PARALLELIZATION OF TASKS WITHOUT A PRIORI KNOWLEDGE OF THEIR DEPENDENCIES

by

Peter S. Aldous

Submitted to Brigham Young University in partial fulfillment of graduation requirements for University Honors

Department of Computer Science

April 2010

Advisor: Jay McCarthy Honors Representative: Bryan Morse

Signature: ____________________ Signature: ____________________

ABSTRACT

PARALLELIZATION OF TASKS WITHOUT A PRIORI KNOWLEDGE OF THEIR DEPENDENCIES

Peter S. Aldous

Department of Computer Science

Bachelor of Science

Computing power has increased rapidly since computers were invented. In recent years, however, computers have had more processors in them instead of more powerful processors. The increase in processors does increase a computer's power, but that increase in power rarely makes individual programs run more quickly because they are designed to run on only one processor.

We **parallelized** `setup-plt`, or split its work between multiple processors. `setup-plt` belongs to a class of programs that are particularly difficult to parallelize. We hoped to make a model that facilitate the parallelization of similar programs. Although we were successful in parallelizing `setup-plt`, we did not decrease its run time as much as we had hoped.

We present our methods and some discussion of the challenges that prevented us from decreasing the run time as much as we had hoped.

Contents

List of Figures

1 Introduction

Computers have been quickly increasing in power since their creation, but it has become more difficult to boost processor speed in recent years. Instead, new computers commonly have an increasing number of processors in them. Although an increased number of processors does increase a machine's computing power, few programs can use multiple processors effectively because most tasks are designed to run **serially**, or on one processor.

Traditionally, computer programs are defined as a sequence of instructions. A computer with one processor executes the instructions in a program in order, one at a time. A computer with two processors can execute two instructions at a time and a computer with four processors can execute four instructions at a time. But because the instructions of a program must be executed in order, a program generally runs on only one processor at a time.

It is possible for a program to be split into separate series of instructions, each of which can be executed independently. This enables a computer with multiple processors to execute the program **in parallel**, or simultaneously on multiple processors. Ideally, a **parallelized** program, or a program split into distinct series of instructions, runs faster than its **serialized** counterpart (the same program in only one series of instructions) by a factor of the number of processors used; that is, a program running in parallel on two processors would run twice as fast as on one processor.

Splitting a program into independent sub-programs, however, is rarely trivial. An instruction may store a value in memory that is read by another instruction after thousands of intervening instructions have been executed. If these two instructions are separated into different instruction sequences and run on separate processors, the latter instruction may read a value from memory long before the correct value is stored there. Many other problems can result from incorrect parallelization.

Research into using multiple processors for the same program is in its infancy. At present, few programs use multiple processors and fewer still use them to the extent that they might. As such, much of the processing power that is available is unused and programs often run no faster on newer computers, even when these newer computers have more computational power.

Although many programs can be parallelized with existing tools and methods, some programs have complex properties that render traditional techniques ineffective. We modified `setup-plt`, one of these complex programs, to run in parallel. Although the parallelized `setup-plt`'s behavior was correct, we were unsuccessful in our attempts to decrease its runtime by a factor of the number of processors used, even after using several different strategies. We instrumented `setup-plt` to create a graph of the different tasks that comprise it in an attempt to determine why our attempts were unsuccessful, but were unable to draw any definite conclusions.

1.1 Vocabulary

As an adequate description of prior work done in this area requires a knowledge of several related concepts, I will include a definition of several important words in this section.

Dependency and Dependency Graphs

Many programs can be represented as a set of interrelated tasks, independent of each other except that some tasks must be done before another can begin. We say that task A **depends** on task B if task B must be completed before task A can begin. We also say that file A **depends** on file B if the action associated with file A cannot begin until the action associated with file B is complete.

Dependencies are frequently represented in a data structure or diagram called a **directed acyclic graph**, or DAG. A graph has nodes, usually drawn as circles, which are connected by edges, drawn as lines or arrows. Figure 1 is a DAG. A dependency graph's nodes represent tasks or files and an edge between two nodes represents a dependency relationship between them. Dependency graphs are directed, which means that an edge between two files or tasks is a one-way arrow. If an edge points from file A to file B, file A depends on file B. Acyclic means that there is no way to return to the same node by following dependency edges in the direction they point. Any cycle in dependencies makes resolution of those dependencies impossible.

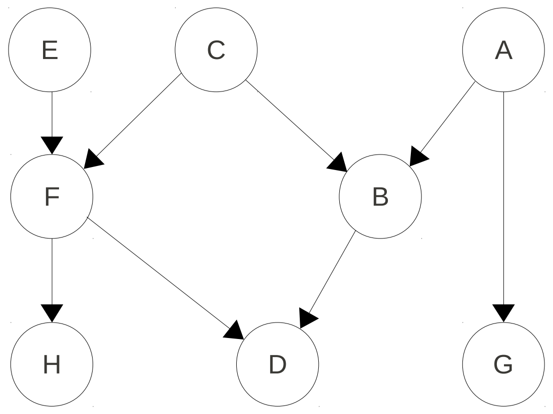

Figure 1: A directed acyclic graph

Compilation, Builds, and `make`

Programs are typically not written in a language computers can use. Instead, they are written in a programming language and then **compiled**, or translated, into an **executable**, or a form that the computer can use directly. A program's behavior is usually written in several different files, called its **source files**. The process of compiling all of a program's source files and combining them as necessary into one executable is called **building** a program.

The process of building a program is more complicated than simply compiling the program's source files. A program can be built faster by skipping files that have already been compiled, but must ensure that it does not skip files whose source has changed since the last compilation. It can also be difficult to combine the compiled files properly. A program called `make` was designed to simplify the process of building a program. It uses a description of a program's source files, their corresponding compiled files, and any dependency relationships between these files to determine which steps need to be performed to build the program and which may be skipped.

1.2 Parallelization with `make`

Given a directed acyclic graph of the source files of a program and their dependencies, it is relatively straightforward to split the graph into subgraphs. `make` assigns subgraphs to different processors, thereby parallelizing the building process.

1.3 Limitations of `make`

Although it is a flexible tool, `make` is not suitable for building programs in every language. PLT Scheme, a programming language, has a macro system that allows programmers to define their own syntax rules, expanding the language to meet their needs. The power of PLT Scheme's macro system makes compilation the only way to discover all of a source file's dependencies, so a dependency graph cannot be created without building the program in the first place. Accordingly, `make` cannot be used to build a PLT Scheme program.

1.4 Terminology specific to PLT Scheme

Programming languages are usually distributed with **libraries**, or already-written programs that provide functionality beyond the basics of the language itself. For example, many languages' mathematical features include only elementary arithmetic operators (such as +, -, *, and /), but the languages' libraries provide specialized mathematical operators and functions, such as exponentiation and trigonometric functions. Libraries in PLT Scheme are called **collections**.

2 Preliminaries

Before parallelizing `setup-plt`, we needed to prepare a set of tools to handle some complications associated with parallel processes. These complications are discussed in Section 2.1. Section 2.2 describes the tools we created to deal with these complications.

2.1 Mutual Exclusion

In order to ensure that the result of our parallelized `setup-plt` was correct, it was necessary to ensure that the different processes did not interfere with each other. It is possible for two processes to compile the same file at the same time, both writing simultaneously to the same output file. If this were to happen, it is almost certain that the file they created would be some mixture of the two programs' output and, therefore, incorrect. When a program is constructed so that only one process may access a **resource**, such as the output file of a compilation, at a time, we say that this program uses **mutual exclusion** to ensure correctness.

2.2 Locking

Many languages provide **semaphores**, which are tools that can be used to provide mutual exclusion. Before accessing a resource, a process can use a predefined semaphore associated with that resource to **lock** it. Other processes that respect the lock will not use that resource until the first process releases its lock. Often, these processes will **block**, or temporarily suspend themselves, when they try and fail to lock a resource. Any function that may block is called a **blocking function** and any call to a function that may block is called a **blocking call**. A function that blocks when it tries and fails to lock a resource is a blocking call. A process that is blocked because it tried to lock some resource is blocked on that resource and will resume when the resource is freed and it succeeds in locking the resource. Some semaphore libraries provide a way to request a lock without blocking (a **non-blocking call**); if the resource is not successfully locked, the process does something that does not involve the locked resource.

PLT Scheme provides a semaphore library, but its semaphores, like its other data structures, cannot be shared between processes. In order to provide multiprocess semaphores, we created a locking mechanism that provided sufficient communication between processes to ensure mutual exclusion in our parallelized `setup-plt`. Our locking mechanism uses the `rename` system call by way of PLT Scheme's `rename-file-or-directory` function. `rename` is guaranteed by each major operating system to be atomic.

Each process only writes a compiled file if it can first lock that file. This guarantees that only one process writes a compiled file at a time. It also ensures that only one process attempts to compile a file at a time. Before compiling, `setup-plt` checks to ensure that there is not already a current compiled version of the file in question. As long as our locking mechanism ensures that this check occurs after locking instead of before, no file will be compiled twice.

The locking system has only a blocking call. The reason for this is that files are locked only when a dependency is discovered and another file must be compiled before the current compilation can continue. Until that dependency is compiled, the process would have nothing to do, so a non-blocking call would be useless.

We tested our locking system with random tests. We created a program that would generate a few different false filenames and a few different processes. In the course of the testing, each process chooses a task at random and executes it: choose one of the filenames at random and lock it, release a lock, or wait for a short period of time. The processes record each attempted task. Once all of these processes have finished, the program validates their records. If any process locks a file that it had already locked, releases a lock that it did not have, etc., that error is reported and the test suite stops.

3 Attempts Made

This section describes the different attempts made to make an efficient, parallelized form of `setup-plt`. Each attempt is first discussed in brief. The Approach subsection of each section describes the attempt in greater depth and then the Results subsection gives and discusses the results of the attempt. In each case, the goal was to decrease the run time from the original run time of 20 minutes.

3.1 Scrambled lists

Because `setup-plt` had never been parallelized before, we began with a simple approach: run multiple instances `setup-plt` at the same time, each one respecting the others' locks and each one compiling files in a different order. The simplest way to ensure that the order of files was different in each instance was for each instance to scramble its list of files to be compiled.

Approach

`setup-plt` compiles over 3,000 files, many of them depending on many others. It is an intricate program, but it works as it is. It begins by finding all of the files in each collection and then compiling each in turn.

Such an intricate program is difficult to modify. However, the intricacy itself suggested that a simple solution might be effective. If these 3,000 compilations were spread over a 20-minute build, it might be possible to have different processors each attempt to compile each file independently. We reasoned that as long as the processes did not compile already-compiled files, they might work together efficiently.

We modified `setup-plt` to lock before compiling to ensure that the different processes would not interfere with each other. We also wrote a program that would start several instances of `setup-plt` and record how long the whole process took.

Results

Our first attempt took longer on two processors than the original, serialized version did. We observed CPU usage charts and found that the CPU utilization was far below what we had expected: only one of the processors was consistently in use, no matter how many processes were running. We determined that the different processes were waiting for each other much of the time. We changed the lock mechanism to wait for a shorter period of time (1 second instead of 10). This change sped the process up enough that the parallelized builds took less time than the serialized build, but not much: they ran in 18 or 19 minutes, not 20. The 4-processor build took about 1.5 minutes longer than the 2- and 3-processor builds, which took just over 18 minutes.

3.2 DAG analysis

Since CPU usage charts in the scrambled lists approach indicated that the processes were waiting for each other, we began examining the dependencies of the files that `setup-plt` compiles in an attempt to find a faster way to parallelize it.

Approach

We instrumented `setup-plt` to build a DAG as it ran so we could analyze the structure of the problem after the fact. **Traversing** this graph, or visiting each node in the graph, is an abstract representation of the completed compilation, so a visual representation of the graph is a representation of the work that `setup-plt` did.

The instrumented `setup-plt` used the dot language for its output. After several attempts, we failed to produce a useful result, as every software tool that we tried was incapable of processing such a huge graph. We were able to simplify the graph by choosing collections rather than source files as nodes. Even with just 30 nodes or so, the graph was too visually large and complex to be very useful. Figure 2 and Figure 3 demonstrate the huge size of this very simplified graph.

We hypothesized that there was a small number of collections, such as `scheme` and `mzlib`, whose compilation accounted for the majority of `setup-plt`'s run time. Accordingly, we changed the structure of `setup-plt`. One process became the master and compiled all of the collections. The other processes first compiled the files inside of the `scheme` collection and then proceeded to the rest of the collections. We hoped that this approach would speed up the compilations that were causing a bottleneck and that the different processes would be able to run independently after that bottleneck cleared.

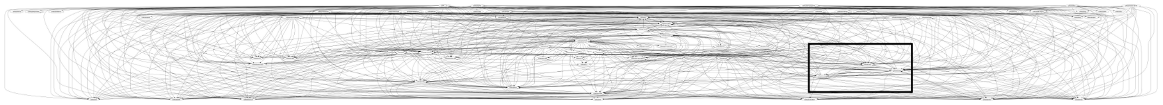

Figure 2: Dependency graph of PLT Scheme collections

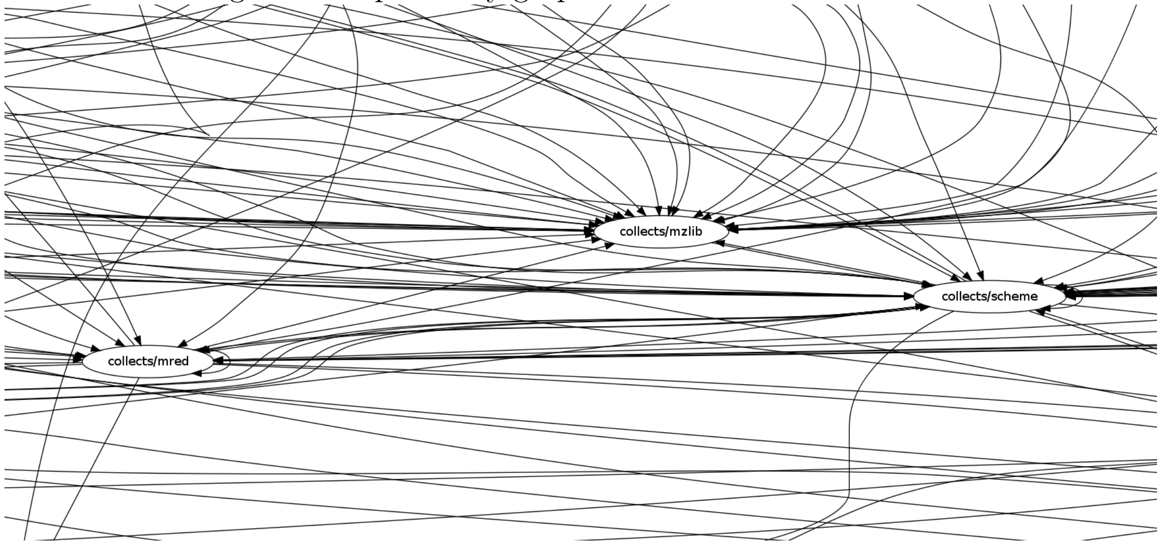

Figure 3: Detail (box) of Figure 2

Results

We ran this version of `setup-plt` 100 times and recorded their running times. The average run time of the 2-process build was about 16 minutes, as was the average run time of the 3-process build. The 2-process build had a small number of run times that were close to 20 minutes. The 4-process build varied more than the others and averaged about 17 minutes.

Once again, adding more processors slowed the process instead of speeding it up. The CPU usage charts showed, as with the scrambled lists approach, the CPUs were less used than expected.

3.3 Job queue

`setup-plt` does several tasks, such as building documentation, in addition to compiling PLT Scheme files. It also does extensive preprocessing before compiling its files. We had hoped that running `setup-plt` would entail minimal duplication of effort, but experimentation proved that this was not the case. We resolved to make a version of `setup-plt` that would create its own helper processes to remove these redundancies.

Approach

We used an approached characterized by a **job queue** and **worker threads** to manage the load (for the purposes of this paper, a **thread** is a process). The job queue is like a list of work to be done and the worker threads are processes that actually perform the work. A **controller thread**, or managing process, maintains the job queue. Worker threads request jobs from the job queue and complete them until all the jobs are complete. In this case, the jobs assigned to worker threads were the compilation of files. The controller thread does the setup before compilation, creates and manages the job queue, and then performs other tasks, such as documentation.

Results

We ran the job queue-based `setup-plt` over 400 times with very consistent results. The 2-processor build averaged about 18 minutes, the 3-processor build between 16 and 17 minutes, and the 4-processor build averaged 16 minutes. Encouraged by the fact that an increase in processors appeared to be correlated with a decrease in run time, we tried a 12-processor build. It took much longer than 20 minutes, indicating that another attempt to take advantage of processor power did not succeed.

4 Conclusions

Parallelization is not a trivial task. Some programs can be modified to run in parallel with only minor modifications, while a method to do the same with other programs has not yet been found. We hoped to find such a method for the class of programs whose work can be represented as a directed acyclic graph, but whose graphs are difficult to analyze a priori. Rather than finding such a method, we succeeded in demonstrating that some techniques are not always effective.

Further work in this area could include experimentation with other approaches. For example, a system that provides a non-blocking call could be useful for some programs, although we did not find a way to use it in the parallelization of `setup-plt`. It would be instructive to try the same techniques we used on other workloads of a similar structure, as other programs may have structures more amenable to simple parallelization.