CS4961 Parallel Programming

Lecture 4: Data and Task Parallelism

Mary Hall
September 3, 2009

Administrative

- Homework 2 posted, due September 10 before class
  - Use the "handin" program on the CADE machines
  - Use the following command:

    "handin cs4961 hw2 <prob1file>"
- Mailing list set up: cs4961@list.eng.utah.edu
- Sriram office hours:
  - MEB 3115, Mondays and Wednesdays, 2-3 PM

Going over Homework 1

- Problem 2:
  - Provide pseudocode (as in the book and class notes) for a correct and efficient parallel implementation in C of the parallel sums code, based on the tree-based concept in slides 26 and 27 of Lecture 2. Assume that you have an array of 128 elements and you are using 8 processors.
  - Hints:
    - Use an iterative algorithm similar to count3s, but use the tree structure to accumulate the final result.
    - Use the book to show you how to add threads to what we derived for count3s.
- Problem 3:
  - Now show how the same algorithm can be modified to find the maximum element of an array. (problem 2 in text). Is this also a reduction computation? If so, why?

Homework 2

Problem 1 (#10 in text on p. 111):

The Red/Blue computation simulates two interactive flows. An n x n board is initialized so cells have one of three colors: red, white, and blue, where white is empty, red moves right, and blue moves down. Colors wrap around on the opposite side when reaching the edge.

In the first half step of an iteration, any red color can move right one cell if the cell to the right is unoccupied (white). On the second half step, any blue color can move down one cell if the cell below it is unoccupied. The case where red vacates a cell (first half) and blue moves into it (second half) is okay.

Viewing the board as overlaid with t x t tiles (where t divides n evenly), the computation terminates if any tile's colored squares are more than c% one color. Use Peril-L to write a solution to the Red/Blue computation.

Homework 2, cont.

Problem 2:

For the following task graphs, determine the following:

(1) Maximum degree of concurrency.

(2) Critical path length.

(3) Maximum achievable speedup over one process assuming than an arbitrarily large number of processes is available.

(4) The minimum number of processes needed to obtain the maximum possible speedup.

(5) The maximum achievable speedup if the number of processes is limited to (a) 2 and (b) 8.

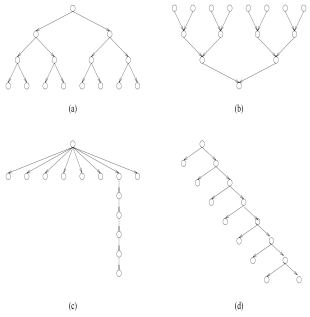


Figure 3.42 Task-dependency graphs for Problem 3.2.

Today's Lecture

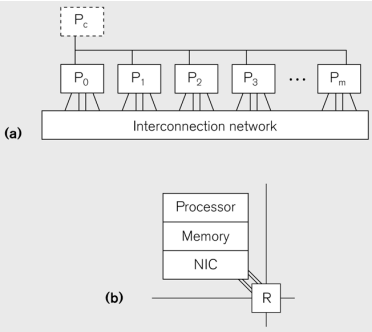
- Data parallel and task parallel constructs and how to express them
- Peril-L syntax
  - An abstract programming model based on C to be used to illustrate concepts
- Task Parallel Concepts
- Sources for this lecture:
  - Larry Snyder, http://www.cs.washington.edu/education/courses/524/08wi/
  - Grama et al., Introduction to Parallel Computing, http://www.cs.umn.edu/~karypis/parbook

Brief Recap of Tuesday's Lecture

- We looked at a lot of different kinds of parallel architectures
  - Diverse!
- How to write software for a moving hardware target?
  - Abstract away specific details
  - Want to write machine-independent code
- Candidate Type Architecture (CTA) Model
  - Captures inherent tradeoffs without details of hardware choices
  - Summary: Locality is Everything!

Candidate Type Architecture (CTA Model)

- A model with *P standard processors, d degree, λ latency*
- Node == processor + memory + NIC
  - All memory is associated with a specific node!
- Key Property: Local memory ref is 1, global memory is λ



Estimated Values for Lambda

- Captures inherent property that data locality is important.
- But different values of Lambda can lead to different algorithm strategies

CMP	AMD	100	Lg λ range => cannot be ignored
SMP	Sun Fire E25K	400-660	
Cluster	Itanium + Myrinet	4100-5100	
Super	BlueGene/L	5000	

Locality Rule

- Definition, p. 53:
  - Fast programs tend to maximize the number of local memory references and minimize the number of non-local memory references.
- Locality Rule in practice
  - It is usually more efficient to add a fair amount of redundant computation to avoid non-local accesses (e.g., random number generator example).

This is the most important thing you need to learn in this class!

Conceptual: CTA for Shared Memory Architectures?

- CTA is not capturing global memory in SMPs
- Forces a discipline
  - Application developer should think about locality even if remote data is referenced identically to local data!
  - Otherwise, performance will suffer
  - Anecdotally, codes written for distributed memory shown to run faster on shared memory architectures than shared memory programs
  - Similarly, GPU codes (which require a partitioning of memory) recently shown to run well on conventional multi-core

Definitions of Data and Task Parallelism

- Data parallel computation:
  - Perform the same operation to different items of data at the same time; the parallelism grows with the size of the data.
- Task parallel computation:
  - Perform distinct computations -- or tasks -- at the same time; with the number of tasks fixed, the parallelism is not scalable.
- Summary
  - Mostly we will study data parallelism in this class
  - Data parallelism facilitates very high speedups; and scaling to supercomputers.
  - Hybrid (mixing of the two) is increasingly common

Parallel Formulation vs. Parallel Algorithm

- Parallel Formulation
  - Refers to a parallelization of a serial algorithm.
- Parallel Algorithm
  - May represent an entirely different algorithm than the one used serially.
- In this course, we primarily focus on "Parallel Formulations".

Steps to Parallel Formulation (refined from Lecture 2)

- Computation Decomposition/Partitioning:
  - Identify pieces of work that can be done concurrently
  - Assign tasks to multiple processors (processes used equivalently)
- Data Decomposition/Partitioning:
  - Decompose input, output and intermediate data across different processors
- Manage Access to shared data and synchronization:
  - coherent view, safe access for input or intermediate data

UNDERLYING PRINCIPLES:

- Maximize concurrency and reduce overheads due to parallelization!
- Maximize potential speedup!

Peril-L Notation

- This week's homework was made more difficult because we didn't have a concrete way of expressing the parallelism features of our code!
- Peril-L, introduced as a neutral language for describing parallel programming constructs
  - Abstracts away details of existing languages
  - Architecture independent
  - Data parallel
  - Based on C, for universality
- We can map other language features to Peril-L features as we learn them

Peril-L Threads

- The basic form of parallelism is a thread
- Threads are specified in the following (data parallel) way:

```
forall <int var> in ( <index range spec> ) {<body> }
```

- Semantics: spawn *k threads each executing body*

```
forall thID in (1..12) {
  printf("Hello, World, from thread %i\n", thID);
}
```

<index range spec> is any reasonable (ordered) naming

Thread Model is Asynchronous (MIMD)

- Threads execute at their own rate
- The execution relationships among threads is not known or predictable
- To cause threads to synchronize, we have

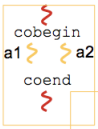
```
barrier;
```

- Threads arriving at barriers suspend execution until all threads in its `forall` arrive there; then they're all released

Common Notions of Task-Parallel Thread Creation (not in Peril-L)

- cobegin/coend

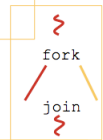
```
cobegin
    job1(a1);
    job2(a2);
coend
```



- Statements in block may run in parallel
- cobegins may be nested
- Scoped, so you cannot have a missing coend

- fork/join

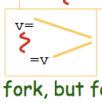
```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```



- Forked procedure runs in parallel
- Wait at join point if it's not finishedC

- future

```
v = future(job1(a1));
… = …v…;
```



- Future expression evaluated in parallel
- Attempt to use return value will wait

- Cobegin cleaner than fork, but fork is more general
- Futures require some compiler (and likely hardware) support

Memory Model in Peril-L

- Two kinds of memory: local and global
  - Variables declared in a thread are local by default
  - Any variable w/ underlined_name is global
- Names (usually indexes) work as usual
  - Local variables use local indexing
  - Global variables use global indexing
- Memory is based on CTA, so performance:
  - Local memory reference are unit time
  - Global memory references take λ time

Memory Read-Write Semantics

- Local Memory behaves like the RAM model
- Global memory
  - Reads are concurrent, so multiple processors can read a memory location at the same time
  - Writes must be exclusive, so only one processor can write a location at a time
  - The possibility of multiple processors writing to a location is not checked and if it happens the result is unpredictable

Shared Memory or Distributed Memory?

- Peril-L is not a shared memory model because
  - It distinguishes between local and global memory costs … that's why it's called "global"
- Peril-L is not distributed memory because
  - The code does not use explicit communication to access remote data
- Peril-L is not a PRAM because
  - It is founded on the CTA
  - By distinguishing between local and global memory, it distinguishes their costs
  - It is asynchronous

Serializing Global Writes

- To ensure the exclusive write Peril-L has

```
exclusive { <body> }
```

- As compared to a lock, this is an atomic block of code.
- The semantics are that
  - a thread can execute <body> only if no other thread is doing so
  - if some thread is executing, then it must wait for access
  - sequencing through exclusive may not be fair

Reductions (and Scans) in Peril-L

- Aggregate operations use APL syntax
  - Reduce: <op>/<operand> for <op> in {+, *, &&, ||, max, min}; as in +/priv_sum
  - Scan: <op>\<operand> for <op> in {+, *, &&, ||, max, min}; as in +\local_finds
- To be portable, use reduce & scan rather than programming them

```
exclusive {count += priv_count; }   WRONG

count = +/priv_count;               RIGHT
```

Reduce/Scan Imply Synchronization

Peril-L Example: Try3 from Lecture 2

```
1  int array[length];                          The data is global
2  int t;                                      Number of desired threads
3  int total=0;                                Result of computation, grand total
4  int lengthPer=ceil(length/t);
5  forall(index in(0..t-1))
6  {
7    int priv_count=0;                         Local accumulation
8    int i, myBase=index*lengthPer;
9    for(i=myBase; i<min(myBase+lengthPer, length); i++)
10   {
11     if(array[i]==3)                         There's no concurrent read since
12     {                                       Array has been partitioned
13       priv_count++;
14     }
15   }
16   exclusive { total+=priv_count; }          Compute grand total
17 }
```

Connecting Global and Local Memory

- CTA model does not have a "global memory". Instead, global data is distributed across local memories
- But #1 thing to learn is importance of locality. Want to be able to place data current thread will use in local memory
- Construct for data partitioning/placement

```
localize();
```

- Meaning
  - Return a reference to portion of global data structure allocated locally (not a copy)
  - Thereafter, modifications to local portion using local name do not incur λ penalty

Scalable Parallel Solution using Localize

```
1  int array[length];                          The data is global
2  int t;                                      Number of desired threads
3  int total;                                  Result of computation, grand total
4  forall(j in(0..t-1))
5  {
6    int size=mySize(array,0);                 Figure size of local part of global data
7    int myData[size]=localize(array[]);
                                               Associate my part of global data with
                                               local variable
8    int i, priv_count=0;                      Local accumulation
9    for(i=0; i<size; i++)
10   {
11     if(myData[i]==3)
12     {
13       priv_count++;
14     }
15   }
16   total =+/priv_count;                      ompute grand total
17 }
```


Summary of Lecture

- How to develop a parallel code
  - Added in data decomposition
- Peril-L syntax
- Next Time:
  - More on task parallelism and task decomposition
  - Peril-L examples
  - Start on SIMD, possibly