Peril-L, Reduce, and Scan

Mark Greenstreet

CpSc 418 – Oct. 23, 2012

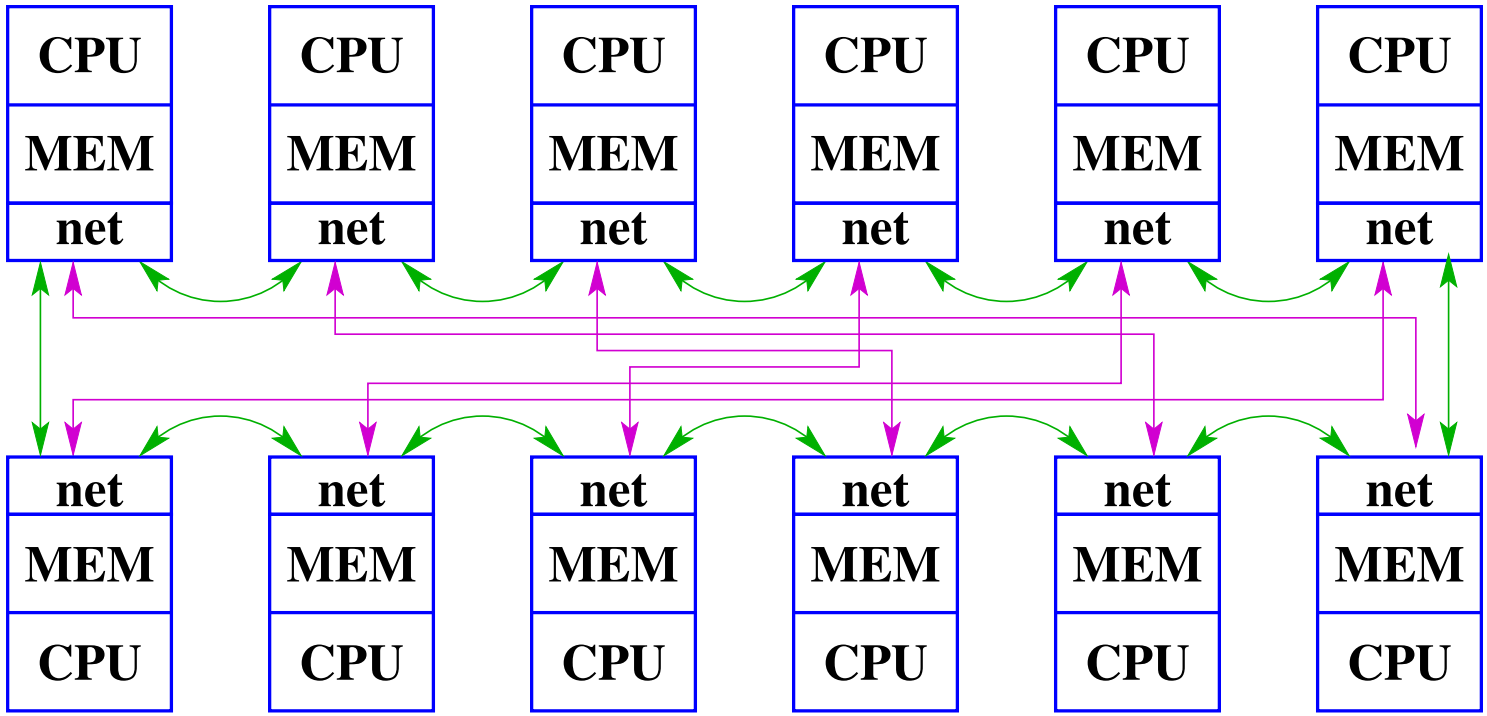
Lecture Outline

- Peril-L
  - A notation based on the CTA model
  - Control and synchronization
  - Local vs. remote data access.
  - Reduce and Scan operators

Peril-L

- "Pidgin" langauge for describing parallel algorithms and programs.
- Based on CTA model
- Uses C-like language with extensions:
  - Each processor has its own, local memory
  - One processor can access the memory of another processor, but such global accesses take λ units of time.
  - `forall`, `exclusive` and `barrier`
  - distinction between `global` and `local` memory
  - `reduce` and `scan` operations

CTA: Summary



CTA	Peril-L
Fixed number of processors	Focus on scalable parallelism
Each processor is a RAM	Unit cost for local operations
Explicit communication	Identify remote accesses
Bounded-degree network	No direct correspondence

forall

Expressing multiple threads in Peril-L:

```
forall (integer variable in range) {
    body
}
```

- One thread is created for each value in *range*.
- The *bodies* for each thread execute in parallel.
- `forall` statements may be nested.
- There is an implicit barrier at the end of the loop:
  - The program does not progress beyond the `forall` statement until all threads have completed their *body*.

Say hello – first attempt

- Program source:

```
char **who = { "Anne", "Bob", "Charlie", "Diane",
               "Elmer", "Francine" };
forall(i in (0..5)) {
   printf(``%s says `hello'.\n'', who[i]);
}
```

- Output:

```
Diane says `hello' Bob.
AnnElmer says says `helloFrancine says Charliee'
say ````hehehehe ssays lllllloo'.
l'lo'o'...

.
```

Why?

exclusive

- `exclusive` { *body* }
- Only one thread can execute *body* at a time.
  - Other threads will block.
  - In other words, `exclusive` provides mutual exclusion.

Say hello, second attempt

- Program source:

```
char **who = { "Anne", "Bob", "Charlie", "Diane",
               "Elmer", "Francine" };
forall(i in (0..5)) {
   exclusive {
      printf(``%s says `hello'.\n'', who[i]);
   }
}
```

- Output:

```
Elmer says `hello'.
Diane says `hello'.
Anne says `hello'.
Charlie says `hello'.
Francine says `hello'.
Bob says `hello'.
```

barriers

- A `forall` statement can include a `barrier` statement.
- All threads of the `forall` must reach the `barrier` before any threads continue beyond it.

```
char **who = { "Anne", "Bob", "Charlie", "Diane", "Elmer
forall(i in (0..5)) {
   exclusive {
      printf(``%s says `hello'.\n'', who[i]);
   }
   barrier;
   exclusive {
      printf(``%s says `good-bye'.\n'', who[i]);
   }
}
```

Variables and Memory

- global(i.e. shared) and local (i.e. per-thread) variables.
- `localize`
- `EF` (“empty/full”) variables.

Global and local variables

- If a variable is declared outside of a `forall` statement, it is global and shared by all threads of the `forall`.
  - In keeping with the CTA model, an access to a global variable cost λ time units.
  - Global variables are indicated by underlining the variable name.
- If a variable is declared in the body of a `forall` statement, it is local. There is a separate, private instance of the variable for each thread.
  - A local variable can be accessed in unit time.

Localize

- In the CTA model, there are global variables but no global memory.
- The `localize` statement is used to partition the storage of an array amongst the processors of a `forall` loop.

```
int allData[n];
forall(threadID in (0..P-1)) {
    int size = mySize(allData[], 0);
    int locData[size] = localize(allData[]);
    ...
}
```

Localize (details)

- `localize` tells the compiler that it should distribute the storage of the array amongst the processors of the `forall`.
- It doesn't actually move data from some (non-existent) global store to local memory; therefore, this declaration doesn't take time at runtime.
- References to the local portion of the array, e.g. `locData`, are local to the processor and performed in unit time.
  - Array indices for the local array start a `0` for each thread, regardless of where the reference is in the global array.
  - `localToGlobal(allData, i, j, ...)`, returns the global index (flattening the array to one dimension, I assume), corresponding to the local indices `i, j, ...`.
- References to the global array, e.g. `allData` are global references and take λ time units.
  - Concurrently accessing an array by its localized and global versions is a bad idea.

Empty/Full variables

- An empty/full variable is a global variable for implementing flow control (e.g. unbuffered messages).
- Writing to an empty variable gives the variable a value, and fills it.
  - Attempting to write to a full variable stalls until the variable is read and thus becomes empty.
- Reading from a full variable gives gets the value and marks the variable as empty.
  - Attempting to read from an empty variable stalls until the variable is written and thus becomes full.
- Reads and writes of empty/full variables are global operations that take λ time units (plus any time for stalls).

Empty/Full in Peril-L

In Peril-L, an empty/full variable is a global variable with a prime at the end of its name:

- E.g. `int q' = 2;`
- If initialized, then initially full, with that value.
- Otherwise, intially empty.
- Full/empty values can be arrays and/or structs, in which case all elements and/or fields are full-empty variables.

Reduce

- Reduce: `op/` *localExpression*
  - Each thread in the current `forall` evaluates `localExpression` and the results are combined using `op` to produce the value for the expression.
  - All threads get the same value – this value can be assigned to a local or global variable or used in a bigger expression.
- Scan: `op\` *localExpression*
  - Each thread in the current `forall` evaluates `localExpression` and the results are combined using `op`.
  - Thread `i` gets the result of applying `op` to the values of `localExpression` for threads `0...i`.
- Both reduce and scan also function as barriers.

Reduce Example

```
double globalData[] = {1.0, ...M};
forall(i in (0..P-1)) {
   localData = localize(globalData);
   localSum = 0.0;
   n = mySize(globalData, 0);
   for(j = 0; i < n; i++)
      localSum += localData[j];
   grandTotal = +/ localSum;
}
```

- Assume `M` $= 1000000$ and `P` $= 10$.
- After the inner for-loop, thread `i` will have

$$\begin{aligned} \texttt{localSum} &= \sum_{j=i*100000+1}^{(i+1)*100000} j \\ &= 100000 * (100000 * i + 50000.5) \end{aligned}$$

- After the reduce, each thread will have `grandTotal` (a separate, local variable for each thread) equal to the sum of all the localSum

Reduce – The Picture

To be drawn on the whiteboard. You can sketch it here.

Scan Example

```
double globalData[] = {1.0, ...M};
forall(i in (0..P-1)) {
   localData = localize(globalData);
   localSum = 0.0;
   n = mySize(globalData, 0);
   for(j = 0; j < n; i++)
      localSum += localData[i];
   grandTotal = +\ localSum;
}
```

- Assume `M` $= 100$ and `P` $= 10$.
- After the inner for-loop, thread `i` will have `localSum` $= 100 * i + 55$ (i.e. thread 0 gets 55, thread 1 gets 155, thread 2 gets 255, ...).
- After the scan, each thread will have `grandTotal` (a separate, local variable for each thread) equal to the sum of all the localSum for threads up to and including its own:
  thread 0 gets 55, thread 1 gets 210, thread 3 gets 465, ...

Scan – The Picture

To be drawn on the whiteboard. You can sketch it here.

Reduce In Peril-L

```
 1.int nodeval'[P];
 2.
 3.forall(index in(0..P-1)) {
 4.    int tally;
 5.    stride=1;
 6.    tally = compute local tally;
 7.    while(stride < P) { % reduce tree
 8.       if(index%(2*stride) == 0) {
 9.          tally = tally + nodeval'[index+stride];
10.          stride = 2*stride;
11.       } else {
12.          nodeval'[index]=tally;
13.          ;
14.       } % if(index...)
15.    } % while(stride ¡ P)
16.} % forall(index...)
```

Generalized Reduce

From the CpSc 418 erlang library, module `wtree`

```
reduce(W, Leaf, Combine, Root) -> Value
    % W: a worker pool
    % Leaf: The function to be applied at each leaf of the tree
       Leaf(ProcState) -> Value
    % Combine: Combine the values from two subtrees
       Combine(LeftValue, RightValue) -> Value
    % Root: Produce the final value
       Root(RawValue) -> ReturnValue
       % The Value returned by Leaf or Combine
       % may have extra "bookkeeping" stuff to allow further combining
       % of results. The Root function cleans this up to provide
       % the desired value.
reduce(W, Leaf, Combine) -> Value
    xcommentSame as reduce/4 where Root is the identity functior
```

Note: unlike the version in Peril-L, `wtree:reduce` doesn't broadcast the final to all workers. You'll need to use `wtree:broadcast` or `workers:update` to do that.

Note (to self): I should add `update` and `retrieve` functions to the `wtree` module.

Generalized Scan

Announcements and reminders

- Oct. 25: work allocation
  Read the rest of Lin & Snyder, Chapter 5, *Assigning Work to Processes Statically* through the end of *Chapter Summary*.
- Oct. 30: introduction to MPI
  - Read Lin & Snyder, Chapter 7, *MPI: The Message Passing Interface* through the end of *MPI: The Message Passing Interface* $\rightarrow$ *Safety Issues* (pp. 202–229).
  - More info on MPI at
    https://computing.llnl.gov/tutorials/mpi/

Review

- How does Peril-L distinguish local and remote memory accesses?
- Does the Peril-L machine model include local memory?
- Does the Peril-L machine model include global memory?
- What is an empty/full variable?
- Try the examples from Lin & Snyder chapter 5, *The Reduce and Scan Abstractions* $\rightarrow$ *Example of Generalize Reduce and Scan*.
- Peril-L has an *exclusive* operator. Think about how you would implement this in Erlang. For example, you might want a process to be able to "lock" `stdout` while writing a multi-line message. Hint: use a process.