

Parallel Algorithm Evaluation

Pangfeng Liu
National Taiwan University

March 5, 2015

Introduction

- Algorithm describes the procedure that solves a problem.
- “Algorithmic thinking” is very important when one wants to use computer to solve problems; in a sense that computers can easily realize an algorithm with its powerful computation capability.
- This is not confined to solving problem with computers, but in this lecture we focus on algorithm for computer.

Computers

- Some algorithms are particularly suitable for computers.
- Consider Sudoku problem. You want to place the number of 1 to 9 into a 9 by 9 matrix so that every row, every column and the nine 3 by 3 sub-matrices have numbers from 1 to 9.
- We can use a “trial-and-error” algorithm to solve this problem. However, this is time consuming and error-prone for human to execute this algorithm.
- In contrast a computer programmer can easily convert the “trial-and-error” algorithm to a computer program that solves this problem in no time.

Definition

- An algorithm must describe the *detailed operations* one wants to perform.
- These operations must be *well-defined* within the underlying model (more on this later).
- The algorithm must also describe the *temporal dependency* of these operations, i.e., loop, synchronization, etc.

Example

- How to pick the largest number from a set of numbers.
- Look through all the numbers one at a time.
- Compare a number with the current largest one you have seen.
- If the number is the larger, replace the current largest one with it.
- Finally you have the largest number.

Example

- The operations, i.e., examine, compare, and replace, are well defined.
- The temporal sequences of these operations are also well defined – “look through all”, “if something happens then does this”.
- This is an algorithm indeed.

Discussion

- Give an example of algorithm.

Parallel Algorithms

- Now we extend the concept to parallel computation.
- A *sequential* algorithm describes the procedure that solves a problem with a computer.
- A *parallel* algorithm describes the procedure that solves a problem with a parallel computer.
- In this course we will focus on parallel algorithms.

Parallel Algorithms

- A sequential algorithm is relatively easy to describe.
 - The timing sequence is for one processor only.
- A parallel algorithm is *harder* to describe since we have to deal with multiple entities working concurrently.
 - The timing constraints are about multiple processors, hence much difficult to describe and analyze.
 - Different processors may need to access the same data, and may have race condition.

Model

- Any algorithm has a underlining assumption on what can be done, for example, a sorting algorithm assumes that keys can be compared in $O(1)$, i.e., a constant amount of time.
- We then follow these assumptions to estimate the **cost** of the algorithm we are considering.
- **The purpose of the model** is to estimate the cost accurately, so it has to be realistic, which means it must **resemble** the real hardware to be meaningful.

Analysis

- The process of estimating the cost of an algorithm is *algorithm complexity analysis*.
- Note that we are not actually estimate the running time of the algorithm since this is a moving target.
- Instead we measure *the number of times* certain operations (e.g., computation or communication), and use them as the estimate on the cost of the algorithm.
- We want to design algorithms will low costs.

Analysis

- A sequential algorithm is usually easy to analyze.
- A parallel algorithm is much more difficult to analysis since different models, i.e., the assumption on how the parallel computer can do, have different algorithmic characteristics.

Analysis

- A shared memory model algorithm on a multiprocessor can be very different from a distributed memory algorithm on a multicomputer.
- Nevertheless there are certain criteria that we may follow, mostly from what the actual CPU can do in a fixed amount of time.
- In this lecture I will try my best to **focus on the parallel computing issues**, instead of the computation models, i.e., I want to have **generic analysis**, instead of model specific analysis.

Discussion

- Give an example of algorithm analysis, like doing an barrier synchronization on a distributed memory parallel computer..

Speedup

- How to determine which parallel algorithm is good, and which is bad?
- We use *speedup* as a metric to evaluate parallel algorithms, which is the ratio between the best sequential time T_s and the parallel time T_p .

$$k = \frac{T_s}{T_p} \quad (1)$$

- Note that we need to use the *best* T_s for a meaningful comparison.

Speedup Improvement

- Remember that performance is paramount for parallel processing (remember the racecar?), so speedup is essential.
- There are two ways to improve speedup.
 - The *right* way is to reduce the parallel time.
 - The *wrong* way is to increase the sequential time. That is why we need to use the best sequential algorithm.

Lesson

When you hear people talking about speedup, always make sure that you know the definition of their “speedup”.


Banana and Orange

- In my opinion, we should always calculate the speedup with *the same basis*.
- What do we get if we compare the sequential time of a sequential program running on a CPU, with the parallel time of the same program running on five GPU's, and get the speedup of 136?

Banana and Orange

- The comparison between one CPU and five GPU's is not *quantitatively* meaningful because we cannot derive any *quantitative* conclusion on how well we are doing.
- We may have a terrible implementation and still get good speedup because the CPU is running slowly, or the GPU's are running fast like hell.
- You are comparing banana and orange.

Relative Speedup

- A speedup comparison is quantitatively meaningful if we can relate the speedup with the extra amount of resources we use in the parallel computation.
- Another speedup metric is to compare the parallel time of using k processor with the *parallel time* of using a single a  single processor. This is usually referred to as *relative speedup*.
- As the definition points out, a *relative speedup* is how well we parallelize a computation.

Discussion

- Describe the concept of speedup by examples.

Overheads

- Note that the execution time of a sequential algorithm may be smaller than a parallel program using a single processor.
- There are inherent overheads in the execution of parallel program, even if you are using only one processor.

Overheads

- A parallel system may need to start up.
- The parallel program may use extra parallel library, which may incur extra overhead.
- There may be synchronization overhead, even if only one processor is involved.
- To keep the following theoretic discussion (on efficiency and work) simple we will assume that these two are the same, but keep in mind that there are always overheads in parallel programming.

Parallelism

- The speedup k alone is not sufficient to evaluate the algorithm, since we do not know how many *processors* are used.
- Let p be the number of processors used in the parallel algorithm.
- The speedup k is between 0 and p .

$$0 < k \leq p \quad (2)$$

- If the speedup is close to p then we have a *linear speedup*.

Theoretical Bounds

We can argue mathematically that speedup k is between 1 and p .

$$1 \leq k \leq p$$



(3)

Proof

- We can use one processor to simulate the sequential algorithm, hence $1 \leq k$.
 - Recall that we assume that we do not have overheads due to parallelization.

Proof



- We can simulate one step of the parallel algorithm with p steps on one processor.
 - If that takes less than k steps, we have an algorithm that runs faster than our optimal sequential algorithm, which is a contradiction.
 - Note that we need to assume that the sequential algorithm is optimal.

In Practice

- In rare occasion we may have $T_s < T_p$. This is usually caused by a very small workload and a tremendous overheads in parallel execution.
- Recall that we have all the overheads due parallelization. If the benefits of parallelization is not enough to compensate the overheads, we do not have any speedup.
- Some problems are better left alone (e.g., inherently sequential problems).

In Practice

- Also in rare occasion we may even have $k > p$, which is *super-linear speedup*.
- This is usually due to the size of the working set of the program.

Caching Effects

- If a problem has a very large working set, then it is impossible to fit it into the cache of a single computer. As a result frequent cache misses degrades performance significantly.
- If we divide the problem into many small sub-problems. and then run them in parallel, then it is likely that each working set of these small sub-problems will fit into the cache of a processor, hence having amazing speedup.



Lesson

Always question theory in the context
of “real world”.



Discussion

- Describe the relation of speedup and the number of processors.

Efficiency



Another metric to evaluate parallel algorithms is *efficiency*, which is defined as the speedup divided by the number of processors.

$$e = \frac{k}{p} = \frac{T_s}{pT_p} \quad (4)$$

Efficiency

- It is easy to see that efficiency e is between 0 and 100%, if the speed up is between 0 and p , where p is the number of processors.
- If we fully parallelize a computation, the efficiency is 100%.
- One can think of the efficiency as “how well we parallelize the computation per processor?” – that is, it is the CP value, or performance/cost ratio.

Discussion

- Describe the relation of efficiency and speedup.

Work

Another metric for parallel algorithm evaluation is *work*, which is the product of the number of the processor and the parallel time.

$$w_p = pT_p \quad (5)$$



Energy

- The concept of *work* is like **man-month**; you used this many processors for this period of time.
- It is like that the energy is the product of power and time; you used this much power for a period of time, then you use this much energy.

Extra Work

- The work done by a sequential algorithm is $w_s = T_s$, and the work done by a parallel algorithm is $w_p = pT_p$.
- We assume that all the work done by the sequential algorithm is *necessary*; that means all the work done by the sequential algorithm is *essential*.
- Now if the work done by the parallel algorithm is larger than the work done by the sequential algorithm, then the difference is *non-essential* in solving this problem.
- This is what we called *overheads*.



Extra Work


- If we parallel a computation well, the two works done by the sequential algorithm and the parallel algorithm should be similar, which implies three things.
 - The Efficiency e should be close to 1.
 - The speedup k should be close to p .
 - w_s should be close to w_p .



$$w_p = pT_p = p \frac{T_s}{k} = \frac{w_s}{e} \quad (6)$$

Discussion

Compute the speedup, efficiency, work for the following, and describe the circumstance each of these action is suitable for

- Using one processor and the time is 40 minutes.
- Using two processors and the time is 24 minutes. 
- Using four processors and the time is 16 minutes.

Amdahl's Law

- The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.



Amdahl's law

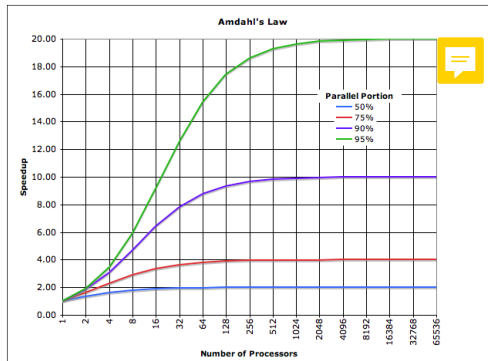
- Let the portion of inherent sequential in a computation be x , the part that can be parallelized will be $1 - x$.
- Assume that the sequential time is 1, then the parallel time will be at least $x + \frac{1-x}{p}$ while using p processors.

$$k = \frac{1}{x + \frac{1-x}{p}} \leq \frac{1}{x} \quad (7)$$

Amdahl's Law

- Amdahl's law says that if you have 20% of you code is inherently sequential, the speedup could not be more than 5.
- That hurts!

Amdahl's Law



1

¹[http:](http://upload.wikimedia.org/wikipedia/commons/6/6b/AmdahlsLaw.png)[//upload.wikimedia.org/wikipedia/commons/6/6b/AmdahlsLaw.png](http://upload.wikimedia.org/wikipedia/commons/6/6b/AmdahlsLaw.png)

Implications

- Every program has an *inherently* sequential part, which limits the speedup.
- If the inherent part is large, then the program is inherently sequential and we do not want to parallelize it; we have to admit that there are computations that cannot be parallelized efficiently.
- The lesson here is to recognize the cases that have only limited parallelism.

Discussion

- Describe Amdahl's Law.

Two Ways

- Remember there are two ways to improve speedup. We either increase the sequential time, or we decrease the parallel time.
- In many cases we can increase the sequential time by increasing the *problem size*. If the impact of the increased problem size on sequential time is larger than on the parallel time, then we have a “better” speedup.

Gustafson's Law

- Computations involving *arbitrarily large data* sets can be efficiently parallelized.
- This is true only when the “inherently sequential” part is a *constant*, i.e., it is not a function of the problem size.

Gustafson's Law

- Let the portion of inherent sequential in a computation be x , the part that can be parallelized will be $1 - x$.
- Instead of a fixed problem size, we increase the number of processors to p . Then after x of inherent sequential work, p processors does $p(1 - x)$ amount of work.



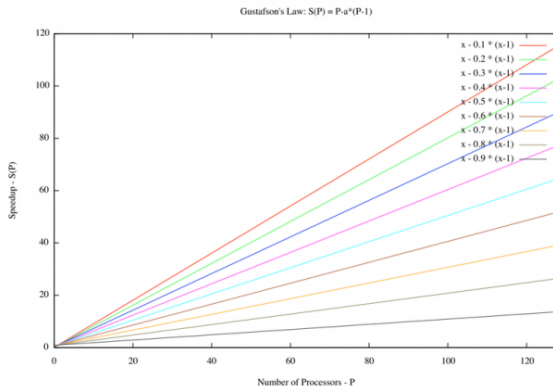
Gustafson's Law

- The sequential time is now at least $x + p(1 - x)$, since the amount of work done by p processors is $p(1 - x)$, and the sequential program only has one processor to do it.
- The parallel time is $x + (1 - x) = 1$ with p processors.
- We can have speedup close to p for any x by increasing p .

$$k = p(1 - x) = p - px \quad (8)$$



Gustafson's Law



2

²<http://slashdot.org/topic/wp-content/uploads/2012/08/>

Screen-Shot-2012-08-29-at-12.19.53-PM-618x425.png

Notes

- We assume that x portion of sequential part is still sufficient even if we increase the work p times.
- In many cases this is true if x only involves setting up the systems, and data can be read in parallel.
- If data cannot be read in parallel, then x will be a function of the problem size, which invalidates the analysis.

Notes

- Despite the possible complications, this observation is generally useful because x is usually a small constant if only system setup is involved. In addition, most computation has complexity like $O(n^2)$ so the work of reading the data, even if done sequentially, is not comparable to the actual computation.

Notes

- If we really really want to have a clear picture about speedup, we would also like to know **the problem size** when the speedup is measured.
- If we have a *good* speedup even if problem size is small, then we really have a good parallel implementation. Otherwise it could be a fabricated phenomenon with the use (or abuse) of Gustafson's Law.
- When you examine a speedup report, always question the **comparison basis and the problem size.**

Discussion

- Describe Gustafson's Law.