.g., Chess Databases)
- Business
  - Rendering computer graphics/imagery (EIS, oil finance, and GM)
  - Financial and economic modeling
  - Transaction processing, web services and search engines
- Defense
  - Nuclear weapons -- test by simulations

Performance Requirements

- Performance terminology
  - FLOPS = floating point operation
  - "FLOPS" = # FLOP/second is the standard metric for computing power

Example: Global Climate Modeling

- Divide the world into a grid (e.g., 10 km spacing)
- Solve fluid dynamics equations for each point & minute
- Requires about 100 FLOPs per grid point per minute
- Weather Prediction (7 days in 24 hours): 56 GFLOPs
- Climate Prediction (50 years in 30 days): 4.8 TFLOPS

Perspective

- Pentium 4 3.0GHz Desktop Processor
  - ~10 GFLOPs
  - Climate Prediction would take ~50 years

Review

- Threads can be awake and ready/running on a core or asleep for sync (or blocking I/O)
- Use PThreads to thread C & fully utilize your multicore processors!
  - pthread_create(), pthread_join(), pthread_exit()
  - pthread_mutex_t, pthread_mutex_lock(), pthread_mutex_unlock()
  - pthread_cond_t, pthread_cond_wait(), pthread_cond_broadcast()
- Domain decomposition is a common technique for multithreading programs
- Watch out for
  - Synchronization overhead
  - Cache issues (sharing data, decomposing)

Amulah's Law & algorithm parallelizability
en.wikipedia.org/wiki/Image:AmdahlsLaw.svg

Performance Requirements

- Performance terminology
  - FLOPS = floating point operation
  - “FLOPS” = # FLOP/second is the standard metric for computing power

Example: Global Climate Modeling

- Divide the world into a grid (e.g., 10 km spacing)
- Solve fluid dynamics equations for each point & minute
- Requires about 100 FLOPs per grid point per minute
- Weather Prediction (7 days in 24 hours): 56 GFLOPs
- Climate Prediction (50 years in 30 days): 4.8 TFLOPS

Perspective

- Pentium 4 3.0GHz Desktop Processor
  - ~10 GFLOPs
  - Climate Prediction would take ~50 years

References:
- http://fah-web.stanford.edu/cgi-bin/main.py?Qtype=osstats

Example Applications

- Science & Medicine
  - Global climate modeling
    - Biogeochemical process models
  - Astrophysical modeling
  - Computational Chemistry, Material Sciences and Nanosciences
- Engineering
  - Nanodevice design
  - Earthquake and structural modeling
  - Fluid dynamics, heat transfer
- Business
  - Rendering computer graphics/imagery (EIS, oil finance, and GM)
  - Financial and economic modeling
  - Transaction processing, web services and search engines
- Defense
  - Nuclear weapons -- test by simulations

Performance Requirements

- Performance terminology
  - FLOPS = floating point operation
  - “FLOPS” = # FLOP/second is the standard metric for computing power

Example: Global Climate Modeling

- Divide the world into a grid (e.g., 10 km spacing)
- Solve fluid dynamics equations for each point & minute
- Requires about 100 FLOPs per grid point per minute
- Weather Prediction (7 days in 24 hours): 56 GFLOPs
- Climate Prediction (50 years in 30 days): 4.8 TFLOPS

Perspective

- Pentium 4 3.0GHz Desktop Processor
  - ~10 GFLOPs
  - Climate Prediction would take ~50 years

References:
- http://fah-web.stanford.edu/cgi-bin/main.py?Qtype=osstats
High Resolution Climate Modeling on NERSC-3
P. Duffy, et al., LLNL

Supercomputing – like those listed in top500.org
- Multiple processors "all in one box / room" from one vendor that often communicate through shared memory
- This is where you find exotic architectures

Distributed computing
- Many separate computers (each with independent CPU, RAM, HD, NIC) that communicate through a network
  - Grids (heterogeneous computers across Internet)
  - Clusters (mostly homogeneous computers all in one room)
  - Google uses commodity computers to exploit "knee in curve" price/performance sweet spot
- It’s about being able to solve "big" problems, not "small" problems faster
  - These problems can be data (mostly) or CPU intensive

Distributed Computing Themes
- Let’s network many disparate machines into one compute cluster
- These could all be the same (easier) or very different machines (harder)
- Common themes
  - "Dispatcher" gives jobs & collects results
  - "Workers" (get, process, return) until done
- Examples
  - SETI@Home, BOINC, Render farms
  - Google clusters running MapReduce

Distributed Computing Challenges
- Communication is fundamental difficulty
  - Distributing data, updating shared resource, communicating results
  - Machines have separate memories, so no usual inter-process communication – need network
  - Introduces inefficiencies: overhead, waiting, etc.
- Need to parallelize algorithms
  - Must look at problems from parallel standpoint
  - Tightly coupled problems require frequent communication (more of the slow part!)
  - We want to decouple the problem
    - Increase data locality
    - Balance the workload

Programming Models: What is MPI?
- Message Passing Interface (MPI)
  - World’s most popular distributed API
  - MPI is “de facto standard” in scientific computing
  - C and FORTRAN, ver. 2 in 1997
  - What is MPI good for?
    - Abstracts away common network communications
    - Allows lots of control without bookkeeping
    - Freedom and flexibility come with complexity
    - 300 subroutines, but serious programs with fewer than 10
  - Basics:
    - One executable run on every node
    - Each node process has a rank ID number assigned
    - Call API functions to send messages
  - Challenges with MPI
    - Deadlock is possible…
      - Seen in CS61A – state of no progress
      - Blocking communication can cause deadlock
        - "crossed" calls when trading information
        - example
          - Proc1 MPI_Recv(Proc2, A); MPI_Send(Proc2, B); Proc2 MPI_Recv(Proc1, B); MPI_Send(Proc1, A);
urt some solutions - MPI_Sendrecv()
    - Large overhead from comm. mismanagement
      - Time spent blocking is wasted cycles
      - Can overlap computation with non-blocking comm.
    - Load imbalance is possible! Dead machines?
      - Things are starting to look hard to code!
A New Hope: Google’s MapReduce

- Remember CS6417:

  ```java
  (reduce + (map square ((1 2 3)) =>
  (reduce + (1 4 9) =>
  14
  ```

- We told you “the beauty of pure functional programming is that it’s easily parallelizable”
  - Do you see how you could parallelize this?
  - What if the reduce function argument were associative, would that help?
  - Imagine 10,000 machines ready to help you compute anything you could cast as a MapReduce problem!
  - This is the abstraction Google is famous for authoring
    - But their reduce not the same as the CS6417 or MPIs reduce!
    - Often, our reduce builds a reverse-lookup table for easy query
    - It hides lots of difficulty of writing parallel code!
    - The system takes care of load balancing, dead machines, etc.

MapReduce Programming Model

Input & Output: each a set of key/value pairs
Programmer specifies two functions:

`map (in_key, in_value) =>
  list(out_key, intermediate_value)`
- Processes input key/value pair
- Produces set of intermediate pairs

`reduce (out_key, list(intermediate_value)) =>
  list(out_value)`
- Combines all intermediate values for a particular key
- Produces a set of merged output values (usu just one)

code.google.com/edu/parallel/mapreduce-tutorial.html

MapReduce WordCount Example

- “Mapper” nodes are responsible for the `map` function
  ```java
  // "I do I learn" => (("I",1), ("do",1), ("I",1), ("learn",1))
  map(String input_key,
  String input_value;
  // input_key : document name (or line of text)
  // input_value : document contents
  for each word w in input_value:
    EmitIntermediate(w, 1);"
  ```

- “Reducer” nodes are responsible for the `reduce` function
  ```java
  reduce(String output_key, Iterator intermediate_values;
  // output_key : a word
  // output_values : a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));"
  ```

Data on a distributed file system (DFS)

MapReduce WordCount Java code

```java
public static class WordCountMapper
  implements Mapper<String, Text, IntWritable, IntWritable>

    public void map(Object key, Text value, Context context)
    
    try
    
    String line = value.toString();
    String[] words = line.split(" ");
    for (String word : words)
      context.write(new Text(word),
      new IntWritable(1));
    
    ```

```java
public static class WordCountReducer
  implements Reducer<IntWritable, IntWritable, IntWritable, IntWritable>

    public void reduce(IntWritable key, Iterable<IntWritable>
    values, Context context)
    
    int result = 0;
    for (IntWritable value : values)
      result += value.get();
    context.write(key, new IntWritable(result));
    
```
MapReduce in CS61A (and CS3?!) 

- Think that’s too much code?
  - So did we, and we wanted to teach the Map/Reduce programming paradigm in CS61A
  - “We” = Dan, Brian Harvey and ace undergrads Matt Johnson, Ramesh Sridharan, Robert Liao, Alex Rasmussen
  - Google & Intel gave us the cluster you used in Lab!
- You live in Scheme, and send the task to the cluster in the basement by invoking the fn mapreduce. Ans comes back as a stream.
  - (mapreduce mapper reducer reducer-base input)
  - www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-34.html

Our Scheme MapReduce interface

Peer Instruction

1. Writing & managing SETI@Home is relatively straightforward, just hand out & gather data
2. Most parallel programs that, when run on N (N big) identical supercomputer processors will yield close to N x performance increase
3. The majority of the world’s computing power lives in supercomputer centers

Peer Instruction Answer

1. The heterogeneity of the machines, handling machines that fail, faulty data. FALSE
2. The combination of Amdahl’s law, overhead, and load balancing take its toll. FALSE
3. Have you considered how many PCs + game devices exist? Not even close. FALSE

Summary

- Parallelism is necessary
  - It looks like it’s the future of computing...
  - It is unlikely that serial computing will ever catch up with parallel computing
- Software parallelism
  - Grids and clusters, networked computers
  - Two common ways to program:
    - Message Passing Interface (lower level)
    - MapReduce (higher level, more constrained)
- Parallelism is often difficult
  - Speedup is limited by serial portion of code and communication overhead
Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation.

To Learn More...

- About MPI...
  - www.mpi-forum.org
  - Parallel Programming in C with MPI and OpenMP by Michael J. Quinn
- About MapReduce...
  - code.google.com/edu/parallel/mapreduce-tutorial.html
  - labs.google.com/papers/mapreduce.html
  - lucene.apache.org/hadoop/index.html

Basic MPI Functions (1)

- **MPI_Send()** and **MPI_Receive()**
  - Basic API calls to send and receive data point-to-point based on rank (the runtime node ID #)
  - We don’t have to worry about networking details
  - A few are available: blocking and non-blocking
- **MPI_Broadcast()**
  - One-to-many communication of data
  - Everyone calls: one sends, others block to receive
- **MPI_Barrier()**
  - Blocks when called, waits for everyone to call (arrive at some determined point in the code)
  - Synchronization

Basic MPI Functions (2)

- **MPI_Scatter()**
  - Partitions an array that exists on a single node
  - Distributes partitions to other nodes in rank order
- **MPI_Gather()**
  - Collects array pieces back to single node (in order)

Basic MPI Functions (3)

- **MPI_Reduce()**
  - Perform a “reduction operation” across nodes to yield a value on a single node
  - Similar to *accumulate* in Scheme
    - (accumulate + '(1 2 3 4 5))
  - MPI can be clever about the reduction
    - Pre-defined reduction operations, or make your own (and abstract datatypes)
    - MPI_Op_create()
- **MPI_AllToAll()**
  - Update shared data resource

MPI Program Template

- Communicators - set up node groups
- Startup/Shutdown Functions
  - Set up rank and size, pass argc and argv
- “Real” code segment

```c
main(int argc, char *argv[]){
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  /* Data distribution */ ... */
  /* Computation & Communication */ ... */
  MPI_Finalize();
}
```