

# COMP 322: Fundamentals of Parallel Programming

## Lecture 12: Abstract Metrics, Parallel Speedup and Amdahl's Law

Mack Joyner  
mjoyner@rice.edu

<http://comp322.rice.edu>



# Abstract Performance Metrics

- Basic Idea
  - Count operations of interest, as in big-O analysis, to evaluate parallel algorithms
  - Abstraction ignores many overheads that occur on real systems
- Calls to `doWork()`
  - Programmer inserts calls of the form, `doWork(N)` within a task (async, future task or data-driven task) to indicate abstract execution of N application-specific abstract operation
    - e.g., in lab 3, we included one call to `doWork(1)` for each double addition, and ignore the cost of everything else
- Abstract metrics are enabled by calling `HjSystemProperty.abstractMetrics.set(true)` at start of program execution
- If an HJ program is executed with this option, abstract metrics can be printed at end of program execution with calls to `abstractMetrics().totalWork()`, `abstractMetrics().criticalPathLength()`, and



# Abstract Performance Metrics

- Pay attention where you put `doWork()` calls
- What does this mean?

```
var bottom = future(() -> . . .);  
var top = future(() -> . . .)  
doWork(1);  
return bottom.get() + top.get();
```

- Correct:

```
var bottom = future(() -> . . .);  
var top = future(() -> . . .);  
  
var bottomVal = bottom.get();  
var topVal = top.get();  
doWork(1);  
return bottomVal + topVal;
```



# Data Races

A data race occurs on location  $L$  in a program execution with computation graph  $CG$  if there exist steps (nodes)  $S1$  and  $S2$  in  $CG$  such that:

1.  $S1$  does not depend on  $S2$  and  $S2$  does not depend on  $S1$ , i.e.,  $S1$  and  $S2$  can potentially execute in parallel, and
  2. Both  $S1$  and  $S2$  read or write  $L$ , and at least one of the accesses is a write.
- A data-race is usually considered an error. The result of a read operation in a data race is undefined. The result of a write operation is undefined if there are two or more writes to the same location.
  - Note that our definition of data race includes the case that both  $S1$  and  $S2$  write the same value in location  $L$ , even if the data race is benign.

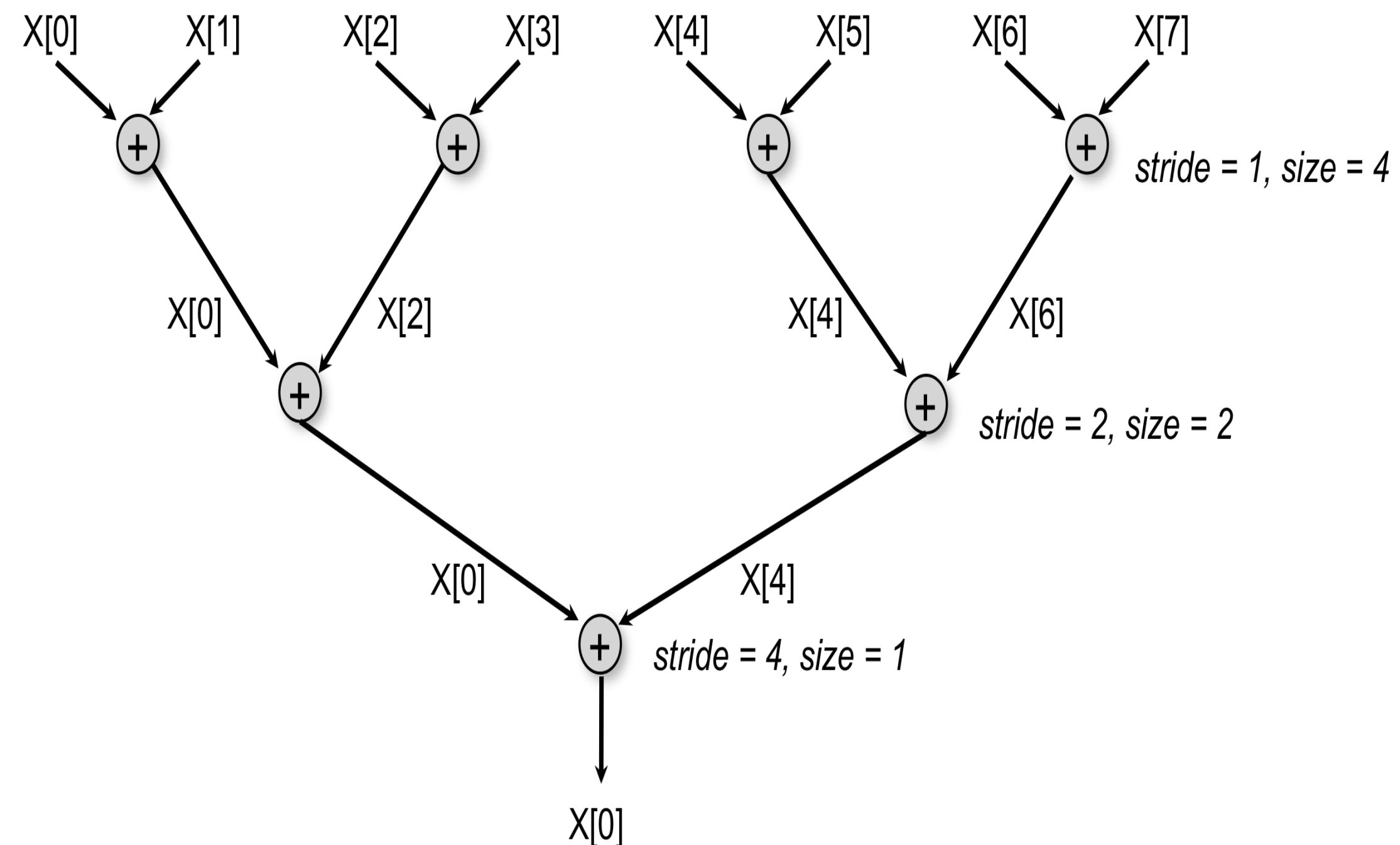


# Parallel Speedup

- Define  $\text{Speedup}(P) = T_1 / T_P$ 
  - Factor by which  $P$  processors speeds up execution time relative to 1 processor, for fixed input size
  - For ideal executions without overhead,  $1 \leq \text{Speedup}(P) \leq P$ 
    - You see this with abstract metrics, but bounds may not hold when measuring real execution times with real overheads
  - Linear speedup
    - When  $\text{Speedup}(P) = k \cdot P$ , for some constant  $k$ ,  $0 < k < 1$
- Ideal Parallelism =  $\text{WORK} / \text{CPL} = T_1 / T_\infty$ 
  - = Parallel Speedup on an unbounded (infinite) number of processors



# Computation Graph for Recursive Tree approach to computing Array Sum in parallel



Assume greedy schedule, input array size  $S$  is a power of 2, each add takes 1 time unit

- $WORK(G) = S-1$ , and  $CPL(G) = \log_2(S)$
- Define  $T(S,P)$  = parallel execution time for Array Sum with size  $S$  on  $P$  processors
- Use upper bound  $T(S,P) \leq WORK(G)/P + CPL(G)$  as a worst-case estimate

$$T(S,P) = WORK(G)/P + CPL(G) = (S-1)/P + \log_2(S) \Rightarrow \text{Speedup}(S,P) = T(S,1)/T(S,P) = (S-1)/((S-1)/P + \log_2(S))$$



# How many processors should we use?

Define  $\text{Efficiency}(P) = \text{Speedup}(P) / P = T_1 / (P * T_P)$

- Processor efficiency --- figure of merit that indicates how well a parallel program uses available processors
- For ideal executions without overhead,  $1/P \leq \text{Efficiency}(P) \leq 1$
- $\text{Efficiency}(P) = 1$  (100%) is the best we can hope for



# How many processors should we use?

---

What should be the minimum efficiency to determine how many processors we should use?





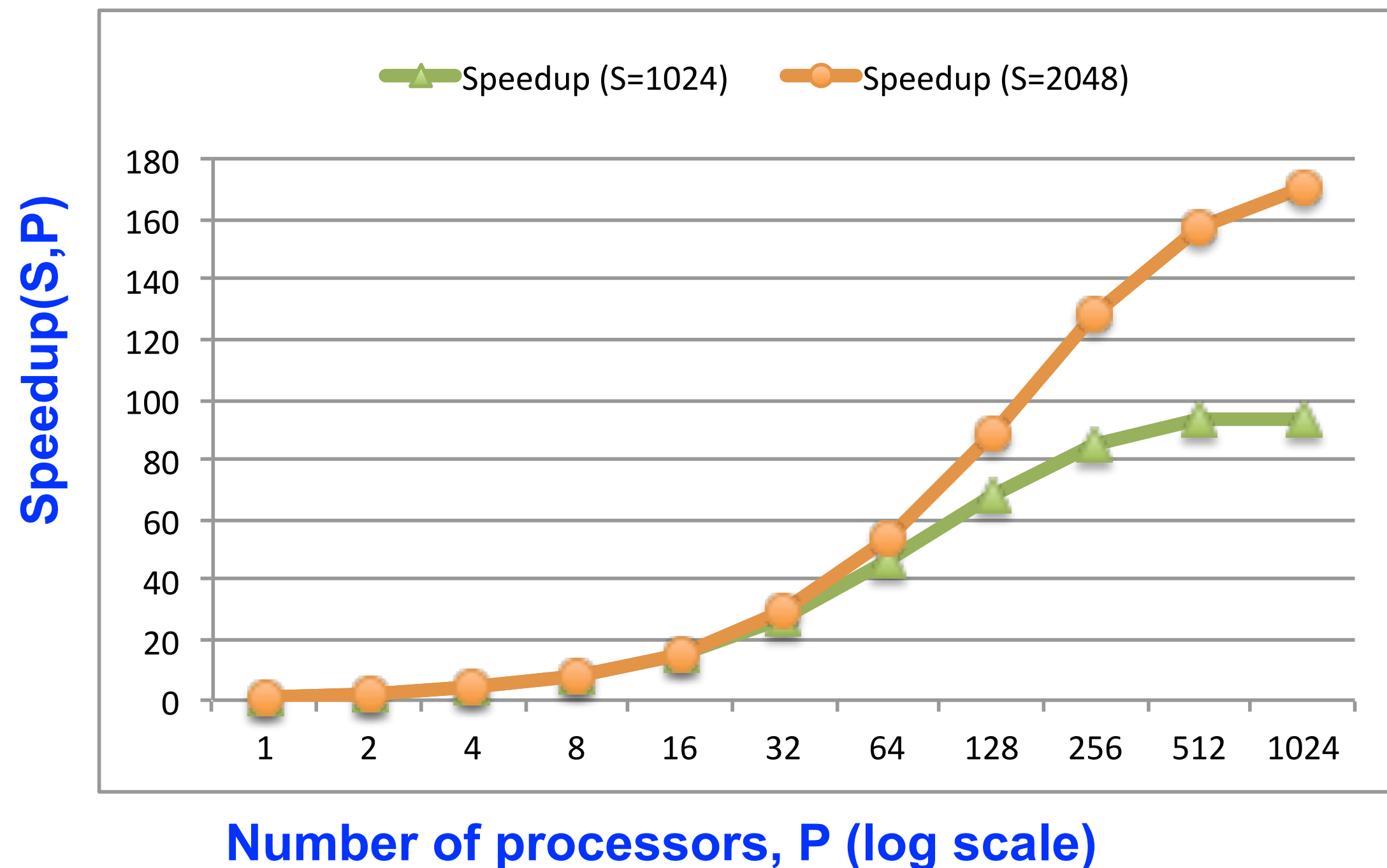
# How many processors should we use?

- Common goal: choose number  $P$  for a given input size,  $S$ , so that efficiency is at least 0.5 (50%)
- **Half-performance metric**
  - $S_{1/2}$  = input size that achieves  $\text{Efficiency}(P) = 0.5$  for a given  $P$
  - Figure of merit that indicates how large an input size is needed to obtain efficient parallelism
  - A larger value of  $S_{1/2}$  indicates that the problem is harder to parallelize efficiently



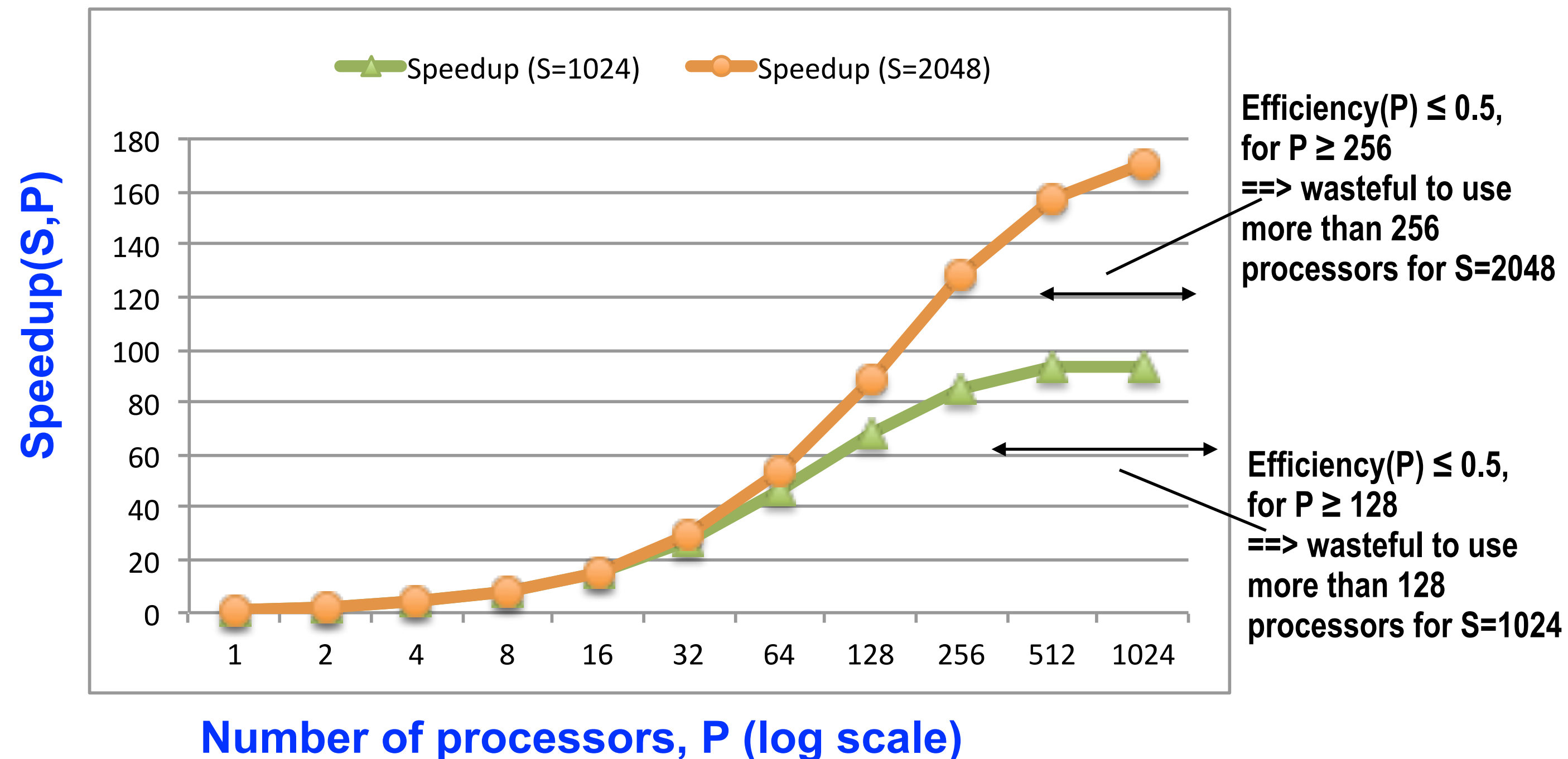
# Array Sum: Speedup as a function of array size $S$ and number of processors $P$

- $\text{Speedup}(S,P) = T(S,1)/T(S,P) = (S-1)/((S-1)/P + \log_2(S))$
- Asymptotically,  $\text{Speedup}(S,P) \rightarrow (S-1)/\log_2 S$ , as  $P \rightarrow \text{infinity}$



# Array Sum: Speedup as a function of array size S and number of processors P

- $\text{Speedup}(S,P) = T(S,1)/T(S,P) = (S-1)/((S-1)/P + \log_2(S))$
- Asymptotically,  $\text{Speedup}(S,P) \rightarrow (S-1)/\log_2 S$ , as  $P \rightarrow \text{infinity}$



# Amdahl's Law

---

If  $q \leq 1$  is the fraction of WORK in a parallel program that must be executed sequentially for a given input size  $S$ , then the best speedup that can be obtained for that program is  $\text{Speedup}(S,P) \leq 1/q$ .



# Amdahl's Law

- Observation follows directly from critical path length lower bound on parallel execution time
  - $CPL \geq q * T(S,1)$
  - $T(S,P) \geq q * T(S,1)$
  - $Speedup(S,P) = T(S,1)/T(S,P) \leq 1/q$
- Upper bound on speedup simplistically assumes that work can be divided into sequential and parallel portions
  - Sequential portion of WORK =  $q$ 
    - also denoted as  $f_s$  (fraction of sequential work)
  - Parallel portion of WORK =  $1-q$ 
    - also denoted as  $f_p$  (fraction of parallel work)



# Illustration of Amdahl's Law: Best Case Speedup as function of Parallel Portion

