Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code

To cite this article: G Marin et al 2008 J. Phys.: Conf. Ser. 125 012087

View the article online for updates and enhancements.

Related content
- Harnessing the power of emerging petascale platforms
  John Mellor-Crummey
- Optical observations of PSR J2021+3651 with the GTC
  E B Ryspaeva, A Yu Kirichenko, Yu A Shibanov et al.
- Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms
  S Ether, W M Tang and Z Lin

Recent citations
- Endong Wang et al
- Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms
  Kamesh Madduri et al
- Karl Fuerlinger et al
Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code

Gabriel Marin, Guohua Jin and John Mellor-Crummey
Department of Computer Science, Rice University, Houston, TX 77005, USA
E-mail: {mgabi,jin,johnmc}@cs.rice.edu

Abstract. Achieving high performance with grand challenge applications on today's large-scale parallel systems requires tailoring applications for the characteristics of the modern microprocessor architectures. As part of the US Department of Energy's Scientific Discovery through Advanced Computing (SciDAC) program, we studied and tuned the Gyrokinetic Toroidal Code (GTC), a particle-in-cell code for simulating turbulent transport of particles and energy in burning plasma, developed at Princeton Plasma Physics Laboratory. In this paper, we present a performance study of the application that revealed several opportunities for improving performance by enhancing its data locality. We tuned GTC by performing three kinds of transformations: static data structure reorganization to improve spatial locality, loop nest restructuring for better temporal locality, and dynamic data reordering at run-time to enhance both spatial and temporal reuse. Experimental results show that these changes improve execution time by more than 20% on large parallel systems, including a Cray XT4.

1. Introduction
As part of the application engagement efforts in the SciDAC Center for Scalable Application Development Software and the Performance Engineering Research Institute, at Rice University we have undertaken analysis and tuning of the Gyrokinetic Toroidal Code (GTC). GTC [0] is a particle-in-cell (PIC) code for simulations of turbulent transport in fusion plasmas such as the International Thermonuclear Experimental Reactor (ITER). GTC is the production code of two DOE SciDAC projects: GPS-TTBP and GSEP. An allocation of 8M processor-hours has been awarded for 2008 to explore GTC applications on the Cray XT4 computer at Oak Ridge National Laboratory.

During each time step, the PIC algorithm repeatedly executes charge, solve, and push operations. In the charge step, it deposits the charge from each particle onto grid points nearby. The solve step computes the electrostatic potential and field at each grid point by solving the Poisson equation on the grid. In the push step, the force on each particle is computed from the potential at nearby grid points. Particles move according to the forces on them. The version of GTC that we studied uses a domain decomposition along the toroidal direction. Each toroidal domain including one or more poloidal planes is assigned to a separate MPI process. Particles move across the boundary poloidal planes through MPI communications. Figure 1(a) shows one-eighth of the donut-shaped torus arrows representing ion movement. Figure 1(b) shows a poloidal plane consisting of rings equally distant from each other (in the radial direction) and uniformly distributed grid cells along each of the rings (in the θ direction).
In our investigation, we used both modeling and measurement to identify and understand opportunities for improving GTC’s performance. We used modeling [0] to understand the principal data reuse patterns in GTC and to identify opportunities for enhancing reuse in cache. We observed that performance degrades as the position of ions in the plasma evolves during the simulation. To alleviate this problem, we periodically reorder particles so that particles nearby in the simulated tokamak are close in memory. We present a mathematical model for computing the optimal sorting interval and an adaptive algorithm that uses this model to control sorting at run-time. We also modified GTC’s communication code to reduce the rate at which locality degrades. This transformation has two benefits: it reduces the need for particle reordering, and it diminishes the performance loss due to locality degradation between each pair of sorting steps.

2. Single-node performance improvements
Tuning the performance of parallel applications requires not only achieving good scalability as the number of processors increases but also maximizing the application’s performance on each individual node. Elsewhere [0], we present in greater detail the process of analyzing and tuning the single-node performance of GTC. For brevity, in this paper we present only a summary of our code transformations for improving data locality in GTC.

- **Static data structure reorganization.** We transposed the ion arrays; this transformation improves spatial reuse and eliminates about half of the cache misses for these arrays.

- **Loop fusion for charge deposition.** Our analysis identified that about 11% of all L3 cache misses are due to reuse of data between two loops that iterate over all particles in the charge deposition phase. By fusing these loops, we eliminated these cache misses.

- **Loop tiling and fusion for particle pushing.** During each time step, the velocities of ions are updated based on the electrical field. This calculation was performed in several loops. Our analysis identified that reuse between these loops accounts for about 20% of all L3 cache misses in the program. Fusing these loops and iterating over subsets of ions improved temporal reuse of the ion data in this phase.

- **Loop interchange for field smoothing.** In the calculation of the electrical field, a single loop nest was contributing a high fraction of TLB misses. Applying loop interchange eliminated all extraneous TLB misses in this phase.

3. Particle reordering
During each time step, each particle deposits charge onto cells related to the particle’s position in space, a potential field is computed using the charge distribution among the cells, and each particle’s position is updated after it is pushed by the resulting electric field. While the potential field computation involves only the cell data structures, the charge deposition and particle position updates involve interactions between particles and cells. For these phases, the best
locality is achieved when all particles in a cell are processed together. At program start, particles are sorted in cell order. However, as the simulation progresses there is a gradual loss of locality as particles migrate from one cell to another, and even to different processors.

Figure 2 shows scatter plots of the particle positions along the radial and $\theta$ directions at the time steps 0 and 20. The data was collected from a single processor in a four-processor simulation. Figures 2(a) and (b) show the initial radial positions of all particles and $\theta$ positions for the first four poloidal rings. Figures 2(c) and (d) show the particle positions at time step 20.

By storing particles in cell order, we can improve data locality in GTC beyond what is achievable using only the data structure and loop transformations described in Section 2. Because performance degrades only gradually over time and since sorting the particles after every time step is expensive, it is preferable to sort the particles only once in a while. The next section describes a mathematical model for computing the optimal sorting interval, $\tau$, based on the cost of sorting and the rate at which performance degrades.

3.1. Understanding the optimal sorting interval

Let $f(x)$ be the function that describes how performance degrades over time. Thus, $f(x)$ represents the execution cost of the $x$th time step since the latest sorting step. We assume that performance per time step returns to the same base level after each sorting step, and that afterwards it degrades in accordance with function $f(x)$. We want to define $G(t)$, the benefit function for sorting at every $t$ time steps and then determine $t = \tau$ for which $G(t)$ is maximized.

Let $C$ be the cost of one sorting step. The benefit function over $N$ time steps when we perform sorting every $t$ time steps can be written as follows.

$$G(t) = \sum_{k=1}^{N-1} \left( \int_{kt}^{(k+1)t} f(x) \, dx - \int_0^t f(x) \, dx - C \right) = \int_0^N f(x) \, dx - \frac{N}{t} \int_0^t f(x) \, dx - \frac{NC}{t} + C \quad (1)$$

Note that $\int_0^N f(x) \, dx$ and $C$ are independent of $t$, and thus we can remove them. Function $G(t)$ is maximized when the equation: $\frac{N}{t} \int_0^t f(x) \, dx + \frac{NC}{t}$ is minimized. The total number of time steps, $N$, is a constant and we can divide the equation by $N$. The problem reduces again to finding the value of $t$ for which the following equation is minimized:

$$h(t) = \frac{1}{t} \left( \int_0^t f(x) \, dx + C \right) \quad (2)$$

This result is not surprising. It tells us that the optimal sorting interval is found when the average execution cost per time step of the program with sorting every $t$ time steps is minimized.

3.2. An adaptive particle sorting algorithm

We could compute the optimal sorting interval analytically if we knew the function $f(x)$ that describes the performance degradation. We also could approximate $f(x)$ by applying regression
to data collected from previous runs. However, we would have to repeat this process for every target architecture, each time we modify the code, and potentially for each program input.

Alternatively, we can compute the optimal sorting interval adaptively at run time. Based on equation 2, we need to know $C$ – the cost of a sorting step, and the cost of the first $t$ time steps – $e(t) = \int_0^t f(x) \, dx$, which we can compute iteratively at run-time. After each time step we update $e(t)$ and evaluate $h(t) = e(t) + C \cdot t$. We stop when the value of $h(t)$ starts to grow, which gives us the value of $t$ for which $h(t)$ is minimized. We added an adaptive sorting algorithm to GTC based on the above observations. The initialization phase of the sorting algorithm executes the following four steps algorithm along the normal execution of the program:

**Step 0** Compute $C$, the execution cost of a sorting step. We let the program run for a fixed number of time steps, after which we invoke our particle reordering routine measuring its execution cost. In our implementation we benchmark the cost of sorting on the tenth time step of the program. The actual initialization phase starts on Step 1.

**Step 1** Evaluate $h(t)$. After each time step we compute the value of $e(t)$ incrementally, evaluate $h(t)$, and update its minimum encountered value, $h_{\text{min}}$, as needed.

**Step 2** Compute a local optimum. If the most recent value of $h(t) > h_{\text{min}}$, we have a candidate value for the local optimal sorting interval: $\tau_{\text{local}} = t - 1$.

**Step 3** Compute a global optimum. For SPMD programs, such as GTC, all processes execute in interlocked steps. In order to achieve good performance, it is essential for all processes to perform particle reordering at the same time. The largest optimal interval across all processes is selected using a reduction operation. We use the maximum value over all processes because from our experience it is more costly to sort excessively than not to sort often enough.

**Step 4** Provide confidence in the global optimum. In practice, the time per time step may fluctuate slightly from one time step to another, and we may find several local minima for $h(t)$ only a few time steps apart. For this reason, we do not stop the initialization algorithm when we find the first minima. Instead, if $\tau$ is our global sorting interval, we continue executing steps 1–3 for $\tau$ more time steps. If any process finds a new local optimum, the new optimal interval will be agreed upon by all processes in Step 3 of the algorithm, and the value of $\tau$ itself will increase in this way. However, if we do not find another minimum during the next $\tau$ steps, we are confident that the interval we found is optimal.

### 3.3. A new particle migration algorithm

In our study of GTC, we found that the strategy for migrating particles between poloidal planes contributed to the rate of locality loss observed for parallel runs. During execution, gaps in the particle array are created as particles migrate between processors. These gaps were being filled with particles from the end of the array, before newly received particles were appended at the end. Using particles from the end of the array that initially reside on the outermost rings of a plane to fill gaps in other rings accelerates the rate with which particles become unsorted.

We expect the number of particles sent and received by any local domain to be approximately equal, since the particle density should be rather uniform along the toroidal dimension. Moreover, we expect the number of particles sent and received through each subsection of a poloidal plane to be approximately equal. With these observations, and since particles are initially sorted in cell order, we modified the particle migration algorithm to fill the gaps in the particle array with newly received particles. Any leftover particles are appended at the end of the array, and any unfilled gaps are filled with particles from the end of the array.
4. Results

We evaluated our optimizations on three parallel systems: a Cray XT4 system at the Lawrence Berkeley National Laboratory with one 2.6 GHz dual-core AMD Opteron processor and 4 GBytes of memory per node; a Cray XD-1 machine with two dual-core 2.2 GHz AMD Opteron CPUs and 8 GBytes of memory per node; and an Intel Itanium 2 cluster with two 900 MHz processors and 4 GBytes of memory per node. The last two clusters are located at Rice University. We compared up to eight versions of the GTC codes on the three parallel systems, with three types of optimizations turned on and off. In naming the versions, “Orig” represents the original version of the code, “Opt” includes the loop and data restructuring optimization, “M” includes the better migration approach, and “S” represents the adaptive sorting. Execution times were collected. The programs are compiled with the Fortran and C compilers from the Portland group with optimizations “-fast -fastsse” on the Cray machines, and with the Intel Fortran and C compilers with optimization “-fast” on the Intel Itanium cluster.

Figure 3(a) presents the single-node performance results of three optimized versions of the GTC codes on the Cray XD1 and Intel Itanium 2. All timing results are normalized to the execution time of the original version. Overall, the version with both optimizations applied achieves the best performance with a 27% reduction of execution time on the Itanium2 cluster.

Figures 3(b) and (c) compare the parallel execution time of all eight versions of the code. In figure 3(b), the optimized version “OptMS” with all three optimizations applied achieved the best performance with about 37% reduction of overall execution time on the Itanium2 cluster. The loop and data restructuring optimization contributed significantly to the overall improvement on the Itanium 2 cluster. In figure 3(c), the best version of the code with all three optimizations applied achieves approximately a 21% execution time reduction against the original version on both Cray machines.

5. Conclusions

Using today’s microprocessor-based leadership computing platforms most effectively requires careful structuring of codes to make the most effective use of their multi-level memory hierarchies by exploiting both spatial and temporal reuse of data. For complex data-intensive scientific codes, exploiting locality is easier said than done. Significantly improving the performance of the Gyrokinetic Toroidal Code required comprehensive changes, including transposition of particle-centric data structures to improve spatial reuse, restructuring of procedures and loop nests to improve temporal reuse, adjustment of data movement associated with communication to avoid unnecessarily introducing disorder in the plasma, and periodically sorting particles by their coordinates to enhance spatial and temporal reuse. By paying careful attention to all of these factors affecting locality, we were able to reduce GTC’s execution time on Opteron-based
parallel systems by just over 20% and on Itanium2-based systems by approximately 37%.

References

[1] Lin Z, Hahm T S, Lee W W, Tang W M and White R B 1998 Turbulent transport reduction by zonal flows: massively parallel simulations Science 281(5384):1835–1837

[2] Marin G and Mellor-Crummey J 2008 Pinpointing and exploiting opportunities for enhancing data reuse Proc. 2008 IEEE Intl. Symposium on Performance Analysis of Systems and Software