Parallel Programming Models

- Fundamental question: what is the “right” way to write parallel programs
  - And deal with the complexity of finding parallelism, coarsening granularity, distributing computation and data, synchronizing, optimizing, etc.
    - Oh, and we want to achieve high performance
    - And make the program portable across different architectures

Parallel Programming Models (from "Multithreaded, Parallel, and Distributed Programming" by Andrews)

		Additional Models
Shared Variables		
Ada	Protected types	
Cilk	Fork/join	OpenMP
Java	Synchronized methods	
SR	Fork/join and semaphores	
Message Passing		
Ada	Rendezvous	
CSP/Occam	Synchronous message passing	
Fortran M	Asynchronous message passing	MPI
Java	Network and remote invocation packages	
SR	Message passing, RPC, rendezvous	
Coordination		
Linda	Tuple space and message-like primitives	
Orca	Data objects and remote operations	
Data Parallel		
C*	C and data layout and parallel execution	
HPF	Data mappings, array statements, reductions	UPC, Chapel, X10
NESL	Nested data parallelism	
ZPL	Data regions and directions, array operations	
Functional		
NESL	Recursive parallelism	MapReduce
Sisal	Iterative (for all) and recursive parallelism	
Abstract Models		
BSP	Bulk synchronous message transfer	
LogP	Distributed-memory processors	
PRAM	Parallel random access to shared variables	

Figure 12.7 Languages and models for parallel programming.

OpenMP

- Add *annotations* to a sequential program, target is multicore machines
  - Language independent---implementations exist in C, C++, Fortran
  - Programmer does not add library calls (whereas the MPI programmer does add them)
- Programmer is still responsible for finding parallelism
  - It’s just easier to use than pthreads
- Modern OpenMP supports GPUs as a target

Parallelizing with OpenMP

- Most common: parallel directive
  - Can be a (1) parallel for loop or (2) task parallelism

```
int N = 100000; int i, a[N];
#pragma omp parallel for shared(a) private(i)
	for (i = 0; i < N; i++)
		a[i] = 2 * i;
```

- Implicit barrier at end of the for loop, unless “nowait” specified
- Number of threads can be specified by programmer, or the default will be chosen

Parallelizing with OpenMP

- Most common: parallel directive
  - Can be a (1) parallel for loop or (2) task parallelism

```
#pragma omp parallel sections
{
  #pragma omp parallel section
  {
      f();
  }
 #pragma omp parallel section
  {
      g();
  }
}
```

Parallelizing with OpenMP

- Other constructs: *single*, *master*
  - indicate that one thread should perform a block
    - (former mandates a barrier, latter does not)

Parallelizing with OpenMP

- Data annotations
  - *shared*: visible to all threads
  - *private*: local to a thread
- For shared, OpenMP runtime system will promote the variable to the global scope (think: program 1)
- For private, OpenMP runtime system will push the variable definition into the thread code
  - So that it's on the thread's private stack

Parallelizing with OpenMP

- Synchronization constructs
  - *critical*: a critical section
  - *atomic*: a one-statement critical section, possibly optimized (by an atomic instruction)
  - *barrier*
  - *reduction*: efficiently handles finding the sum, product, max, or min of a shared variable when many threads are updating the shared variable

```
#pragma omp parallel for reduction(+:sum)
for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
```

Parallelizing with OpenMP

- Loop scheduling
  - *static*: equal sized chunk (parameter set by user) per thread
  - *dynamic*: threads can get more work if they finish their assigned work (again, chunk is size of block of iterations assigned to threads)
    - Chunk size must be sufficiently small; if, for example, there are *p* threads and *p* chunks, dynamic is the same as static
  - *guided*: threads get a successively smaller chunk each time they finish their work
    - Goal: minimize overhead and eliminate tail-end load imbalance

Parallelizing with Cilk

- Supports efficiently recursive parallelism
- Classic recursive example is computing Fibonacci numbers (an idiotic implementation but a good example)

```
int fib(int n) {
  if (n < 2)  return 1;
  else {
    x = fib(n-1);
    y = fib(n-2);
    return x + y;
  }
}
```

Parallelizing with Cilk

```
cilk int fib(int n) {
  if (n < 2)  return 1;
  else {
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return x + y;
  }
}
```

Parallelizing with Cilk

- Extremely simple model
  - Can quickly adapt sequential recursive programs
  - But, can only adapt sequential recursive programs
- What does Cilk runtime system have to do?
  - Quite a bit, actually
  - Must deal with:
    - Not creating too much parallelism
    - Efficiently allowing work stealing to happen between processors to avoid load balancing

Parallelizing with Data Parallel Languages

- Example: HPF (High Performance Fortran)
- Idea: data parallel language, plus annotations that determine data distributions
  - Step 3 in developing a parallel program
- If the compiler knows the data distribution, it is straightforward to distribute the work and determine where to insert communication
  - If the rule is "owner computes"
- Challenge:
  - Figuring out the most efficient data distribution isn't easy for the programmer

Example HPF code fragment

```
double A[N,N], B[N,N] {BLOCK, BLOCK}
…
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
     A[i][j] = 0.25 * (B[i][j+1] + B[i][j-1] +
                       B[i+1][j] + B[i-1][j]);
     …
  }
}
```

HPF Compiler Analysis

- From number of processors and distribution annotations, determines what each processor *owns*
- Determines what can legally execute in parallel
  - User could annotate here, conceptually
- Divides up iterations between processors based on data distribution annotations
- Determines necessary communication by looking at array references along with data owned; then, inserts that communication

Partitioned Global Address Space Languages

- Idea: "flat MPI" (one MPI process per core) is not sustainable for many-core machines
  - Copying cost (even though MPI processes will commonly be on same machine)
  - Memory overhead (what about boundary rows as number of MPI processes increases?)
- Possibilities to deal with this
  - MPI + pthreads (your first programming assignment)
    - In general, MPI + (some shared memory model)
  - PGAS model

Partitioned Global Address Space Languages

- Provides a global (shared) address space
  - Any thread can access any data element, just as if one were on a multicore machine
  - **However**, all data is either local or global (not hidden from programmer, unlike NUMA on a multicore machine)
- Specific PGAS example: UPC (others include Chapel and X10)
  - Adds restriction that parallelism is SPMD, with owner-computes
  - User writes a multithreaded program
  - User can declare variables as private (the default) or shared (use shared as little as possible)---similar idea to OpenMP
  - User can give shared variables a distribution, just as with HPF Leads to a hybrid shared/distributed programming model
    - Sits in the middle between a multithreaded model and a message-passing model in terms of programming difficulty and efficiency

Example UPC code fragment

```
shared [*][*] double A[N,N], B[N,N]
…
upc_forall (i = 0; i < N; i++) {
  upc_forall (j = 0; j < N; j++) {
     A[i][j] = 0.25 * (B[i][j+1] + B[i][j-1] +
                        B[i+1][j] + B[i-1][j]);
     …
  }
}
```

Parallelizing with Functional Languages

- It’s “easy”---the user does nothing but write the sequential program
- (Pure) functional languages do not have side effects
  - As soon as the arguments are available to a function, the function can be executed
  - A function can operate only on local data (again, assuming a pure functional language)
  - Therefore, determining what executes in parallel is a trivial problem…but limiting the parallelism (i.e., coarsening granularity) becomes a problem (as in the Fibonacci example)
  - Also, functional languages do not handle data distribution, communication, optimization

Parallelizing with Functional Languages

- One approach was a *single-assignment* language
  - Note that functional languages are basically also single assignment
- Best known was SISAL, developed to be a competitor to Fortran in the 1980s
- Every variable can be written to at most once
  - Can express as: $x = old\ x$, meaning that conceptually there is a new variable on the left-hand side (otherwise you would violate single assignment)

Parallelizing with Functional Languages

- Think about Jacobi; an array is a variable, so every assignment (in the inner loop!) is to a new entire array
  - Compiler must determine when it can **update in place**.

```
double A[N,N]
for (first sweep) {
  A[i][j] = 0.25 * (old B[i][j+1] + old B[i][j-1] +
                    old B[i+1][j] + old B[i-1][j]);
}
for (second sweep) {
  B[i][j] = 0.25 * (old A[i][j+1] + old A[i][j-1] +
                    old A[i+1][j] + old A[i-1][j]);
}
```

Parallelizing with Coordination Languages

- Best known example is Linda
- Not a language itself, but a set of primitives added to any language
  - Similar to OpenMP in that sense, but here we use primitives and not pragmas
- Linda creates a tuple space
  - Shared, associative memory
  - Basically a shared communication channel
- Javaspaces is based on Linda
  - Also, tuple spaces have been developed for Python and Ruby

Parallelizing with Coordination Languages

- Linda primitives
  - OUT (similar to send)
  - IN (similar to receive)
  - RD (similar to a "peek", which means view message but don't receive it; no immediate analogy in MPI)
  - EVAL (similar to fork)
  - INP and RDP (nonblocking versions of IN and RD)

Parallelizing with Coordination Languages

- A tuple has the form (“tag”, value1, …, valueN)
- OUT therefore has the form
  - OUT(“tag”, expr1, …, exprN)
  - Means: put this tuple into the tuple space
- IN has the form
  - IN(“tag”, field1, …, fieldN)
  - Means: block until there is a matching tuple in the tuple space, and then remove the tuple from the tuple space, placing its values into the respective fields
  - field1, …, fieldN must be an l-val, takes the form ?var
- EVAL takes a function to execute, like fork

Parallelizing with Coordination Languages

- Mostly used for "bag of tasks" paradigm
- Takes the following form: each process does…

```
Init: OUT("task", initX, initY, …)
while (not done) {
    IN("task", ?x, ?y, …)
    do work using x, y, …
    if (need to create more work) {
        OUT("task", x1, y1, …)
        OUT("task", x2, y2, …)
    }
}
```

Coordination Languages: Example (Adaptive Quadrature)

```
Init: OUT("quad", a, b)
while (RD("quad", ?x, ?y) {   // terminate in deadlock
    IN("quad", ?a, ?b)
    c = (a+b)/2
    compute area of each half and area of whole
    if (close)
      localSum += area of whole
    else {
        OUT("quad", a, c)
        OUT("quad", c, b)
    }
}
```

This code is executed by each process

Finalization code: do a sum-reduction

Parallelizing with MapReduce

- Allows massive parallelization on large numbers of nodes
- Functional programming model
  - Restricts application domain
  - Map(f, nil) = nil
  - Map(f, (cons(e, L)) = cons(f(e), Map(f, L))
    - Map takes list as input, applies function to each element, and produces a list as output
  - Reduce(f, z, nil) = z
  - Reduce(f, z, cons(e, L)) = f(e, Reduce(f, z, L))
    - Reduce takes a list as input and applies a function to the entire list, producing a value

Parallelizing with MapReduce

- Google's MapReduce involves:
  - First, applying a map function to each logical record in the input
    - Produces a set of intermediate key/value pairs
  - Then, applying a reduce function to all values that have a common key
- Critical that this is a functional model so there are no side effects
  - Will become quite important in the implementation

MapReduce Example (dumb example!)

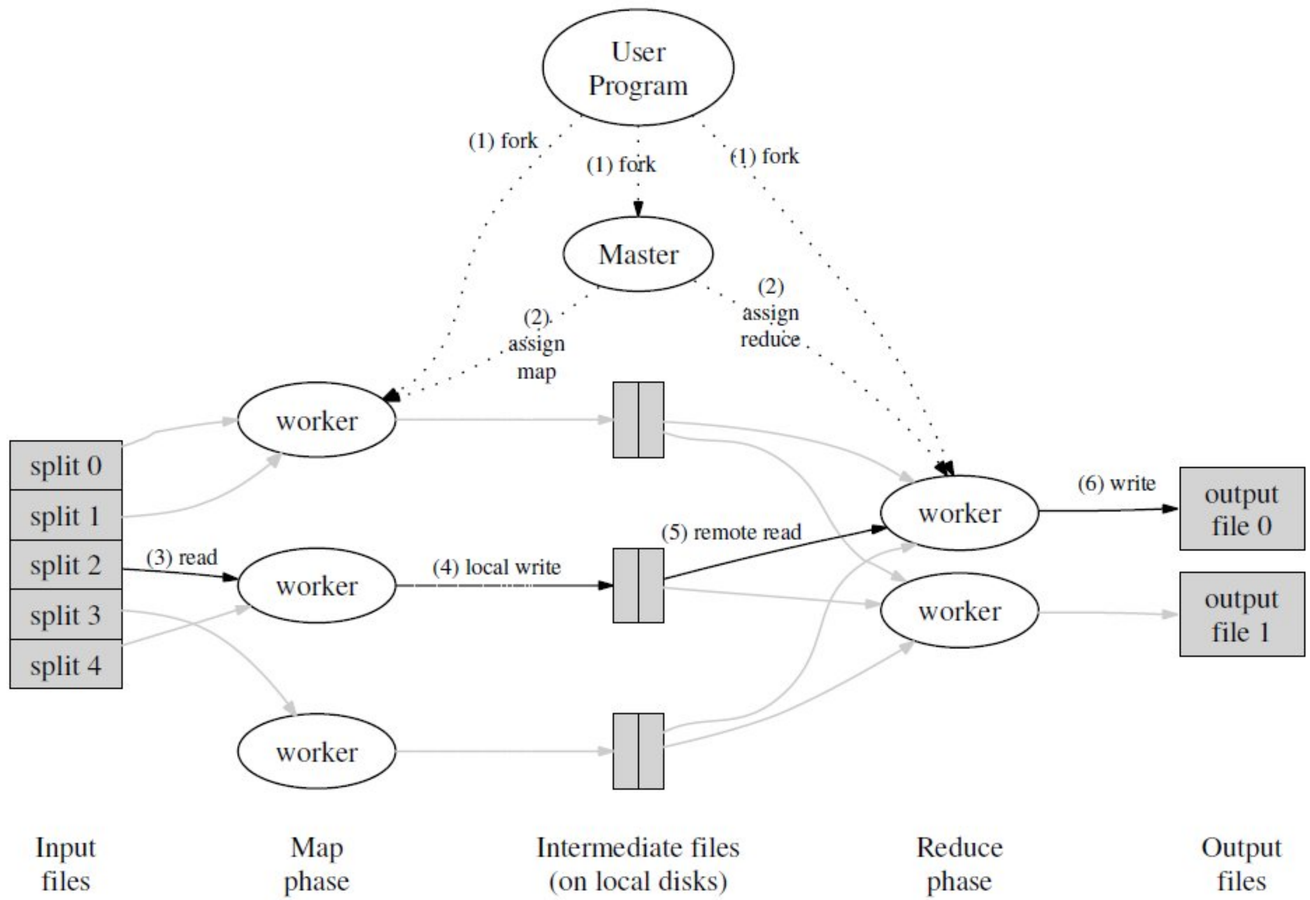
```
Map(String key, String value):
    For each word w in value:
        EmitIntermediate(w, “1”);

Reduce(String key, Iterator values):
    Int result = 0;
    For each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Other MapReduce Examples

- URL Access Frequency (same idea as WordCount)
  - Map outputs <URL, 1> for each URL
  - Reduce adds for same URL
- Reverse Web-Link Graph
  - Map outputs <target, source> for each link to a target URL found in source
  - Reduce concatenates source URLs for each target
- Inverted Index
  - Map outputs <word, documentID>
  - Reduce outputs <word, list(documentID)>

MapReduce Implementation
(Picture from MapReduce paper [Dean and Ghemawat])



MapReduce Implementation

- Basic idea (for clusters of commodity machines):
  - 1. Split input files into M chunks
  - 2. Split reduce tasks into R pieces (hash on key)
  - 3. Use Master/Worker paradigm; one master
    - Master assigns M map tasks and R reduce tasks to workers
  - 4. Workers doing a map task write key/value lists into different files per R piece
    - If the key “indicates” the i'th reduce task, write to file "i".
    - Pass back this file info to master, who tells reduce tasks

MapReduce Implementation

- 5. Reduce worker grabs its data from all local disks, sorts, and reduces.
  - Sorting occurs because may be many keys assigned to each reduce task
  - One call to user's reduce function per unique key; parameter is list of all values for that key
  - Appends output into final output file.
- 6. When everything done, wake up MapReduce call.

Fault Tolerance with MapReduce

- Critical to handle fault tolerance because there will be thousands of machines involved
  - Lessons learned from RAID
- Master keeps track of each map and reduce task
  - Marks it as idle, in progress, or completed
  - Master assumed to never fail (could checkpoint to alleviate this problem)

Fault Tolerance with MapReduce

- For workers:
  - Ping periodically; if no response mark worker as "failed"; mark worker's task as idle
  - Completed map tasks are re-executed because the worker's local disk is assumed inaccessible
  - Notify reduce tasks if a map task changes hands, so reduce tasks know where to read from

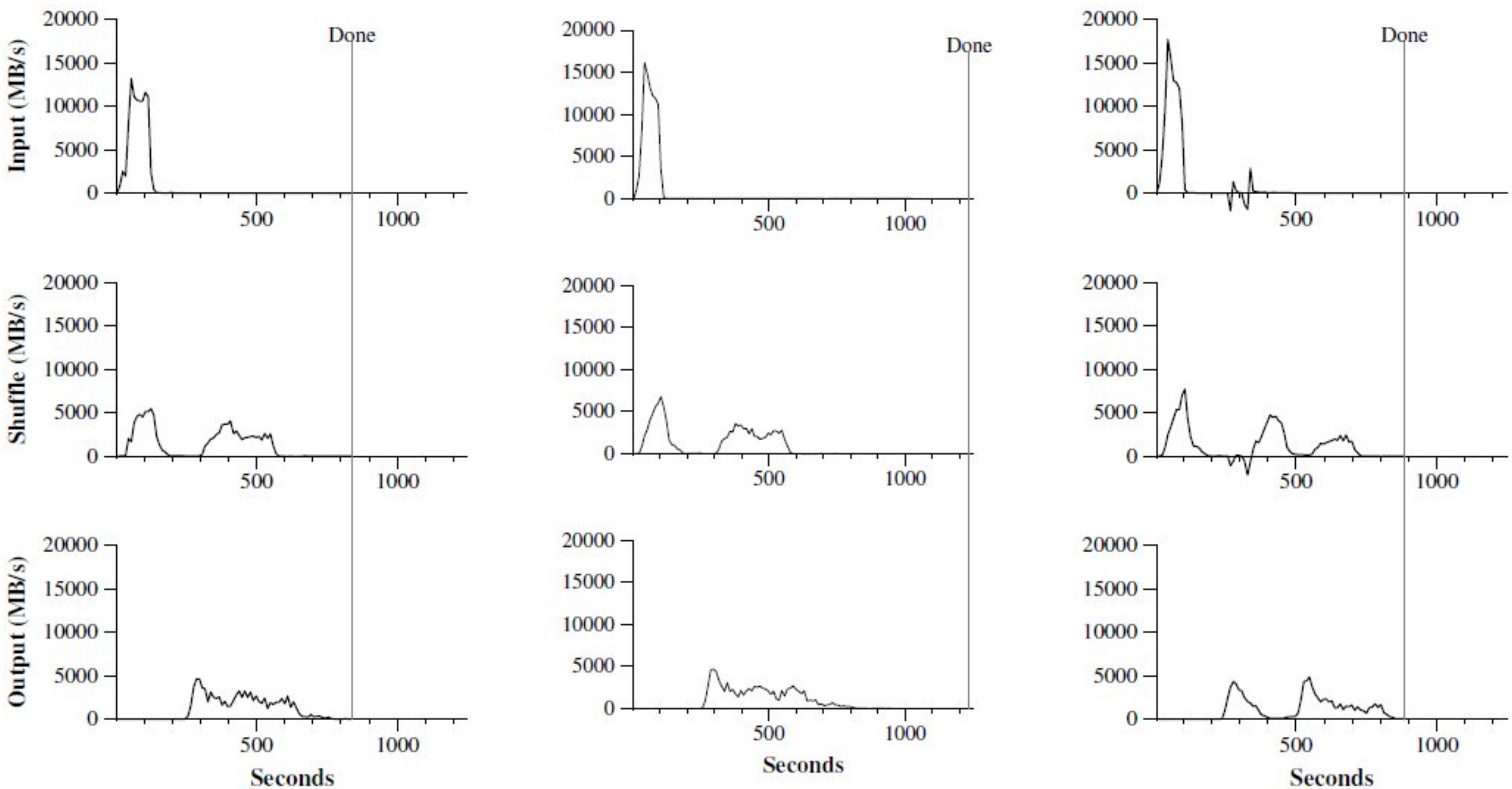
Fault Tolerance with MapReduce

- Fault tolerance is simple because MapReduce is a functional model
  - Can have duplicate tasks, for example---no side effects
    - Very important to removing tail-end load imbalance of reduce workers
  - Depends on atomic commits of map and reduce task outputs
    - Map task sends completion message after writing R files; includes names in message (to master)
    - Reduce task writes to temp file, then renames to final output file---depends on atomic rename operation from file system in case many machines execute rename on same reduce task

MapReduce "Optimizations"

- There are a host of these
  - Many of them are completely obvious
    - E.g., "Combiner function" to avoid sending large amounts of easily reducible data
    - For WordCount, there will be many ("the", 1) records; may as well just add them all up locally
    - Also can "roll your own" partitioning function
  - All "optimizations" affect only performance, not correctness

MapReduce Performance



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed