Performance of parallel programs

Outline

- Sources of overhead
- Total parallel overhead
- Speedup
- Efficiency
- Scalability
- Communication cost
- Timing

Sources of overhead

Overhead - factors which make the parallel code run slower than expected, when compared to the serial code

- Communication

- Idling, load imbalance, synchronization, presence of serial components

- Excess computation

  The fastest known serial algorithm may be difficult or impossible to parallelize. Hence may have to use a poorer, but easier to parallelize algorithm

  The difference in computation performed by the parallel program and the best serial program is the excess computation overhead incurred by the parallel program

Total parallel overhead

Denote serial running time by T_s and parallel running time on p processors by T_p

(We assume solving problems of the same size)

Total overhead is defined as

$$T_o = pT_p - T_s$$

Ts is the time for the fastest serial algorithm solving the problem on 1 processor

pT_p is the total time spent over all processors solving the problem

Speedup

Speedup is defined as

$$S = \frac{T_s}{T_p}$$

Usually 0 < S < p

If S = p, we have linear speedup

Theoretically, S can never exceed p

In practice, it may happen that S > p : superlinear speedup

Example of superlinear speedup: cache effect

Say you are running a program that requires memory M = 100 MB

The processors available to you all have 8 MB cache. Memory access to cache memory is MUCH faster that access to main RAM memory.

Assume program can be parallelized to run on p processes, and work divided in such a way that only M/p memory is used by each process.

When p is increased sufficiently for M/p to fit completely within the 8 MB cache (i.e. p>12), there will be no cache misses (i.e. slow access to main RAM memory) and a superlinear increase in performance may be observed.

This is highly dependent on the particular program (and its parallel overhead) and the computing hardware one is using.

Efficiency

Efficiency is a measure of processor utilization in a parallel program

That is a measure of the fraction of time for which a processor is employed

$$E = \frac{S}{p}$$

If E =1 , linear speedup

If E < 1/p, slowdown (serial program is faster)

Scalability

Efficiency as p grows, problem size fixed

Efficiency can be written as

$$E = \frac{S}{p} = \frac{T_s}{pT_p} = \frac{1}{1 + \frac{T_o}{T_s}}$$

T_o is an increasing function of p

- every program must contain some serial component, or
  one that cannot be efficiently parallelized
  Example: MPI_Init which initializes MPI

- To give the most common example, if the time for the serial component on process 0 is t_s, p-1 processors will be idle for $(p-1)t_s$

- hence T_o grows at least linearly

- because of communication, idling, and excess computation, it may grow superlinearly

For a given problem size efficiency goes down as p increases

Efficiency as problem size grows, p fixed

T_o depends on problem size and p

In many cases, T_o grows sublinearly with respect to program size

Efficiency increases as p is fixed and problem size is increased

Scalable: the efficiency can be kept constant as the number of processing elements is increased, provided that the problem size is increased

It follows that, when testing parallel programs for efficiency, it is important to select the correct problem size.

For example, it may be acceptable for a parallel program to show poor efficiency for small problem size (for which one would usually use serial code anyway), as long as it displays good efficiency for large problem sizes, for which running in parallel may be the only way to obtain a solution to the problem

Is running a low efficiency parallel code ever acceptable?

Broadly speaking, no. If for some p processes the code has poor efficiency, it is better to run multiple runs with a lower number of processes and benefit from the higher efficiency that usually results. This assumes that to some degree the problem can be broken into multiple independent runs, which is the case in the vast majority of problems.

In fact, it is quite common to find that running many serial process is best. Such an approach is called “serial farming”.

If a problem cannot be broken down into parts in any way and so it must be run on a large number of processes (because it will not fit into memory otherwise, for example), then lower efficiency may be justified.

On SHARCNET running low efficiency code is discouraged, to ensure fairness in distributing resources to users.

For some mission critical software, speedup may trump efficiency. For example, a company that can increase profits if it is able to compute results faster may be willing to buy a cluster with many machines to achieve that speedup, even in cases where efficiency is quite low.

Amdahl’s Law

Model for relationship between expected speedup of a partially parallelized algorithm as compared to serial algorithm

Highly relevant as most parallel programs are obtained by taking the serial code and parallelizing its most time consuming parts

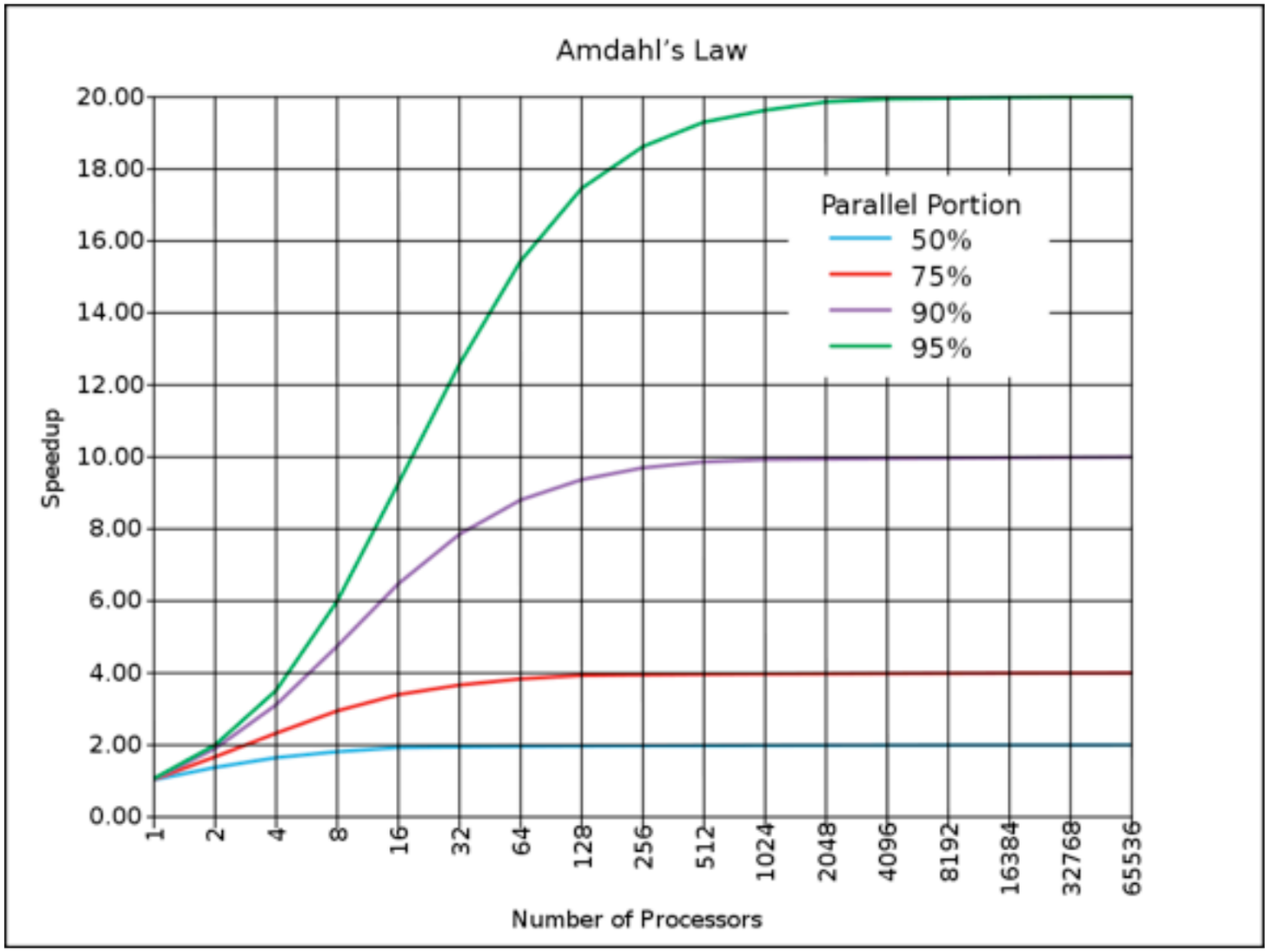
Assume that in a serial code, fraction P of the code (as measured by computational time needed) has been parallelized. Let’s assume that parallelization has been perfect, with no overhead and hence linear speedup (i.e. best case scenario).

Amdahl’s law states that the maximum speedup achievable in that case when running with p processors is

$$S = \frac{1}{(1-P) + \frac{P}{p}}$$

$$S = \frac{T_s}{T_p}$$

$$T_p = (1-P)T_s + \frac{PT_s}{p}$$



As P goes to infinity

$$S = \frac{1}{1-P} = constant$$

One can parallelize more and more of the code to progressively reduce P, but this costs increasing amounts of effort

Law of diminishing returns will eventually set in

Timing

One can use a special MPI function to time parallel code:

```
double MPI_Wtime(void);
```

It returns time elapsed (in seconds) since some point in the past which will not change during the execution of the task. Note that this MPI function does not return an integer error code.

Example use:

```
t1=MPI_Wtime();
... Code we wish to time ...
t2=MPI_Wtime();
time_diff=t2-t1; /* how long it too to execute the code */
```

May not be precise enough for some tasks (eg. for measuring a single communication function call), but may be OK if one averages over many short tasks

Other timing functions in C may work better (eg. gettimeofday)

Output of SHARCNET job scripts provides some useful information about the total runtime of your program