Lecture 20 – Multi-Processor Performance

- CPU scalability
- Memory scalability
- Interconnection network
- Bandwidth and latency issues
- Problem size and granularity
- *How many processors can we use efficiently?*

- In the last class, we discussed Amdahl's law for single and multiple processors. Let's first go back and review the multiple-processor model.

Amdahl's Law - Parallel Processing

- Ideally, if a computation can be carried out in *p* equal parts, the total execution time will be nearly *1/p* of the time required by a single processor
- Suppose t_j denotes the wall clock time required to execute a task with j processors
- Speedup, S_p, for p processors is defined as

$$S_p = \frac{t_1}{t_p}$$

- Where t_1 is the time required for the most-efficient sequential algorithm to complete the calculation, and t_p is the time required for the most efficient parallel implementation of the same algorithm, from beginning to end, using p processors.

Amdahl's Law - Parallel Processing

- The *computational efficiency* using *p* processors is defined as

$$E_p = \frac{S_p}{p} \qquad 0 \le E_p \le 1$$

- Then, the *total execution time* using *p* processors is given by

$$t_p = \frac{ft_1}{p} + (1-f)t_1 = \frac{t_1(f+(1-f)p)}{p} \ge (1-f)t_1$$

Amdahl's Law - Parallel Processing

- Therefore, the speedup on p processors is then

$$S_p = \frac{p}{(f+(1-f)p)} \le \frac{1}{1-f}$$

(Ware's Law)

- This shows that the speedup is considerable reduced even for pretty large values of f (close to 95%)

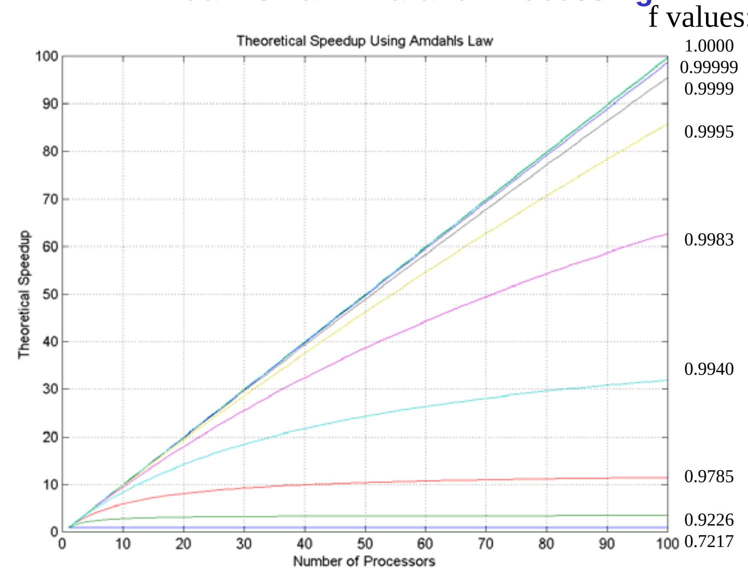
Amdahl's Law - Parallel Processing

- ***Parallel overhead*** **is the additional amount of work that is required on the parallel implementation of a sequential algorithm arising from the use of a parallel computer:**
  - Inter-processor communication
  - Load imbalance
  - Additional computation resulting from the algorithm that is being parallelized not being as efficient as the most efficient serial algorithm.
- **If the total time spent in solving a problem over all processing elements is pT_p and T_s is the time spent doing useful work (consider this the time for a single processor), then T_o is the overhead time:**

$$T_o = pT_p - T_s$$

- **This is the overhead time spent in communication**

Amdahl's Law - Parallel Processing



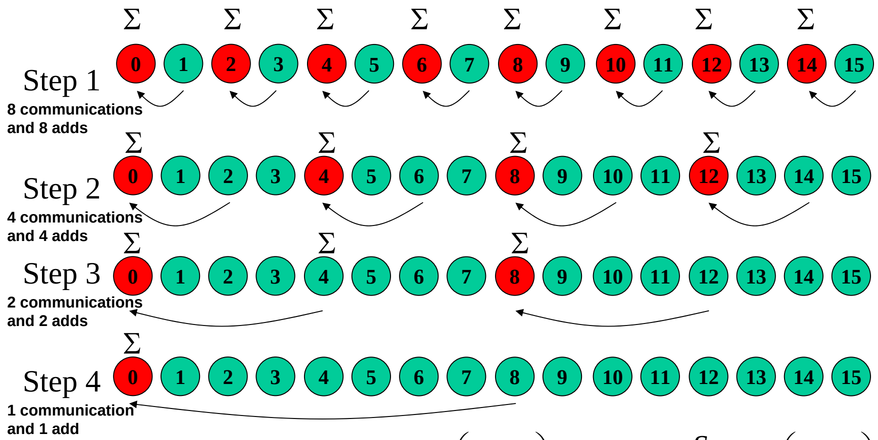
Amdahl's Law - Parallel Processing

- **Amdahl's law is a simplistic, yet powerful way of looking at the problem of scalability.**
- **In a naïve way, it points out that a large number of processors cannot be used on any computational task, since f needs to be very close to 1. For example, f=0.999 would allow the use of a maximum number of processors equal to 1000.**

Example - Adding on a Hypercube

- **Consider adding *n* numbers using *n* processors of a hypercube (*n* is a power of two)**
- **Initially, each processor gets one number, and at the end, one processor has the sum of all of them**
- **Addition and communication take each 1 unit of time.**
  - **The addition can be performed in some constant time, t_c.**
  - **The communication of a single word can be performed in time t_s+t_w where**
    - t_s is the start-up time to handle message at the sending and receiving nodes
    - t_w is the per-word transfer time

Example - Adding on a Hypercube
(sum 16 numbers on 16 processors)



$$T_P = \Theta(\log n) \qquad S_p = \Theta\left(\frac{n}{\log n}\right) \qquad E_p = \frac{S_p}{n} = \Theta\left(\frac{1}{\log n}\right)$$

15 communications and 16 adds

see Grama, et. al Chapter 4

Example - Adding on a Hypercube

$$T_P = \Theta(\log n) \qquad S_p = \Theta\left(\frac{n}{\log n}\right) \qquad E_p = \frac{S_p}{n} = \Theta\left(\frac{1}{\log n}\right)$$

- **Performance not impressive:**
  - For 16 numbers and 16 processors, E_p = 25% (Note that all log functions in Introduction to Parallel Computing by Grama et al are in base 2. See Appendix A)

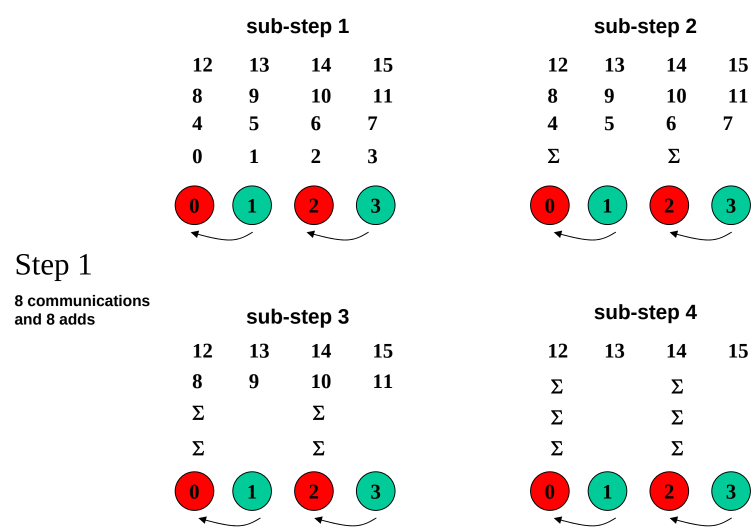
$$\log_2 x = \frac{\log_{10} x}{\log_{10} 2} = \frac{\log_{10} x}{.301}$$

- **Maybe we are using too many processors for this problem? Maybe the problem size is too small?**
- **Let's try again.**

Example - Adding on a Hypercube

- **Assign larger pieces of data to each processor (more than one number per processor). We are increasing the work-load of each processor by doing this.**
- **Consider adding *n* numbers using *p* processors of a hypercube, *p* < *n*, both *p* and *n* are powers of two. Using fewer processors than the maximum number of elements is called *scaling down*.**
- **Initially, each processor gets *n/p* numbers, and at the end, one processor has the sum of all of them**
- **Addition and communication take each 1 unit of time**

Example - Adding on a Hypercube (Non-optimal Algorithm)
(Sum 16 numbers on 4 processors – mimic operations of 16 processors)

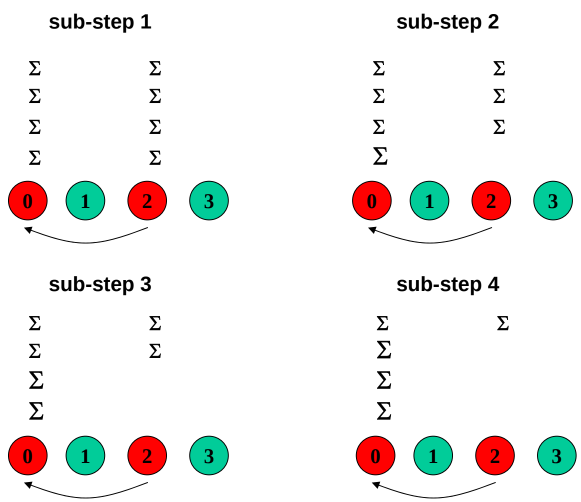


Example - Adding on a Hypercube (Non-optimal Algorithm)

(sum 16 numbers on 4 processors (non-optimally))

Step 2

4 communications and 4 adds

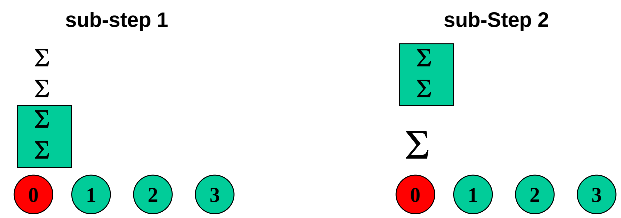


Example - Adding on a Hypercube (Non-optimal Algorithm)

(sum 16 numbers on 4 processors (non-optimally))

Step 3

2 adds

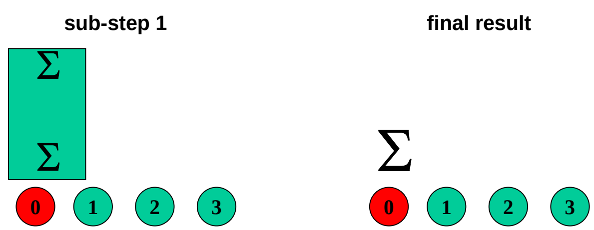


Example - Adding on a Hypercube (Non-optimal Algorithm)

(sum 16 numbers on 4 processors (non-optimally))

Step 4

1 add



- **First *log p* steps (until only one processor left) of the *log n* steps are done in**
  - *(n/p) log p* steps on *p* processors
- **Remaining steps require no communication, therefore, adding *n/p* numbers takes *n/p* time**

12 communications and 15 adds

Example - Adding on a Hypercube

(sum 16 numbers on 4 processors (non-optimally))

- **Expected computational time from our previous example of n-processor hypercube:**

$$T_P = \Theta\left(\frac{n}{p}\log n\right)$$

- **However what we find in this analysis for a p-processor hypercube and n-words:**

$$T_P = \Theta\left(\frac{n}{p}\log p\right)$$

and the speedup and efficiency would be

$$S_p = \Theta\left(\frac{n}{\frac{n}{p}\log p}\right) = \Theta\left(\frac{p}{\log p}\right) \qquad E_p = \frac{S_p}{n} = \Theta\left(\frac{p}{n\log p}\right)$$

- **For 16 words on 4 processors, E_p =12% (worse – because it is non-optimal). Increasing work-load doesn't help!!**

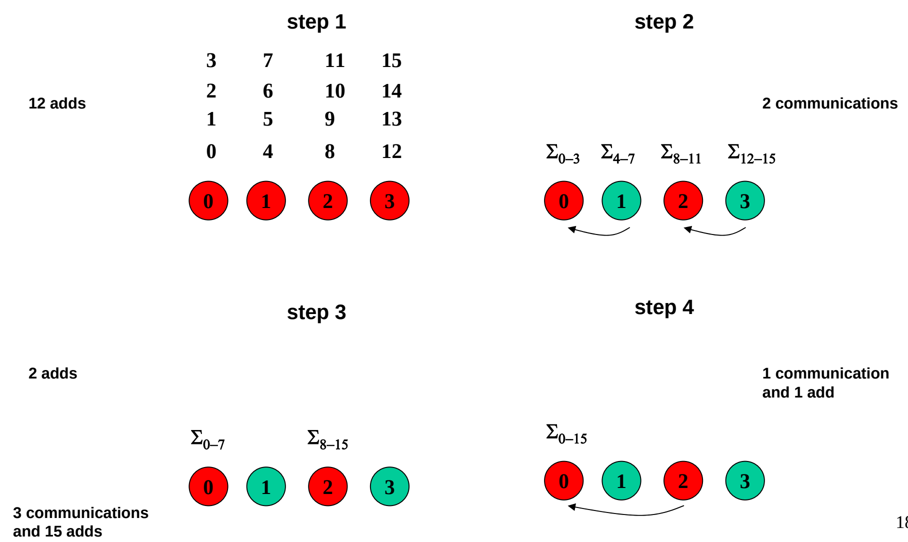
Example - Adding on a Hypercube

(sum 16 numbers on 4 processors (non-optimally))

- **This example shows that if you have a bad parallel algorithm, increasing work-load will not necessarily help**
- **We must go back and change the algorithm to make it more optimal**

Example - Adding on a Hypercube (Optimal)

(Sum 16 numbers on 4 processors optimally)



Example - Adding on a Hypercube

(sum 16 numbers on 4 processors optimally)

- **Total computation time, assuming it takes one unit of time to add two numbers and one unit of time to transfer a number between neighboring processors**

$$T_P = \left(\frac{n}{p} - 1\right) + 2\log p \approx \frac{n}{p} + 2\log p$$

- **Speedup**

$$S_p = \frac{n}{\frac{n}{p} + 2\log p} = \frac{np}{n + 2p\log p}$$

Example - Adding on a Hypercube

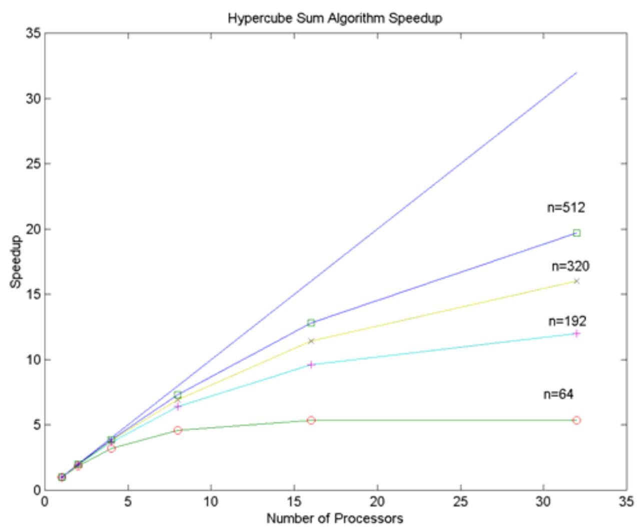
(sum 16 numbers on 4 processors optimally)

- **Efficiency:** $$E_p = \frac{S_p}{p} = \frac{n}{n + 2p\log p}$$
  - For 16 numbers on 4 processors, E_p= 50% (much better!!!)

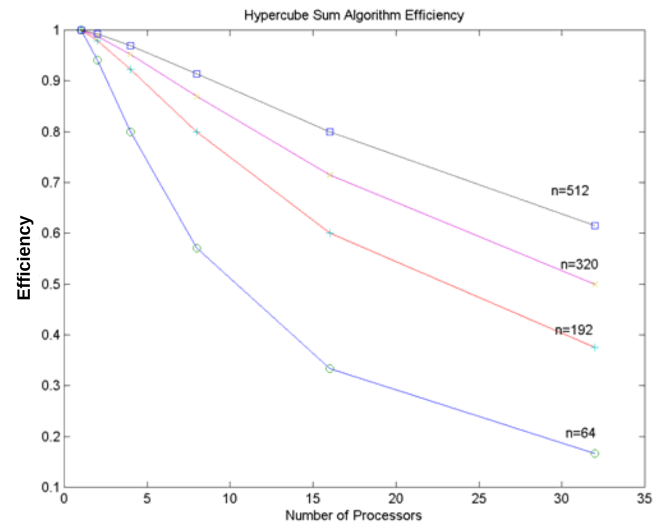
P	Sp	Ep
2	1.6	0.8
4	2.0	0.5
8	2.0	0.25
16	1.78	0.11

- **Note that:**
  - Speedup does not increase linearly with the number of processors
  - Speedup increases with larger vector sizes (for fixed *p*)
  - If we scale the problem at the same time we increase the number of processors, and efficiency remains constant, we have a *scalable system and algorithm*

Example - Adding on a Hypercube

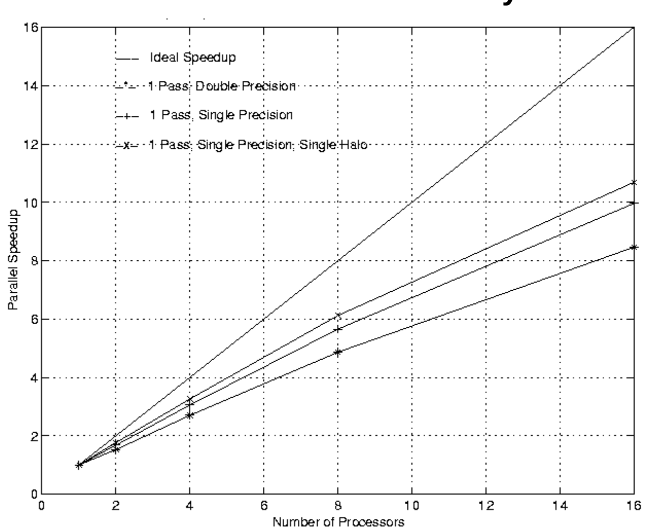


Example - Adding on a Hypercube

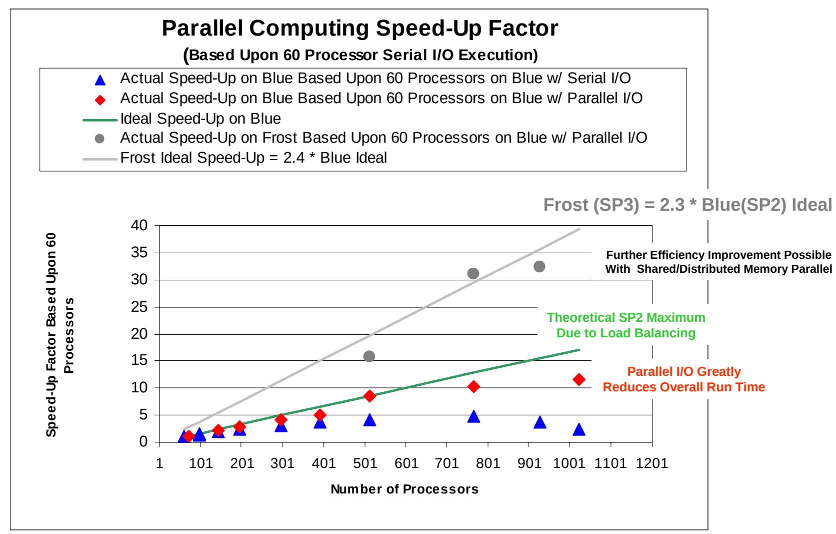


Realistic Test Case

- **A multi-block flow solver. Not very different!!!**



A Scalable Version of Another Multi-Block Flow Solver



Granularity

- Often, the concept of *granularity* is what matters most.
- ***Granularity***: Ratio of amount of *communication* that a processor requires *to* the amount of *computation* that it performs
- ***Coarse grain*** parallelism (low granularity): typical MPI applications. Decrease communication overhead. Most suitable for low performance networks (but also for high performance, of course).

Granularity

- ***Fine grain*** parallelism (high granularity): most typical in OpenMP and GPU applications where small portions of computation (such as operations inside a loop) are divided among multiple processors.
- **Fine grain parallelism requires high performance network/memory subsystems.**

Granularity

- **Note the inherent benefit in scaling up the problem size for a block of typical dimension, *L*:**
  - Amount of communication is proportional to the surface area of the block $\approx L^2$
  - Amount of computation is proportional to the volume of the block $\approx L^3$
- **Therefore, as the problem gets larger, the granularity decreases** $\approx \frac{1}{L}$

Bandwidth

- ***Bandwidth*** **is the *rate of information transfer* that a communication subsystem can maintain (Mb/sec)**
  - Bandwidth that matters is software bandwidth (using MPI, for example)
  - More is better (… and much more expensive)
  - To minimize communication overhead, send only necessary information
  - Typical number: 60 Mb/sec for Origin2000, 85 Mb/sec for Matrx, 775 Mb/sec for Davistron, 440 Mb/sec for Vortex or Wopr

Latency

- ***Latency*** **is defined as the time it takes to send a zero-length message from one processor to another (measured in microseconds, typically)**
  – Less is better (… and more expensive)
  – Software latency (MPI, for example) matters
  – Impacts short messages mostly (and algorithms that rely heavily on short messages, i.e. multigrid)
  – Try to agglomerate messages if possible to decrease communication cost.
  – Typical number: 15 microseconds for SGI Origin2000, 50 microseconds for Matrx, 1.3 microseconds for Davistron, 2 microseconds for Vortex

Homework 7

- **Read Chapter 7 of <u>Introduction to Parallel Computing</u> by Grama et al**
  – Read about shared-memory parallel computing and OpenMP
  – A good tutorial and users' manual on OpenMP is: https://computing.llnl.gov/tutorials/openMP/