AN INTRODUCTION TO PARALLEL COMPUTING

CHAPTER 1 INTRODUCTION

BY FRED ANNEXSTEIN, PHD

1. INTRODUCTION TO PARALLEL COMPUTING

CHAPTER 1 OBJECTIVES

The goal of this chapter is for students to understand the modern context for parallel computing. We hope students gain an understand for a variety of abstract models for the design of parallel algorithms, and understand how to design parallel programs through a process of task decomposition, agglomeration, and mapping to processors. We also consider a variety of performance measures and metrics, which we can use to compare our designs. Students will be exposed to some basic laws concerning the scalability of parallel computing, and exposed to several structured software design patterns to provide context to use cases for computing with parallelism.

BACKGROUND

Today, multiple cores and general-purpose graphics processing units (GPUs) are common computing devices found on laptops and handheld devices, and many productivity tools depend on distributed services.

Supercomputers containing heterogeneous collections of powerful processors, large multicore arrays and GPUs, all serve science and engineering research by piecing together very large clusters of commodity hardware.

Modern microprocessors have become incredibly complicated, as they make use of the ever increasing number and density of transistors in order to improve the performance of execution of a single instruction stream, using techniques such as pipelining and superscalar execution. They have also devoted an increasing fraction of the chip to high speed cache memory, in an attempt to compensate for slower access to the main memory.

There exists a scalability issue in the model that many modern kernels use. Cache coherence overhead restricts the ability to scale to many-cores. Because of these technical issues, multicore processors have only a limited number of cores. High-end systems today have in the range of 8-24 cores.

Other architectural designs use higher density of processors, so-called manycore processors.

These designs enable many more cores (1000x) to be placed on a single chip, and therefore provide more parallel execution streams. These two general types of architectures multicore vs manycore have also been compared as latency-oriented versus throughput-oriented designs.

Not surprisingly, computer systems today typically include both these designs: notebooks, consoles, and tablets all increasingly have both CPUs and compute-capable GPUs. The open question in the computing industry today is not whether a single application will be spread across different kinds of cores, but how different the cores can and should be. The transition of industry to heterogeneous computing platforms now seems permanent, since different kinds of computations naturally run faster and/or use less power on different kinds of cores.

Parallel architectures continue to evolve in both speeds and complexity at a very rapid pace, and an abstract representation of hardware and the mapping of software to hardware components is an important skill for computing students to master.

PARALLEL PROGRAMMING MODELS

Parallel programming models exist as an aid to developers designing software targeted to parallel machines. These models attempt to provide an abstraction of hardware and memory architectures useful to software developers. Generally, these models are not specific and do not refer to any particular types of machines or memory architectures. These programming models present to the developer a high-level view of hardware and guide the way software must be implemented to efficiently perform parallel computation. An important aspect of each model is the way data and information is shared and communicated with other processors.

The most widely used models for parallel programming are characterized as follows: Shared memory model, Message passing or distributed memory model, Multithread control model, and Data-parallel control model.

THE SHARED MEMORY MODEL

In the shared memory model multiple tasks share a single memory area in which we can read and write memory independently and asynchronously. As part of the model there are mechanisms that allow the software designer to control the access to shared variables. For example, the model may include locks or semaphores to specify access control. Shared memory models are generally considered easier for the software design in so far that it offers the advantage of not having to clarify the details of communications between tasks, since shared variables are available. An important disadvantage of the model is that it becomes more difficult to manage data locality, that is managing caches in order to keep data local to the processor that is reading and writing it.

THE MESSAGE PASSING MODEL

The message passing model is usually applied in cases where each processor has its own memory (referred to as a distributed memory system). Multiple tasks reside on either the same physical processor or distributed on an arbitrary number of interconnected processors. Software developers using this model are responsible for determining the amount of parallelism, the number of processors and the data exchange that occurs through the explicit sending and receiving messages.

Today there is a de facto standard specification for users of the Message passing model called the Message Passing Interface or simply MPI.

THE MULTITHREAD CONTROL MODEL

In the multithreaded control model, a process can have multiple flows of execution. Each flow is often referred to as a thread. For example, a single parent thread may specify a sequential execution, and subsequently, a series of child tasks are created or spawned from the parent that each can be executed independently in parallel.

Multi-threaded programming can be a complex and error-prone practice. Threads, typically, live within a processes, and consist of a program counter, a stack, and a set of registers as well as an identifier. Threads are generally the smallest unit of execution to which a processor can allocate time. Threads are generally able to interact with shared resources, and so this type of model is used on shared memory architectures.

When threads operate on shared memory, programmers must prevent multiple threads from updating the shared memory locations in a way that affects the correctness of the computation (i.e., a race condition).

The current-generation of CPUs are multithreaded in software and hardware. POSIX (short for Portable Operating System Interface) threads are classic examples of the implementation of multithreading using software. Intel's Hyper-Threading technology implements multithreading on hardware by switching between two threads when one is stalled or waiting on I/O.

THE DATA-PARALLEL MODEL

In the data-parallel model there are tasks that operate on the same data structure, usually arrays. Each task operates on a different portion of the data array, but essentially does the same operation on each element, which greatly simplifies the control needed to execute.

The simplified control used in the data parallel model helps programmers avoid the race conditions often arising in multi-threaded control model.

Today, graphics processors known as GPUs can provide a great deal of parallelism to data-parallel codes. They are able to do this with large numbers of symmetric multiprocessors that have limit shared memory.

DESIGNING PARALLEL PROGRAMS

The initial design of parallel programs is often based on a series of high level operations, which must be carried out for the program to perform the job correctly without erroneous results. These high level operations that must be carried out for parallelization of an algorithm are as follows: Task decomposition, Task assignment, Agglomeration, and Mapping.

TASK DECOMPOSITION

In the task decomposition phase the software program is divided into discrete tasks, that is a set of instructions that can then be executed on different processors to implement parallelism. To perform this division, two decomposition methods are often applied.

In *domain decomposition* the data of the problems is decomposed into separate (possibly overlapping) units. This methodology is often used when we have a large amounts of independent data that must be processed.

In *functional decomposition* the problem task is split into set of subtasks each performing a particular operation on the available data.

TASK AGGLOMERATION

Agglomeration of tasks is the process of combining smaller tasks with larger ones in order to improve performance though increased granularity. Granularity is measured by the ratio of the amount computation to the amount or cost of communication. Agglomerating related tasks can often significantly lower the potential costs of communication. Generally speaking, communication costs are proportional to the amount of data transferred plus some fixed overhead cost for every communication operation. If there are too many small tasks, then this fixed cost can easily make the design inefficient.

TASK ASSIGNMENT AND MAPPING

Task assignment requires a mechanism to specify how subtasks are distributed among the various processors or processes. This phase establishes the distribution of workload among the various processors, and it determines the level of load balancing across the application. Poor load balancing can result in processors existing in an idle state for a long time.

In addition to load balancing, the task assignment often will account for the heterogeneity of the system, for example assigning more tasks to better-performing processors. Finally, effective task assignment accounting for locality will help limit the costs of communication between processors.

Tasks mapping has some similar goals to agglomeration in that tasks that communicate frequently should be placed in the same processor to increase locality. However, tasks that can be executed concurrently should be placed in different processors to increase the potential for parallelism.

Optimal mapping is a very difficult computational problem, and is known to be NP-complete. For tasks of equal size and tasks with easily identified communication patterns near optimal mappings are achievable. However, if the tasks have communication patterns that are hard to predict or the amount of work varies per task, then it can be hard to design an efficient mapping and agglomeration scheme.

Task mapping may also be dynamic in nature. For example a basic dynamic mapping scheme may have worker tasks that connect to a centralized manager. The manager can repeatedly send work-tasks to the set of workers and collect the results. Another example is where groups of two or more workers coordinate to share the load of task processing.

PERFORMANCE METRICS

Designers and developers need performance metrics in order to decide whether designs will be successful. To facilitate performance metric analysis we will compare the parallel algorithm obtained to the original, sequential algorithm from where it is derived. The performance metrics quantify the number of threads and/or the number of processes used. We now introduce a few performance metric indexes: Work and Execution Time, Flops, Speedup, Efficiency, and Scaling.

WORK AND EXECUTION TIME

Consider a single program that has some amount of computation to perform, measured as **work**. The simplest metric of performance when executing a non-interactive program on a computer is to assume that the computer delivers constant *compute speed*, which is measured by the quantity of work performed per time unit. For instance, a program with 100 units of work would run in 50 seconds on a core with a speed of 2 units of work per second. This 50 second elapsed time is called the program's **execution time**.

Generalizing the above example, for a given amount of work to perform there is a linear relationship between the program's execution time and the speed of the core on which it is executed:

execution_time = work / compute_speed

There are many options for choosing an appropriate way to quantify work. One possibility is to use a measure that is specific to what the program does. For instance, if the program renders movie frames, a good measure of work would be the number of frames to render. One would then want to measure a core's speed in terms of the number of frames that can be rendered per second (assuming all frames require the roughly the same amount of computation).

Another possibility is to use a more generic measure, for instance, the number of basic instructions. The work of a program would then be measured by its number of basic instructions (e.g., the number of hardware instructions the program performs) and the speed of a core would be in number of instructions per second. This approach is known to have problems, as basic instructions are not all equal, especially across different families of processors. Therefore, a processor that delivers fewer instructions per seconds than another could actually be preferred for having a faster execution time for some programs.

FLOPS AND MIPS

Defining a universal unit of work is not possible. One popular option is the measure of million of instructions per second (MIPS) measure of compute speed. Another popular metric is focused on the number of floating-point operations, or FLOP. This is the metric that we will primarily use in this book. We measure the speed of a core in number of Flop per second, which is commonly used in the field of high-performance scientific computing.

Like any single measure of work, the Flop speed is imperfect as programs also do non-floating-point computations and floating-point operations are not all the same. Fortunately, all the concepts we learn are agnostic to the way in which we measure work. And so we use Flop counts to be a way to be consistent throughout.

Example 1: Say a program performs work of 100 Tflop (“one hundred TeraFlop”) and is executed on a core with a rated speed 35 Gflop/sec (“35GigaFlop per second”). The program’s execution time would then be:

execution_time = work / speed =
$100\times10^{12}\text{Flop} / 35\times10^{9}\text{Flop/sec}$
$\approx 2{,}857.14$ sec

Example 2: Say a program performs work of 12 Gflop and runs in 5 seconds on a core, then the speed of this core in Mflop/sec (“MegaFlop per second”) is:

$\text{speed} = 12\times10^{9}\text{Flop} / 5\text{sec} \times 1/10^{6}$
$= 2{,}400\text{Mflop/sec}$

SPEEDUP

Parallel speedup is simply a ratio of sequential speed to parallel speed that is used to measure the benefit of decreased execution time of solving a problem in parallel. The time taken to solve a problem sequentially on a single processor is Ts. And the time required to solve the same problem on p identical processors is Tp. We denote speedup as follows:

Speedup = Ts / Tp

If *Speedup=p*, then it means that the speed of execution increases with the number of processors, and we call it *linear speedup.* This can occur in an ideal case of *embarrassingly parallel* tasks without communication. A worthy goal is to design parallel programs that have speedup that is proportional to p.

EFFICIENCY

It is usually not reasonable to expect linear speedups, since time is often wasted in either idling or communicating. Efficiency, denoted by ***Eff***, is a measure of how much of the execution time a processing element puts toward doing useful work, given as a fraction of the time spent. Efficiencies are always measured by ***Eff***

<=1, and algorithms with linear speedup have an efficiency value of ***Eff = 1***. In general we have,

Eff = Speedup / p = Ts / p*Tp

SCALABILITY

Scalability is a metric defined as the ability to remain efficient on a parallel machine as p increases. By increasing the size of the problem and, at the same time, the number of processors to which we assign tasks, a scalable system will present no loss in terms of efficiency as p grows. A scalable system is one that must maintain the same efficiency or potentially improve it as p grows.

TIME SHARING

If you execute multiple programs at once on your computer (e.g., your Web browser and a text editor), then this is called multi-programming. It is something that operating systems have supported since the 1960's. Considering **a single core**, the operating system allows a program to run for a while, then another program, and so on until we cycle back and repeat. This is called **time sharing**.

At a high-level, when running *n* programs at the same time on one core, each of them proceeds at *1/n*-th of the core's compute speed. This is not true in practice, as time sharing has some overhead. Also, programs compete for some hardware resources, such as caches, and thus can slow each other down.

With our ideal model, say that at time 0 two programs are started on a single core with speed 1Gflop/sec. If both programs have work 5 Gflop, then they both complete at time 10 sec. Say now that the second program has work of 10 Gflop. Then the first program will complete at time 10, and the second program at time 15. During the first 10 seconds, both programs proceed at speed 0.5 Gflop/sec. At time 10 the first program thus completes (because it has performed all its work), at which point the second program proceeds at full 1 Gflop/sec speed. Since the second program still has 5 units of work to complete, it runs for another 5 seconds and completes at time 15.

As you go through upcoming models, you will note that we almost always avoid time sharing altogether by never running two programs at the same time on a single core. This is typical when one focuses on high performance. However, the reasoning needed to compute how time sharing would impact program execution times is general and applicable to other settings (e.g., sharing of network bandwidth).

There are several exercises concerning work and execution speed at the end of the chapter. For these problems you should be familiar with the following orders of magnitudes.

```
- K(ilo): 10^3
- M(ega): 10^6
- G(iga): 10^9
- T(era): 10^12
- P(eta): 10^15
- E(exa): 10^18
```

AMDAHL'S AND GUSTAFSON'S LAWS

There are two well-known and contrasting laws in parallel computing. The first provides a pessimistic projection and the second an optimistic projection of the potential for parallel speedup.

Amdahl's law states that the maximum speedup that can be achieved is limited by the serial component of the work of any program. If an algorithm expresses that 10% of the program must be executed serially, or equally 90% of code is parallelizable, then even with an unbounded number of processors the speedup is bounded by

$$Speedup < Ts / Tp < Ts / 0.1*Ts = 10$$

In general, we can express with Amdahl's Law that speedup is always bounded by ***Speedup <= 1 / 1-P ,*** *w*here ***1 – P*** denotes the percentage of the serial component (not parallelized) of a program.

In practice, this barrier to performance given in Amdahl's law is not as significant as it appears for properly designed parallel programs. Parallel file systems for example can remove the I/O sequential bottleneck for applications operating on large data sets. Also, the sequential portion of a program often has a lower complexity that the parallel portion, and may vanish as the problem size grows large.

Gustafson's law stands in contrast to Amdahl's law, which assumes that the overall workload of a program does not change with respect to the number of processors. Gustafson's law is parameterized by p, the number of processors, and states the following result about parallel speedup:

$$Speedup(p) = p - \alpha(p-1)$$

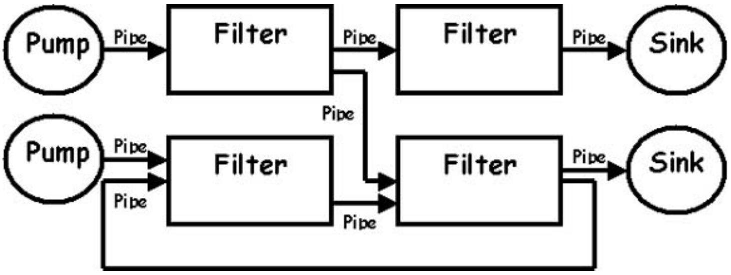
Here, p is the number of processors, Speedup(p) is the speedup factor with p processors, and α is the non-parallelizable fraction of any parallel process.

Gustafson's law is optimistic and suggests that designers first set the time bound allowed for solving a problem in parallel. Then based on that time set the size the problem to be solved in parallel. The faster the parallel system is, the larger the problems that can be solved over the same period of time.

STRUCTURAL PATTERNS IN PARALLEL SOFTWARE

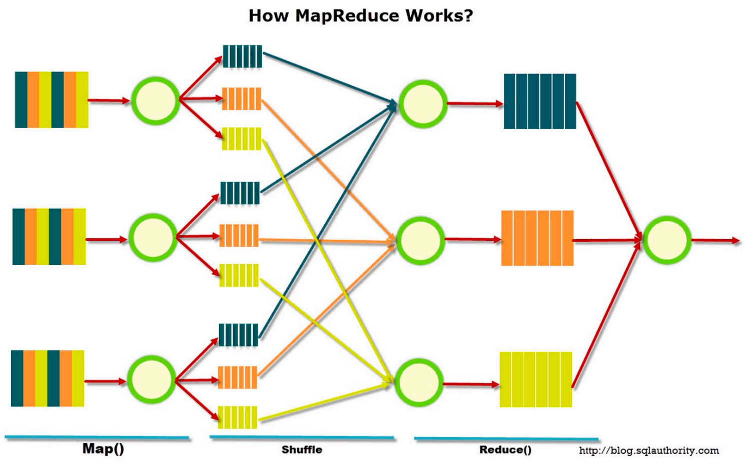
In order to make parallel programming more concrete we now provide the reader with a sense of the range and the types of computational problems we consider by considering a small number (four) of common structural patterns for parallel software. Indeed these four patterns cover a tremendous number of modern applications using parallel systems.

Pipe-and-filter: this pattern is used to solve problems characterized by data flowing through



modular phases of a computation. The solution constructs the program as filters (computational elements) connected by pipes (data communication channels). Alternatively, such algorithms can be viewed as a graph with computations or operations as vertices and communication along edges. Data flows through the succession of stateless filters, taking input only from its input pipe(s), transforming that data, and passing the output to the next filter via its output pipe.

Map-Reduce: the map-reduce pattern is used to solve problems where the same function is independently applied across

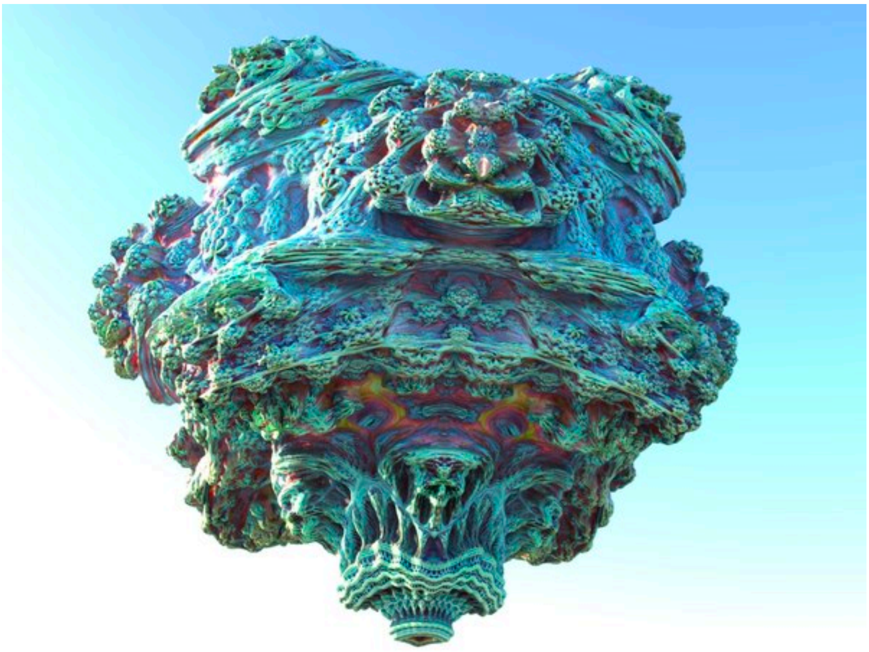


distributed data. The main issue here is to optimally exploit the computational efficiency and parallelism latent in this structure. The solution is to define a program structured as two or three distinct phases: map, shuffle, reduce. The map phase applies a function to distributed data; the shuffle phase reorganizes the data after the mapping to bring related items together locally; and the reduction phase is typically a summary computation, or merely a data reduction. This can be generalized to a Bulk-Synchronous Parallel Strategy which will be discussed in a later chapter.

Agents and Repositories: the agents and repository pattern is used to solve problems organized as a collection of data elements that

are modified at irregular times by a flexible set of distinct operations executed by agents. The solution is to structure the computation in terms of a centrally managed data repository, a collection of autonomous agents that operate upon the data, and a manager that schedules and coordinates the agents' access to the repository and enforces consistency of updates.

Iterative refinement: the iterative refinement pattern is used to solve a class of problems where a set of operations are applied over and over to a system until a predefined goal is realized or constraint is met. The number of



applications of the operation in question may not be predefined, and the number of iterations through the loop may not be able to be statically determined. The solution to these problems is to wrap   flexible iterative framework around the operations as follows: the iterative computation is performed; the results are checked against a termination condition; depending on the results of the check, the computation completes or proceeds to the next iteration. Fractal patterns and related images are often carried out by an iterative refinement process.

EXERCISES

Part A. Short answer questions.

1. What is the difference between the terms concurrency and parallelism?
2. What is the difference between shared memory and message-passing models?
3. What is the difference between multi-threaded control parallelism and data parallelism?
4. In the design of parallel programs what is the difference between task decomposition, task assignment, and task agglomeration.
5. What is the difference between speedup and efficiency of parallel computation?
6. What is the relationship between the work and the execution time of a parallel computation?
7. How does Amdahl’s Law differ from Gustafson’s Law?
8. How does the pipe-and-filter pattern differ from map-reduce?

9. For the pattern map-reduce, how many bulk computation steps and how many synchronization steps are there?

Part B. Calculate Work and Execution time questions

10. You have to run a program that performs 4000 Gflop, and your core computes at speed 30 Tflop/sec. How long will the program run for in seconds?
11. A program just ran in 0.32 seconds on a core with speed 2 Tflop/sec, how many Gflop does the program perform?
12. You have to run a program that performs 2000 Tflop, and your core computes at speed 450 Gflop/sec. How long will the program run?
13. A program that performs 3000 Gflop just ran in 1.5 minutes on a core. What is the core speed in Tflop/sec?
14. On a given core, a program just ran in 14 seconds. By what factor should the core speed be increased if you want the program to run in 10 seconds?
15. At time 0 on a core with speed 2 Tflop/sec you start two programs. Program *A*'s work is 200 Gflop, and program *B*'s work is 220 Gflop. At what times do the programs complete?
16. Two programs, each with work 800 Tflop, were started at the same time on a core and

both completed simultaneously after one hour. What is the core's speed in Gflop/sec?

17. Two programs are started at the same time on a core. These programs both have work 1 Tflop, and both complete after 220 seconds. What was the core speed in Gflop/sec?
18. Three programs, A, B, and C, were started at the same time on a core with speed 600 GFLop/sec. After 10 seconds, A and C complete. Then, 2 seconds later, program B completes. What is the work (in Gflop) of each of the three programs?
19. A program, A, with work 4 Tflop is started on a core of speed 500 Gflop/sec. 5 seconds later another program B, is started. Both programs finish at the same time. What is the work of B measured in Tflop?