Lecture 5: Futures — Tasks with Return Values

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu
Solution to Worksheet 4

- Estimate $T(S,P) \sim \frac{\text{WORK}(G,S)}{P} + \text{CPL}(G,S) = \frac{S-1}{P} + \log_2(S)$ for the parallel array sum computation shown in slide 4.

- Assume $S = 1024 \Rightarrow \log_2(S) = 10$

- Compute for 10, 100, 1000 processors
  - $T(P) = \frac{1023}{P} + 10$
  - $\text{Speedup}(10) = \frac{T(1)}{T(10)} = \frac{1033}{112.3} \approx 9.2$
  - $\text{Speedup}(100) = \frac{T(1)}{T(100)} = \frac{1033}{20.2} \approx 51.1$
  - $\text{Speedup}(1000) = \frac{T(1)}{T(1000)} = \frac{1033}{11.0} \approx 93.7$

- Why does the speedup not increase linearly in proportion to the number of processors?
  - Because of the critical path length, $\log_2(S)$, is a bottleneck
Worksheet 4 - Speedup Chart
(linear scale)

\[ speedup = \frac{1033}{\left( \frac{1023}{P} \right) + 10} \]
Functional Parallelism: Adding Return Values to Async Tasks

Example Scenario (PseudoCode)

// Parent task creates child async task
future<Integer> container = future { return computeSum(X,low,mid); };

// Later, parent examines the return value
Integer sum = container.get();

Two issues to be addressed:
1) Distinction between container and value in container (box)
2) Synchronization to avoid race condition in container accesses

Parent Task

Child Task

\(\text{container} = \text{future}\{\ldots\}\)
\(\text{computeSum}(\ldots)\)

\(\text{container.get()}\)

\(\text{return value}\)
future<T> f = future {Stmt-Block}

- Creates a new child task to execute Stmt-Block, which returns a value of type T
- The future expression has type future<T>

Expr.get()

- Evaluate Expr, and wait until Expr’s value is becomes available
- Unlike finish which waits for all tasks in the finish scope, a get() operation only waits for the specified future task
future\(<T>\) f = future \{Stmt-Block\}

- Creates a new child task to execute \texttt{Stmt-Block}, which \texttt{return}\s a value of type \texttt{T}
- The future expression has type \texttt{future}\(<T>\)

\texttt{Expr.get()}

- Evaluate \texttt{Expr}, and wait until \texttt{Expr}'s value is becomes available
- Unlike finish which waits for all tasks in the finish scope, a \texttt{get()} operation only waits for the specified future task

Only one task can possibly modify the contents of the container!
Future Task Declarations and Uses

• Variable of type future is a reference to a future object
  — Container for return value from future task
  — The reference to the container is also known as a “handle”

• Two operations that can be performed on variable V of type future:
  — Assignment: V can be assigned value of type future
  — Blocking read: V.get() waits until the future task referred to by V has completed, and then propagates the return value
  — Supports waiting on a specific task, in contrast to finish which waits on all tasks in scope
Example: Two-way Parallel Array Sum using Future Tasks

1. // Parent Task T1 (main program)
2. // Compute sum1 (lower half) & sum2 (upper half) in parallel
3. future<Integer> sum1 = future { // Future Task T2
4.   int sum = 0;
5.   for(int i = 0; i < X.length / 2; i++) sum += X[i];
6.   return sum;
7. }
8. future<Integer> sum2 = future { // Future Task T3
9.   int sum = 0;
10.  for(int i = X.length / 2; i < X.length; i++) sum += X[i];
11.  return sum;
12. }
13. // Task T1 waits for Tasks T2 and T3 to complete
14. int total = sum1.get() + sum2.get();
Future Tasks vs. Regular Async Versions of Two-Way Array Sum

- Future task version initializes two references to future objects, sum1 and sum2
- No finish construct needed in this example
  - Instead parent task waits for child tasks by performing sum1.get() and sum2.get()
- Easier to guarantee absence of race conditions in Future Task version
  - No race on sum because it is declared as a local variable in both tasks T2 and T3
  - No race on future variables, sum1 and sum2, because of blocking-read semantics
Recursive Array Sum (Sequential version)

Sequential divide-and-conquer pattern:

1. int sum = computeSum(X, 0, X.length-1); // main
2. static int computeSum(int[] X, int lo, int hi) {
3.     if ( lo > hi ) return 0;
4.     else if ( lo == hi ) return X[lo];
5.     else {
6.         int mid = (lo+hi)/2;
7.         int sum1 =
8.             computeSum(X, lo, mid);
9.         int sum2 =
10.            computeSum(X, mid+1, hi);
11.         return sum1 + sum2;
12.     }
13. } // computeSum
Recursive Array Sum using Future Tasks
(Two futures per method call)

Parallel divide-and-conquer pattern:

1. int sum = computeSum(X, 0, X.length-1); // main
2. static int computeSum(int[] X, int lo, int hi) {
3.     if ( lo > hi ) return 0;
4.     else if ( lo == hi ) return X[lo];
5.     else {
6.         int mid = (lo+hi)/2;
7.         future<int> sum1 = future {
8.             computeSum(X, lo, mid); 
9.         };
10.        future<int> sum2 = future {
11.            computeSum(X, mid+1, hi); 
12.        };
13.        // Parent now waits for the container values
14.        return sum1.get() + sum2.get();
15.     }
16. } // computeSum
Computation Graph Extensions for Future Tasks

• Since a get() is a blocking operation, it must occur on boundaries of CG nodes/steps
  — May require splitting a statement into sub-statements e.g.,
  - 12: int sum = sum1.get() + sum2.get();
    can be split into three sub-statements
    - 12a: int temp1 = sum1.get();
    - 12b: int temp2 = sum2.get();
    - 12c: int sum = temp1 + temp2;

• Spawn-edge connects parent task to child future task, as before
• Join-edge connects end of future task to Immediately Enclosing Finish (IEF), as before
• Additional join edges are inserted from end of future task to each get() operation on future object
CG for Two-way Parallel Array Sum using Future Tasks

1. `int sum = computeSum(X, 0, X.length-1);` // main
2. `static int computeSum(int[] X, int lo, int hi) {
3. if ( lo > hi ) return 0;
4. else if ( lo == hi ) return X[lo];
5. else {
6. int mid = (lo+hi)/2;
7. `future<int> sum1 = future {
8. computeSum(X, lo, mid);`;
9. `future<int> sum2 = future {
10. computeSum(X, mid+1, hi);`;
11. `// Parent now waits for the container values
12. return sum1.get() + sum2.get();
13. }` // computeSum
14. } // computeSum

Computation graph of the program from Slide 10 when input array has length of 2

Stmt8 = Future task for sum1
Stmt10 = Future task for sum2
CG for Two-way Parallel Array Sum using Future Tasks

1. int sum = computeSum(X, 0, X.length-1); // main
2. static int computeSum(int[] X, int lo, int hi) {
3.   if ( lo > hi ) return 0;
4.   else if ( lo == hi ) return X[lo];
5.   else {
6.     int mid = (lo+hi)/2;
7.     future<int> sum1 = future {
8.       computeSum(X, lo, mid); }
9.     future<int> sum2 = future {
10.    computeSum(X, mid+1, hi); }
11.   // Parent now waits for the container values
12.   return sum1.get() + sum2.get();
13. }
14. } // computeSum

// Where should doWork() for + be placed?
12a: int temp1 = sum1.get();
12b: int temp2 = sum2.get();
12c: int sum = temp1 + temp2;

Computation graph of the program from Slide 10 when input array has length of 2

Stmt8 = Future task for sum1
Stmt10 = Future task for sum2
Announcements & Reminders

• IMPORTANT:
  – Watch video & read handout for topic 2.2 for next lecture on Friday, Jan 18th

• HW1 was posted on the course web site (http://comp322.rice.edu) on Jan 9th, and is due on Jan 23.

• Quiz for Unit 1 (topics 1.1 - 1.5) posted today and is due by Friday, Jan 25th on Canvas

• See course web site for all work assignments and due dates

• Use Piazza (public or private posts, as appropriate) for all communications re. COMP 322

• See Office Hours link on course web site for latest office hours schedule.