CSE524 Parallel Computation

Lawrence Snyder
www.cs.washington.edu/CSEp524

17 April 2007

1

Announcements

- Graded homework returned at end of class
- New homework assigned at end of class
- Thanks to all those who have suggested improvements to the book

We will use a || computer at MS

2

Review Key Points

- Amdahl's Law is a fact but it doesn't impede us much
- Inherently sequential problems (probably) exist, but they don't impede us either
- Latency hiding could hide the impact of λ with sufficiently many threads and much (interconnection) bandwidth
- Impediments to parallel speedup are numerous: overhead, contention, inherently sequential code, waiting time, etc.

Review Key Points (continued)

- Concerns while parallel programming are also numerous: locality, granularity, dependences (both true and false), load balance, etc.
- Happily: Parallel and sequential computers are different: More hardware means more fast memory (cache, RAM), implying the possibility of superlinear speedup
- Measuring improvement is complicated

Peril-*L* …

- A pseudo-language to assist in discussing algorithms and languages
- Don't panic--the name is just a joke
- Goals:
  - Be a minimal notation to describe parallelism
  - Be universal, unbiased towards languages or machines
  - Allow reasoning about performance (using the CTA)

I'm interested how well this works

5

Base Language is C

- Peril-*L* uses C as its notation for scalar computation
- Advantages
  - Well known and familiar
  - Capable of standard operations & bit twiddling
- Disadvantages
  - Low level
  - No goodies like OO

This is not the way to design a || language

6

Threads

- The basic form of parallelism is a thread
- Threads are specified by

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

- Semantics: spawn *k* threads running *body*

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

<index range spec> is any reasonable naming

7

Thread Model is Asynchronous

- 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

8

Memory Model

- Two kinds of memory: local and global
  - All variables declared in a thread are local
  - Any variable w/ name extended by `_G` 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

Notice that the default are local variables

9

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 the result is the last change

In PRAM terminology, this is CREW

10

Example: Try 1

- Shared memory programs are expressible
- The first (erroneous) Count 3s program is

```
int *array_G, length_G, count_G;
forall thID in (0..t-1) {
  int i;
  for (i=0; i < length_G; i++) {
    if (array_G[i] == 3)
      count_G++;
  }
}
```

- Variable usage is now obvious

11

Why Is This Not Shared 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 a PRAM because
  - It is founded on the CTA
  - It distinguishes between local and global memory, and by extension their costs
  - It is asynchronous

These may seem subtle but they matter

12

Getting Global Writes Serialized

- To insure the exclusive write Peril-L has

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

- The semantics are that a thread can execute *<body>* only if no other thread *of its* `forall` is doing so; if some thread is executing, then it must wait for access; sequencing through `exclusive` may not be fair

Exclusive gives behavior, not mechanism

13

Example: Try 4

- The final (correct) Count 3s program

```
int *array_G, length_G, count_G;
forall thID in (0..t-1) {
  int i, priv_count=0; len_per_th=length_G/t;
  int start=thID * len_per_th;
  for (i=start; i<start+len_per_th; i++) {
    if (array_G[i] == 3)
      priv_count++;
  }
  exclusive {count_G += priv_count; }
}
```

Padding is irrelevant … it’s implementation

14

Materialized Memory

- Memory usually works like information:
  - Reading is repeatable w/o “emptying” location
  - Writing is repeatable w/o “filling up” location
- Matter works differently
  - Taking something from location leaves vacuum
  - Placing something in location requires it’s empty
- Materialized Memory: Apply the idea to memory … the presence or absence of the materialized memory helps serializing

Use the **_GO** suffix to identify MM

Treating memory as matter

- A location can be read only if it’s filled
- A location can be written only it’s empty

Location contents	Variable Read	Variable Write
Empty	Stall	Fill w/value
Occupied	Take value	Stall

- Scheduling stalled threads may not be fair

We’ll find uses for this next week

Reduce and Scan

- 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`
- Use reduce & scan rather than programming them to be portable

```
exclusive {count_G += priv_count; }   WRONG
count_G = +/priv_count;               RIGHT
```

Reduce/Scan Imply Synchronization

17

Reduce/Scan and Memory

- When reduce/scan involve local memory

```
priv_count= +/priv_count;
```

  - The local is assigned the global sum
  - This is an implied broadcast

```
priv_count= +\priv_count;
```

  - The local is assigned the prefix sum to that pt
  - No implied broadcast
- Assigning a reduce/scan value to a local forces a barrier, but assigning reduce value to a global does not

18

Peril-*L* Summary

- Peril-*L* is a pseudo-language
- No implementation is implied, though performance is
- Discuss: How efficiently could Peril-*L* run on previously discussed architectures?
  - CMP, SMPbus, SMPx-bar, Cluster, BlueGeneL
  - Features: C, Threads, Memory (G/L/M), /, \

19

Thinking About Parallel Algorithms

- Computations need to be reconceptualized to be effective parallel computations
- Three cases to consider
  - Unlimited parallelism -- issue is grain
  - Constrained ||ism -- issue is performance
  - Scalable parallelism -- get all performance that is *realistic* and *build in flexibility*
- Consider the three as an exercise in
  - Learning Peril-*L*
  - Thinking in parallel and discussing choices

20

The Problem: Alphabetize

- Assume a linear sequence of records to be alphabetized
- Technically, this is parallel sorting, but the full discussion on sorting must wait
- Solutions
  - Unlimited: Odd/Even
  - Constrained: Local Alphabetize
  - Scalable: Ranking (one round)

21

Unlimited Parallelism, Part 1

```
continue_G = true;
while continue_G do {                          Odd/Even
 forall (i in (1:n-1:2))      {                Stride by 2
   done = true;                                Set for term. test
   if (strcoll(L_G[i].x,L_G[i+1].x)>0){Misordered?
       temp = L_G[i];
       L_G[i] = L_G[i+1];
       L_G[i+1] = temp;
       done = false;
   }
  continue_G = !(&&/done);                     Changes ?
}
```

Data is referenced globally

22

Unlimited Parallelism, Part 2

```
  forall (i in (2:n-2:2)){                    Even/Odd
    done = true;                              Set for term. test
    if (strcoll(L_G[i].x,L_G[i+1].x)>0){Misordered?
        temp = L_G[i];
        L_G[i] = L_G[i+1];
        L_G[i+1] = temp;
        done = false;
    }
    continue_G = continue_G && (!(&&/done));Changes ?
  }
}                                             End while
```

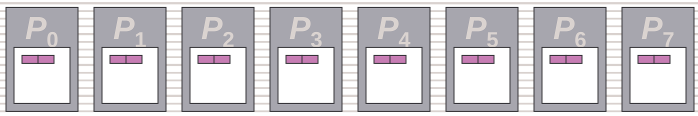
The final round is essentially verification

Reflection on Unlimited Parallelism

- Is solution correct … are writes exclusive?
- What's the effect of process spawning overhead?
- How might this algorithm be executed for *n*=10,000, *P*=1000
- What is the performance?
- Are the properties of this solution clear from the Peril-*L* code?

1 More Problem w/Unlimited Model

- Needing to "multiplex" the threads rather than "enlarge" is not the only problem with unlimited parallelism
- Consider this model



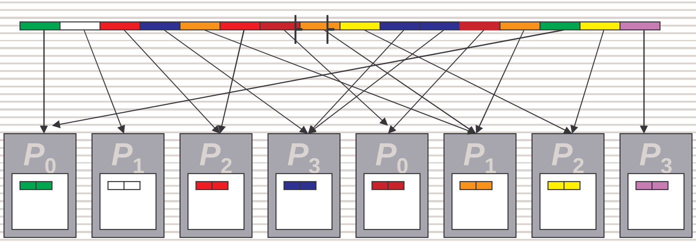
- Imagine data shifts left one item … what's the cost for 100,000 local values?

Generalizing "trivialized" operations is hard

25

Cartoon of Constrained Solution

- Move locally



- Sort
- Return

26

Constrained ||ism, Part 1

- Begin by finding out size of local storage

```
forall (index in (0..25) ) {A thread for each letter
   count = 0;                  Number of records with my letter
   for (i=0; i<n; i++) {       Count number w/each letter to
                                 get size of local storage alloc
      if (index == letRank(charAt(L_G[i].x,0))){
         count += 1 };
      }
   }
   Temp = new DBrecord array [count];Allocate local storage
   j = 0;
```

27

Constrained ||ism, Part 2

- Grab records, sort them and return

```
  for (i=0; i<n; i++) {          Move records locally
     if (index == letRank(charAt(L_G[i].x,0)) ) {
         Temp[j++] = L_G[i];
     }
  }
  startPt = +\count;              Scan counts # records ahead
  alphabetizeInPlace(Temp);       Alphabetize within this letter
  j = startPt - count;            Find my starting index
  for (i=0; i<count; i++) {       Move back to original memory
     L_G[j++] = Temp [i];
  }
}
```

28

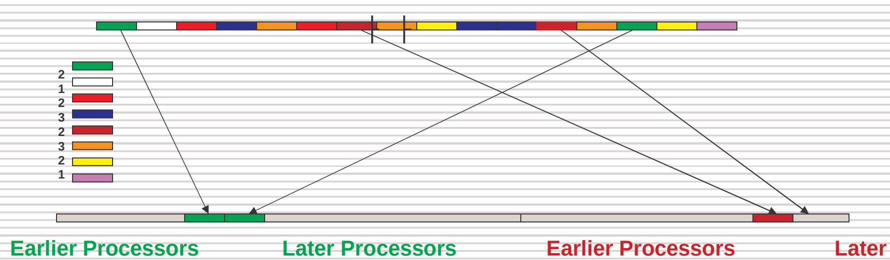
Reflection on Constrained ||ism

- Is solution correct … are writes exclusive?
- Is “moving the data twice” efficient?
- How might this algorithm be executed for *n*=10,000, *P*=1000
- What is the performance?
- Are the properties of this solution clear from the Peril-*L* code?

29

Cartoon of Scalable Solution

- Count number of each letter that are local
- Plan where they go
- Move them there



30

Scalable Parallelism, Part 1

```
DBrecord L_G[n];                                    Declare problem data
forall (index in (0..P-1) ){                        Thread for each processor
  int myRec = L_G_myHi – L_G_myLo + 1;  Define size
  DBrecord L[myRec]=is_local(L_G[]);  Equate array w/ local portion
  DBrecord Lt[myRec];                               Strictly local temporary variable
  alphaCount  = new int array[26];  Number of local letters
  globalCount = new int array[26];  Total records for each letter
  prefixCount = new int array[26];  Records w/ letters earlier in list
  int i, oldval, temp, let, pos;
  for (i=0; i<myRecs; i++) {                        How many of each letter are local
     alphaCount[letRank(charAt(L[i].x,0))] += 1;
     Lt[i] = L[i];                                  While touching data, move out of way
 }
 globalCount = +/alphaCount;                        Reduce local counts for global count
 prefixCount = +\alphaCount;                        Scan is contribution of earlier threads
 oldval = 0;
```


Scalable Parallelism, Part 2

```
 for (i=0; i<26; i++) {                   Local scan of globalCountto compute

     temp = globalCount[i];                  position where each letter starts
     globalCount[i] = oldval;
     oldval = oldval + temp;
 }
 prefixCount = prefixCount – alphaCount; Make exclusive
 for (i=0; i<myRecs; i++) {               Place records in final positions
     let = letRank(charAt(Lt[i].x,0)); Which letter is this record
     pos = globalCount[let]               Position is sum of total preceding letters
           + prefixCount[let]               plus slots used by earlier processors
           +(sofar[let]++);                 plus the number from this thread
     L_G[pos] = Lt[i];                    Move data to global space
  }
}
```

Reflection on Scalable ||ism

- Is solution correct … are writes exclusive?
- If data not preassigned, how does one get it
- How might this algorithm be executed for *n*=10,000, *P*=1000
- What is the performance?
- Are the properties of this solution clear from the Peril-*L* code?

33

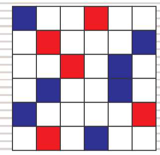
Features of Scalable Solution

- Does the solution scale, i.e. does it match the analysis from last week?
- Discuss: what were the good/bad features of the scalable solution?

34

Homework

- Part 1: Red/Blue Computation
  - Give n x n grid of colored cells
  - Red moves right, blue moves down
  - Board is toroidal
  - First 1/2 step, red moves 1 step; second 1/2 step, blue moves 1 step; can't move into occupied cell
  - Red vacate then blue move into cell is OK
  - Initial state is input; stop if any t x t block 90% one color



Be a friend to your TA…

35

Homework

- Part 2
  - One page essay of your thoughts on this conventional belief:

    "Quicksort is an example of a sequential algorithm that is a good candidate to be incrementally transformed into a parallel algorithm"
  - There is no "right" answer; it is credit/no credit
  - To receive credit, you need to make thoughtful points

36