Part IV: Programming Strategies

Goal: Introduce scalable algorithms and strategies for developing scalable solutions

Red Blue Discussion

- Regarding the Red/Blue computation
  - How did you allocate the array? Why?
  - How was the work assigned?
  - How do the threads communicate?
Data and Task Parallelism

- 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
**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

```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

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 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

```c
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 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

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 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

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

Reduce/Scan Imply 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

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), /, \

Using Peril-L

- 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

Slick Tree Accumulate Using F/E

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

index (in hex)
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;  
4   nodeval'[index] = val2accum;  
5   Begin logic for tree
6   while (stride < P) {  
7       if (index % (2*stride) == 0) {  
8           nodeval'[index]=nodeval'[index]+nodeval'[index+stride];  
9       }  
10      else {  
11          break;  
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;  
4   nodeval'[index] = val2accum;  
5   Begin logic for tree
6   while (stride < P) {  
7       if (index % (2*stride) == 0) {  
8           nodeval'[index]=nodeval'[index]+nodeval'[index+stride];  
9       }  
10      else {  
11          break;  
12       }  
13   }  
14 }

Caution: This implementation is wrong ...
## 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>0</td>
<td>1</td>
</tr>
</tbody>
</table>

index (in hex)

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

index % (2 * stride)

nodeval[index]

Caution: This implementation is wrong ...

---

## But What If P₂ is Slow, P₀ Fast?

<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>0</td>
<td>1</td>
</tr>
</tbody>
</table>

index (in hex)

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

index % (2 * stride)

nodeval[index]

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

1 int nodeval'[P];
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) { Begin logic for tree
6 if (index % (2*stride) == 0) {
7 nodeval'[index]=nodeval'[index]+nodeval'[index+stride];
8 stride = 2*stride;
9 } 10 barrier;
11 12 }
13 14}

Barrier Stops Until Stable State

<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>0</td>
<td>1</td>
</tr>
</tbody>
</table>

index (in hex)

<table>
<thead>
<tr>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

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

```
int nodeval[P];
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;
            break;
        }
    }
}
```
The “full” Applies To Root Only

<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>0</td>
<td>1</td>
</tr>
</tbody>
</table>

index (in hex)

<table>
<thead>
<tr>
<th>3</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

nodeval[index]

Critique of Tree Accumulate

- 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/P;
  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;
    }
  }
}
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
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 misordered?
7             temp = L[i]; Yes, fix
8             L[i] = L[i+1];
9             L[i+1] = temp;
10         } Data is referenced globally
11     }
```

Unlimited Parallelism (O/E Sort, II)

```c
12     forall (i in (0:n-2:2)) { Stride by 2
13         rec temp;
14     bool done = true; Set up for termination test
15     if (strcmp(L[i].x,L[i+1].x)>0) { Is e/odd pair misordered?
16         temp = L[i]; Yes, interchange
17         L[i] = L[i+1];
18         L[i+1] = temp;
19         done = false; Not done yet
20     }
21     continue = !(&&/ done); Were any changes made?
22 }
23 }
```
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

- 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”

---

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
Move locally

Sort

Return

Fixed Part 1: Introduce 2 functions

The data is global
A thread for each letter
Number of local items
Make data locally ref-able
Count # of each letter
Count number w/each letter
Number of records of my letter
Alloc local mem for records
Fixed Part 2

13  j = 0;                              Index for local array
14  for(i=0; i<n; i++) {                Grab records for local alphabetize
15    if(index==letRank(charAt(L[i].x,0)))
16      Temp[j++]= L[i];                Save record locally
17  }
18  alphabetizeInPlace(Temp[]);        Alphabetize within this letter
19  startPt=+\myLet;                   Scan counts # records ahead of these; scan synchs, so OK to overwrite L, post-sort
20  for(i=0; i<count; i++) {           Find my start index in global
21    L[j++]=Temp[i];                 Return records to global mem
22  }
23
24 }

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

![Ascending Descending \(\Leftrightarrow\) Lower Bitonic Upper Bitonic](image)

- 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 \((-, 0)\), \(d\) counts up, \(p\) down from \(d-1\)
- \(p = \) process pairs
- \(d = \) direction is \(d^{th}\) 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 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

\[
\begin{align*}
\text{P}_i & \quad \text{free'} \quad \text{ready'} \\
\text{BufK} & \\
\end{align*}
\]

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

\[
\begin{align*}
\text{P}_i & \quad \text{free'} \quad \text{ready'} \\
\text{BufK} & \\
\end{align*}
\]
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_i
free' ready'
BufK
```

26
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

```
null
null
free' ready'
BufK
null
free' ready'
BufK
null
```

Details on Data Transfer

20 alphabetizeInPlace(K[], bit(index, 0));  \textit{Local sort, up or down based on bit 0}
21 for (d=1; d<=m; d++) {  \textit{Main loop, m phases}
22 for (p=d-1; p<0; p--) {  \textit{Define p for each sub-phase}
23 stall = free'[neigh(index, p)];  \textit{Stall till I can give data}
24 for (i=0; i<size; i++) {  \textit{Send my data to neighbor}
25 BufK[neigh(index, p)][i] = K[i];
26 }
27 ready'[neigh(index, p)] = true;  \textit{Release neighbor to go}
28 stall = ready'[index];  \textit{Stall till my data is ready}
29 ...  \textit{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 $p_o$
  - $p_o$ 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 $p_0$
- $p_0$ 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?

Summary

- 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 that minimizing process-interactions
HW for Next Week

- 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`

HW Goals

- 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