

CSE 260 – Introduction to Parallel Computation

Class 5: Parallel Performance

October 4, 2001

Speed vs. Cost

How do you reconcile the fact that you may have to pay a LOT for a small speed gain??

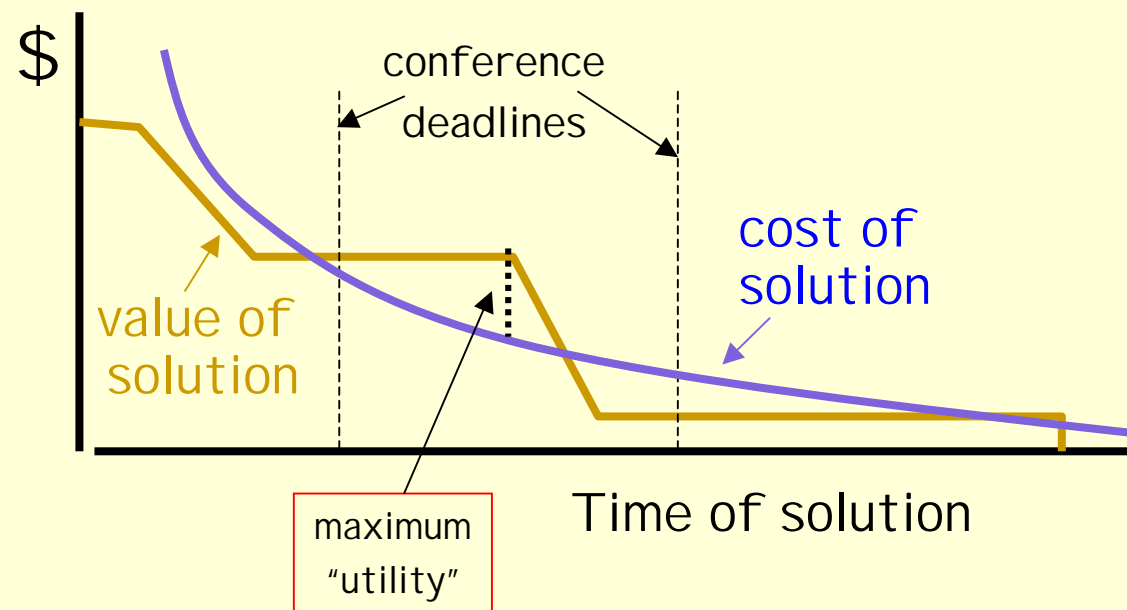
Typically, the fastest computer is much more expensive per FLOP than a slightly slower one.

And if you're in no hurry, you can probably do the computation for free.

Speed vs. Cost

How do you reconcile the fact that you may have to pay a LOT for a small speed gain??

Answer from economics: One needs to consider the "time value of answers" - for each situation, one can imagine two curves:



Speed vs. Cost (2)

But this information is difficult to determine

Users often don't know value of early answers.

So one of two answers are usually considered:

Raw speed – choose the fastest solution
independent of cost

Cost-performance – choose the most economical
independent of speed

Example: Gordon Bell prizes awarded yearly
for fastest and best cost-performance.

Measuring Performance

The following should be self-evident:

- The best way to compare two algorithms is to measure the execution time (or cost) of well-tuned implementations on the same computer.
 - Ideally, the computer you plan to use.
- The best way to compare two computers is to measure the execution time (or cost) of the best algorithm on each one.
 - Ideally, for the problem you want to solve.

How to lie with performance results

The best way to compare two algorithms is to measure their execution time (or cost) on the same computer.

- Measure their FLOP/sec rate.
- See which "scales" better.
- Implement one carelessly.
- Run on different computers, try to "normalize".

The best way to compare two computers is to measure execution time (or cost) of the best algorithm on each

- Only look at theoretical peak speed.
- Use the same algorithm on both computers
 - even though it's not well suited to one.

Speed-Up

- **Speedup:** $S(p) = \frac{\text{Execution time on one CPU}}{\text{Execution on } p \text{ processors}}$

Speedup is a measure of how well a program “scales” as you increase the number of processors.

- We need more information to define speedup:
 - What problem size?
 - Do we use the same problem for all p's?
 - What serial algorithm and program should we use for the numerator?
 - Can we use different algorithms for the numerator and the denominator??

Common Definitions*

Speedup = Fixed size problem

“Scaleup” or scaling = Problem size grows as p increases

Choice of serial algorithm:

1. Serial algorithm is parallel algorithm with “ $p = 1$ ”. (May even include code that initializes message buffers, etc.)
2. Serial time is fastest known serial algorithm running on a one processor of the parallel machine.

Choice 1 is OK for getting a warm fuzzy feeling.

- doesn't mean it's a good algorithm.
- doesn't mean it's worth running job in parallel.

Choice 2 is much better.

* Warning: terms are used loosely. Check meaning by reading (or writing) paper carefully.

What is “good” speedup or scaling?

- Hopefully, $S(p) > 1$
- Linear speedup or scaling:
 - $S(p) = p$
 - Parallel program considered perfectly scalable
- Superlinear speedup
 - $S(p) > p$
 - This actually can happen! Why??

What is maximum possible speedup?

- Let f = fraction of program (algorithm) that is serial and cannot be parallelized. For instance:
 - Loop initialization
 - Reading/writing to a single disk
 - Procedure call overhead
- Amdahl's law gives a limit on speedup in terms of f .

$$T_s = fT_s + (1 - f)T_s$$
$$T_p = fT_s + \frac{(1 - f)T_s}{n}$$
$$S(n) = \frac{T_s}{fT_s + \frac{(1 - f)T_s}{n}} = \frac{n}{nf + 1 - f} = \frac{1}{(n - 1)f + 1}$$
$$\lim_{n \rightarrow \infty} = \frac{1}{f}$$

These "n"s should be "p"s.

Example of Amdahl's Law

- Suppose that a calculation has a 4% serial portion, what is the limit of speedup on 16 processors?

$$16 / (1 + (16 - 1) * .04) = 10$$

- What is the maximum speedup, no matter how many processors you use?

$$1 / 0.04 = 25$$

Variants of Speedup: Efficiency

Parallel Efficiency: $E(p) = S(p)/p \times 100\%$

- Efficiency is the fraction of total potential processing power that is actually used.
 - A program with linear speedup is 100% efficient

Important (but subtle) point!

- Speedup is defined for a fixed problem size.
 - As p get arbitrarily large, speedup must reach a limit ($T_s/T_p < T_s/\text{clock period}$).
 - Doesn't reflect how big computers are used - people run bigger problems on bigger computers.

How should we increase problem size?

- Several choices:
 - Keep amount of “real work per processor” constant.
 - Keep size of problem per processor constant.
 - Keep efficiency constant

Vipin Kumar's “Isoefficiency”

Analyze how problem size must grow.

Speedup vs. Scalability

- Gustafson* questioned Amdahl Law's assumption that the proportion of a program doing serial computations (f) remains the same over all problem sizes.
 - Example: suppose the serial part includes $O(N)$ initialization for an $O(N^3)$ algorithm. Then initialization takes a smaller fraction of time as the problem size grows. So f may become smaller as the problem size grows larger.
 - Conversely, the cost of setting up communication buffers to all $(p-1)$ other processors may increase f (particularly if work/processor is fixed.)

*Gustafson: <http://www.scl.ameslab.gov/Publications/FixedTime/FixedTime.html>

Speedup vs. Scalability

Interesting consequence of increasing problem size:

There is no bound to speedup as $n \rightarrow$ infinity;

Scaled speedup can be superlinear!

Doesn't happen often in practice.

Is efficiency

- Kumar argues, for many algorithms, there is a problem size that keeps efficiency constant (e.g. 50%)
 - Make problem size small enough, it's sure to be inefficient.
 - Good parallel algorithms will get more efficient as problems get larger.
 - Serial fraction decreases
 - Only problem is if there's enough local memory

I soefficiency

- The “isoefficiency” of an algorithm is the function $N(p)$ that tells how much you must increase problem size to keep efficiency constant (as you increase p = number of procs).
 - A small (e.g. $N(p)=p \lg p$) isoefficiency function means it's easy to keep parallel computer working well.
 - Large isoefficiency function (e.g. $N(p) = p^3$) indicate the algorithm doesn't scale up very well.
 - I soefficiency doesn't exist for unscalable methods.

Example of isoefficiency

Tree-based algorithm to add list of N numbers:

- Give each processor N/p number to add
- Arrange processors in tree to communicate results.

Time for problem of size N on p procs.

Time to send one number

Parallel time is $T(N,p) = N/p + c \lg(p)$

Log base 2

Efficiency is $N/(p T(N,p)) = N/(N+cp \lg(p))$
 $= 1/(1+cp \lg(p)/N)$

For isoefficiency, must keep $cp \lg(p)/N$ constant, that is, we need $N(p) = c p \lg(p)$.

Excellent isoefficiency: per-processor problem size increases only with $\lg(p)$.

More problems with scalability

For many problems, it's not obvious how to increase the size of a problem.

Example: sparse matrix multiplication

keep the number of nonzeros per row constant?

keep the fraction of nonzeros per row constant?

keep the fraction of nonzeros per matrix constant?

keep the matrices square??

NAS Parallel Benchmarks made arbitrary decisions (class S, A, B and C problems)

Scalability analysis can be misused!

Example: sparse Matrix Vector product $y = Ax$

Suppose A has k non-zeros per row and column. Processors only need to store non-zeros. Vectors are dense.

- Each processor "owns" a $p^{1/2} \times p^{1/2}$ submatrix of A and $(1/p)$ -th of x and y .
- Each processor must receive relevant portions of x from owners, compute local product, and send resulting portions of y back to owners.
- We'll keep number of non-zeros per processor constant.
 - work/processor is constant
 - storage/processor is constant

A scalable $y = Ax$ algorithm:

Each number is sent via a separate message.

When A is large, each non-zero owned by a processor will be in distinct columns.

Thus, it needs to get M messages (M = memory)

Thus, the communication time is constant, even as P grows arbitrarily large.

Actually, this is a *terrible* algorithm!

Using a separate message per non-zero is very expensive!

There are much more efficient ones (for reasonable size problems) – but they aren't "perfectly scalable".

Cryptic details in Alpern&Carter, "Is Scalability Relevant?", '95 SIAM conf. on Parallel Processing for Scientific Computing.

Little's Law

Transmission time x bits/time unit = bits in flight

Translation by Burton Smith:

Latency (in cycles) x Instructions/cycle = Concurrency

Note: concurrent instructions must be independent

Relevance to parallel computing

Coarse-grained computation needs lots of independent chunks!