Tuning space optimization for stencil-based applications on multi-core

Víctor Martínez, Philippe Navaux
Informatics Institute (INF)
Federal University of Rio Grande do Sul (UFRGS),
Av. Bento Gonçalves, 9500, Campus do Vale,
91501-970, Porto Alegre, Brazil
{victor.martinez,navaux}@inf.ufrgs.br

Fabrice Dupros, Hideo Aochi
BRGM, 3 Av. Claude Guillemin, Orléans, France
{f.dupros,h.aochi}@brgm.fr

Márcio Castro
Department of Informatics and Statistics (INE)
Federal University of Santa Catarina (UFSC),
Campus Reitor João David Ferreira Lima, Trindade,
88040-970, Florianópolis, Brazil
marcio.castro@ufsc.br

Abstract

Stencil computations are fundamental to solve Partial Differential Equations (PDEs) used in numerical simulations. However, their performance is very sensitive to many optimization parameters such as architectural features, compiler flags, memory contention and multi-threading. Fine-tuning these parameters relies on finding the best configuration that results in optimal performance for a given stencil application. In this work, we improve performance of OpenMP stencil programs on multicore architectures through tuning input parameters. We create a large configuration set of input values (problem size, threads number, chunk size, scheduling algorithm), until we reach the peak of best performance, and analyze how these parameters affect the performance.

1. Introduction

The performance of HPC applications depends on many factors: architecture, code optimization, compiler and runtime frameworks. An example of HPC applications are stencil-based applications (nearest-neighbor), which are used to solve many problems related to Partial Differential Equations (PDE). Then, optimizing these computations improve performance in many simulations.

One extended methodology to get best performance is application tuning, in which several parameters are adjusted to achieve the best performance. But it depends on several variables of a large set: machine architecture, parallelization strategy, domain decomposition, compiler flags, scheduling and load balancing algorithms at runtime, etc. Finding best space of input parameters requires to search on this large set of configurations. In [3] the authors optimize stencil computations for multiple architectures (multicore and accelerators).

Our work is oriented to obtain the best performance on multicore architectures for stencil computations, that are implemented in diverse areas as electromagnetics, fluid dynamics and wave propagation (i.e., seismic simulations). This paper presents our approach to find best tuning space of input parameters for stencil computations. The paper is organized as follows. Section 2 discusses the fundamentals of our stencil based model on single seven-point Jacobi. Then, Section 3 presents the testbed used and explains the methodology of experiments. Section 4 discusses the performance of simulations and how we can obtain the best performance. Finally, Section 5 concludes this paper.

2. Stencil Model

2.1. Stencil Equation

The stencil model is given by the explicit 3D heat equation [4]:

$$\begin{aligned} B_{i,j,k} =& \alpha A_{i,j,k} \\ &+ \beta(A_{i-1,j,k} + A_{i,j-1,k} + A_{i,j,k-1} \\ &+ A_{i+1,j,k} + A_{i,j+1,k} + A_{i,j,k+1}) \end{aligned} \quad (1)$$

This seven-point stencil performs a single Jacobi (out-of-place) iteration. Thus, reads and writes occur in two distinct arrays (A, B), where each subscript represent the 3D index into array A or B. For each grid point, this stencil will execute 8 floating point operations [3]. The stencil sweep can be expressed as a triply nested loop presented in algorithm 1.

Algorithm 1 Pseudocode for stencil algorithm

```
1: for each timestep do
2:     Compute in parallel
3:     for each block in X-direction do
4:         for each block in Y-direction do
5:             for each block in Z-direction do
6:                 Compute stencil(3D tile)
7:             end for
8:         end for
9:     end for
10: end for
```

3. Experimental Setup

In this section we present the configurations considered in this work: stencil algorithms, the multicore architectures and the input and performance vectors.

3.1. Stencil algorithms

In order to obtain best performance we used three algorithms implemented in [5] with optimizations explained in [1].

3.1.1. Naive: First algorithm is the standard implementation of the triple nested loops coming from the three spatial dimensions. This allows a very straightforward use of OpenMP directives. Unfortunately, this standard implementation offers a poor cache reuse.

3.1.2. Blocking: Second algorithm uses the cache blocking technique. The main idea is to exploit the inherent data reuse available in the triple nested loop of the elastodynamic kernel by ensuring that data remains in cache across multiple uses. Dependency between the velocity and the stress components is exploited to implement a space-time decomposition.

3.1.3. Skew: Third algorithm is also based on cache misses reduction. These new approaches decompose the stencil using both the space and the time directions. Indeed, the spacetime domain is computed in a specific order, which means that the computations begin with the subdomains closest to the left boundary and then extend to the right boundary to honor the data dependencies [2].

The advantage on Blocking and Skew algorithms is to improve the computational intensity by keeping a relatively small amount of data in cache memory and by performing many more floating-point operations on them.

3.2. Experimental testbed

We used two multicore architectures to run our experiments: one machine with a single Intel Haswell processor (single socket) to avoid cache transferences and one NUMA platform composed of 4 Intel Nehalem processors. Configurations of our testbed are listed in Table 1.

	Node 1	*Node 2*
Processor	i5-4570	Xeon X7550
Clock (GHz)	3.20	2.0
Cores	4	8
Sockets	1	4
Threads	4	64
L3 cache size (MB)	6	18
Compiler	gcc-5.3.1	gcc-4.6.4

Table 1: Experimental testbed configurations.

3.3. Input and performance space

For each stencil experiment, we created one configuration vector and obtained one performance vector. Input parameters are related to code optimization and execution runtime. Fore code optimization we use one parameter for parallel looping in OpenMP (`omp task` or `omp parallel for`). For the runtime, we used two parameters for thread counting (total available threads and used threads), one parameter for the chunk size and one parameter for scheduling policy used by OpenMP (static, guided and dynamic). Each one of input configurations were executed 15 times to compute the averages. We used PAPI library [6] to measure cache-related information.

The performance measures are obtained with following values: Total cache misses L3 (`PAPI_L3_TCM` event),Total

cache access L3 (`PAPI_L3_TCA` event), Time (which corresponds to the total execution time to solve the stencil), Performance in GFLOPS (obtained from the execution time and stencil size).

4. Experimental Results

4.1. Algorithms

We first analyze the impacts of algorithms on the performance. Each algorithm presents a different performance for each machine. Figure 1 shows that best performance on node 1 is achieved with Skew algorithm (left) whereas on node 2 the Naive algorithm achieves the best performance. Blocking and Skew algorithms showed performance losses.

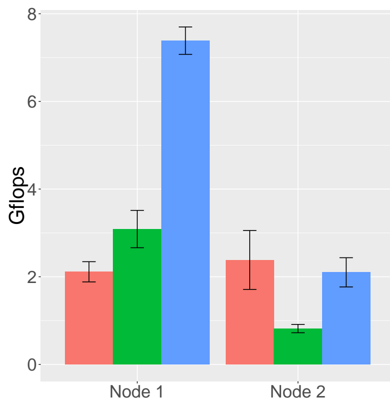

Figure 1: Performance (left) for each stencil algorithm: Naive (magenta), Blocking (green) and Skew(cyan).

As it can be observed in Figure 1, we obtained the expected behavior on node 2: better performance is quite related to the amount of L3 cache misses, as we can see low rate of cache misses is presented with high Gflops values and poor performance has high number of cache misses. We found that Skew and Blocking algorithm reduces cache misses significantly as it was reported in [5].

4.2. Scalability

We now analyze the scalability of the algorithms on node 2. The results presented in Figure 2 (left) show an expected behavior: when the number of threads is increased the performance of Naive algorithm also increases.

The Skew algorithm showed limited scalability on node 2, reaching a peak performance with 8 threads. The Blocking algorithm presented poor scalability on both platforms. We analyze this unexpected behavior with the other input parameters.

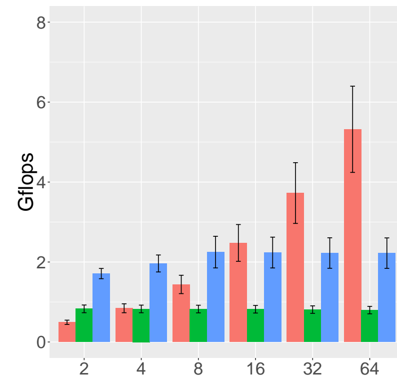

Figure 2: Performance for different number of threads on node 2. Algorithms: Naive (magenta), Blocking (green) and Skew(cyan).

4.3. Code optimization: parallel loop vs tasking

Our 3D stencil is calculated by three `for` instructions, as explained in Section 2. In this section, we compare two possible parallel implementations: i) we collapsed all `for` loops and performed the parallelization with `#pragma omp for`; and (ii) we used `#pragma omp task` in the inner `for` to create parallel tasks. Figure 3 presents the results for each node using the maximum number of available threads in each platform.

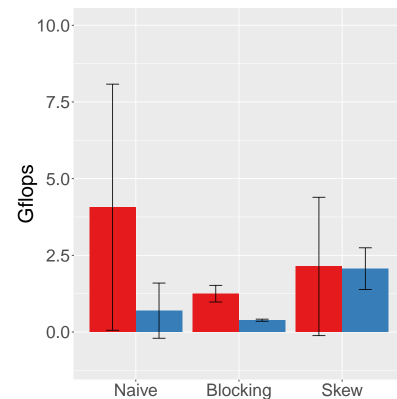

Figure 3: Performance for code optimization on Node 2. Method: parallefor (red) and Tasking (blue)

As it can be observed, the `parallel for` implementation achieved better performance than tasking in most of cases. The main reason is twofold: it creates a lot of tasks that need to access more cache levels and threads have to synchronize with each other in the `taskwait` clause. To

confirm this fact we traced the execution with pajeNG[1] and found that the OpenMP implicit barrier takes more time to synchronize all threads.

4.4. Scheduling

Loop scheduling on OpenMP is defined by two parameters: chunk size and policy [7]. Chunk size defines number of loop iterations to be assigned to each thread. Now, we analyze how this parameter influences the overall performance. For small chunk size we have more data communication between threads. We analyze scheduling on Node 2 (more cores, more threads, more cache levels and communications).

Figure 4 presents results of changing chunk size and scheduling for Naive algorithm. Three strategies are available in OpenMP: dynamic, guided and static. Then we found that the scheduling policy does not affect the performance for guided scheduling, while dynamic and static achieve better performance when the chunk size is increased.

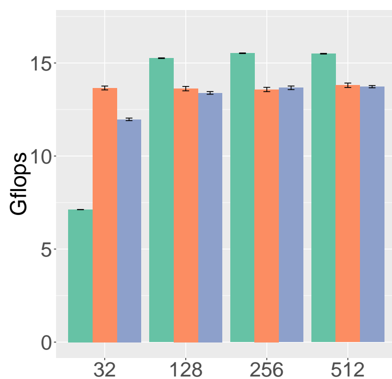

Figure 4: Performance for different scheduling with `omp parallel for` on Node 2: Dynamic (green), Guided (orange) and Static (gray). Algorithms: Blocking (left), Naive (center), Skew (right).

5. Conclusions and Future Work

In this work we studied the influence of several configurations and algorithms on the performance of stencil computations. We observed that two known algorithms (Blocking and Skew) may present poor scalability in several scenarios. Moreover, we observed that tasking achieves good performance when the algorithm does not use cache intensively (Skew). Chunk size and scheduling algorithms play an important role and can contribute to achieve a peak of performance if threads do not perform intensive data communications. As a future work, we intend to develop an auto-tuning approach to automatize the choice of the input parameters. One possibility is to use Machine Learning algorithms to perform that task.

6. Acknowledgments

This work have been granted by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and Bureau de Recherches Géologiques et minières (Institut Carnot BRGM). Research has received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E Project, grant agreement n° 689772.

References

[1] C. Andreolli. Eight optimizations for 3-dimensional finite difference (3dfd) code with an isotropic (iso). `https://software.intel.com/en-us/articles/openmp-loop-scheduling`. Accessed: 2016-01-01.

[2] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] K. Datta, S. W. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. *Scientific Computing with Multicore and Accelerators*, chapter Auto-Tuning Stencil Computations on Multicore and Accelerators. CRC Press, Taylor & Francis Group, 2010.

[5] F. Dupros, F. Boulahya, H. Aochi, and P. Thierry. Communication-avoiding seismic numerical kernels on multicore processors. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 330–335, Aug 2015.

[6] ICL. Papi reference. `http://icl.cs.utk.edu/projects/papi/wiki/Main_Page`. Accessed: 2016-01-01.

[7] Intel. Openmp loop scheduling. `https://software.intel.com/en-us/articles/openmp-loop-scheduling`. Accessed: 2016-01-01.

1 https://github.com/schnorr/pajeng