CZ4102 – High Performance Computing
Lecture 10:
Analytical Modeling of Parallel Systems

- A/Prof Tay Seng Chuan

Reference: “Introduction to Parallel Computing” – Chapter 5.

Basics

- A sequential algorithm is evaluated by its runtime as a function of input size.

- In general, the asymptotic runtime (i.e., the trend of runtime when the input size is being increased) of a sequential program is identical on any serial platform.

- The parallel runtime of a program depends on the input size, the number of processors, and the communication parameters of the machine. An parallel algorithm must therefore be analyzed in the context of the underlying platform.

Performance Measure

- A number of performance measures such as the wall clock time or CPU time are intuitive.

- Wall clock time - the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble. This can be measured by system time (not by stop watch). But how does wall clock time scale when the number of processors is changed or the program is ported to another machine altogether?

- How much faster is the parallel version? What is the basis for comparison? Can we use a sub-optimal serial program to make our parallel program look better? These questions have been discussed in tutorial 1.
Sources of Overhead in Parallel Programs

- If I use two processors, shouldn't my program run 2 times faster? I used sweeping this classroom as an example and gave 4 answers: (i) Yes, (ii) Faster but less than 2 times, (iii) Faster than 2 times, (iv) Slower than the time needed by 1 sweeper.

- Overheads in parallel execution include wasted computation, communication, idling, and contention which cause degradation in performance.

![Execution Time Chart]

- Essential/Excess Computation
- Interprocessor Communication
- Idling

Perfomance Metrics for Parallel Systems:

- Serial runtime (denoted by $T_S$ or $T_1$) of a program is the elapsed time between the start and the end of its execution on a sequential computer.

- The parallel runtime with $p$ processing elements (denoted by $T_P$) is the elapsed time from the moment the first processor starts to the moment the last processor finishes its execution.

- Let $T_{all}$ be the total time collectively spent by all the processing elements. We have $T_{all} = p \times T_P$.

- Observe that $T_{all} - T_S$ is the total time spent by all combined processors in non-useful work. This is called the **total overhead**. The overhead function ($T_o$) is therefore given by $T_o = p \times T_P - T_S$.

Sources of Overheads in Parallel Programs

- Interprocess interactions: Processors working on any non-trivial parallel problem will need to communicate to each other. Eg, queuing simulation.

- Idling: Processes may be idle because of load imbalance, synchronization, or serial components. Eg, the building-up phase in parallel quicksort.

- Excess Computation: This refers to the portion of the computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication. Eg, the consolidation of grid contents after row-sort to establish heap structure.

Performance Metrics for Parallel Systems:

- What is the benefit from parallelism? After all, what we want is that the program execution should be completed at a shorter time. So we need to compute the speedup.

- Speedup ($S$) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with $p$ identical processing elements.

$$S = \frac{T_S}{T_P}$$

- We also express speedup as follows:

$$S(p) = \frac{T_S}{T_P}$$

- The secondary objective of using more than 1 PEs is to take advantage of the larger consolidated amount of cache memory to run a big program. Running a big program on a single PE can incur too much cache misses thus causing thrashing effect.
Performance Metrics: Example

- Consider the problem of adding \( n \) numbers by using \( n \) processing elements.

- If \( n \) is a power of two, we can perform this operation in \( \log n \) steps by propagating partial sums down a logical binary tree of processors.

![Diagram of binary tree for summing numbers](image)

In Summation Form

Computing the global sum of 16 partial sums using 16 processing elements. \( \Sigma_i \) denotes the sum of numbers with consecutive labels from \( i \) to \( j \).

![Diagram of binary tree for summing numbers](image)

Performance Metrics: Example (continued)

- If an addition takes constant time say \( t_c \), and communication of a single word takes time \( t_s + t_w \), we have the parallel time \( T_p = \Theta(\log n) \).

- We know that \( T_S = \Theta(n) \), which can be implemented by a for loop as follows: for \( (i=0; i<n; i++) \) sum+= a[i];

- Speedup \( S \) is given by \( S = \frac{T_s}{T_p} = \Theta\left(\frac{n}{\log n}\right) \).

- For a given problem, there might be many serial algorithms available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees. What if \( T_S \) (or \( T_I \)) is not optimal? Have we over represented the speedup?

- For the purpose of computing speedup, we always consider the best sequential program as the baseline.

Performance Metrics: Speedup Example

- Consider the problem of parallel bubble sort. The serial time for bubblesort is 150 seconds.

- The parallel time for odd-even sort is 40 seconds. The speedup would appear to be \( \frac{150}{40} = 3.75 \)! Is this really a fair assessment of the system?

- What if serial quicksort only took 30 seconds? In this case, the speedup is \( \frac{30}{40} = 0.75 \). This is a more realistic assessment of the system.
Performance Metrics: Speedup Bounds

- Speedup can be as low as 0 (the parallel program never terminates). This happens for infinite loop, deadlock and/or livelock situations.
- Speedup, in theory, should be upper bounded by $p$ - after all, we can only expect a $p$-fold speedup if we use $p$ times as many resources.
- A speedup greater than $p$ is possible only if each processing element spends less than time $T_S/p$ solving the problem. This is called super-linear speedup and cannot be guaranteed. Why?

Answer: Otherwise we can construct a new sequential algorithm using time-sliced approach to simulate the parallel algorithm to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup calculation.

Performance Metrics: Super-linear Speedups

One reason for super linearity is that the parallel version does less work than corresponding serial algorithm.

Searching an unstructured tree for a node with a given label, 'S', on two processing elements using depth-first traversal. The two-processor version with processor-0 searching the left subtree and processor-1 searching the right subtree expands only the shaded nodes before the solution is found. The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.

$T_1 = 14c$
$T_2 = 5c$
$S = \frac{14c}{5c} = 2.8$ (super-linear as $p = 2$)

Performance Metrics: Super-linear Speedup Due to Higher Cache Hit Rate

We call it resource-based super-linearity - the higher aggregate cache/memory bandwidth can result in better cache-hit ratios, and therefore super-linearity. Example: A processor with 64KB of cache yields an 80% hit ratio. If cache access time is 2 ns, and DRAM (Dynamic Random Access Memory) access time is 100 ns, The effective memory access time is

$2 \text{ ns} \times 0.8 + 100 \text{ ns} \times 0.2 = 21.6 \text{ ns}.$

If the computation is memory bound and perform 1 FLOP/memory access, the processing rate is $\frac{1}{21.6\text{ns}} = 46.3$ MFLOPS.

Now if two processors are used, since the problem size/processor is smaller, the hit ratio goes up to 90%. Of the remaining 10% access, 8% come from local memory and 2% from remote memory (arbitrary figures for explanation purpose). If the remote memory access time is 400 ns, the effective memory access time is

$2 \text{ ns} \times 0.9 + 100 \text{ ns} \times 0.08 + 400 \text{ ns} \times 0.02 = 17.8 \text{ ns}.$

The processing rate in each processor is $\frac{1}{17.8\text{ns}} = 56.18$ MFLOPS.

The total processing rate for 2 processors is $56.18 \times 2 = 112.36$ MFLOPS, and this corresponds to a speedup of $112.36/46.3 = 2.43$ – superlinear!

Performance Metrics: Efficiency

- Efficiency is a measure of the fraction of time for which a processing element is usefully employed.
- Mathematically, it is given by $E = \frac{S}{p}$.
- Following the bounds on speedup, efficiency in general can be as low as 0 and as high as 1. $E > 1$ can happen but cannot be guaranteed.
- The speedup of adding numbers on processors is given by

$S = \frac{n}{\log n}$

and the efficiency is given by

$E = \Theta\left(\frac{n}{\log n}\right)$
Parallel Time, Speedup, and Efficiency Example

Consider the problem of edge-detection in images. The problem requires us to apply a $3 \times 3$ template to each pixel. If each multiply-add operation takes time $t_c$, the serial time for an $n \times n$ image is given by $T_S = t_c n^2$.

Example of edge detection: (a) an $8 \times 8$ image; (b) typical templates for detecting edges; and (c) partitioning of the image across four processors with shaded regions indicating image data that must be communicated from neighboring processors to processor 1.

Edge Detection

• One possible parallelization partitioning scheme is to divide the image equally into vertical segments, each with $n^2/p$ pixels—which is the number of pixels to be processed by each PE.

• The boundary of each segment is $2n$ pixels ($n$ pixels on the left, and $n$ pixels on the right). This is also the number of pixel values that will have to be communicated. This takes time $2(t_c + t_w n)$.

• Templates (with 9 values) may now be applied to all $n^2/p$ pixels in time $9 t_c n^2/p$.

• The total time for the edge detection algorithm is therefore given by the sum of computation time and communication time: $T_P = 9t_c n^2/p + 2(t_c + t_w n)$.

• The corresponding values of speedup ($S$) and efficiency ($E$) are given by:

$$S = \frac{9t_c n^2}{9t_c n^2/p + 2(t_c + t_w n)}$$

and

$$E = \frac{1}{1 + \frac{2p(t_c + t_w n)}{9t_c n^2}}$$
**Cost of a Parallel System**

- Cost is the product of parallel runtime and the number of processing elements used ($p \times T_p$).
- Cost reflects the sum of the time that each processing element spends solving the problem.
- A parallel system is said to be cost-optimal if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost.
- Since $E = \frac{(T_s)}{p} = \frac{T_s}{p}$, for cost optimal systems $E = \frac{p}{p} = O(1)$.
- Cost ($p \times T_p$) is sometimes referred to as work or processor-time product.

**Cost of a Parallel System: Example**

- Consider the problem of adding numbers on processors.
  
  - We have, $T_p = \log n$ (for $p = n$).
  
  - The cost of this system is given by $p \times T_p = n \log n$.
  
  - Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal. **What is the cause?**

**Impact of Non-Cost Optimality**

Consider a sorting algorithm that uses $n$ processing elements to sort a list in time $(\log n)^2$ — eg, Odd-Even Merge Sort.

- Since the serial runtime of a (comparison-based) sort is $n \log n$, the speedup and efficiency of this algorithm are given by $\frac{n}{\log n}$ and $\frac{1}{\log n}$ respectively.
- The $p \times T_p$ product of this algorithm is $n (\log n)^2$ showing that this algorithm is not cost optimal by a factor of $\log n$.
- If $p < n$, assigning $n$ tasks to $p$ processors gives $T_p = n (\log n)^2 / p$.
- The corresponding speedup ($S_p = T_s / T_p$) of this formulation is $\frac{p}{\log n}$. This speedup goes down as the problem size $n$ is increased for a given $p$! This is the side effect of using the same task allocations with increasing problem size.

**Effect of Granularity on Performance**

- Often, using fewer processors improves performance of parallel systems. The task allocations has been changed with fewer processors. Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called scaling down a parallel system. (In real life this is called downsizing, or trimming the fats.)
- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign (or to cluster) these virtual processors equally to scale down the available processors.
- Since the number of processing elements decreases by a factor of $\frac{n}{p}$, the computation at each processing element increases by a factor of $\frac{n}{p}$.
- The communication cost should not increase by this factor since some of the virtual processors assigned to a physical processors might communicate to each other (intra-processor communication). This is the basic reason for the improvement from building granularity. (A person takes zero time to communication with himself/herself.)
Scalability of Parallel Systems

How do we extrapolate performance from small problems and small systems to larger problems on larger configurations?

Consider three parallel algorithms for computing an \( n \)-point Fast Fourier Transform (FFT) on 64 processing elements. Asymptotic effect is observed for 2D transpose.

Scaling Characteristics of Parallel Programs

- The efficiency of a parallel program can be written as:
  \[
  E = \frac{S}{p} = \frac{T_0}{pT_P}
  \]
  give
  \[
  E = \frac{1}{1 + \frac{fS}{T_0}}
  \]

- The total overhead function \( T_o \) is an increasing function of \( p \), ie, \( T_o = pT_p - T_s \).

- For a given problem size (i.e., the value of \( T_s \) remains constant), as we increase the number of processing elements, \( T_o \) or overhead increases. Consequently, the overall efficiency of the parallel program goes down. This is the case for all parallel programs.

Amdahl's Law

A small number of sequential operations can ultimately limit the speedup of a parallel algorithm.

Let \( f \), \( 0 < f < 1 \), be the fraction of operations in a computation that must be performed sequentially. What is the best speedup if the computation is performed in parallel?

We have

\[
T_p = f \times T_f + \frac{(1-f) \times T_i}{p}
\]

\[
S(p) = f \times T_f + \left(1-\frac{f}{p}\right) \times T_i
\]

When \( p \) is a large number, we have \( S(p) = \frac{1}{f} \). This imposes an upper bound on speedup.

Change in Speedup when the number of PEs is increased

Speedup tends to saturate and efficiency drops as a consequence of Amdahl's law.