New Multithreaded Code for Calculating Longitudinal Collective Instabilites using Computers with Multiprocessors

C.Y. Tan*, FNAL, Batavia, IL 60510, USA

Abstract

We have developed a new mulithreaded code with pthreads for calculating longitudinal collective instabilites on computers with multiprocessors. We have selected pthreads as the basis for multiprocessing because it is portable, as such we are able to port this code to Solaris, IRIX and OS/2 platforms. We will demonstrate that when there are four cavities and 36 bunches in the simulation, our code shows a speed increase of $> 3\times$ compared to single processor code when run on a symmetric multiprocessing (smp) machine.

1 INTRODUCTION

There are many well established codes used in longitudinal simulations of beam instabilities, like ESME [1], which run on a single processor. With the introduction of computers with many processors (multiprocessors) like SUN Sparcs with four or more processors, it seemed to be a waste not to use the other processors in our simulations. Modern workstations with many processors are usually symmetric multiprocessing[1] and the vendors have written their operating systems to support this architecture. This means that the vendor has already written the appropriate libraries which allow the programmer to easily write programmes which can run on multiprocessors. The usual challenge which faces the physicist is to come up with a parallel processing algorithm which takes advantage of the hardware. We will show in this paper that for longitudinal simulations with many RF cavities, the parallel algorithm which we will choose is very natural and indeed takes advantage of the multiprocessor machine. The next challenge which we will face is the choice of libraries and language. The choice for us will again obvious: we will choose a library which is cross-platform and an object-oriented language.

We will, in the rest of the paper, introduce the idea of threads and the pipeline model used in the simulation. Plus the speed increases compared with single processor code.

2 THREADS

A thread is defined as a sequence of instructions to be executed within a programme. Most programmes are single threaded code which consists of one thread of execution which starts in main(). Before the invention of threads, the UNIX way of doing things in parallel is to use the fork/exec model. This model spawned several processes each a thread of execution which can then be run in parallel. However, each new process that is created by fork/exec require the operating system to make a child process from its parent process by copying over the instruction, user-data and system-data segments of the parent and then executing the child process as well as the parent process. This method is expensive because of the overhead required for each process creation. Furthermore, communication between parent and child need external channels like pipes, sockets, memory maps (mmaps) etc. because resources held by parent and child are private and neither parent or child can peek into each others resources without using these mechanisms. The fork/exec model can therefore be thought of as many processes each having its own thread of execution.

Contrast this with a multithreaded programme where there is one process with many threads of execution. This means that different parts of the same code can be executing in parallel. The advantages over fork/exec are immediate : all threads share the same resources and the overhead of creating a new process is eliminated. The downside is that because all threads share the same resources, there must be software mechanisms, like locks and semaphores, which preserve the integrity of the data and prevent race and deadlock conditions. For example, suppose there are two threads A and B which use a piece of shared data. At some point in the process, thread A updates the data, so thread A must prevent thread B from reading the data before thread A completes the update, i.e. prevention of a race condition. Furthermore, if a situation arises where thread A cannot finish the update unless thread B reads the data then we have a deadlock condition. This means that our application is stuck in a never ending wait between threads A and B. To prevent these type of problems, judicious use of locks and semaphores is essential. Unfortunately, this adds to the overhead of multithreaded code when compared with single threaded code which means that there are scenarios where single threaded code will run faster than multithreaded code.

2.1 Pthreads

POSIX [2] threads or pthreads is a cross-platform implementation of threads whose programming interface is specified by IEEE POSIX 1003.1c standard (1995). This standardization allow us to port the code with minimal changes to two distinct UNIX platforms: Solaris, IRIX and one PC platform: OS/2. Although pthreads is highly portable, there are some shortcomings. For example, a feature which we have thought will be useful is the explicit specification of the mapping between thread and CPU. This feature does not exist in pthreads and thus under this programming en-

* cytan@fnal.gov

[1] Symmetric multiprocessing (smp) means that all CPUs are treated the same, as opposed to asymetric multiprocessing where some CPUs are special and others which are slaves.

vironment, the programmer defers the dispatch of threads to CPUs to the operating system. If the programmer really wants to force a thread to a CPU, (s)he must use non-portable system calls.

3 MODEL

We will describe here the pipeline model used to parallelize the longitudinal simulation code. The pipeline model, which is extremely similar to the Ford assembly line, is shown in Figure 1. We start first with a computer with M processors. For illustration, let us suppose that in our simulations we have $1 < n \leq M$ RF cavities. We can naturally place a single thread of RF cavity code on each CPU as shown in Figure 1. We suppose that we have N bunches numbered bunch0, bunch1,. . ., bunch$(N-1)$ for the simulation. At the start of the simulation, we put bunch0 into RF Cavity 0 with the other bunches waiting to be processed. Once RF Cavity 0 is done with bunch0, it is sent to RF Cavity 1 and bunch1 is sent into RF Cavity 0. Both RF Cavity 0 and RF Cavity 1 can then work on their bunches in parallel. Once RF Cavities 0 and 1 are done with their respective bunches, bunch0 goes to RF Cavity 2, bunch1 goes into RF Cavity 1 and bunch2 goes into RF Cavity 0 where they are again processed in parallel. As this process is repeated for the remaining bunches, we can see that eventually all the processors will be working in parallel. The RF cavities themselves form a pipeline and the bunches propagate through this pipeline starting from RF Cavity 0 and move towards RF Cavity$(n-1)$ which is at the end of the pipeline. The first bunch to reach the end is bunch0 which means that bunch0 has completed one turn through the accelerator and is ready to be fed back into the beginning of the pipeline at RF Cavity 0 again. This is done *ad infinitum* for all the bunches until the required number of turns have been met.

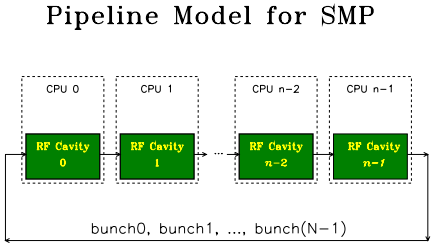

Figure 1: This figure shows the pipeline model used in the simulation. Each RF cavity is placed on a CPU and each bunch is evolved through each cavity like a pipeline.

The advantage of this model is that when there is more than one CPU in the computer and more than one RF cavity in the simulation we can efficiently utilize the computer resources to speed up the calculations by placing a thread on each CPU. However, when there is only one RF cavity in the simulation, the overhead used to set up and maintain the pipeline, which basically consists of one thread only, will be much greater than single threaded code. This means that the simulation will run slower using multithreaded code compared with single threaded code. Similarly, for the case when there is only one CPU and many RF cavities in the simulation, i.e. many threads and one CPU, there is also a comparable slowdown.

It is also important to keep the pipeline from stalling. Consider the case when there is only one bunch and many RF cavities. Then all the RF cavities except one are sitting idle in the simulation, thus again the thread overhead overwhelms any advantage of using multiprocessors. We can always tell that the pipeline is stalled because when the number of bunches is increased with the number of RF cavities fixed, the speed of the simulation will increase and tend to an asymptotic limit which is when all the CPUs in the pipeline are working with full steam.

Finally, it is also not too much of a stretch of the imagination to think of each RF cavity as an object which is mapped to each CPU and each bunch is an object which is manipulated by each RF cavity. Thus the natural language to code the simulation in is in some object oriented language like *C++*.

4 RESULTS

For amusement we show the results of *top* in Figure 2. This shows that our programme *bl* is using 330% more CPU than the next highest user thus leaving him in the dust! For a definitive check of the speedups, we compare completion times with single threaded code and multithreaded code and saw $> 3\times$ increase in speed when we have four RF cavities and 36 bunches in the simulation. Table 2 shows the actual speed increase with timed results on a Sparc workstation with six processors (unfortunately we can only use four, see *CONCLUSION* for the reason) and running SunOS 5.6. We define the speedup factor to mean

$$\text{speedup factor} = \frac{\text{wall-clock time with single thread}}{\text{wall-clock time with multithreads}} \quad (1)$$

Notice from Table 1 that there is a cross-over point when a single threaded simulation is *faster* than a multithreaded simulation when we have two RF cavities and < 4 bunches in the simulation. The single threaded simulation is faster than the multithreaded simulation because the pipeline is stalled from too few bunches.

The other extreme case is when we have more RF cavities than CPUs, which is illustrated with the eight RF cavities case, and even then, the multithreaded performance is still much better than single threaded performance.

5 CONCLUSION

As can be seen from Table 1, the speedup factor is dependent on the number of RF cavities as well as the number of bunches. Thus to determine whether there is an advantage in doing a simulation with multiprocessors, we

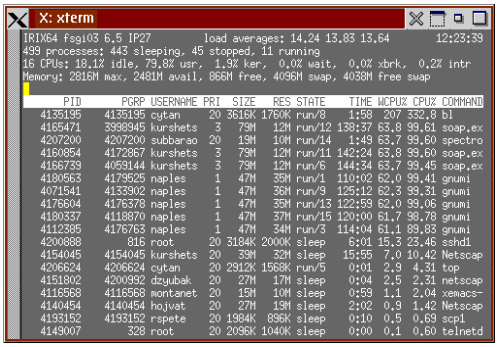

Figure 2: This shows the output of *top* which is used to monitor CPU usage. As can be seen from the line containing USERNAME cytan, our programme *bl* is using 332.8% CPU while other users are using $<$ 100% CPU. This particular machine has 16 CPUs and we are using 4 RF cavities and 36 bunches in the simulation.

Table 1: Comparison between Single and Multithreaded Performance

#RF cav.	# CPUs	# bunches	speedup factor
8	4	36	3.2
4	4	36	3.2
2	2	36	1.7
2	2	8	1.2
2	2	4	1.0
2	2	2	0.5

must first do a computation with a short simulation time to determine which method is better before committing ourselves. Furthermore, not only do we have to consider the non-linear scaling between the speedup factor and the number of CPUs, there are also hard and soft speed limits which prevent us from going infinitely fast

- A hardware limit which is the overhead in communications between processors. Even when we have four available processors on the Sparc to simulate four RF cavities, we do not get $4\times$ increase in speed.
- A software limit put in by the system administrators which prevent us CPU hogs from using more than four processors.

Notwithstanding the above two limits, we have shown that multithreaded code which runs on modern smp machines can indeed make our simulations run much faster and clearly in the 21st century this is the way to go.

6 REFERENCES

[1] J.A. MacLachlan, http://www-ap.fnal.gov/ESME

[2] D.Y. Butenhof, “Programming with POSIX Threads”, Addison Wesley Publishing Company, 1997.

[3] C.Y. Tan, “New Multithreaded Code for Calculating Longitudinal Collective Instabilites using Computers with Multiprocessors”, Fermilab Technical Note, TM-2140, for more details.