COMP 322: Fundamentals of Parallel Programming

Lecture 3: Computation Graphs, Abstract Performance Metrics

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Announcements

- Coursera forum on HJ Environment and Setup Issues
  - Please post your issues, and also respond to postings by other students when you can help
- Instructor's office hours are during 2pm - 3pm on MWF
  - Please stop by if you have problems with any of the following
    - Accessing the Module 1 handout
    - Using the turnin script
    - You did not receive any email sent to comp322-all
- Homework 1 has been posted
  - Contains written and programming components
  - Due by 5pm on Wednesday, Jan 23rd
  - Must be submitted using "turnin" script introduced in Lab 1
    - In case of problems, email a zip file to comp322-staff at mailman.rice.edu before the deadline
  - See course web site for penalties for late submissions

Complexity Measures for Computation Graphs (Recap)

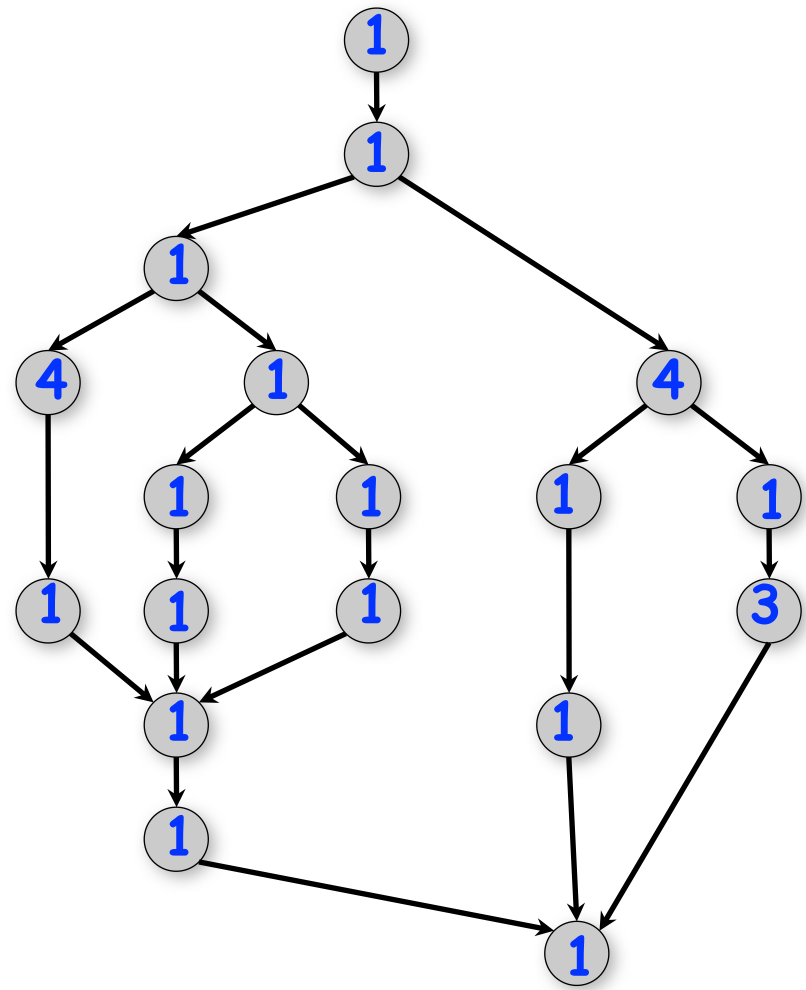
Define

- TIME(N) = execution time of node N
- WORK(G) = sum of TIME(N), for all nodes N in CG G
  - WORK(G) is the total work to be performed in G
- CPL(G) = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called critical paths
  - CPL(G) is the length of these paths (critical path length)
  - CPL(G) is also the smallest possible execution time for the computation graph

Ideal Parallelism (Recap)

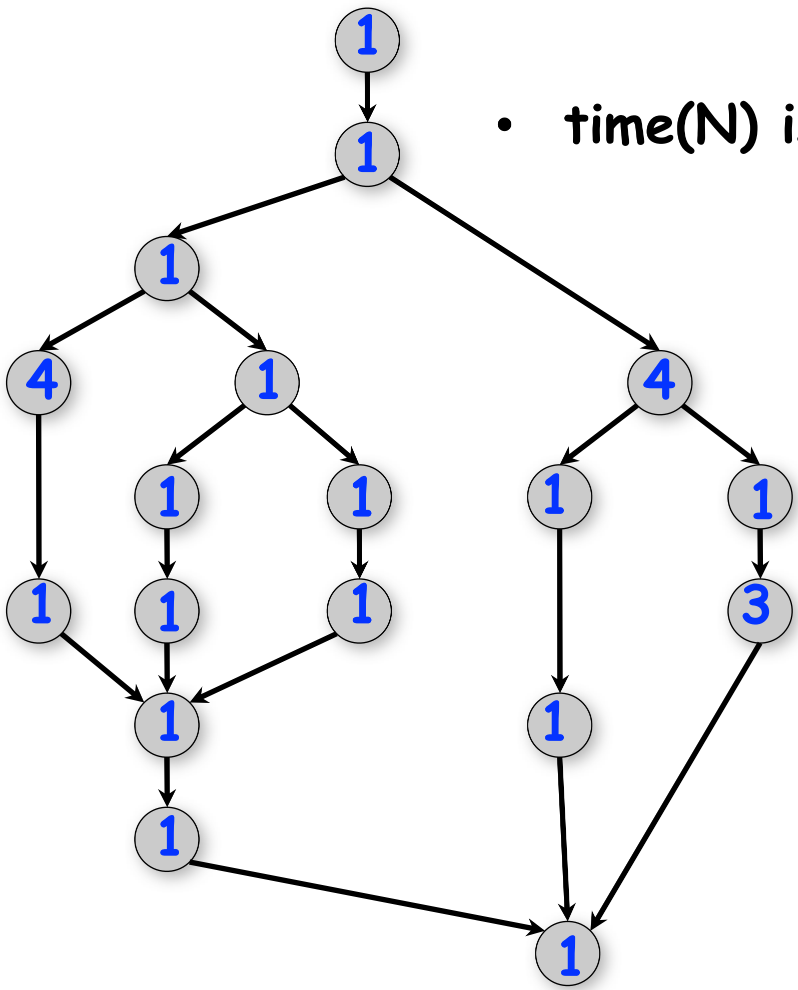
Define ideal parallelism of Computation Graph G as the ratio, WORK(G)/CPL(G)

Ideal Parallelism is independent of the number of processors that the program executes on, and only depends on the computation graph



Solution to Worksheet #2: what is the critical path length and ideal parallelism of this graph?

CPL(G) = length of a longest path in computation graph G



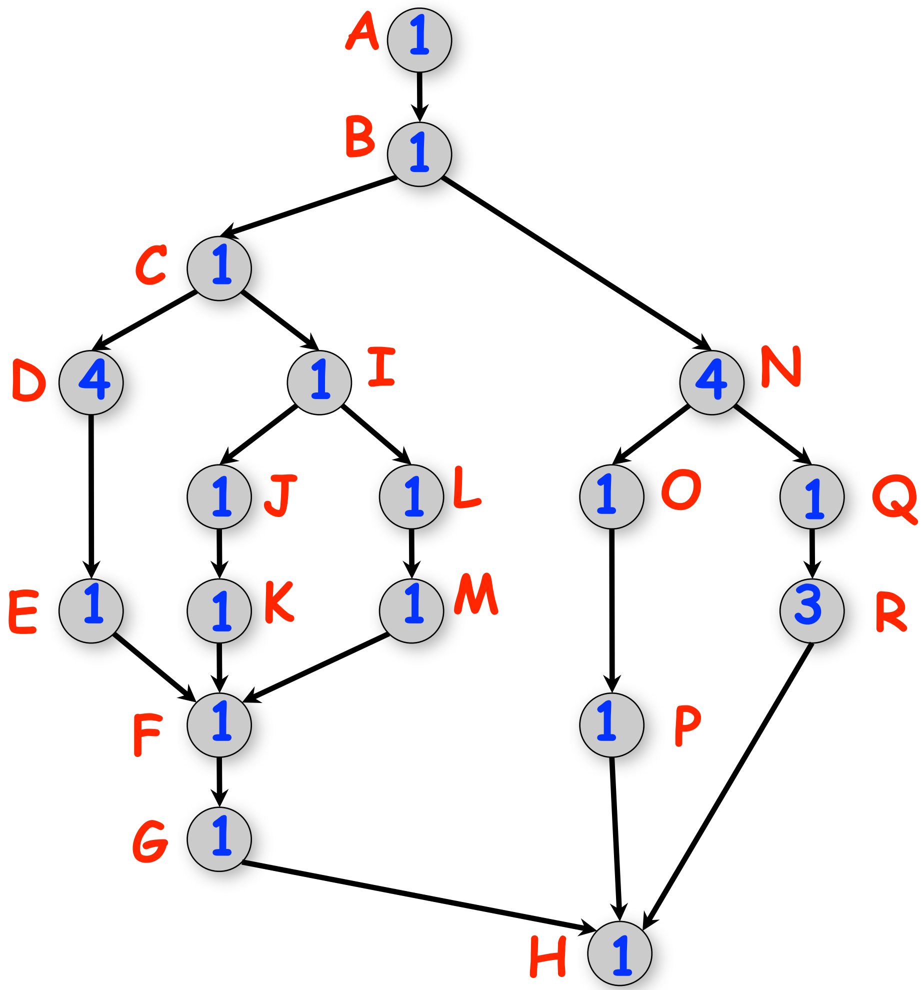
- time(N) is labeled for all nodes N in the graph

WORK(G) = 26

CPL(G) = 11

Ideal Parallelism
= WORK(G)/CPL(G)
= 26 / 11 ~ 2.36

Scheduling of a Computation Graph on a fixed number of processors: Example



Start time	Proc 1	Proc 2	Proc 3
0	A		
1	B		
2	C	N	
3	D	N	I
4	D	N	J
5	D	N	K
6	D	Q	L
7	E	R	M
8	F	R	O
9	G	R	P
10	H		
11			

Scheduling of a Computation Graph on a fixed number of processors, P

- Assume that node N takes TIME(N) regardless of which processor it executes on, and that there is no overhead for creating parallel tasks
- A schedule specifies the following for each node
  - START(N) = start time
  - PROC(N) = index of processor in range 1...P

  such that
  - START(i) + TIME(i) <= START(j), for all CG edges from i to j (Precedence constraint)
  - A node occupies consecutive time slots in a processor (Non-preemption constraint)
  - All nodes assigned to the same processor occupy distinct time slots (Resource constraint)

Lower Bounds on Execution Time of Schedules

- Let T_P = execution time of a schedule for computation graph G on P processors
  - Can be different for different schedules
- Lower bounds for all greedy schedules
  - Capacity bound: $T_P \geq WORK(G)/P$
  - Critical path bound: $T_P \geq CPL(G)$
- Putting them together
  - $T_P \geq \max(WORK(G)/P, CPL(G))$

Greedy Schedule

- A greedy schedule is one that never forces a processor to be idle when one or more nodes are ready for execution
- A node is ready for execution if all its predecessors have been executed
- Observations
  - T_1 = WORK(G), for all greedy schedules
  - T_∞ = CPL(G), for all greedy schedules

Upper Bound on Execution Time of Greedy Schedules

Theorem [Graham '66]. Any greedy scheduler achieves

$$T_P \leq WORK(G)/P + CPL(G)$$

Proof sketch:

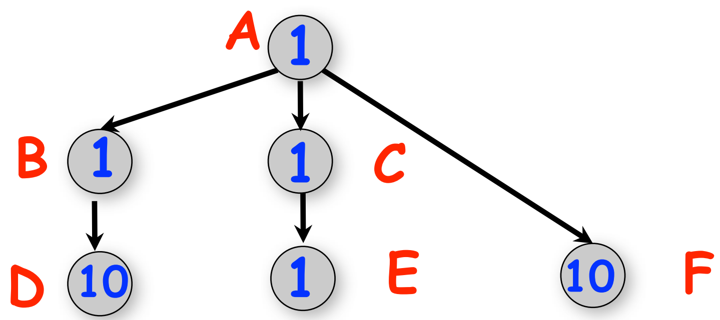
Define a time step to be complete if $\geq$ P nodes are ready at that time, or incomplete otherwise

complete time steps $\leq$ WORK(G)/P

incomplete time steps $\leq$ CPL(G)

Start time	Proc 1	Proc 2	Proc 3
0	A		
1	B		
2	C	N	
3	D	N	I
4	D	N	J
5	D	N	K
6	D	Q	L
7	E	R	M
8	F	R	O
9	G	R	P
10	H		
11			

What are the best-case and worst-case schedules that we can obtain for this example on 2 processors?



- WORK(G) = 24
- CPL(G) = 12
- For P=2, WORK(G)/P = 12
- Lower bound = max(12,12) = 12
- Upper bound = 12 + 12 = 24
- Best (13) and worst (14) values for T_2 are in the range, 12 ... 24

Best case, T_2 = 13

Start time	Proc 1	Proc 2
0	A	
1	B	F
2	D	F
3	D	F
4	D	F
5	D	F
6	D	F
7	D	F
8	D	F
9	D	F
10	D	F
11	D	C
12		E
13		

Worst case, T_2 = 14

Start time	Proc 1	Proc 2
0	A	
1	F	B
2	F	C
3	F	E
4	F	D
5	F	D
6	F	D
7	F	D
8	F	D
9	F	D
10	F	D
11		D
12		D
13		D
14		

Bounding the performance of Greedy Schedulers

Combine lower and upper bounds to get

$\max(WORK(G)/P,\ CPL(G)) \le T_P \le WORK(G)/P + CPL(G)$

Corollary 1: Any greedy scheduler achieves execution time T_P that is within a factor of 2 of the optimal time (since max(a,b) and (a+b) are within a factor of 2 of each other, for any $a \ge 0, b \ge 0$).

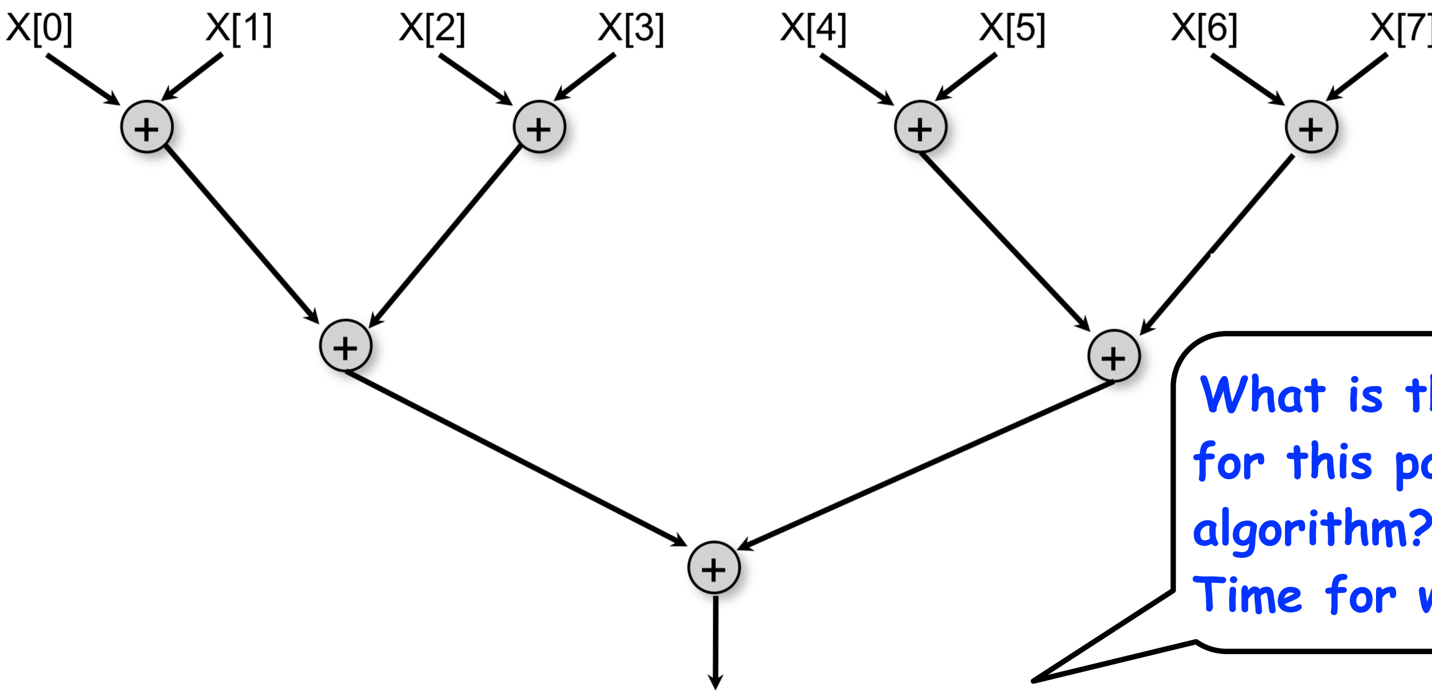
Corollary 2: Lower and upper bounds approach the same value whenever

- There's lots of parallelism, WORK(G)/CPL(G) >> P
- Or there's little parallelism, WORK(G)/CPL(G) << P

Strong Scaling and Speedup

- Define Speedup(P) = T_1 / T_P
  - —Factor by which the use of P processors speeds up execution time relative to 1 processor, for a fixed input size
  - —For ideal executions without overhead, 1 <= Speedup(P) <= P
  - —Linear speedup
    - – When Speedup(P) = k*P, for some constant k, 0 < k < 1
- Referred to as “strong scaling” because input size is fixed

Reduction Tree Schema for computing Array Sum in parallel

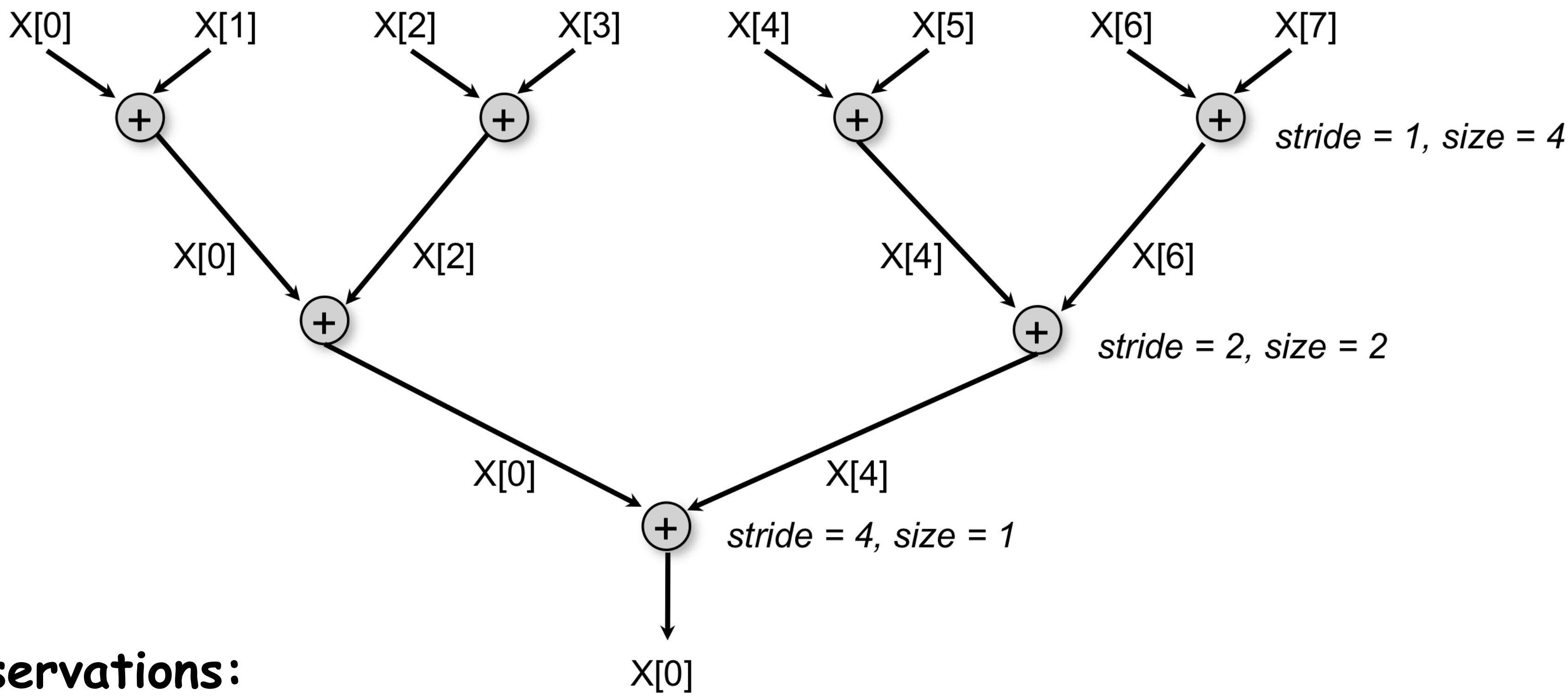


What is the speedup for this parallel algorithm?
Time for worksheet #3!

Assume input array size = S, and each add takes 1 unit of time:

- WORK(G) = S-1
- CPL(G) = log2(S)
- Assume T_P = WORK(G)/P + CPL(G) = (S-1)/P + log2(S)
  - Within a factor of 2 of any schedule's execution time

Algorithm based on updates to array



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)

Async-Finish Parallel Program for Array Sum (for X.length = 8)

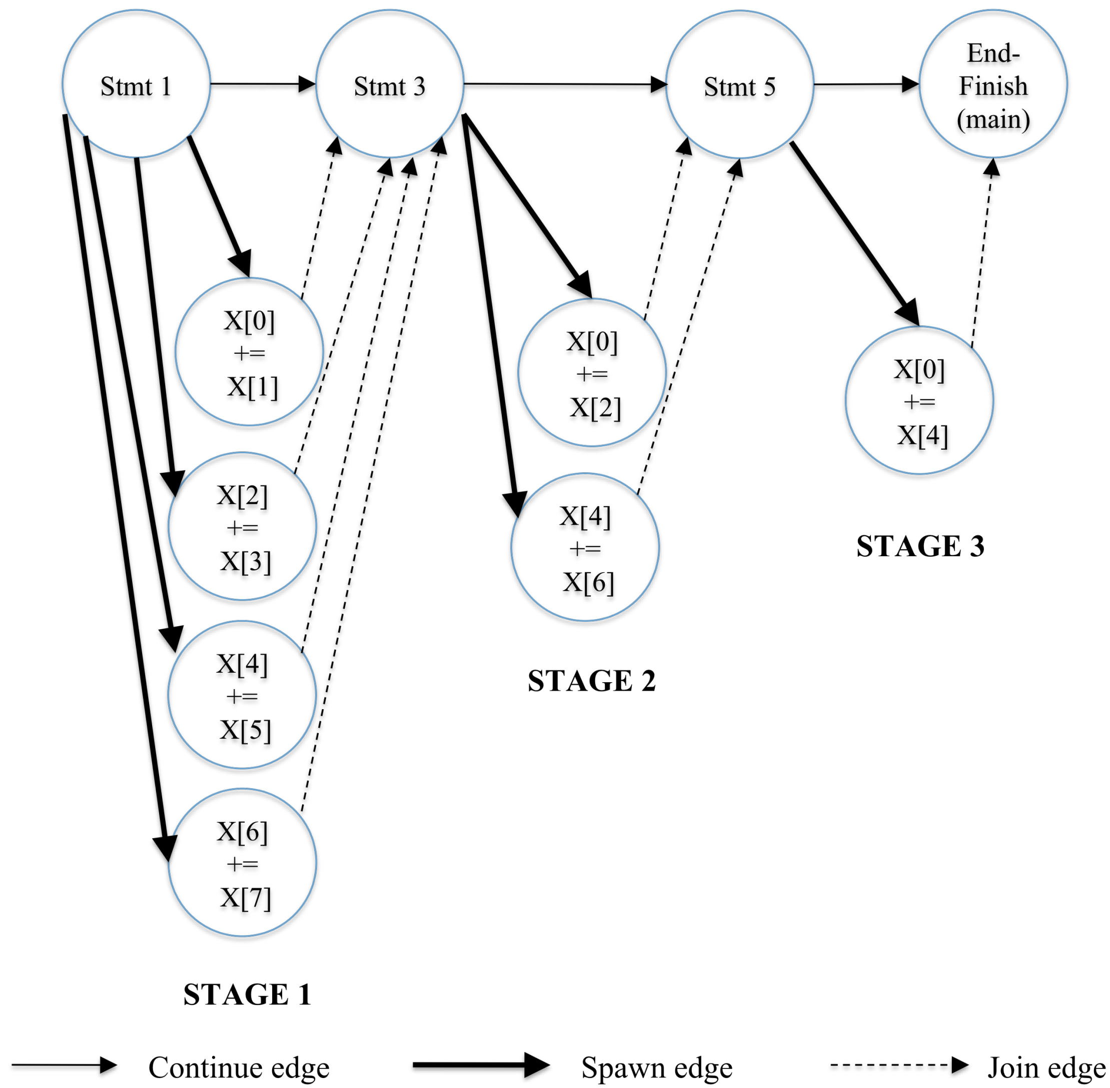
```
finish { //STAGE 1: stride = 1, size = 4 parallel additions
  async X[0]+=X[1]; async X[2]+=X[3];
  async X[4]+=X[5]; async X[6]+=X[7];
}
finish { //STAGE 2: stride = 2, size = 2 parallel additions
  async X[0]+=X[2]; async X[4]+=X[6];
}
finish { //STAGE 3: stride = 4, size = 1 parallel additions
  async X[0]+=X[4];
}
// Final sum is now in X[0]
```

Generalization to arbitrary sized arrays (ArraySum1)

```
for ( int stride = 1; stride < X.length ; stride *= 2 ) {
  // Compute size = number of adds 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-async
} // for

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

Computation Graph for ArraySum1



HJ Abstract Performance Metrics

- **Basic Idea**
  - Count operations of interest, as in big-O analysis
  - Abstraction ignores overheads that occur on real systems

- **Calls to perf.doWork()**
  - Programmer inserts calls of the form, perf.doWork(N), within a step to indicate abstraction execution of N application-specific abstract operations
    - e.g., adds, compares, stencil ops, data structure ops
  - Multiple calls add to the execution time of the step

- **Enabled by selecting "Show Abstract Execution Metrics" in DrHJ compiler options (or -perf=true runtime option)**
  - If an HJ program is executed with this option, abstract metrics are printed at end of program execution with WORK(G), CPL(G), Ideal Speedup = WORK(G)/ CPL(G)

Inserting call to perf.doWork() in ArraySum1

```
for ( int stride = 1; stride < X.length ; stride *= 2 ) {
  // Compute size = number of adds 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 ) {
        perf.doWork(1);
        X[2*i*stride] += X[(2*i+1)*stride];
      }
    } // finish-for-async
} // for
```

Worksheet #3: Strong Scaling for Array Sum

Name 1: ______________________ Name 2: ______________________

- Assume T(S,P) ~ WORK(G,S)/P + CPL(G,S) = (S-1)/P + log2(S) for a parallel array sum computation
- Strong scaling
  - Assume S = 1024 ==> log2(S) = 10
  - Compute Speedup(P) for 10, 100, 1000 processors
    - T(P) = 1023/P + 10
    - Speedup(10) = T(1)/T(10) =
    - Speedup(100) = T(1)/T(100) =
    - Speedup(1000) = T(1)/T(1000) =
  - Why is it worse than linear?