Parallelism and Pedagogy in Weather Modeling

Skylar Thompson

November 5, 2012

Abstract

Weather modeling software can be computationally intensive and can also produce cryptic output. To attempt to solve the first problem I tried various techniques to run a weather model using multiple processors. For the second problem, I engineered some tools to increase the readability and usability of the output.

Contents

1 Introduction

Since the beginning of computer science, computers have been used to solve problems in other disciplines. While computers are rarely necessary in solving problems, they can greatly increase the speed at which problems ranging from chemistry to mathematics are solved.

One such application is that of weather modeling. Weather modeling involves taking the current environmental conditions (temperature, precipitation, etc.) and producing likely conditions for a given time and place. The more fine-grained the data is, the more precise the model, which means that the computational requirements of weather modeling are among the highest of any problem.

While in the past, weather modelers typically turned to supercomputers (i.e. the single-image multi-computers mentioned in Section 2.2.1), recently clusters of computers have been built to perform the same task as the supercomputer at a fraction of the cost. This involves tailoring the execution of the application to run on many separate computers, and the focus of this paper is on the methods and results of the parallelization of the CLIGEN weather modeling package from Purdue.

CLIGEN was originally designed by Purdue and the Department of Agriculture to measure soil erosion from rainfall. It is now maintained by Purdue and the Forest Service as a general-purpose climate model for the continental United States, as well as certain international stations.

2 Parallel Computation

The current trend in high-performance computation is away from single specialized computers, and more towards networked general-purpose computers. While this complicates the interactions between different parts of a single system, this cost is small compared to the benefits brought by the use of smaller, more numerous components. [**?**]

2.1 Paradigms

There are several ways of parallelizing a serial algorithm. The most common approaches are partitioning (whether by data or by function calls), divide-and-conquer, and pipelining. The decision of which one to use is specific to the problem or problem class. The wrong approach can substantially reduce the performance of a parallel algorithm compared to its serial implementation.

Partitioning Partitioning is the simplest method of attacking parallelization problem. It involves a mother node sending a chunk of information to each compute node, and each one performs the same operation on that chunk. The mother node then coalesces all the data before terminating. [**?**, page 106] See figure 1.

Divide-and-conquer Divide-and-conquer is related to partitioning, but is more complex. It is frequently used in developing a parallel algorithm for a problem that can be recursively defined, such as tree and sorting operations. The problem is successively divided into smaller chunks, and the leaves are solved and then passed up one level towards the root. The new leaves are then solved, and the process continues until there are no more leaves to be made. [**?**, page 111] See figure 2.

Pipeline Pipelining is an alternative to partitioning, and is much closer to the traditional architecture of processing units. Rather than engineer for each compute node to work on the same task as all the rest, each one specializes in one stage, and then passes on its work to the next node in the sequence. The advantage to this

Figure 1: Example partitioning scheme.

Figure 2: Example divide-and-conquer scheme.

Figure 3: Example pipeline scheme.

comes in cache coherency on the CPUs of the nodes, because each step can be smaller relative to the all-encompassing steps that partitioning involve. Cache coherency allows processors to maintain the same instructions in fast memory close to the processor, rather than having to reload them for each cycle of the computation. [**?**, page 140] See figure 3.

2.2 Types of Parallel Computers

Parallel computers fall into two distinct types, depending on the degree of resource sharing. One type is a shared-memory multicomputer, and the other is a message-passing multicomputer. [**?**]

2.2.1 Shared-memory multicomputers

Multicomputers that use a shared-memory architecture allocate all memory for processes and data within a single address space, which makes programming for them trivial. Typically all that is needed is pragmas within the program's source code that direct the compiler to add some form of parallelization, which are typically threads. Alternatively, if the compiler is not instructed to process the directives, they will be ignored and the program will run serially without further modification. [**?**, page 17]

Due to the simplicity of the interface to the parallelization, modifying large, existing serial programs can be trivial compared to the call-level interface of message-passing. There have also been some studies showing that large data-sets do not benefit significantly from message-passing as compared to shared memory implementations. [**?**]

2.2.2 Message-passing multicomputers

Message-passing multicomputers are less tightly integrated than shared-memory multicomputers. They share only a network connection with the other processors, and each processor has its own dedicated memory and address space. Any data that is to be shared with other processors has to be explicitly sent and received by each node. While this might seem to be a limitation, it eliminates scoping issues without extra work.

Another advantage is the ability to use COTS [1] components. Standard PC hardware can be used, along with standard Ethernet networking gear. Not only does this reduce the initial cost, but it also reduces the maintenance costs by avoiding components only supplied by a specific vendor.

Message-passing clusters tend to scale more easily as well, to a point. Because of the commodity nature of the equipment, new interconnects and nodes can easily and cheaply be added. The problem comes in when too many nodes are needed on a given problem. Because parallelization adds an inherent communication overhead, the cost of nodes talking to each other can reduce the degree to which the problem can scale. [**?**, page 16]

3 Weather Modeling

3.1 General Concepts

Weather modeling is dependent on many factors, but all models are implemented as a grid with each section containing data on certain characteristics. Whether manual or computerized, it is a technique that is and will continue to be useful, particularly in regions that are already vulnerable to climate change. [**?**, page 435]

Weather models are dependent on many factors, such as atmosphere, oceans, snow cover, land, and biological presence. [**?**, page 436] The precision on these data can obviously vary widely, and the size of each data point can also clearly vary widely as well. As Haltiner points out, weather models are "limited by computing

[1] Commodity Off-The-Shelf

capability and is expected to remain so for the foreseeable future", so the potential for parallelization is high. [?, page 437]

Weather models have many uses. For example, here in Wayne County, weather models are becoming increasingly important as there is a move to green energy sources such as wind and solar generation plants. There is little point in building a green-energy infrastructure if the site cannot support a return-on-investment. [?, page 15]

3.2 Parallelization

In order to parallelize a program, one must first determine which parts or functions of the program take the most time, and then which of those are most amenable to parallelization. After choosing sections of the code to parallelize, one must then choose one of the above-mentioned parallel programming paradigms that will best improve the performance of the program.

3.2.1 Profiling

One of the most popular techniques for determining areas that are candidates for parallelization is profiling. Profiling data is collected while the program is running, and then is dumped to a file or directly to the screen after execution is completed. The profiler will return the number of times each function is called and the amount of time the program spent in each function. If this is run on a multitasking time-sharing system, the time reported should compensate for other possible loads on the system. [?, page 5]

To profile my code, I used the GNU gprof profiler. Support for gprof is added at compile-time using the "`-pg`" flag on GCC compilers. This adds in benchmarking code to each function, which uses counters and high-resolution timers to produce the data mentioned above. In addition, gprof produces graphs showing the common paths through the program. Using this, I can not only see what functions the program spends a lot of time in; I can also see from where these functions are frequently called.

As an example, I have included two sections of gprof data in Figures 4 and 5. Figure 4 shows the amount of time CLIGEN spent in each subroutine (but not in its children), and the number of times that subroutine was called. This is useful information, but in a program the size of CLIGEN—which is over 8000 lines of FORTRAN with an additional 2000 lines in common blocks—that kind of data is not enough. It is also necessary to know where these functions are being called from, and that is the purpose of the call graph. Using Figure 5, which is adapted from the text call graphs in the gprof output, it is possible to see precisely the areas of the

program that are called the most and how the program gets there. Unfortunately, it cannot provide data on the path of communication in a parallel environment. [**?**, page 4] For this, a tool like XMPI could be used to show visually the ratio of computation to communication.

3.2.2 My approach

In order to be useful, my approach must be fast [**?**, page 428], and as near to real-time as possible [**?**, page 539]. This is so that people can experiment as much as possible with the data without having to idle.

A similar problem achieved a best-case 4.11x speedup using PVM and 16 processors. [**?**, page 542] PVM is a message-passing library, and they discovered that their major bottleneck was communication overhead. In my case, the call-based interface to message-passing libraries is a significant impediment to parallelization. In order to avoid initialization of the MPI environment multiple times I would have to split out the subroutines that I am parallelizing, which could then interrupt the GOTO-based flow control of the model.

After doing more tests, Sabot tried their weather model on a Thinking Machines CM-5, which is a highly specialized shared-memory multicomputer. They exceeded the maximum speedup on their runs by two-fold, primarily due to a fast communication system and integrated address space. [**?**, page 544]

For the reasons mentioned above, I chose to use a shared-memory approach. While this limits the amount of parallelization I can do to the available hardware, it also limits the disruption to the existing source code. I don't have an in-depth knowledge of weather modeling and it is difficult to ensure the integrity of the model, so the fewer changes I make the more sure I am that the model will remain correct.

Unfortunately, while the random number generator consumes the majority of the execution time, it is also where CLIGEN is most sensitive to errors. [**?**, page 6] According to the CLIGEN website, much effort has been put into ensuring the quality of the random number generator, so I am reluctant to touch it without knowing the reasons behind their changes. I instead chose to attempt to parallelize the χ^2 portion of the random number generator, which seemed a less intrusive change.

Despite attempts with both message-passing and shared-memory implementations, the loops in the χ^2 subroutine showed little speedup, and occasionally even negative speedup. After looking through the rest of the program, I determined that the problem is likely the use of global variables in FORTRAN common blocks. I then declared temporary variables for use in the parallelized loops, but the coalescing of the temporary and global variables after the parallelization ate up any benefit that

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
33.25	54.51	54.51	301811185	0.00	0.00	randn
27.76	100.02	45.51	601368080	0.00	0.00	dstn1
25.57	141.94	41.92	12000	0.00	0.01	ranset
5.38	150.75	8.82	9860726	0.00	0.00	ks_tst
2.42	154.72	3.96	6856268	0.00	0.00	conflm
2.37	158.60	3.89	6856268	0.00	0.00	gratio

Figure 4: Number of seconds and number of calls for the top subroutines in CLIGEN.

the parallelization might have gained.

4 User interface

The second part of my project involved writing a user-friendly interface to CLIGEN. The first portion of this involved writing a set of scripts to load and extract data in a SQL database. The second uses this base to generate graphs, but could be extended in many ways to analyze the data.

First, I wrote a script to take a CLIGEN Option-6 input file and load it into a SQL database. This allows an easy migration for users with existing Option-6 data, but otherwise is unnecessary if users already are using their own database.

I then wrote a second script that extracts data from the SQL database into CLIGEN Option-6 format. Originally I planned to allow CLIGEN to read directly from the database, but this would involve writing a set of libraries in C and passing the data back into FORTRAN for CLIGEN. This interface involved a significant amount of work, and the amount of time to generate an Option-6 file from the database is trivial from both a software engineering and user perspective.

A third script runs CLIGEN with the prepared input file, and loads CLIGEN's output into another SQL database. The last script then takes this data and uses gnuplot to produce graphs of precipitation, temperature, etc.

The graphs I believe are incredibly useful, especially for a lay person who is not used to the format of the data. My scripts allow the specification of a date range, and condense thousands of data points into something that is easily analyzed. Future work would easily allow the addition of a web interface that would allow users anywhere to access the data.

Figure 5: Call Graph for CLIGEN execution

Figure 6: Graph of maximum temperature variations.

Figure 7: Graph of precipitation.

Figure 8: Graph of wind speed.

5 Future Work

This project leaves several avenues for future work. The parallelization obviously needs to be completed. I think learning more about random number generators and particularly parallel implementations of random number generators would be very useful.

For the pedagogical aspect of the project, I think writing a web interface of some kind would allow those people without access to a command-line interface to utilize my scripts. It would also eliminate many dependencies my scripts have on certain Perl modules being installed, and not require users to have direct access to the database.

Acknowledgments

- Many thanks to Charles Meyer of Purdue University for his timely help with the running of CLIGEN.
- Charlie Peck was invaluable in his knowledge of FORTRAN and a variety of parallelization techniques.