Parallelizing AT with Open Multi-Processing and MPI (Postprint)

Authors: LUO Cheng-Ming, TIAN Shun-Qiang, WANG Kun, ZHANG Man-Zhou, Qing-Lei Zhang, JIANG Bo-Cheng

Date: 2023-06-18T00:00:00+00:00

Abstract

Simulating charged particle motion through the elements is necessary to understand modern particle accelerators. The particle numbers and the circling turns in a synchrotron are huge, and a simulation can be time-consuming. Open multi-processing (OpenMP) is a convenient method to speed up the computing of multi-cores for computers based on share memory model. Using message passing interface (MPI) which is based on non-uniform memory access architecture, a coarse grain parallel algorithm is set up for the Accelerator Toolbox (AT) for dynamic tracking processes. The computing speedup of the tracking process is 3.77 times with a quad-core CPU computer and the speed almost grows linearly with the number of CPU.

Full Text

Preamble

Parallelizing AT with Open Multi-Processing and MPI

LUO Cheng-Ming (罗承明)[1],[2], TIAN Shun-Qiang (田顺强)[1], WANG Kun (王坤)[1], ZHANG Man-Zhou (张满洲)[1], ZHANG Qing-Lei (张庆磊)[1],[2], and JIANG Bo-Cheng (姜伯承)[1],†

[1]Shanghai Institute of Applied Physics, Chinese Academy of Sciences, Shanghai 201800, China
[2]University of Chinese Academy of Sciences, Beijing 100049, China

(Received September 26, 2014; accepted in revised form March 5, 2015; published online June 20, 2015)

Abstract: Simulating charged particle motion through accelerator elements is essential for understanding modern particle accelerators. The enormous number of particles and circulation turns in a synchrotron make such simulations

computationally intensive. Open Multi-Processing (OpenMP) provides a convenient method to accelerate computations on multi-core computers using a shared memory model. Using the Message Passing Interface (MPI), which is based on non-uniform memory access architecture, we have implemented a coarse-grain parallel algorithm for the Accelerator Toolbox (AT) to speed up dynamic tracking processes. The tracking process achieves a speedup factor of 3.77 on a quad-core CPU computer, with performance scaling nearly linearly with the number of CPUs.

Keywords: Accelerator Toolbox, Open multi-processing, Message passing interface, Parallel computing
DOI: 10.13538/j.1001-8042/nst.26.030104

Introduction

The Accelerator Toolbox (AT) code was developed at the Stanford Synchrotron Radiation Lightsource (SSRL) to model particle accelerators and beam transport lines within the MATLAB environment [**?**]. AT enables users to develop custom functions and applications by building upon its source code, and its simulation results show excellent agreement with experimental measurements. At the Shanghai Synchrotron Radiation Facility (SSRF), AT has been utilized for several years [**?**, **?**]. Storage ring simulations, particularly dynamic aperture tracking and lattice design optimization, demand highly computation-intensive algorithms. To complete these calculations within acceptable timeframes, code efficiency must be improved. The primary performance bottleneck in AT stems from massive repeated calls to particle tracking functions that are independent of one another, making parallel computing a straightforward approach to accelerate computations [**?**].

Two common parallelization strategies exist: GPU computing and multi-core CPU multi-threading. Since AT's tracking processes involve extensive cache operations, the frequent data exchange between computer memory and GPU makes GPU computing less attractive when particle numbers are relatively small. Consequently, multi-core CPU computing represents the better option. Open Multi-Processing (OpenMP) is a standardized model for shared-memory computing that supports C/C++ compilers, achieving a 3.7× speedup on a local quad-core computer. To further enhance performance, the Message Passing Interface (MPI), based on a message-passing model, is employed on dual-CPU servers, with computational speed growing linearly with the number of computers.

II. Experimental

A. Accelerator Physics and Optimization Analysis

In AT, particle motion is modeled by representing each particle as a point in 6-dimensional phase space coordinates:

$$X = (x, p_x, y, p_y, (p - p_0)/p_0, c\tau)^T$$

where x and y are transverse coordinates, p_x and p_y are divergence angles, $(p-p_0)/p_0$ represents momentum spread, and $c\tau$ denotes longitudinal position. Evolution of the phase space point through a magnetic accelerator element can be modeled using second-order transport matrices [**?**]:

$$X_{\text{final},j} = \Delta X_j + \sum R_{jk} X_k + \sum \sum T_{jkl} X_{kX} l$$

Computing the entire process of evolving a particle through a single magnetic element thus requires numerous multiply-accumulate operations. Particles are assumed sufficiently relativistic that inter-particle interactions can be neglected, allowing the same operation to be applied to all particles in parallel [**?**]. Different accelerator elements have different transport matrices, which in AT code are calculated in various 'passmethod' functions. These functions are called millions of times during the tracking process by AT's 'ringpass' function to compute particle trajectories. Tracking multiple particles through accelerator elements constitutes a SIMD (Single Instruction Multiple Data) operation, making it well-suited for OpenMP and MPI processing [**?**].

Parallel computing employs multiple computing resources to simultaneously work on different parts of a problem [**?**], providing an efficient means to accelerate computations. Our focus is on adapting AT for parallel processing with OpenMP and MPI. OpenMP is an application programming interface (API) for multi-threaded programming in C/C++ and Fortran, offering a highly abstract description of parallel computing through compiler directives, library routines, and environment variables that affect runtime behavior [**?**]. By introducing OpenMP routines and directives into the existing AT source code, we enable AT to follow a Uniform Memory Access (UMA) model where all processor cores share the same physical memory uniformly. This requires moderate modifications to the 'passmethod' functions written in C, as OpenMP can be implemented through MATLAB's MEX compilation function [**?**].

MPI is the standard for distributed-memory models using explicit methods to control parallel computing. MatlabMPI is a MATLAB implementation of the MPI standard that enables any MATLAB program to exploit multiple processors [**?**]. We construct a Non-Uniform Memory Access (NUMA) architecture using OpenMP and MPI. In this model, multiple machines run independently with their own local memory and communicate via bus interconnect. [Figure 1: see original paper] schematically illustrates the NUMA model.

B. Parallel AT with OpenMP and MPI

As mentioned above, accelerating the passmethod functions directly increases overall computational speed. We employ OpenMP and MPI to parallelize these

functions. The approach to creating a parallel program from existing code involves identifying sections that can be processed simultaneously; the necessary code changes and compilation methods are detailed in the appendix. The parallelized portions use library functions `omp_{{{get}}_{{num}}}_{threads}()` and `omp_{{{get}}_{{thread}}}_{num}()` to obtain the number of threads in the parallel zone and the ID of the working thread (numbered from zero). The start index serves as an offset for each computing core, with different threads working on different data based on their thread ID, thereby achieving data and task parallelism.

Taking DriftPass.c [**?**], which calculates particle states after passing through a drift element, as an example, we used Intel VTune Amplifier [**?**] to test the efficiency of the parallelized version. With 3,920 particles and 500,000 loops on a quad-core computer, the results are shown in Table 1 , where CPU time represents the sum of CPU time across all threads, and overhead & spin time represents the time an active thread spends acquiring synchronous constructs. These two components constitute most of the CPU time, with overhead & spin time accounting for approximately 9.3% of total CPU time—this being the primary reason why parallel program speedup cannot reach the theoretical maximum.

The achieved speedup is 2.98.

TABLE 1. Computation time for parallel and non-parallel DriftPass

Type of computing	CPU time (s)	Elapsed time (s)	Overhead & spin time (s)
Parallel			
Non-parallel			

In this implementation, we use a general distributed memory model. Upper-level computers employ MatlabMPI to control parallel computing and communication, while lower-level computers use OpenMP for computation. Message-passing functions distribute data to lower computers, and their computational results are transferred back and combined into the final result. This pattern allows us to use the shared-memory computing interface to manage local task distribution for each CPU while enabling AT to function within a global distributed memory model, thereby avoiding data conflicts that occur when two processors attempt to access the same memory. If only the UMA model were used, such conflicts would cause longer CPU spin times, and in some cases, the data-conflict delay could make two-processor execution slower than single-processor execution.

OpenMP is used to parallelize all 'passmethod' functions called by 'ringpass', and since all passmethod codes are parallelizable, ringpass can be treated as a parallel program.

III. Results and Discussion

Frequency Map Analysis (FMA) is a method for analyzing amplitude-dependent frequency shifts within the dynamic aperture. The program flow traces particles, obtains particle output data through N turns, and applies a first-order Hamming filter to the data [**?**]. FMA serves as a frequency scanning tool to reveal information about nonlinear resonances and guide frequency optimization [**?**]. Particle tracking consumes the majority of computational time, and reprogramming AT with OpenMP significantly reduces this time, potentially saving days or weeks. [Figure 2: see original paper] shows FMA computation results using parallel and non-parallel methods with varying numbers of particles on an Intel i7-3770 CPU with 4GB RAM.

FMA execution time grows almost linearly with particle number. The non-parallel method is up to 3.16 times slower than the parallel method. According to Amdahl's law [**?**]:

$$S = [f_{\text{par}}/P + (1 - f_{\text{par}})]^{-1}$$

where f_{par} is the parallel fraction of the code, P is the speedup for the parallel part, and S is the speedup of the entire program. The profile command measures time spent in parallel and non-parallel portions of the program flow. The parallel portion is the 'ringpass' function, while the main non-parallel portion is the FMA function; remaining parts consume negligible time. Table 2 shows timing results for parallel and non-parallel programs, where N is the number of particles, T_f is the time taken by the FMA function, and T_{all} is the total program time.

TABLE 2. Time profile (in seconds) using parallel and non-parallel computing

N	Parallel computing T_f/T_{all}	Non-parallel computing T_f/T_{all}

From Table 2, the T_f/T_{all} ratio remains stable for both computing types. According to Amdahl's law, the speedup of the parallel part can be obtained from Eq. (4):

$$[(1 - 0.07)/P + 0.07]^{-1} = 3.16$$

where 0.07 represents the non-parallel fraction of the computing process. Thus, the speedup $P = 3.77$, which never exceeds 4 even with larger particle numbers. Since the CPU is quad-core, a maximum of 4 computing threads can execute simultaneously, and thread synchronization further reduces computational speed.

To leverage the speedup from OpenMP and MPI, a Dell R720 server was utilized for particle tracking in the slow extraction process of the Shanghai Proton

Therapy Synchrotron. The R720 features 2 processors with 16 cores each, acting as two compute nodes. The speedup for one node is 6.23, while for two nodes it reaches 12.18—nearly double the single-node speedup. Since the computing processes are independent and communication accounts for only about 2.3% of total runtime, the speedup of two nodes should approximately double that of one node. This demonstrates that the speedup grows linearly with the number of CPUs using OpenMP and MPI.

IV. Conclusion

This paper describes the workings of AT and demonstrates how OpenMP and MPI can be used to parallelize such programs. The parallelized AT achieves significantly faster computation. For other accelerator physics programs that can be parallelized, OpenMP and MPI can be applied similarly. This approach is convenient to implement and achieves speedup close to the theoretical limit for a single computer or cluster. With additional compute nodes, larger problems can be solved.

Appendix

```
#include <omp.h>

... some computation and initialization

Omp_{{{set}}_{{num}}}_{threads}(4)
#pragma omp parallel private(i) shared(start_{index}, n)
thread_{id} = omp_{{{get}}_{{thread}}}_{num}();
num_{threads} = omp_{{{get}}_{{num}}}_{threads}();
start = start_{index} + n * thread_{id} / num_{threads};
if (thread_{id} == num_{threads} - 1) end = n - 1;
end = n * (thread_{num} + 1) / num_{threads} - 1;
for (i = start; i <= end; i++) {
    ... computation
}
```

References

[1] Terebilo A. Accelerator toolbox for MATLAB. SLAC-PUB-8732, 2001.
[2] Tian S Q, Jiang B C, Zhou Q G. Lattice design and optimization of the SSRF storage ring with super-bends. Nucl Sci Tech, 2014 25: 010102. DOI: 10.13538/j.1001-8042/nst.25.010102
[3] Jiang B C, Liu G M, Zhao Z T. Simulation of a transverse feedback system for the SSRF storage ring. High Energ Phys Nucl, 2007, 31: 956–961.
[4] Yan X Y, Zhang W W, Bu S H. Parallel optimization of three-dimension particle simulation based on mixed MPI/OpenMP Programming. Journal of South China University of Technology (Natural Science Edition), 2012, 40:

71–78. DOI:10.3969/j.issn.1000-565X.2012.04.011
[5] Grote H, Iselin F C. The MAD Program (Methodical Accelerator Design) Version 8.13/8 User's Reference Manual. Geneva, Switzerland. Jan. 18, 1994.
[6] Appleby R, Bailey D, Higham J, et al. High performance stream computing for particle beam transport simulations. J Phys Conf Ser, 2008, 119: 042001. DOI: 10.1088/1742-6596/119/4/042001
[7] Chen G L, Sun G Z, Xu Y, et al. Integrated research of parallel computing: Status and future. Chinese Sci Bull, 2009, 54: 1845–1853. DOI: 10.1007/s11434-009-0261-9
[8] Dagum L and Menon R. OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng, 1998, 5: 46–55. DOI: 10.1109/99.660313
[9] Zhang Y. Solving large-scale linear programs by interior-point methods under the Matlab Environment. Optim Method Softw, 1998, 10: 1–31. DOI: 10.1080/10556789808805699
[10] Kepner J. Parallel programming with MatlabMPI. arXiv: astro-ph/0107406
[11] Marowka A. On performance analysis of a multithreaded application parallelized by different programming models using intel Vtune. Lect Notes Comput Sc, 2011, 6873: 317–331. DOI: 10.1007/978-3-642-23178-0_{28}
[12] Laskar J. Frequency map analysis and particle accelerators. IEEE Part Acc Conf, 2003, 1: 378–382. DOI: 10.1109/-PAC.2003.1288929
[13] Tian S Q, Liu G M, Li H H, et al. Tune optimization of the third generation light source storage ring based on Frequency Map Analysis. Chinese Phys C, 2009, 33: 224–. DOI: 10.1088/1674-1137/33/3/012
[14] Chandra R, Dagum L, Kohr D, et al. Parallel programming in OpenMP. San Francisco (USA): Morgan Kaufmann Publishers, 2001, 16–17.

Note: Figure translations are in progress. See original paper for figures.