SLAC-TN-11-009

Parallelizing AT with MatlabMPI

Evan Y. Li

Office of Science, Science Undergraduate Laboratory Internship (SULI)

Brown University

SLAC National Accelerator Laboratory

Menlo Park, CA

August 20, 2010

Prepared in partial fulfillment of the requirements of the Office of Science, Department of Energy's Science Undergraduate Laboratory Internship under the direction of Xiaobiao Huang at the Stanford Synchrotron Radiation Lightsource (SSRL).

Participant: ______________________________

Signature

Research Advisor: ______________________________

Signature

Work supported in part by US Department of Energy contract DE-AC02-76SF00515.

TABLE OF CONTENTS

Abstract ii

Introduction 1

Dynamic Aperture Determination 2

Discussing `ringpass` 3

Parallelization Strategy 4

Installing and Running 5

Results and Discussion 7

Conclusions and Future Work 8

Acknowledgments 9

References 10

ABSTRACT

Parallelizing AT with MatlabMPI. EVAN Y. LI (Brown University, Providence, RI 02912) XIAOBIAO HUANG (Stanford Synchrotron Radiation Lightsource (SSRL), Menlo Park, CA)

The Accelerator Toolbox (AT) is a high-level collection of tools and scripts specifically oriented toward solving problems dealing with computational accelerator physics. It is integrated into the MATLAB environment, which provides an accessible, intuitive interface for accelerator physicists, allowing researchers to focus the majority of their efforts on simulations and calculations, rather than programming and debugging difficulties. Efforts toward parallelization of AT have been put in place to upgrade its performance to modern standards of computing. We utilized the packages MatlabMPI and pMatlab, which were developed by MIT Lincoln Laboratory, to set up a message-passing environment that could be called within MATLAB, which set up the necessary pre-requisites for multithread processing capabilities. On local quad-core CPUs, we were able to demonstrate processor efficiencies of roughly 95% and speed increases of nearly 380%.

INTRODUCTION

As the field of accelerator physics continues to grow, so too does the need for faster, more efficient methods of beam simulation in computationally intensive tracking functions. The Accelerator Toolbox (AT) for MATLAB, which has been in development at the Stanford Synchrotron Radiation Lightsource (SSRL), has grown increasingly popular throughout the scientific community in recent years. This is due to its ability to combine the efficacy of modern-day, industry-standard tracking algorithms with the flexibility, intuitiveness, and efficiency of the MATLAB working environment, making it the modeling code of choice in SLAC projects such as the SPEAR3 Synchrotron Light Source [1].

Though it functions as a powerful tool, AT's methods and implementations for particle simulation still show room for further development. A number of the most difficult tasks in storage ring simulation, particularly processes such as lattice design optimization and dynamic aperture tracking, require highly computation-intensive algorithms. With plans to upgrade the performance of the SPEAR3 storage ring, SSRL now requires even more powerful calculations to be implemented for optimization processes. This drastically increases the amount of time and processing power needed to produce satisfactorily accurate models of particle beam motion. While AT's current methods are presently capable of performing these tasks, code can take a sizable amount of time to run to completion, detracting from the efficiency that AT strives to provide for its users.

Our goal is to upgrade AT's functionality with the efficiency standards of modern computing. We have worked to implement design changes to our computing models by transitioning the passmethods from their original serial-code construction to a generalized parallel-computing structure, allowing these functions to take advantage of both the symmetric multi-processing capabilities of most modern-day computers and the overwhelming computational power of supercomputing clusters present in large-scale research facilities. It should be noted

that parallel computing has proven to be one of the most straightforward and promising approaches toward hastening the computing speed in accelerator physics, as the main source of bottlenecks in AT's performance stems from heavily reiterative calls of particle tracking functions which are largely independent of each other. In addition, tests in computational accelerator physics have demonstrated promising values for processor efficiencies [2], allowing us to reduce computing times by hundreds of factors with the use of a large computer cluster.

The first revisions we implemented were conducted using OpenMP specifications for C/C++ compilers (OpenMP has emerged as the standardized model for shared memory computing). By exploiting the multi-core processing capabilities of common workstations, we were instantly able to observe runtime speed increases to nearly 400% of their original values. Once we were able to successfully demonstrate the efficacy of parallel computing on a local machine, we then worked to implement parallelization adhering to a distributed memory model by using the MatlabMPI implementations of the standardized Message Passing Interface (MPI), which allowed us to run AT on multi-CPU setups and supercomputing clusters. By introducing these modifications, we were able to demonstrate enormous upgrades in AT's computational abilities, and can potentially reduce processing times to less than 1% of those associated with the previous implementations.

DYNAMIC APERTURE DETERMINATION

The dynamic aperture is a property of accelerator structures which describes certain boundary conditions, inside of which a particle can continue to exhibit a stable orbit as it passes repeatedly through the electromagnetic fields of the apparatus. Conversely, particles located outside of the dynamic aperture will tend to exhibit chaotic orbits, causing their transverse displacements from the axis of the beam to amplify, resulting in collisions with the physical

aperture of the machine. This chaotic behavior is a highly problematic experimental factor, as it results in low injection efficiency, beam loss, and increased radiation hazards. To ensure a successful design, the dynamic aperture must be calculated repeatedly and the beam optics adjusted many times to maximize the stability of the beam.

In the particle accelerator design process, tasks of calculating the dynamic aperture and finding the optimal lattice design have proven to be some of the most computationally intensive problems that accelerator physicists face, with optimization algorithms demonstrating runtimes on the order of several hours, days, or even weeks. Currently, there is promising evidence to suggest that these are the algorithms that will benefit most from implementations of parallel computing. Our project will focus on `ringpass`, which defines the function that AT uses to carry out this computing procedure and thus will be the focus of our parallelization efforts.

DISCUSSING RINGPASS

AT models accelerator particle motion by representing the electromagnetic properties of the fields in the lattice as passmethod functions, M, which act upon the vector representation of the particle to produce the evolved trajectory vector:

$$\vec{X}(n) = M_n M_{n-1} \cdots M_2 M_1 \vec{X}_0$$

Calls to `ringpass` will evolve the particle trajectories as the script is run. Each function call receives three input arguments, which correspond to the following variables: the particle accelerator's lattice of beam optics, the particle beam's initial coordinates, and the number of turns for which the beam should be tracked. Instances of `ringpass` often involve millions of reiterative calls to the AT library functions and passmethods. Although intensive and seemingly inefficient, these are calculations that must be executed due to the chaotic, non-

linear behavior of high-order magnets. This behavior results in beam motion that cannot be analytically described, thus calling for heavily taxing numerical calculations to obtain an accurate visualization of the beam's behavior.

PARALLELIZATION STRATEGY

Parallel computation has become an increasingly universal method for performing large-scale computations in an efficient manner. The parallel model allows jobs to be broken down into discrete sets of instructions, which can be executed simultaneously on multiple processors, allowing for dramatic increases in performance quality. Our methods focused primarily on the various approaches for making AT compatible with parallel processing.

As parallel computing has grown increasingly popular, a number of methods have emerged as the standardized systems for the varying forms of parallel computation (Figures 1, 2). Our first goal was to introduce OpenMP directives and routines to the existing AT source code, allowing the toolbox to be used on systems following a Uniform Memory Access (UMA) model. This is a straightforward procedure which requires only minor alterations to the code as the passmethods are written in C, for which OpenMP has a direct implementation that is compatible with MATLAB's MEX compiling function [3].

The shared memory model can be further improved upon by expanding and adhering the structure to a general distributed memory model. However, the transition toward a distributed memory model is less straightforward. While MathWorks has developed packages for cluster parallelization, standard versions of MATLAB do not currently support methods for message-passing between CPUs. We solve this issue by invoking MatlabMPI and pMatlab [4], which can be used by MATLAB as implementations of the Message Passing Interface (MPI), a library that has emerged as the universal standard for message-passing programs. pMatlab operates by adhering to a Single Instruction, Multiple Data (SIMD) computing

structure by using distributed array datatypes, which partition data amongst CPUs to be processed separately in accordance with a single, defined set of instructions.

Once we set up the distributed computing environment for MATLAB, we were able to combine our implementations of OpenMP and MPI by using the message-passing functions to distribute data to a cluster of multi-core machines. This allows us to utilize the shared memory computing interface for managing local task distribution for each of the individual CPUs, while functioning in a global distributed memory interface. To perform all of these tasks, we set up the MATLAB environment and AT toolbox on all of the machines in the cluster and utilize the MPI routines on the host machine to distribute partitions of the particle beam vector representation amongst each of the worker machines. At the end of the computing procedures, information from each CPU will be consolidated and processed, allowing us to receive outputs from beam tracking functions in a fraction of the original serial-code's runtime.

INSTALLING AND RUNNING

Configuring pMatlab and MatlabMPI

Our parallelized implementation of `ringpass` uses the packages MatlabMPI and pMatlab from MIT Lincoln Laboratory. It can be installed using the following instructions:

- The user must extract the pMatlab library to a directory which is visible to all CPUs in the communicator

- Each MATLAB startup.m file on individual nodes should be modified to run the scripts contained in the pMatlab/startup.m file

- PC users should also append `addpath .\MatMPI` to the startup file.

This should allow MATLAB to run pMatlab scripts from the home process. To address further difficulties, it would be helpful to consult the pMatlab/MatlabMPI documentation, located at http://www.ll.mit.edu/mission/isr/pmatlab/pmatlab.html

MatlabMPI sets up the necessary I/O communication processes to run MATLAB scripts on multiple computers, while pMatlab implements higher level functions to automate tasks such as data distribution and memory management. It is imperative that it is set up properly on the system before proceeding. They can be tested using the examples located in the pMatlab/examples and pMatlab/MatlabMPI/examples directory.

Testing parringpass

We provide the functions `prpTest`, `prpTestLaunch`, and `samplePlot` to simulate a test run of AT's parallel capabilities. To run these files, the user should navigate the MATLAB working directory to the /examples folder and run the `prpTestLaunch` script. If the script completes, the `samplePlot` script can be run as well to obtain a picture of the phase space plots of each tracked particle (Figure 3).

Running parringpass

The MATLAB script, `parringpass`, is designed to function as a parallelized version of AT's `ringpass` function. It should not be directly called from the command line, but rather from within the `parringpass`launch function, which specifies the settings corresponding to the communication directories and the number of threads/processors. For help setting up communications, consult both the pMatlab documentation and the MatlabMPI documentation, located within the pMatlab/MatlabMPI path.

Before the launch script is run, `parringpass` itself has a number of parameters that should be specified from within the function. This is due to the process by which pMatlab carries out its calculations. Parameter arguments cannot be directly passed into the func-

tions; they must be directly instantiated from within the script. The `parringpass` code provides example formats for declaring these variables, which should be changed according to the desired calculations:

```
% ARGUMENTS
SPEAR3; % Script which instantiates the corresponding version of the SPEAR3
  lattice.
numberParticles = 1024; % Specifies the number of particles to track
Rin = zeros(6,numberParticles); % Instantiates an array of zeros representing
  the initial coordinates
NT = 1024; % Integer specifying number of passes through ring
```

pMatlab uses distributed array computing to run its parallel processes. For this reason, arrays must be instantiated using the `zeros()`, `ones()`, or `rand()` functions, which are overloaded by the pMatlab libraries to be compatible with distributed matrix computing.

RESULTS AND DISCUSSION

Though `parringpass` has yet to be tested on a multi-CPU cluster due to configuration issues, we were able to demonstrate enormous performance upgrades by taking advantage of parallel capabilities on a multi-core system and running it in a simulated multi-CPU environment with access to four discrete nodes. The resultant computation runtimes are demonstrated (Figure 4), as well as the processor performance graphs recorded using the Windows Task Manager (Figure 5).

From the test runs utilizing a quad-core processor in a simulated distributed memory environment, we observe a speed factor increase of 3.7, correspond to an approximate efficiency of 95% per processor, strikingly close to an ideal, linear speed increase for parallel computing applications. Studies in parallel computing and computational accelerator physics have demonstrated similarly high efficiencies on high-performance supercomputing clusters, predicting processor efficiencies for `parringpass` of over 80% [2], even with communications set up between hundreds of discrete nodes.

CONCLUSIONS AND FUTURE WORK

By exploiting the efficacy of modern-day parallel computing, we were able to demonstrate incredibly efficient speed increments per processor in AT's beam-tracking functions. Extrapolating from prediction, we can expect to reduce week-long computation runtimes to less than 15 minutes. This is a huge performance improvement and has enormous implications for the future computing power of the accelerator physics group at SSRL. However, one of the downfalls of `parringpass` is its current lack of transparency; the pMatlab and MatlabMPI packages must first be well-understood by the user before the system can be configured to run the scripts. In addition, the instantiation of argument parameters requires internal modification of the source code. Thus, `parringpass`, cannot be directly run from the MATLAB command line, which detracts from its flexibility and user-friendliness. Future work in AT's parallelization will focus on development of external functions and scripts that can be called from within MATLAB and configured on multiple nodes, while expending minimal communication overhead with the integrated MATLAB library.

ACKNOWLEDGMENTS

I would first like to acknowledge the U.S. Department of Energy and the Accelerator Physics Group at the Stanford Synchrotron Radiation Lightsource (SSRL), located at the SLAC National Accelerator Laboratory for their roles in funding and hosting the program. I would especially like to thank Xiaobiao Huang for his endless support since the project's inception, and all of his efforts in making this the best summer experience of my life. Special thanks to Eric Shuphert, Shannon Ferguson, and Steve Rock for organizing the SULI program, as well as all of the SULI interns, for making the internship such a rewarding nine weeks.

REFERENCES

1. Terebilo, A. "Accelerator Toolbox for MATLAB." SLAC-PUB-8732 (2001).

2. Pelegant: A Parallel Accelerator Simulation Code for Electron Generation and Tracking Y. Wang and M. Borland, AIP Conf. Proc. 877, 241 (2006), DOI:10.1063/1.2409141

3. Barney, Blaise. "Introduction to Parallel Computing". Lawrence Livermore National Laboratory. 6/13/10 ¡https://computing.llnl.gov/tutorials/parallel_comp/¿.

4. Kepner, Jeremy. "MIT Lincoln Laboratory: MatlabMPI". Massachusetts Institute of Technology. 07/16/10 ¡http://www.ll.mit.edu/mission/isr/matlabmpi/matlabmpi.html¿.

5. Wille, Klaus. The Physics of Particle Accelerators. New York: Oxford University Press, 2000.

FIGURES

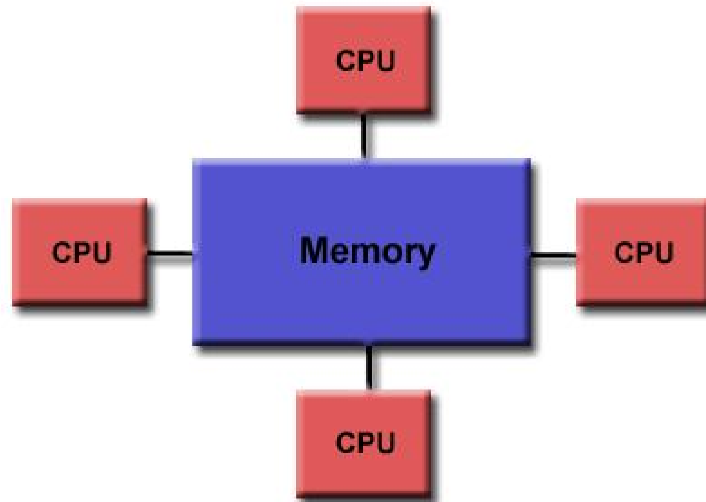

Figure 1: Shared Memory (UMA) Schematic: The image above demonstrates setup of a shared memory multiprocessing unit. All four processors run independently of one another, but have access to a universally distributed memory source. This setup will allow AT to run significantly faster even on low-cost workstations.

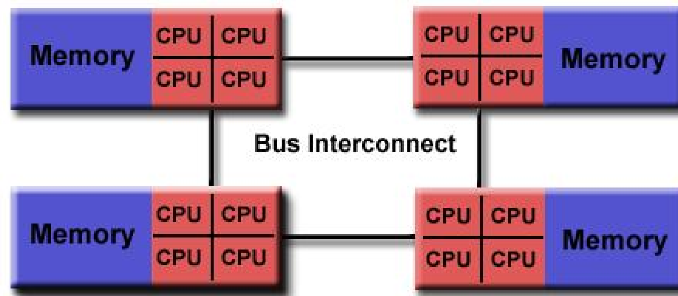

Figure 2: Distributed Memory (NUMA) Schematic: The image above demonstrates setup of a distributed memory multiprocessing unit. In this system, multiple machines run independently of one another with access only to a local memory source, but also implement connections for communications and passing of messages. By utilizing this setup, we can increase AT's processing power to over 100 times its current performance capabilities. Note that UMA systems may be embedded within a larger NUMA system.

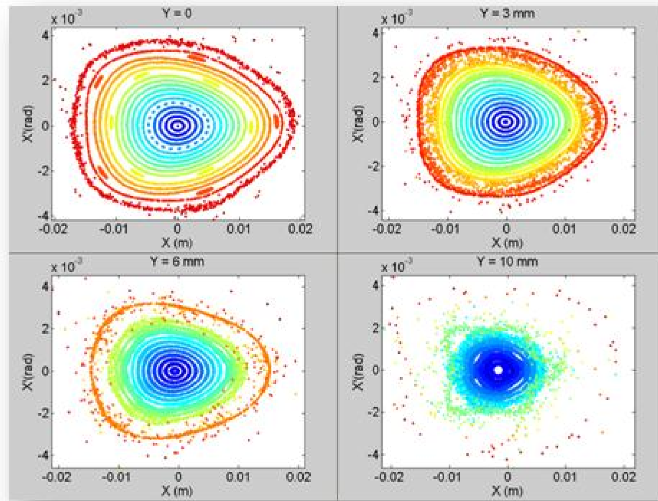

Figure 3: Phase Space Plots: The above plots show Poincare maps of particle trajectories as they are evolved through the SPEAR3 ring lattice. Each color corresponds to a discrete particle with a different set of initial conditions.

Number of Processors	Single	Dual	Quad
Computation Time (s)	*78.2974*	*41.1318*	*21.1746*
Communication Time (s)	*0.0012*	*1.4272*	*1.5732*
Total Runtime (s)	*78.2986*	*42.5590*	*22.7478*

Figure 4: Processor Runtimes: The above table demonstrates the decrease in processor times in a simulated Distributed Memory environment as the code is run on additional threads.

Figure 5: Processor Activity Recordings: These figures show the recorded processor activity on a quad-core processor in a simulated distributed-memory computing environment for one, two, and four processors, respectively. As the number of threads increases, we can note clear increases in processing activity and decreases in computing time.