Peril-L

*Peril-L* is the book’s pseudocode language

- Start with C

- Add high-level constructs for parallelism:
  - `forall` to start parallelism
  - `exclusive` and `barrier`
  - `full/empty t`
  - `global` versus `local` variables
  - `localize`, `mySize`, `localToGlobal`
  - `<op>`/ and `<op>\"
forall

forall(⟨variable⟩ in(⟨range⟩))
{
  ⟨body⟩
}

- The ⟨range⟩ indicates $N$ integers
- Runs ⟨body⟩ in $N$ threads concurrently
- The ⟨variable⟩ is bound in ⟨body⟩ to a value from ⟨range⟩, a different value for each thread
- forall finishes when all threads complete
- Nested forall is allowed
forall(i in (1..3))
{
    printf("Hello %i\n", i);
}

produces

Hello 2
Hello 1
Hello 3
forall Example

forall(i in (1..3))
{
    printf("Hello %i\n", i);
}

... or ...

Hello 3
Hello 2
Hello 1
forall Example

forall(i in (1..3))
{
    printf("Hello %i\n", i);
}

... or

Hello 3
1 Hello o 2
1
exclusive

exclusive { ⟨body⟩ }

• Globally restricts threads so only one runs ⟨body⟩ at a time
exclusive Example

forall(i in (1..3))
{
exclusive
{
printf("Hello %i\n", i);
}
}

Like previous example, but no mixing of lines
barrier

barrier;

Waits until all other threads within the immediately enclosing forall reach the same place
exclusive Example

forall(i in (1..3))
{
    exclusive { printf("Hello %i\n", i); }  
    barrier;
    exclusive { printf("Goodbye %i\n", i); }  
}

All Hello lines print before all Goodbye lines
Global and Local Variables

• *global* variables are underlined
  ○ access cost is $\lambda$
  ○ always shared by all threads

• *local* variables are not underlined
  ○ access cost is 1
  ○ never shared by any threads
Global and Local Variables

```c
int data[n];

forall(i in (0..n-1))
{
    data[i] = -data[i];
}
```

Negates `data` in parallel

Note that thread-specific index `i` is local, while `data` is global
**localize**

\[
\langle \text{local variable} \rangle = \text{localize}(\langle \text{global variable} \rangle)
\]

Produces a pointer to a local portion of \(\langle \text{global variable} \rangle\)

- \(\langle \text{local variable} \rangle\) write/read \(\Rightarrow\) \(\langle \text{global variable} \rangle\) write/read, but without \(\lambda\) penalty
- \(\langle \text{local variable} \rangle\) in different threads is a different part of \(\langle \text{global variable} \rangle\)
- \(\langle \text{local variable} \rangle\) is indexed from 0
- \(\langle \text{local variable} \rangle\) location/distribution within \(\langle \text{global variable} \rangle\) is unspecified but order is preserved?
Example

```c
int data[n];
int t;

forall(i in (0..t-1))
{
    int size = n / t;
    int mydata[] = localize(data);

    for (int j = 0; j < size; j++) {
        mydata[j] = -mydata[j];
    }
}
```

Same as previous example, but with only \( t \) threads
mySize

mySize(⟨global variable⟩,  ⟨dimen⟩)

Avoids assumption that all threads get the same amount of data
int data[n];
int t;

forall(i in (0..t-1))
{
    int size = mySize(data, 0);
    int mydata[] = localize(data);

    for (int j = 0; j < size; j++) {
        mydata[j] = -mydata[j];
    }
}

Same as previous example, but more abstract
**localToGlobal**

`localToGlobal(global variable, index, dimen)`

Exposes mapping of local to global data

Example where this is needed:

```c
int data[n];
for (int i = 0; i < n; i++) {
    data[i] += i;
}
```
localToGlobal Example

```cpp
int data[n];
int t;

forall(i in (0..t-1))
{
    int size = mySize(data, 0);
    int mydata[] = localize(data);

    for (int j = 0; j < size; j++) {
        mydata[j] += localToGlobal(data, j, 0);
    }
}
```

Parallel version of sequential example
Variable name with a tick is either \textit{empty} or \textit{full}

\begin{itemize}
  \item $t' = \langle \text{value} \rangle \& \text{empty } t' \Rightarrow \text{full } t' \text{ with } \langle \text{value} \rangle$
  \item $t' \& \text{full } t' \text{ with } \langle \text{value} \rangle \Rightarrow \text{empty } t'$, return $\langle \text{value} \rangle$
  \item $t' = \langle \text{value} \rangle \& \text{full } t' \Rightarrow \text{wait until } t' \text{ is empty}$
  \item $t' \& \text{empty } t' \Rightarrow \text{wait until } t' \text{ is full}$
\end{itemize}

Full/empty variables are implicitly global
Full/empty Variables Example

```c
int data[n];
int obuf', ebuf';

forall(i in(0 ..n))
{
  if (i & 1) {
    obuf' = data[i];
    data[i] = ebuf';
  } else {
    ebuf' = data[i];
    data[i] = obuf';
  }
}
```

Swaps each even slot with a random odd slot in `data`
Reduce and Scan

For associative, commutative \( \langle \text{op} \rangle \):

- \( \langle \text{op} \rangle / \langle \text{expr} \rangle = \text{parallel reduce} \)
  - Fold \( \langle \text{op} \rangle \) over an array to produce one value

- \( \langle \text{op} \rangle \backslash \langle \text{expr} \rangle = \text{parallel scan} \)
  - Fold \( \langle \text{op} \rangle \) over an array to produce prefix array

These operations imply synchronization when the source and destination are local

Prefer these forms over other ways of solving a problem
Reduce Examples

• +/data

  Sums all elements of data

• ||/data

  Determines whether any element of data is non-zero

• int data[n], w;
  forall(i in(0..n))
  {
    int v = data[i] / 2;
    w = +/v;
  }

  Sums halved elements of data
Reduce Examples

- $+/\text{data}$
  Sums all elements of $\text{data}$

- $||/\text{data}$
  Determines whether any element of $\text{data}$ is non-zero

- $\text{int data}[n], w;$
  $\text{forall}(i\ \text{in}(0..n))$
  {
    $w = +/(\text{data}[i] \ / \ 2);$  
  }

  Also sums halved elements of $\text{data}$
Scan Examples

- **data = +data**
  
  Sums elements of data, recording prefix

- `int data[n];`
  `forall(i in(0..n))`
    `{`
    `  int v = data[i] / 2, w;`
    `  w = +v;`
    `  data[i] = -w;`
    `}`
  
  Sums halved elements of data, records negated prefix
int array[n], count;

forall(i in (0..n-1))
{
    count = +/((array[i] == 3) ? 1 : 0);
}

Clear!
Concise!
Difficult to compile to efficient code!
int array[n], count;
int t;

forall(i in (0..t-1))
{
    int size = mySize(array);
    int myArray = localize(array);
    int myCount = 0;

    for (int j = 0; j < size; j++)
        if (myArray[j] == 3) myCount++;

    count = +/myCount;
}