Compsci 590.3: Introduction to Parallel Computing

Alvin R. Lebeck

Slides based on this from the University of Oregon
Admin

Logistics
• Homework #3
  ▪ Use script…
• Project Proposals
  ▪ Document: see web site
    » Due 10am Wed 10/7
  ▪ Lightning talk
    » 10-11 Groups
    » One person per group
    » 5 minutes / group
    » 2-3 slides, no more!
    » Send to me PPT by Tuesday @ 8pm

• No Office Hours Tues 10/6
• Office Hours Wed 10/7 3-4pm

Outline
• What is speedup?
• What does a speedup graph look like?
• MIC performance
Fork-Join Pattern

- Control flow divides (forks) into multiple flows, then combines (joins) later
- During a fork, one flow of control becomes two
- Separate flows are “independent”
  - Does “independent” mean “not dependent”?
  - No, it just means that the 2 flows of control “are not constrained to do similar computation”
- During a join, two flows become one, and only this one flow continues
  - The “master”
Fork-Join Pattern

- Fork-Join directed graph:

Independent work

Is it possible for B() and C() to have dependencies between them?
Fork-Join Pattern

- Typical **divide-and-conquer** algorithm implemented with fork-join:

```c
void DivideAndConquer( Problem P ) {
    if( P is base case ) {
        Solve P;
    } else {
        Divide P into K subproblems;
        Fork to conquer each subproblem in parallel;
        Join;
        Combine subsolutions into final solution;
    }
}
```
Fork-Join Pattern for Divide-Conquer
Fork-Join Pattern for Divide-Conquer

\[ K = 2 \quad (\text{2-way fork-join}) \]

\[ N = 3 \quad (\text{3 levels of fork-join}) \]
Fork-Join Pattern for Divide-Conquer

$2^3 = 8$-way parallelism
Fork-Join Pattern

• Selecting the base case size is critical
• Recursion must go deep enough for plenty of parallelism
• Too deep, and the granularity of sub-problems will be dominated by scheduling overhead
• With K-way fork-join and N levels of fork-join, can have up to KN-way parallelism
Fibonacci Example

• Recursive Fibonacci is simple and inefficient

```c
long fib ( int n ) {
    if (n < 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

• Are there dependencies between the sub-calls?
• Can we parallelize it?
Fibonacci in Parallel Example

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

How do we actually do a fork and join?
Programming Model Support for Fork-Join

- Cilk Plus:

```
cilk_spawn B(); — Fork
C();
cilk_sync; ———— Join
```

- B() executes in the child thread
- C() executes in the parent thread
Parallel Fibonacci in Cilk / Cilk Plus

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

© Alvin R. Lebeck based on slides from UOregon
Programming Model Support for Fork-Join

- Cilk Plus:

```c
cilk_spawn B();
cilk_spawn C();
/* nil */
cilk_sync;
```

Bad form! Why?
Programming Model Support for Fork-Join

- Cilk Plus:

```c
cilk_spawn B();
C();
cilk_sync;
```

```c
---------
cilk_spawn A();
cilk_spawn B();
cilk_spawn C();
D();   // Not spawned, executed in spawning task
cilk_sync;   // Join
---------
for ( int i=0; i<n; ++i )
    if ( a[i]!=0 )
        cilk_spawn f(a[i]);
cilk_sync;
```
Programming Model Support for Fork-Join

• TBB
  ▪ parallel_invoke()
    » For 2 to N way fork
    » Joins all tasks before returning
  ▪ Tbb::task_group
    » For more complicated cases
    » Provides explicit join

```cpp
task_group g;
for ( int i=0; i<n; ++i )
  if ( a[i] != 0 )
    g.run( [=,&a]{f(a[i]);} ); // Spawn f(a[i]) as child task
  g.wait(); // Wait for all tasks spawned from g
```
Programming Model Support for Fork-Join

- OpenMP:

```
#pragma omp task
B();
C();
#pragma omp taskwait
```

Forked task
Performed by spawning task
Programming Model Support for Fork-Join

• OpenMP:

```c
#pragma omp task
B();
C();
#pragma omp taskwait
```

Forked task can also be a compound statement:

```c
{B(); C(); D();}
```
Programming Model Support for Fork-Join

- OpenMP:

```c
#pragma omp parallel

#pragma omp task
B();
C();
#pragma omp taskwait
```

Must be enclosed in an OpenMP parallel construct
More to the OpenMP Fork-Join Story

- OpenMP uses a fork-join model of parallel execution as a fundamental basis of the language.
- All OpenMP programs begin as a single process:
  - Master thread executes until a parallel region is encountered.
- OpenMP runtime systems executes the parallel region by forking a team of (Worker) parallel threads:
  - Statements in parallel region are executed by worker threads.
- Team threads join with master at parallel region end.

© Alvin R. Lebeck based on slides from UOregon
OpenMP – General Rules

• Most OpenMP constructs are compiler directives
• Directives inform the compiler
  ▪ Provide compiler with knowledge
  ▪ Usage assumptions
• Directives are ignored by non-OpenMP compilers!
  ▪ Essentially act as comment for backward compatibility
• Most OpenMP constructs apply to structured blocks
  ▪ A block of code with one point of entry at the top and one point of exit at the bottom
  ▪ Loops are a common example of structured blocks
    » excellent source of parallelism
Coarser-Grain Parallelism

What’s going on here? Is this possible? When?

Is this better? Why?
SECTIONS Directive

- C/C++

```c
#pragma omp sections [clause [clause …]]
{
    [#pragma omp section]
    block
    …
}
```
SECTIONS Directive: Details

• Sections are assigned to threads
  ▪ Each section executes once
  ▪ Each thread executes zero or more sections

• Sections are not guaranteed to execute in any order

```c
#pragma omp parallel
#pragma omp sections
{
    X_calculation();
    #pragma omp section
    y_calculation();
    #pragma omp section
    z_calculation();
}
```
OpenMP Fork-Join Summary

• OpenMP parallelism is Fork-Join parallelism
• Parallel regions have logical Fork-Join semantics
  ▪ OMP runtime implements a Fork-Join execution model
  ▪ Parallel regions can be nested!!!
    » can create arbitrary Fork-Join structures
• OpenMP tasks are an explicit Fork-Join construct
Recursive Implementation of Map

- **Map** is a simple, useful pattern that fork-join can implement.
- Good to know how to implement map with fork-join if you ever need to write your own map with novel features (fusing map with other patterns).
- Cilk Plus and TBB implement their map constructs with a similar divide-and-conquer algorithm.
Map: Recursive Implementation

- Cilk_for Map
  \[
  \text{cilk\_for ( unsigned int } i = \text{lower; } i < \text{upper; } i++) \\
  \text{f(i);}
  \]

- Can be implemented with a divide and conquer approach
- Conceptually…

  \[
  \text{if (lower < upper)} \\
  \text{recursive\_map(lower, upper);}
  \]
Recursive Implementation of Map

```cpp
1  template<typename Func>
2  void recursive_map( unsigned lower, unsigned upper, unsigned grainsize, Func f ) {
3      if( upper-lower<=grainsize )
4          // Parallel base case
5              for( unsigned i=lower; i<upper; ++i )
6                  f(i);
7      else {
8          // Divide and conquer
9              unsigned middle = lower+(upper-lower)/2u;
10             cilk_spawn recursive_map( lower, middle, grainsize, f );
11             recursive_map( middle, upper, grainsize, f );
12         }
13     // Implicit cilk_sync when function returns
14  }
```
Recursive Implementation of Map

- recursive_map(0,9,2,f)
Choosing Base Cases (1)

• For parallel divide-and-conquer, two base cases:
  - Stopping parallel recursion
  - Stopping serial recursion

• For a machine with P hardware threads, we might think to have P leaves in the spawned functions tree

• This often leads to poor performance
  - Scheduler has no flexibility to balance load
Choosing Base Cases (2)

• Given leaves from spawned function tree with equal work, and equivalent processors, system effects can effect load balance:
  § Page faults
  § Cache misses
  § Interrupts
  § I/O
• Best to over-decompose a problem
• This creates parallel slack
Choosing Base Cases (3)

• **Over-decompose**: parallel programming style where more tasks are specified than there are physical workers. Beneficial in load balancing.

• **Parallel slack**: Amount of extra parallelism available above the minimum necessary to use the parallel hardware resources.
Load Balancing

• Sometimes, threads will finish their work at different rates
• When this happens, some threads may have nothing to do while others may have a lot of work to do
• This is known as a load balancing issue
Sharing Memory Between Tasks

• Two tasks need to increment the same location?
  ▪ E.g., two tasks insert/delete to/from a shared linked list
  ▪ E.g., Two tasks increment same counter

• What do we need to do?
TBB and Cilk Plus Work Stealing (1)

- TBB and Cilk Plus use work stealing to automatically balance fork-join work
- In a work-stealing scheduler, each thread is a worker
- Each worker maintains a stack of tasks
- When a worker’s stack is empty, it grabs from the bottom of another random worker
  - Tasks at the bottom of a stack are from the beginning of the call tree – tend to be a bigger piece of work
  - Stolen work will be distant from stack’s owner, minimizing cache conflicts
• TBB and Cilk Plus work-stealing differences:

Cilk Plus

TBB

Steal continuation

Steal child
Performance of Fork/Join

• Let A||B be interpreted as “fork A, do B, and join”
  ▪ Work: $T(A||B)_1 = T(A)_1 + T(B)_1$
  ▪ Span: $T(A||B)_\infty = \max(T(A)_\infty, T(B)_\infty)$
Implementing Scan with Fork-Join

- We saw that the map pattern can be implemented with the fork-join pattern.
- Now we will examine how to implement the scan operation with fork-join.
- Input: initial value, initial, and sequence, \( r_0, r_1, \ldots, r_n \)
- Output: exclusive scan, \( s_0, s_1, \ldots, s_n \)
- **Upsweep** computes a set of partial reductions on tiles of data.
- **Downsweep** computes final scan by combining partial reductions.
Implementing Scan with Fork-Join
Implementing Scan with Fork-Join
Implementing Scan with Fork-Join

- During the upsweep, each node computes a partial reduction of the form:
  
  $$ r_{i:m} = r_{i:k} \bigoplus r_{i+k:m-k} $$
Implementing Scan with Fork-Join

- During the downsweep, each node computes subscans of the form:
  - $s_i = \text{initial} \oplus r_{0:i}$ and $s_{i+k} = s_i \oplus r_{i:k}$
Summary

• Fork Join Pattern
• Create tasks
• Wait for child task completion
• Need a base case
• Chapter 8 in Text has more details and examples