Part IV: Programming Strategies

Goal: Introduce scalable algorithms and strategies for developing scalable solutions

Data and Task Parallelism

- Many definitions … parallelize the data or work?
- In a data parallel computation the parallelism is applied by performing the same operation to different items of data at the same time; the parallelism grows with the size of the data
- In a task parallel computation the parallelism is applied by performing distinct computations -- or tasks -- at the same time; with the number of tasks fixed, the parallelism is not scalable

Contrast solutions to preparing a banquet
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

Base Language is C

- Peril-L uses C as its notation for scalar computation, but any scalar language is OK
- 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
Threads

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

\[
\text{forall } \langle \text{int var} \rangle \text{ in } (\langle \text{index range spec} \rangle) \{ \langle \text{body} \rangle \}
\]

- Semantics: spawn \( k \) threads running \( \text{body} \)

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

- \( \langle \text{index range spec} \rangle \) is any reasonable (ordered) naming

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

\[
\text{barrier;}
\]

- Threads arriving at barriers suspend execution until all threads in its \text{forall} arrive there; then they’re all released
Memory Model

- Two kinds of memory: local and global
  - All variables declared in a thread are local
  - Any variable with 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 $\lambda$ time

Notice that the default vars are local vars

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

In PRAM terminology, this is CREW
Example: Try 1

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

```c
int *array, length, count;
forall thID in (0..t-1) {
    int i;
    int length_per=length/t;
    int start=thID*length_per;
    for (i=start; i<start+length_per; i++) {
        if (array[i] == 3) {
            count++;
        }
    }
}
```

- Variable usage is now obvious

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
  - By distinguishing between local and global memory, it distinguishes their costs
  - It is asynchronous

These may seem subtle but they matter
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 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

---

Announcements

- Class Friday at the usual time: 128 Allen
- Should be reading … Chapter 4
- New T2000 computer from Sun … perhaps will be a platform
Discussion

- Write a program for the Red/Blue computation
  - What happened?
  - What numbers did you explore?

---

Example: Try 4

- The final (correct) Count 3s program

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

Padding is irrelevant … it’s implementation
Full/Empty 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
- Full/Empty: Applies matter idea to memory
  ... F/E variables help serializing

Use the apostrophe’ suffix to identify F/E

Treating memory as matter

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

<table>
<thead>
<tr>
<th>Location contents</th>
<th>Variable Read</th>
<th>Variable Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Stall</td>
<td>Fill w/value</td>
</tr>
<tr>
<td>Full</td>
<td>Empty of value</td>
<td>Stall</td>
</tr>
</tbody>
</table>

- Scheduling stalled threads may not be fair

We’ll find uses for this next week
Slick Tree Accumulate Using F/E

Idea: Let values percolate up based on availability in F/E memory

Tree Accumulation Using F/E

```c
1 int nodeval'[P]; // Global full/empty vars to save right child val
2 forall ( index in {0..P-1} ) {
3   int val2accum; int stride = 1; // val2accum: locally computed val
4   nodeval'[index] = val2accum; // Assign initially to tree node
5   while (stride < P) {
6     if (index % (2*stride) == 0) {
7       nodeval'[index] = nodeval'[index] + nodeval'[index+stride]);
8       stride = 2*stride;
9     } else {
10       break; // Exit, if not now a parent
11     }
12   }
13 }
14}
```

Caution: This implementation is wrong …
Round 1 of Tree Accum …

index (in hex)

0 1 0 1 0 1 0 1 0 1 0 1

index % (2 * stride)

3 1 3 1 3 1 3 1 3 1 3 1

4 4 4 4 4 4 4 4 4 4 4 4

8 8 8 8 8 8 8 8 8 8 8 8

16 16 16 16 16 16 16 16 16 16 16 16

32 32 32 32 32 32 32 32 32 32 32 32

nodeval[index]

Caution: This implementation is wrong …
Tree Accumulation Using F/E

```c
int nodeval'[P];  // Global full/empty vars to save right child val
forall ( index in (0..P-1) ) {
    int val2accum;  int stride = 1;
    while (stride < P) {
        if (index % (2*stride) == 0) {
            val2accum = val2accum + nodeval'[index+stride];
            stride = 2*stride;
        }
        else {
            nodeval'[index] = val2accum;  // Assign val to F/E memory
            break;  // Exit, if not now a parent
        }
    }
}
```

Corrected version of Figure 5.1

Round 1 of Tree Accum ...

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index (in hex)</th>
<th>index % (2 * stride)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

nodeval[index]
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`

- To be portable, use reduce & scan rather than programming them

```
exclusive (count += priv_count; )  // WRONG
count = +/priv_count;              // RIGHT
```

Reduce/Scan Implies Synchronization

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
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/f/e), /, \ 

Thinking About Parallel Algorithms

- Computations need to be reconceptualized to be effective parallel computations
- Three cases to consider
  - Unlimited parallelism -- issue is grain
  - Fixed ||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
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
- Fixed: Local Alphabetize
- Scalable: Batcher’s Sort

Unlimited Parallelism (O/E Sort, I)

```c
bool continue = true;
rec L[n]; The data is global
while (continue) do {
forall (i in 1:n-2:2) { Stride by 2
rec temp;
if (strcmp(L[i].x, L[i+1].x) > 0) { Is o/even pair misordered?
    temp = L[i]; Yes, fix
    L[i] = L[i+1];
    L[i+1] = temp;
}
}
} Data is referenced globally
```
Unlimited Parallelism (O/E Sort, II)

forall (i in (0:n-2:2)) {  \textit{Stride by 2}
rec temp;
bool done = true; \textit{Set up for termination test}
if (strcmp(L[i].x,L[i+1].x)>0){ \textit{Is e/odd pair misordered?}
temp = L[i]; \textit{Yes, interchange}
L[i] = L[i+1];
L[i+1] = temp;
done = false; \textit{Not done yet}
}
continue = !(&&/ done); \textit{Were any changes made?}
}

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

---

Recall …

- We are illustrating the Peril-L notation for writing machine/language independent parallel programs
  - The “unlimited parallel solution” is O/E Sort
    - All data references were to global data
    - Threads spawned for each half step
    - Ineffective use of parallelism requiring threads to be created and implemented literally
  - Now consider a “fixed parallel solution”
Fixed Algorithm

- Postulate a process for handling each letter of the alphabet -- 26 Latin letters
- Logic
  - Processes scan records counting how many records start with their letter handle
  - Allocate storage for those records, grab & sort
  - Scan to find how many records ahead precede

Cartoon of Fixed Solution

- Move locally
- Sort
- Return
Fixed Part 1

```java
1 rec L[n];                    The data is global
2 forall (index in (0..25)) { A thread for each letter
3   int myAllo = mySize(L, 0); Number of local items
4   rec LocL[] = localize(L[]); Make data locally ref-able
5   int counts[26] = 0;      Count # of each letter
6   int i, j, startPt, myLet;
7   for (i=0; i<myAllo; i++) { Count number w/each letter
8     counts[letRank(charAt(L[i].x,0))]++;  
9   }
10  counts[index] = +/ counts[index]; Figure no. of each letter
11  myLet = counts[index];     Number of records of my letter
12  rec Temp[myLet];          Alloc local mem for records
13  alphabetizeInPlace(Temp[]); Alphabetize within this letter
14  startPt=+\myLet;          Scan counts # records ahead
15  for(i=0; i<count; i++) {   of these; scan synchs, so
16    L[j++]=Temp[i];        OK to overwrite L, post-sort
17  }
18  alphabetizeInPlace(L[]);  Alphabetize within this letter
19  for(i=0; i<count; i++) {   Return records to global mem
20    L[j++]=Temp[i];        
21  }
22```
Reflection on Fixed ||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?

Scalable Sort

☐ Batcher’s algorithm -- not absolute best, but illustrates a dramatic paradigm shift
☐ Bitonic Sort is based on a bitonic sequence:
☐ a sequence with increasing and decreasing subsequences

☐ Merging 2 sorted sequences makes bitonic
Batcher’s Sort

Skip recursive start; start w/ local sort

Control by thread ID of paired processes

$(p, d)$ controls it: start at $(\cdot, 0)$, $d$ counts up, $p$ down from $d-1$

$p = \text{process pairs}$

$d = \text{direction is } d^{th} \text{ bit}$

Logic of Batcher’s Sort

- Assumption: $2^x$ processes, ascending result
- Leave data in place globally, find position
  - Reference data locally, say $k$ items
  - Create (key, input position) pairs & sort these
  - Processes are asynch, though alg is synchronous
  - Each process has a buffer of size $k$ to exchange data -- write to neighbor’s buffer
  - Use F/E var to know when to write (other buffer empty) and when to read (my buffer full)
  - Merge to keep (lo or hi) half data, and insure sorted
  - Go till control values end; use index to grab original rec
Data Exchange

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor's buffer can be filled
  - ready' is empty until local buffer is filled

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P_i</td>
<td>P_j</td>
</tr>
<tr>
<td>free'</td>
<td>free'</td>
</tr>
<tr>
<td>ready'</td>
<td>ready'</td>
</tr>
<tr>
<td>BufK</td>
<td>BufK</td>
</tr>
</tbody>
</table>

Data Exchange

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor's buffer can be filled
  - ready' is empty until local buffer is filled

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P_i</td>
<td>P_j</td>
</tr>
<tr>
<td>free'</td>
<td>free'</td>
</tr>
<tr>
<td>ready'</td>
<td>ready'</td>
</tr>
<tr>
<td>BufK</td>
<td>BufK</td>
</tr>
</tbody>
</table>
Data Exchange

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor's buffer can be filled
  - ready' is empty until local buffer is filled

```
  Pi     Pj
  free'  free'
  ready' ready'
  BufK   BufK
```

Data Exchange

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor's buffer can be filled
  - ready' is empty until local buffer is filled

```
P_i     P_j
free'   free'
ready'  ready'
BufK    BufK
```
Data Exchange

- Use one buffer per processor plus to F/E variables: free' and ready'
  - free' is full when neighbor’s buffer can be filled
  - ready' is empty until local buffer is filled

\[ \text{P}_i \quad \text{P}_j \]

<table>
<thead>
<tr>
<th>free'</th>
<th>free'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ready'</td>
<td>ready'</td>
</tr>
<tr>
<td>BufK</td>
<td>BufK</td>
</tr>
</tbody>
</table>

Details on Data Exchange

20  alphabetizeInPlace(K[],bit(index,0));  \text{Local sort, up or down based on bit 0}

21  for(d=1; d<=m; d++)  \{  \text{Main loop, m phases}  
22  for(p=d-1; p<0; p--)  \{  \text{Define p for each sub-phase}  
23  stall=free'[neigh(index,p)];  \text{Stall till I can give data}  
24  for(i=0; i<size; i++)  \{  \text{Send my data to neighbor}  
25  BufK[neigh(index,p)][i]=K[i];  
26  \}  
27  ready'[neigh(index,p)]=true;  \text{Release neighbor to go}  
28  stall=ready'[index];  \text{Stall till my data is ready}  
29  \}  \text{Merge two buffers, keeping half}  
30  \}  
31  \}


Bitonic Sort In Text

- Details are in the book …
- **Discussion** Question: What, if any, is the relationship between Bitonic Sort and Quick Sort?

Sample Sort

- The idea of sending data to where it belongs is a good one … the Fixed Solution works out where that is, and Batcher’s Sort uses a general scheme
- Can we figure this out with less work?
  - Estimate where the data goes by sampling
  - Send a random sampling of a small number (log $n$?) of values from each process to $p_1$
  - $p_1$ sorts the values and picks the $P-1$ “cut points”, sends them back to all processors
  - Sample size depends on the values of $n$ and $P$
Sample Sort (Continued)

- After receiving the “cut points” each process...
  - Sends its values to the process responsible for each range
  - Each process sorts
  - A scan of the actual counts can place the “cut points” into the right processes
  - An adjustment phase “scooches” the values into final position

Cartoon of Sample Sort Solution

- Sample v values from all processors to $p_1$
- $p_1$ sorts and figures $P-1$ cutpoints
- Move them there
- Adjust position
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?

Know Your Way Around the Book

☐ Intro Stuff: 1-5
☐ Languages:
  ■ Threading: 6
  ■ Message Passing: 7
  ■ Hi Level Languages: 8
  ■ Critique: 9
☐ Promising Technologies: 10
☐ Projects & programming: 11
Expected Uses

- Chapter 3: Terms like speed-up; issues like dependences; trade-offs granularity: load_bal
- Language chapters contain core details of programming languages, but not complete
- Chapter 11: Getting on a parallel machine (first sections) and more about experimental method
- For Friday: Read Chapter 6, OpenMP

Next Assignment

- For Monday
  - Parallelize your sequential program using OpenMP
  - Compare performance to your best hand-threaded program: reporting usual stuff
    - Experimental set-up (processor characteristics, n…)
    - Speed-up for $p=1$, $p=2$, … for both solutions
    - Commentary of what the data means; 1 page max
    - Record of raw numbers, code listing, etc.
    - Click together in pdf.

Such “reporting” assignments are graded
For Wednesday (Feb 13)

- Write an MPI program for Red/Blue using the “vertical panel” approach 
- attu has MPI installed, or use other parallel machine
- Compare the MPI program to a threaded solution (probably on attu)
- Report the “usual stuff”