Part IV: Programming Strategies

Goal: Introduce scalable algorithms and strategies for developing scalable solutions
Regarding the Red/Blue computation

- How did you allocate the array? Why?
- How was the work assigned?
- How do the threads communicate?
Many definitions ... parallelize the data or work?

In a data parallel computation the parallelism is applied by performing the same (or similar) operations 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
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

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

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

```plaintext
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

- Threads execute at their own rate
- The execution relationships among threads are 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
- Reference to the `forall` index identifies the thread
Two kinds of memory: local and global
  - All variables declared in a thread are local
  - Any variable with underlined_name is global
Names (usually indexed) work as usual
  - Local variables use local indexing
  - Global variables use global indexing
Memory is based on CTA, so performance:
  - Local memory references 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, but it’s not a PRAM
Example: Try 1

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

```plaintext
int *array, length, count, t;
... initialize globals here ...
forall thID in (0..t-1) {
  int i, 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 \texttt{exclusive}
  may not be fair

Exclusive gives behavior, not mechanism
Example: Try 4

The final (correct) Count 3s program

```c
int *array, length, count, t;
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
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 requires the location be 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 if 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
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

```plaintext
exclusive {count += priv_count; } "WRONG"
count = +/priv_count; "RIGHT"
```

Reduce/Scan Imply Synchronization
When reduce/scan involve local memory

- 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), /, \
The point of a pseudocode is to allow detailed discussion of subtle programming points without being buried by the extraneous detail.

To illustrate, consider some parallel computations...

- Tree accumulate
- Balanced parens
Idea: Let values percolate up based on availability in F/E memory
Naïve F/E Tree Accumulation

```
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         }
10     } else {
11         break;  // Exit, if not now a parent
12     }
13 }
14 }
```

Caution: This implementation is wrong ...
Naïve F/E Tree Accumulation

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     } 8 9  index
10   } 0 1  Odd?
11   else {
12     break;  Exit, if not now a parent
13   }
14 } 4

Caution: This implementation is wrong ...
Caution: This implementation is wrong...
But What If $P_2$ is Slow, $P_0$ Fast?

index (in hex)

index $\% (2 \times \text{stride})$

nodeval[index]

Caution: This implementation is wrong ...
Introduce Barrier to Synch Levels

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         }
10     } else {
11         break; // Exit, if not now a parent
12     }
12.5 barrier;
13 }
Barrier Stops Until Stable State

index (in hex)

index % (2 * stride)

nodeval[index]
The Problem With Barriers

- In many places barriers are essential to the logic of a computation, but ...
- In many cases they are just an implementational device to overcome (for example) false dependences
- Avoid them when possible
  - They force the ||-ism to drop to zero
  - Often costly even when all threads arrive at once
Asynchronous Tree Accumulate

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;
4   while (stride < P) {
5     Begin logic for tree
6       if (index % (2*stride) == 0) {
7         val2accum=val2accum+nodeval'[index+stride];
8         stride = 2*stride;
9       }
10      else {
11         nodeval'[index]=val2accum; Assign val to F/E memory
12         break; Exit, if not now a parent
13       }
14   }
15 }
The “full” Applies To Root Only

index (in hex)

index % (2 * stride)

nodeval[index]
Both the synchronous and asynchronous accumulates are available to us, but we usually prefer the asynch solution. Notice that the asynch solution uses data availability as its form of synchronization.
char *symb[n];

forall pID in (0..P-1) {
    int i, len_per_th=length/t;
    int start=pID * len_per_th;
    int o=0, c=0;
    for (i=start; i<start+len_per_th; i++) {
        if (symb[i] == "(" )
            o++;
        if (symb[i] == ")" ) {
            o--;
            if (o < 0) {
                c++; o = 0;
            }
        }
    }
}
Break
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
1 bool continue = true;
2 rec L[n];
3 while (continue) do {
4  forall (i in (1:n-2:2)){ Stride by 2
5     rec temp;
6     if (strcmp(L[i].x,L[i+1].x)>0){ Is o/even pair misorderened?
7         temp = L[i]; Yes,fix
8         L[i] = L[i+1];
9         L[i+1] = temp;
10     }
11  }
12 }

Data is referenced globally
Unlimited Parallelism (O/E Sort, II)

forall (i in (0:n-2:2)) {  Stride by 2
  rec temp;
  bool done = true;  Set up for termination test
  if (strcmp(L[i].x,L[i+1].x)>0) {  Is e/odd pair misordered?
    temp = L[i];  Yes, interchange
    L[i] = L[i+1];
    L[i+1] = temp;
    done = false;  Not done yet
  }
  continue = !(&&/ done);  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?
The criticism of fine-grain logical processes is they will usually be *emulated*; it’s much slower than doing the work directly.

Can we compile logical threads to tight code?

Possibly, but consider this model

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

Generalizing “trivialized” operations is hard
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 w/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: Introduce 2 functions

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;` Count number w/each letter
7. `for (i=0; i<myAllo; i++) {` Count number w/each letter
8. `  counts[letRank(charAt(LocL[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
j = 0;  // Index for local array
for(i=0; i<n; i++) {
  if(index==letRank(charAt(L[i].x,0)))
    Temp[j++]=L[i];  // Save record locally
}
alphabetizeInPlace(Temp[]);  // Alphabetize within this letter
startPt=+\myLet;  // Scan counts # records ahead of these; scan synchs, so OK to overwrite L, post-sort
j=startPt-\myLet;
for(i=0; i<count; i++){
  L[j++]=Temp[i];
}
}
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?
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
Skip recursive start; start w/ local sort
Control by thread ID of paired processes \((p,d)\) controls it: start at \((-,0)\), \(d\) counts up, \(p\) down from \(d-1\)
\(p\) = process pairs
\(d\) = direction is \(d^{th}\) bit
Bitonic Sort, Closer Look
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 Transfer

- 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

\[ P_i \]

\[ P_j \]

\[ \text{free'} \quad \text{ready'} \]

\[ \text{BufK} \]

\[ \text{free'} \quad \text{ready'} \]

\[ \text{BufK} \]
Data Transfer

- 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

\[ P_i \]

- free' ready' BufK

\[ P_j \]

- free' ready' BufK
Data Transfer

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

- 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
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
20  alphabetizeInPlace(K[],bit(index,0));  Local sort, up or
down based on bit 0
21  for(d=1; d<=m; d++)  {  Main loop, m phases
22      for(p=d-1; p<0; p--)  { Define p for each sub-phase
23          stall=free'[neigh(index,p)];  Stall till I can give data
24          for(i=0; i<size; i++)  { Send my data to neighbor
25              BufK[neigh(index,p)][i]=K[i];
26          }
27          ready'[neigh(index,p)]=true; Release neighbor to go
28          stall=ready'[index]; Stall till my data is ready
29          ...
30      } Merge two buffers, keeping half
31  }
Details are in the book ...

Discussion Question: What, if any, is the relationship between Bitonic Sort and Quick 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 $p_0$
- $p_0$ 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 $P$
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
Sample \( v \) values from all processors to \( p_o \)
- \( p_o \) 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?
Peril-L is a useful notation for sketching a solution – you will probably implement it w/o much language support
- Ideally, we should have language support
- Hopefully, it helps working out subtle points, like synchronization behavior

In algorithm design, maximizing parallelism is much less important than minimizing process-interactions
Work out the basic logic of Sample Sort and program it in Peril-L

Focus only on finding the “cuts,” determining where the data goes, and “adjusting” for balanced final allocation

- Data is initially placed where you want it – but say where that is
- Assume any “local” functions you wish, such as `loc_sort()` that sorts data locally in place
- `n` is a multiple of `P`, whose values are in `n` and `P`
The purpose of this assignment is
- Familiarity with Peril-\( L \)
- Understand the ideas behind Sample sort

Turn in
- Peril-L code with “coarse grain” commenting
- Your thoughts about the usefulness of the CTA in developing the algorithm, and any comments about Peril-\( L \)