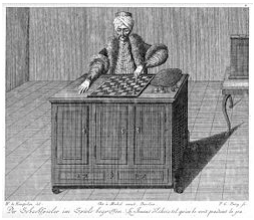
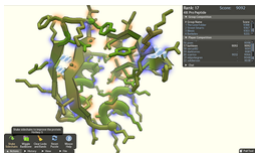
inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #27 – Parallelism in Software
2008-8-06





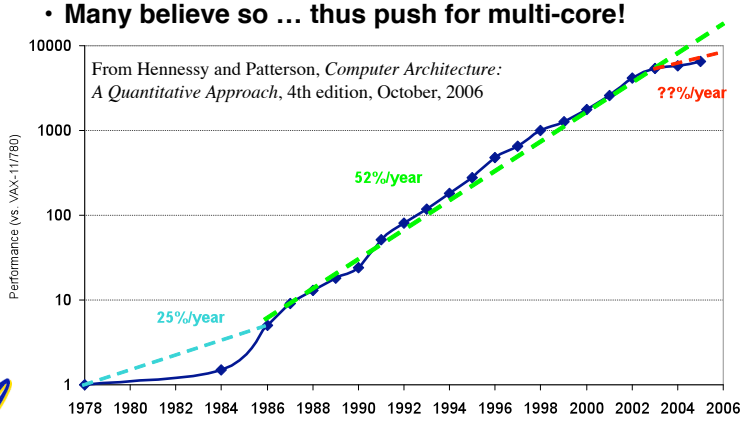
http://fold.it/portal/adobe_main

Amazon Mechanical Turk
http://www.mturk.com/mturk/welcome

Albert Chae, Instructor

What Can We Do?

- **Wait for our machines to get faster?**
  - **Moore's law tells us things are getting better; why not stall for the moment?**
- **Moore on last legs!**
  - **Many believe so … thus push for multi-core!**



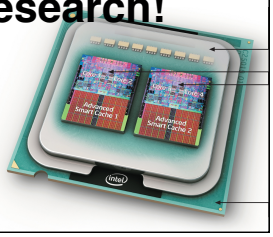
The Future of Parallelism

"Parallelism is the biggest challenge since high level programming languages. It's the biggest thing in 50 years because industry is betting its future that parallel programming will be useful."

– David Patterson

Review: Multicore everywhere!

- Multicore processors are taking over, *many*core is coming
- The processor is the "new transistor"
- This is a "sea change" for HW designers and especially for programmers
- Berkeley has world-leading research! (RAD Lab, Par Lab, etc.)



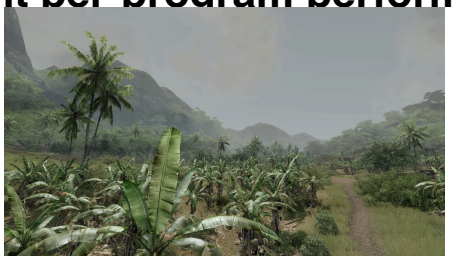
Outline for First Half of today

- Motivation and definitions
- Synchronization constructs and PThread syntax
- Multithreading pattern: domain decomposition
- Speedup issues
  - Overhead
  - Caches
  - Amdahl's Law

How can we harness (many I multi)core?

- Is it good enough to just have multiple programs running simultaneously?

- We want per-program performance gains!

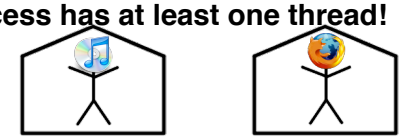


Crysis, Crytek 2007

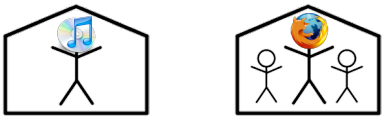
- The leading solution: *threads*

Definitions: threads v.s. processes

- A *process* is a "program" with its own address space.
  - A process has at least one thread!



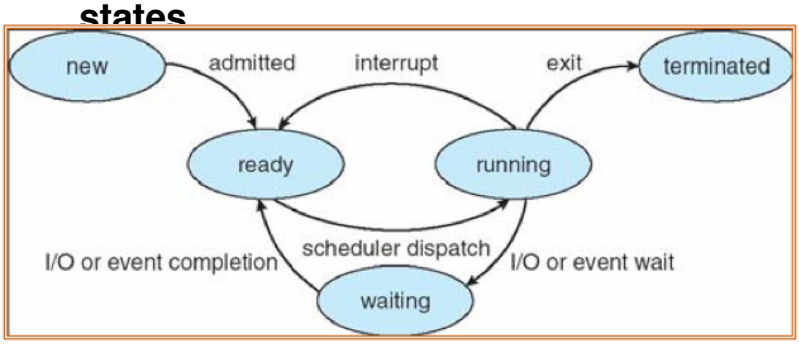
- A *thread of execution* is an independent sequential computational task with its own control flow, stack, registers, etc.
  - There can be many threads in the same process sharing the same address space



  - There are several APIs for threads in several languages. We will cover the PThread API in C.

How are threads *scheduled?*

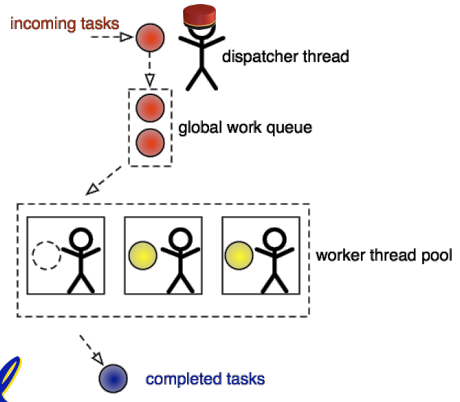
- Threads/processes are run sequentially on one core or simultaneously on multiple cores
  - The operating system schedules threads and processes by moving them between states



From Prof. Kubiatowicz's CS 162, originally from Silberschatz, Galvin, and Gagne

Side: threading without multicore?

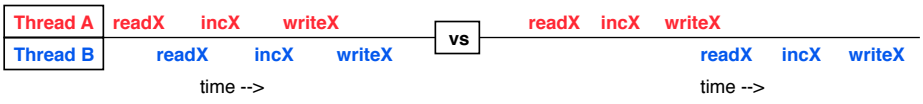
- Is threading useful without multicore?
  - Yes, because of I/O blocking!
- Canonical web server example:




```
global workQueue;

dispatcher() {
  createThreadPool();
  while(true) {
    task = receiveTask();
    if (task != NULL) {
      workQueue.add(task);
      workQueue.wake();
    }
  }
}

worker() {
  while(true) {
    task = workQueue.get();
    doWorkWithIO(task);
  }
}
```


How can we make threads cooperate?

- If task can be completely decoupled into independent sub-tasks, cooperation required is minimal
  - Starting and stopping communication
- Trouble when they need to share data!
- Race conditions:



- We need to force some serialization
  - Synchronization constructs do that!

Lock / mutex semantics

- A *lock* (mutual exclusion, mutex) guards a *critical section* in code so that only one thread at a time runs its corresponding section
  - *acquire* a lock before entering crit. section
  - *releases* the lock when exiting crit. section
  - Threads share locks, one per section to synchronize
- If a thread tries to acquire an in-use lock, that thread is put to sleep
  - When the lock is released, the thread wakes up *with the lock*! (blocking call)

Lock / mutex syntax example in PThreads

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int x;

threadA() {
  int temp = foo(x);
  pthread_mutex_lock(&lock);
  x = bar(x) + temp;
  pthread_mutex_unlock(&lock);
  // continue…
}
```

```
threadB() {
  int temp = foo(9000);
  pthread_mutex_lock(&lock);
  baz(x) + bar(x);
  x *= temp;
  pthread_mutex_unlock(&lock);
  // continue…
}
```

Thread A	readX	…	acquireLock => SLEEP				WAKE w/ LOCK	…	releaseLock
Thread B	…	acquireLock	readX	readX	writeX	releaseLock	…		

time -->

- But locks don't solve everything…
  - And there can be problems: deadlock!

```
threadA() {
  pthread_mutex_lock(&lock1);
  pthread_mutex_lock(&lock2);
}
```

```
threadB() {
  pthread_mutex_lock(&lock2);
  pthread_mutex_lock(&lock1);
}
```

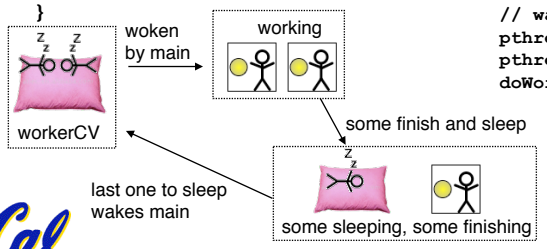
Condition variable semantics

- *A condition variable* (CV) is an object that threads can sleep on and be woken from
  - *Wait* or *sleep* on a CV
  - *Signal* a thread sleeping on a CV to wake
  - *Broadcast* all threads sleeping on a CV to wake
  - I like to think of them as thread pillows…
- *Always* associated with a lock!
  - Acquire a lock before touching a CV
  - Sleeping on a CV releases the lock in the thread's sleep
  - If a thread wakes from a CV it will have the lock
- Multiple CVs often share the same lock

Condition variable example in PThreads

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mainCV = PTHREAD_COND_INITIALIZER;
pthread_cond_t workerCV = PTHREAD_COND_INITIALIZER;
int A[1000];
int num_workers_waiting  = 0;

mainThread() {
  pthread_mutex_lock(&lock);
  // set up workers so they sleep on workerCV
  loadImageData(&A);
  while(true) {
    pthread_cond_broadcast(&workerCV);
    pthread_cond_wait(&mainCV,&lock);
    // A has been processed by workers!
    displayOnScreen(A);
  }
}
```

```
workerThreads() {
while(true) {
   pthread_mutex_lock(&lock);
   num_workers_waiting += 1;
   // if we are the last ones here…
   if(num_workers_waiting == NUM_THREADS){
      num_workers_waiting = 0;
      pthread_cond_signal(&mainCV);
   }
   // wait for main to wake us up
   pthread_cond_wait(&workerCV, &lock);
   pthread_mutex_unlock(&lock);
   doWork(mySection(A));}}
```



Creating and destroying PThreads

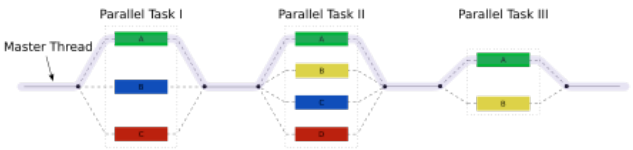
```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5
pthread_t threads[NUM_THREADS];

int main(void) {
  for(int ii = 0; ii < NUM_THREADS; ii+=1) {
    (void) pthread_create(&threads[ii], NULL, threadFunc, (void *) ii);
  }

  for(int ii = 0; ii < NUM_THREADS; ii+=1) {
    pthread_join(threads[ii],NULL); // blocks until thread ii has exited
  }

  return 0;
}

void *threadFunc(void *id) {
  printf("Hi from thread %d!\n",(int) id);
  pthread_exit(NULL);
}
```

To compile against the PThread library, use gcc's -lpthread flag!

Side: OpenMP is a common alternative!

- PThreads aren't the only game in town
- OpenMP can automatically parallelize loops and do other cool, less-manual stuff!

```
#define N 100000
int main(int argc, char *argv[]){
   int i, a[N];
   #pragma omp parallel for
   for (i=0;i<N;i++)
      a[i]= 2*i;
   return 0;
}
```



Domain decomposition

- *Domain decomposition* refers to solving a problem in a data-parallel way
  - If processing elements of a big array can be done independently, divide the array into sections (domains) and assign one thread to each!
  - (Common data parallelism in Scheme?)

Speedup issues: overhead

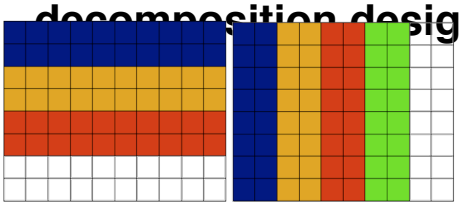
More threads does not always mean better!

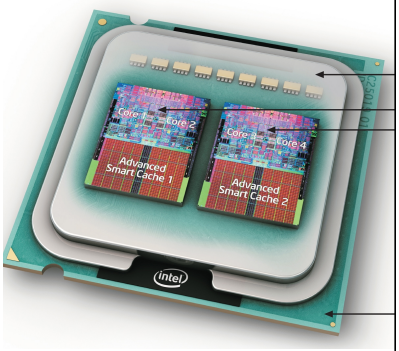
- With more threads, too much time could be spent on *synchronizing* (e.g. waiting on locks and condition variables)

- Synchronization is a form of overhead
  - Also communication and creation/ deletion overhead

Speedup issues: caches

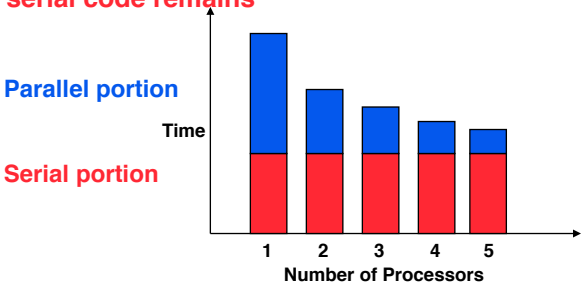
- Caches are often one of the largest considerations in performance
- For multicore, common to have independent L1 caches and shared L2 caches
- Can drive domain decomposition design





Speedup Issues: Amdahl's Law

- Applications can almost never be completely parallelized; some serial code remains



- s is serial fraction of program, P is # of processors
- Amdahl's law:

Speedup(P) = Time(1) / Time(P)

≤ 1 / (s + ((1-s) / P)), and as P → ∞

≤ 1/s

- Even if the parallel portion of your application speeds up perfectly

Peer Instruction

- Multicore is hard for architecture people, but pretty easy for software
- Multicore made it possible for Google to search the web

Peer Instruction Answers!

- Multicore is hard for architecture people, but pretty easy for software
  False: parallel processors put the burden of concurrency largely on the SW side
- Multicore made it possible for Google to search the web
  False: web search and other Google problems have huge amounts of data. The performance bottleneck becomes RAM amounts and speeds! (CPU-RAM gap)

Administrivia

- **Proj3 due TODAY 8/6**
  - Face to face grading?
- Proj4 out soon. Find a partner.
- Final 8/14 – 9:30-12:30pm in 105 North Gate

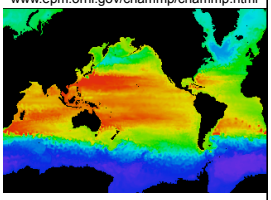
Big Problems Show Need for Parallel

- Simulation: the Third Pillar of Science
  - Traditionally perform experiments or build systems
  - Limitations to standard approach:
    - Too difficult – build large wind tunnels
    - Too expensive – build disposable jet
    - Too slow – wait for climate or galactic evolution
    - Too dangerous – weapons, drug design
  - Computational Science:
    - Simulate the phenomenon on computers
    - Based on physical laws and efficient numerical methods
- Search engines needs to build an index for the entire Internet
- Pixar needs to render movies
- Desire to go green and use less power
- Intel, Microsoft, Apple, Dell, etc. would like to sell you a new computer next year

Performance Requirements

- Performance terminology
  - the FLOP: FLoating point OPeration
  - Computing power in FLOPS (FLOP per Second)
- 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 3GHz Desktop Processor
    - ~6-12 Gflops
  - Climate Prediction would take ~50-100 years

www.epm.ornl.gov/chammp/chammp.html



Reference: http://www.hpcwire.com/hpcwire/hpcwireWWW/04/0827/108259.html

What Can We Do? Use Many CPUs!

- 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 (heterogenous 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

Challenges with MPI

- Deadlock is possible…
  - Blocking communication can cause deadlock
    - "crossed" calls when trading information
    - example:
      - Proc1: `MPI_Receive(Proc2, A); MPI_Send(Proc2, B);`
      - Proc2: `MPI_Receive(Proc1, B); MPI_Send(Proc1, A);`
    - There are 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 CS61A?

```
(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 CS61A's or MPI's reduce)
    - Builds a reverse-lookup table
  - 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)

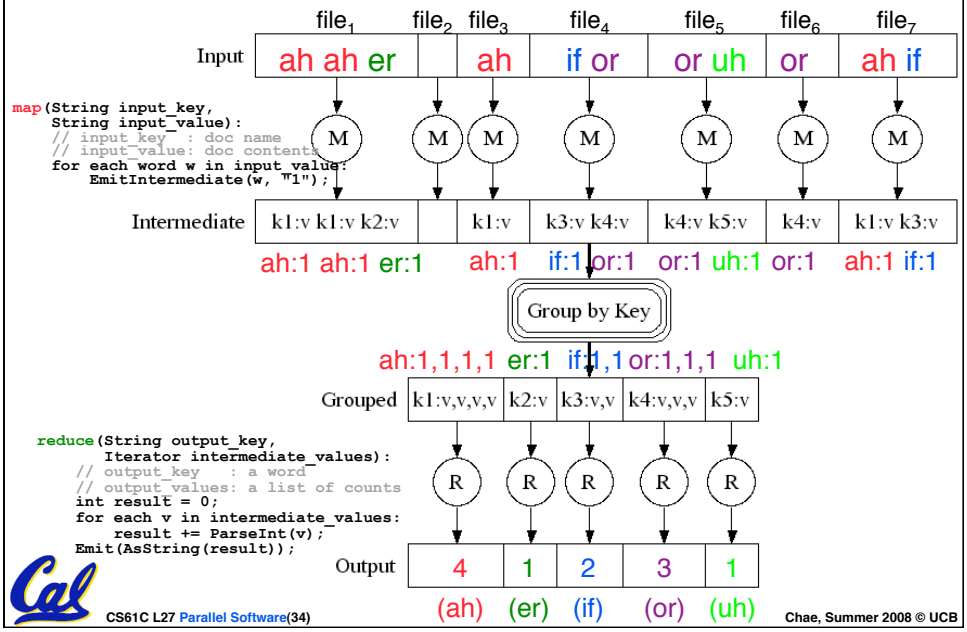
code.google.com/edu/parallel/mapreduce-tutorial.html

MapReduce Code Example

```
map(String input_key,
    String input_value):
    // input_key  : document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

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));
```

- "Mapper" nodes are responsible for the `map` function
- "Reducer" nodes are responsible for the `reduce` function
- Data on a distributed file system (DFS)

MapReduce Example Diagram



MapReduce Advantages/Disadvantages

- Now it's easy to program for many CPUs
  - Communication management effectively gone
    - I/O scheduling done for us
  - Fault tolerance, monitoring
    - machine failures, suddenly-slow machines, other issues are handled
  - Can be much easier to design and program!
- But… it further restricts solvable problems
  - Might be hard to express some problems in a MapReduce framework
  - Data parallelism is key
    - Need to be able to break up a problem by data chunks
  - MapReduce is closed-source – Hadoop!

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

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT

Peer Instruction Answer

1. The heterogeneity of the machines, handling machines that fail, falsify 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

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

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT

Summary

- Threads can be *awake* and ready/running on a core or *asleep* for sync. (or blocking I/O)
- Use PThreads to thread C code and use your multicore processors to their full extent!
  - `pthread_create()`, `pthread_join()`, `pthread_exit()`
  - `pthread_mutex_t`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`
  - `pthread_cond_t`, `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`
- Domain decomposition is a common technique for multithreading programs
- Watch out for
  - Synchronization overhead
  - Cache issues (for sharing data, decomposing)
  - Amdahl's Law and algorithm parallelizability

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

Bonus

To Learn More…

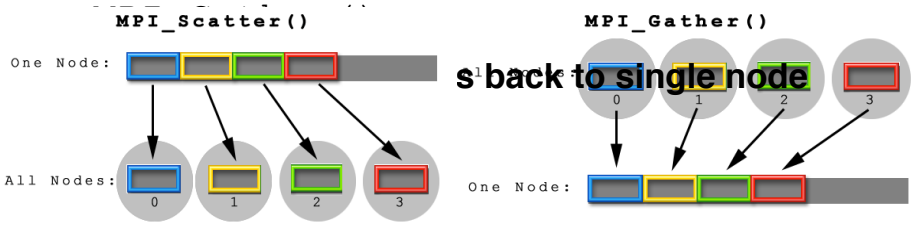
- **About MPI…**
  - `www.mpi-forum.org`
  - **Parallel Programming in C with MPI and OpenMP by Michael J. Quinn**
- **About MapReduce…**
  - `code.google.com/edu/parallel/mapreduce-tutorial.html`
  - `labs.google.com/papers/mapreduce.html`
  - `lucene.apache.org/hadoop/index.html`

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**



s back to single node

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**

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