

PARALLEL COMPUTING

Lecture Notes

Azali Bin Saudi
Software Engineering Program
Universiti Malaysia Sabah
azali@ums.edu.my
www.azalisaudi.com

April 2008

1. INTRODUCTION

1.1 Background

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed which produces a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a given time - after that instruction is finished, the next is executed.



Figure 1.1: The Cray-2 was the world's fastest computer from 1985 to 1989.

Parallel computing on the other hand uses multiple processing elements simultaneously to solve a problem. The problem is broken into parts which are independent so that each processing element can execute its part of the algorithm simultaneously with others. The processing elements can be diverse and include resources such as a single computer with multiple processors, a number of networked computers, specialized hardware or any combination of the above.

Frequency scaling was the dominant reason for computer performance increases from the mid-1980s until 2004. The total runtime of a program is proportional to the total number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all computation-bounded programs.

However, power consumption in a chip is given by the equation $P = C \times V^2 \times F$, where P is power, C is the capacitance being switched per clock cycle (proportional to the number of transistors whose inputs change), V is voltage, and F is the processor frequency (cycles per second). Increases in frequency thus increase the amount of

power used in a processor. Increasing processor power consumption led ultimately to Intel's May 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.

Moore's Law is the empirical observation that transistor density in a microprocessor doubles every 18 to 24 months. Despite power issues, and repeated predictions of its end, Moore's law is still in effect. With the end of frequency scaling, these additional transistors (which are no longer used to facilitate frequency scaling) can be used to add extra hardware to facilitate parallel computing.

From Wikipedia:

Parallel computing is a form of computing in which many instructions are carried out simultaneously. Parallel computing operates on the principle that large problems can almost always be divided into smaller ones, which may be carried out concurrently ("in parallel"). Parallel computing exists in several different forms: bit-level parallelism, instruction level parallelism, data parallelism, and task parallelism. It has been used for many years, mainly in high performance computing, but interest in it has become greater in recent years due to physical constraints preventing frequency scaling. Parallel computing has recently become the dominant paradigm in computer architecture, mainly in the form of multicore processors.

Parallel computer programs are harder to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks is typically one of the greatest barriers to getting good parallel program performance. In recent years, power consumption in parallel computers has also become a great concern. The speed up of a program as a result of parallelization is given by Amdahl's law.

1.2 Amdahl's law and Gustafson's law

The performance of an algorithm on a parallel computing platform depends on parallelizing the algorithm to achieve performance so it is important to be aware of Amdahl's law, originally formulated by Gene Amdahl in the 1960's. It states that a small portion of the program which can't be parallelized will limit the overall speedup available from parallelization. Any large math or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. This relationship is given by Amdahl's law:

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

where S is the speedup of the program (as a factor of its original sequential runtime), and P is the fraction that is parallelizable. If the sequential portion of a program is 10% of the runtime, we can get no more than a 10x speedup, regardless of how many processors are added. This puts an upper bound on the usefulness of adding more parallel execution units.

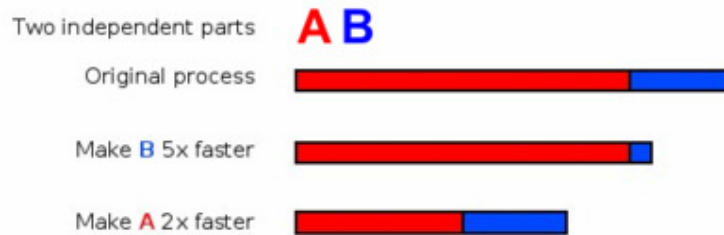


Figure 1.2: A graphical representation of Amdahl's law. Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. With effort, a programmer may be able to make this part 5 times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A be twice as fast. This will make the computation much faster than by optimizing part B, even though B got a greater speed-up, (5x versus 2x).

Gustafson's law is another law in computer engineering, closely related to Amdahl's law. Gustafson's law can be formulated as:

$$S(P) = P - \alpha(P - 1)$$

where P is the number of processors, S is the speedup, and α the non-parallelizable part of the process. Amdahl's law assumes a fixed-problem size and that the size of the sequential section is independent of the number of processors, whereas Gustafson's law does not make these assumptions.

1.3 Dependencies

Understanding data dependencies is one of the foundations of knowing how to implement parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since the fact that the calculations are dependent introduces an ordering of executions. Fortunately, most algorithms do not consist of a long chain of dependent calculations and little else, but instead there are many opportunities for executing independent calculations in parallel.

Let P_i and P_j be two program fragments. Bernstein's conditions describe when the two are independent and can be executed in parallel. Let I_i be all of the input variables to P_i and O_i the output variables, and likewise for P_j . P_i and P_j are independent if they satisfy

- $I_j \cap O_i = \emptyset$
- $I_i \cap O_j = \emptyset$
- $O_i \cap O_j = \emptyset$

Violation of the first condition introduces a flow dependency, corresponding to first statement producing a result used by the second statement. The second condition represents an anti-dependency, when the first statement overwrites a variable needed by the second expression. The third, and final condition q is an output dependency. When two variables write to the same location, the final output must be the one of the second statement.

For example, consider the following function:

```
1: function Dep(a, b)
2:   c := a·b
3:   d := 2·c
4: end function
```

Operation 3 in Dep(a,b) cannot be executed before (or even in parallel) operation 2, due to the fact that operation 3 uses a result from operation 2. It violates condition 1, and thus introduces a flow dependency.

```
1: function NoDep(a, b)
2:   c := a·b
3:   d := 2·b
4:   e := a+b
5: end function
```

In this example there are no dependencies between the instructions, so they can all be run in parallel.

Bernstein's conditions do not allow memory to be shared between different processes. For that, some means of enforcing an ordering between accesses, such as semaphores, barriers or some other synchronization method is needed.

1.4 Race conditions, mutual exclusion, synchronization, and parallel slowdown

Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. However, "threads" is generally accepted as a generic term for subtasks.

Threads will often need to update some variable that is shared between them. The instructions between the two programs may be interleaved in any order. For example, consider the following program:

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A: Write back to variable V	3B: Write back to variable V

If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition. The programmer must use a lock to provide mutual exclusion. A lock is a programming language construct that allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked. The thread holding the lock is free to execute its critical section (the section of a program that requires exclusive access to some variable), and unlock the data when it is finished.

Therefore, in order to guarantee correct program execution, the above program can be rewritten to use locks:

Thread A	Thread B
1A: Lock variable V	1B: Lock variable V
2A: Read variable V	2B: Read variable V
3A: Add 1 to variable V	3B: Add 1 to variable V
4A: Write back to variable V	4B: Write back to variable V
5A: Unlock variable V	5B: Unlock variable V

One thread will successfully lock variable V, while the other thread will be locked out - unable to proceed until V is unlocked again. This guarantees correct execution of the program. Locks, while necessary to ensure correct program execution, can greatly slow down a program.

Locking multiple variables using non-atomic locks introduces the possibility of program deadlock. An atomic lock is a lock that locks multiple variables all-at-once. If it cannot lock all of them, it does not lock any of them. If two threads each need to lock the same two variables using non-atomic locks, it is possible that one thread will lock one of them and the second thread will lock the second variable. In such a case, neither thread can complete, and deadlock results.

Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier. Barriers are typically implemented using a software lock. One class of algorithms, known as lock-free and wait-free algorithms, avoid altogether the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures.

Not all parallelization results in speed-up. Generally, as a task is split up into more and more threads, those threads spend an ever-increasing portion of their time communicating with each other. Eventually, the overhead from communication dominates the time spent solving the problem, and further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This is known as parallel slowdown.

1.5 Fine-grained, coarse-grained, and embarrassing parallelism

Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

1.6 Consistency models

Parallel programming languages and parallel computers must have a consistency model (also known as a memory model). The consistency model defines rules for how operations on computer memory occur and how results are produced.

One of the first consistency models was Leslie Lamport's sequential consistency model. Sequential consistency is the property of a parallel program that its parallel execution produces the same results as a sequential program. Specifically, a program is sequentially consistent if "...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Software transactional memory is a common type of consistency model. Software transactional memory borrows from database theory the concept of atomic transactions and applies them to memory accesses.

Mathematically, these models can be represented in a number of ways. Process calculus is the branch of mathematics dealing with concurrency. Process calculus can be subdivided into ambient calculus, calculus of communicating systems, and communicating sequential processes. Petri nets were an early attempt to codify the rules of consistency models. Dataflow theory later built upon these. Dataflow architectures were later created that physically implement the ideas of dataflow.

1.7 Flynn's taxonomy

Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy. Flynn classified programs and computers by whether they were operating using a single or multiple sets of instructions, whether or not those instructions were using a single or multiple sets of data.

The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set. This is commonly done in signal processing applications. Multiple-instruction-single data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Figure 1.1: Flynn's taxonomy

According to David A. Patterson and John L. Hennessy, "Some machines are hybrids of these categories, of course, but this classic model has survived because it is simple, easy to understand, and gives a good first approximation. It is also—perhaps because of its understandability—the most widely used scheme."