Outline of this lecture:

1. Summing Example
2. Hyperobjects
3. Series and Parallel Relations

1 Summing Example

\[
\text{SUM}(n)
\]
1 \quad \text{result} = 0
2 \quad \text{for } i \leftarrow 0 \text{ to } n
3 \quad \text{do result} \leftarrow \text{result} + \text{compute}(A[i])
4 

If we want to achieve parallelism, we need to be careful of race conditions. Consider the following cilk code

\[
\text{CILK}_\text{SUM}(n)
\]
1 \quad \text{result} = 0
2 \quad \text{cilk_for } i \leftarrow 0 \text{ to } n
3 \quad \text{do result} \leftarrow \text{result} + \text{compute}(A[i])
4 

Determinacy Race exists in the cilk code. Also consider another cilk code in the following.

\[
\text{SUM}(n)
\]
1 \quad \text{mutex } L
2 \quad \text{result} = 0
3 \quad \text{cilk_for } i \leftarrow 0 \text{ to } n
4 \quad \text{do temp} \leftarrow \text{compute}(myArray[i])
5 \quad \text{L.lock}()
6 \quad \text{result} \leftarrow \text{result} + \text{temp}
7 \quad \text{L.unlock}()
8 

While there is no data race in the above code, there is still a determinacy race. Additionally, we suffer from lock overhead and lock contention, which can destroy the benefits of parallelism.
2 Hyperobjects

Cilk provides hyperobjects, such as reducers, to mitigate determinacy races on local variables without the need for locks

\[
\text{SUM(n)}
\]

1 \hspace{1em} \text{res} = 0
2 \hspace{1em} \text{cilk} :: \text{reduce} < \text{cilk} :: \text{opadd} < \text{int} >> \text{result}_r()
3 \hspace{1em} \text{result}_r.\text{move_in}(\text{res})
4 \hspace{1em} \text{cilk_for} i \leftarrow 0 \text{ to } n
5 \hspace{1em} \text{do} \hspace{1em} \ast result_r \leftarrow \ast result_r + \text{compute(myArray[i])};
6 \hspace{1em} \text{result}_r.\text{move_out}(\text{res})

**Intuition Behind Reducers**

A reducer is defined over an associative operation, such as addition in the example above. Threads can update a reducer, as each worker has its own view of the reducer. The cilk runtime system is responsible for keeping track of these views, and combining them when appropriate.

**Programming With Reducers**

A **monoid** is a triple \((T, \otimes, e)\) in which

1. \(T\) is a set
2. \(\otimes\) is an associative binary operator on elements of \(T\)
3. \(e \in T\) is an identity element for \(\otimes\)

   where **Associative** is \(a \otimes (b \otimes c) = (a \otimes b) \otimes c\) and **Identity** is \(a \otimes e = e \otimes a = a\)

Monoids can be represented in cilk on type \(T\) objects by creating a C++ class that inherits from \(\text{cilk} :: \text{monoid_base} < T >\) and defines

1. A member function \text{reduce()} that implements the binary operator \(\otimes\)
2. A member function \text{identity()} that constructs a fresh identity \(e\)
3. Other updating operations

**More on Monoids**

- Upon declaration, the default constructor of a reducer initializes it’s initial view with identity \(e\)
- In a parallel region, the underlying local view can be accessed by dereferencing the reducer
- The final value resulted from the updates can be safely retrieved by using the \text{move_out()} function
The `cilk::reducer` library

- `reducer::move_in`: Populates the underlying view, irrespective of what happened before that point.
- `reducer::move_out`: Extracts content out of the reducer.
- Both operations take a single object to swap content with as an argument. Therefore, the move is destructive to the content of the source.

3 Series and Parallel Relations

Series Relations

- A strand $s_1$ logically precedes strand $s_2$ ($s_1 \prec s_2$) if there exists a path from $s_1$ to $s_2$ in the computational dag. We also say that $s_2$ succeeds $s_1$ ($s_2 \succ s_1$).
  - Two strands $s_1$ and $s_2$ are in series if $s_1 \prec s_2$ or $s_2 \prec s_1$.

![Diagram showing series and parallel relations](image)

- a and b are in series ($a \prec b$)

Parallel Relations

- A strand $s_1$ is logically parallel to strand $s_2$ ($s_1 \parallel s_2$) if no path exists from $s_1$ to $s_2$ or from $s_2$ to $s_1$ in the computational dag.
a and b are in parallel (a \parallel b)

Definitions

- **Tetrachotomy Lemma**: For any two strands $s_1$ and $s_2$, exactly one of the following holds:
  - $s_1 = s_2$
  - $s_1 \parallel s_2$
  - $s_1 \prec s_2$
  - $s_1 \succ s_2$

- **Transitivity Lemma**: $s_1 \prec s_2$ and $s_2 \prec s_3$ implies $s_1 \prec s_3$.

- **Peers**: The peers of a strand $s$ is the set of strands that are logically parallel to $s$. 

the peers of c are highlighted in peer above

Stable Views

- **Serial Walk** of a computation dag is the list of instructions encountered in a depth-first execution in which spawn edges are followed before continue edges

- **Stable-View Theorem**: Let $r$ be a reducer with an associative operator $\otimes$. Consider a series walk of $G$, and let $a_1, \ldots, a_k$ be the update amount to $r$ after strand $x$ and strand $y$. Denote the view for $r$ in $x$ by $r_x$ and the view in $y$ by $r_y$. If $\text{peers}(x) = \text{peers}(y)$, then we have:

$$r_y = r_x \otimes a_1 \otimes a_2 \otimes \ldots \otimes a_k$$

In this case, we say that $r_y$ is stable with respect to $r_x$
**Corollary:** If \( r \) is only updated through its associative operator \( \otimes \), the result of \( a_1 \otimes ... \otimes a_k \) is deterministic

**How Cilk Maintains Views**

- **Upon a cilk_spawn:**
  - the child owns the view \( h \) owned by the parent before the cilk_spawn
  - the parent owns a new view \( h' \), initialized to the identity \( e \)

- **After a spawned child returns:**
  - the parent owns the child’s view \( h \), which is reduced with the parent’s view \( h' \) sometime before the cilk_sync, and \( h' \) is destroyed

**Combining Hypermaps**

- A worker maintains a hashtable per trace that maps from the reducer object to the appropriate local view for the trace

- An access to the view of a reducer \( r \) causes the worker to look up the local view for \( r \) in the hypermap. If the view of \( r \) does not exist in the hypermap, the worker creates a new view with value \( e \).

- When a worker finishes its subcomputation, hypermaps are combined using the appropriate reduce() functions
  - Each full frame in the steal-tree contains two place holders for hypermaps
  - Each frame in the steal-tree contains three hypermaps: user, left child(lchild), and right sibling (rsib)
  - Whenever a full frame is returning, it first combines its user with lchild and rsib (if there is one), and deposits itself into the appropriate placeholder.