COMP 322: Fundamentals of Parallel Programming

Lecture 5: Parallel Array Sum and Array Reductions

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Announcements

• Homework 2 is due by 5pm today

• Homework 3 will be assigned on Monday, Jan 24th and will be due two weeks later on Monday, Feb 7th
  – This is a programming assignment with abstract performance metrics
  – To prepare for HW3, please make sure that you can compile and run the programs from Lab 2 on your own, using the -perf option. In case of problems, please send email to comp322-staff @ mailman.rice.edu

• Graded Homework 1 assignments will be emailed to you by Monday, Jan 24th
Acknowledgments for Today’s Lecture

- COMP 322 Lecture 5 handout
Sequential Array Sum Program (Lecture 1)

```java
int sum = 0;
for (int i=0 ; i < X.length ; i++)
    sum += X[i];
```

- The original computation graph is sequential
- We studied a 2-task parallel program for this problem
- How can we expose more parallelism?
Reduction Tree Schema for computing Array Sum in parallel

Observations:

- This algorithm overwrites \( X \) (make a copy if \( X \) is needed later)
- \( stride = \) distance between array subscript inputs for each addition
- \( size = \) number of additions that can be executed in parallel in each level (stage)
Parallel Program that satisfies dependences in Reduction Tree schema (for X.length = 8)

```cpp
finish { // STAGE 1: stride = 1, size = 4 parallel additions
    async X[0]+=X[1];  async X[2]+=X[3];
}

finish { // STAGE 2: stride = 2, size = 2 parallel additions
    async X[0]+=X[2];  async X[4]+=X[6];
}

finish { // STAGE 3: stride = 4, size = 1 parallel additions
    async X[0]+=X[4];
}
```
Generalization to arbitrary sized arrays
(ArraySum1)

for ( int stride = 1; stride < X.length ; stride *= 2 ) {
   // Compute size = number of additions to be performed in stride
   int size=ceilDiv(X.length,2*stride);
   finish for(int i = 0; i < size; i++)
      async {
         if ( (2*i+1)*stride < X.length )
            X[2*i*stride]+=X[(2*i+1)*stride];
      } // finish-for-asnc
} // for

// Divide x by y, round up to next largest int, and return result
static int ceilDiv(int x, int y) { return (x+y-1) / y; }
Complexity Analysis of ArraySum1

- Define $n = X.length$
- Assume that each addition takes 1 unit of time
  - Ignore all other computations since they are related to the addition by some constant
- Total number of additions, $WORK = n-1 = O(n)$
- Critical path length (number of stages), $CPL = \text{ceiling}(\log_2(n)) = O(\log(n))$
- Ideal parallelism = $WORK/CPL = O(n) / O(\log(n))$
- Consider an execution on $p$ processors
  - Compute partial sum for $n/p$ elements on each processor
  - Use ArraySum1 program to reduce $p$ partial sums to one total sum
  - $CPL$ for this version is $O(n/p + \log(p))$
  - Parallelism for this version is $O(n) / O(n/p + \log(p))$
  - Algorithm is optimal for $p = n / \log(n)$, or fewer, processors - why?
Observations:

- Computation graph has extra dependences relative to schema e.g., \( X[0] += X[2] \) must follow \( X[4] += X[5] \).

- Extra dependences can make a difference if computations in same stage take different times e.g., if \( X[4] += X[5] \) and \( X[0] += X[2] \) take 100 time units each.

- How can we write a program that avoids these extra dependences?
Extra dependences in ArraySum1 program

- - - - - 

Extra dependence edges due to finish-async stages
Summing an arbitrary sized array using a Recursive method and Future Tasks (ArraySum2)

```java
static int computeSum(int[] X, int lo, int hi) {
    if ( lo > hi ) return 0;
    else if ( lo == hi ) return X[lo];
    else {
        int mid = (lo+hi)/2;
        final future<int> sum1 =
            async<int> {return computeSum(X, lo, mid);};
        final future<int> sum2 =
            async<int> {return computeSum(X, mid+1, hi);};
        return sum1.get() + sum2.get();
    }
} // computeSum
int sum = computeSum(X, 0, X.length-1); // main program code
```

Can be replaced by finish-async, but future tasks are more natural.
Parallel Array Reductions

• Why all this focus on array sum?

• ArraySum1 and ArraySum2 programs can easily be adapted to reduce any associative function \( f \)
  
  \(- f(x,y) \) is said to be associative if \( f(a,f(b,c)) = f(f(a,b),c) \) for any inputs \( a, b, \) and \( c \)

• Sequential version of array reduction:

  ```java
  int result=X[0];
  for(int i=1 ; i < X.length ; i++ ) result=f(result,X[i]);
  ```

• General reductions have many interesting applications in practice, as you will see when we learn about Google’s Map Reduce framework

• Motivates complexity analysis where evaluation of a single call to \( f() \) is assumed to take 1 unit of time (could be much larger than an integer add, and justify the use of an async)
Extension of ArraySum1 to reduce an arbitrary associative function, \( f \)

```java
for ( int stride = 1; stride < X.length ; stride *= 2 ) {
    // Compute size = number of additions to be performed in stride
    int size=ceilDiv(X.length,2*stride);
    finish for(int i = 0; i < size; i++)
        async {
            if ( (2*i+1)*stride < X.length )
                X[2*i*stride] = f(X[2*i*stride], X[(2*i+1)*stride]);
        } // finish-for-async
} // for

// Divide x by y, round up to next largest int, and return result
static int ceilDiv(int x, int y) { return (x+y-1) / x; }
```
Extension of ArraySum2 to reduce an arbitrary associative function, f

```java
static int computeSum(int[] X, int lo, int hi) {
    if (lo > hi) return identity();
    else if (lo == hi) return X[lo];
    else {
        int mid = (lo+hi)/2;
        final future<int> sum1 =
            async<int> {return computeSum(X, lo, mid);};
        final future<int> sum2 =
            async<int> {return computeSum(X, mid+1, hi);};
        return f(sum1.get(), sum2.get());
    }
} // computeSum

int sum = computeSum(X, 0, X.length-1); // main program code
```