Study and Optimization of a Finite Volume Application

Brito, Rui
PG22781
Department of Informatics
University of Minho
ruibrito666@gmail.com

Alves, José
PG22765
Department of Informatics
University of Minho
zealves.080@gmail.com

1 Introduction

This study presents an analysis of the application conv-diff. This analysis consists in the application profile, the use of several optimization techniques and the discussion of its results. The program conv-diff calculates the heat transfer through an area using a Finite Volume method.

The goal of this study is to research different optimization techniques and to achieve an improvement in the application performance.

The study was divided in several stages. In the first stage a profile of the application was created, identifying its troublesome areas and improving the sequential version. A second stage dedicated to a shared-memory version using OpenMP. The third stage to work with a GPU version, using its massive parallelism capabilities, was made in CUDA. In the fourth stage a naive distributed-memory version was researched using MPI. Finally in the fifth and final stage one or more methods of the previous stages were to chosen and improved upon. Based in our previous results the progression to an improved OpenMp version seemed obvious and was pursued.

In this extended abstract the contents are presented according to the stages of the project. A brief section explaining the case study followed by its profile will be presented first. With this sections providing a thorough analysis of the program, a string foundation is created to develop the optimizations. From here, the optimized versions are presented starting with the sequential version, and followed by the OpenMP and CUDA versions. After a brief explanation of the MPI version a more extended section is presented introducing the final optimization. A final section for a cocnlusion of the project is also presented.

2 Case Study

The application analyzed for this study is *conv-diff(Convection-Diffusion)*.This application simulates the way heat is transferred in a fluid using the finite-volumes method.To compute the heat diffusion, the surface is represented as a mesh. Being represented by cells and edges, the algorithm will traverse all edges, calculating the contribution of the adjacent cells. This application rests in a Finite Volume Library (FVLib), which handles the structures and some of the logic functions necessary for the problem's solution.

The application's main objective is to compute a vector $\overline{\phi}$ such that $\overline{\phi} \longrightarrow G(\overline{\phi}) = \begin{pmatrix} 0 \\ 0 \\ \vdots \end{pmatrix}$ This is accomplished in three different stages:

1. We begin with a candidate vector ϕ
2. For each edge, we compute the flux F_{ij}, with i and j being the indexes of the adjacent cells
3. For each cell, we compute $\sum |e_{ij}|F_{ij} - |c_i|f_i$

Thus: $\phi = \begin{pmatrix} \phi_1 \\ \vdots \\ \phi_I \end{pmatrix} \longrightarrow G = \begin{pmatrix} G_1 \\ \vdots \\ G_I \end{pmatrix}$

The output can be converted to msh format, which is compatible with gmsh, for following data visualization(fig. 2).

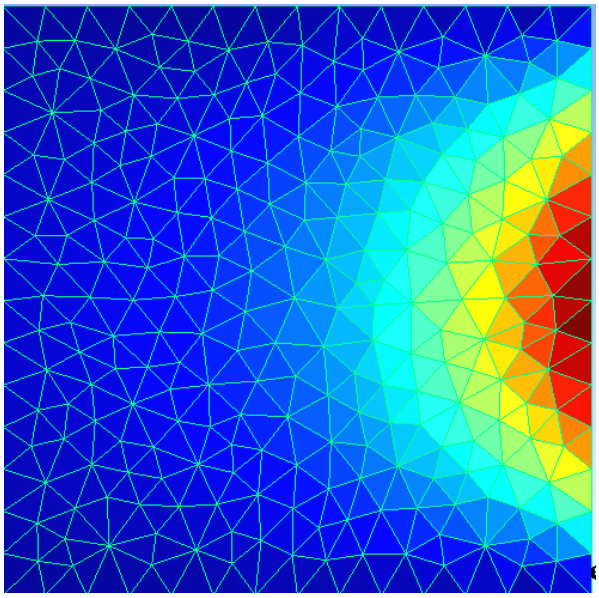

Figure 1: Output example.

3 Naïve Optimization

The program consists of four major parts, reading the initial mesh from a file. Then, using the functions *makeResidual*, which calls the function *makeFlux*, the flux contributions are calculated and the vector phi is built, thus achieving a matrix free implementation. Following this, to calculate the deviation in the results from the previous operations, the function *LUFactorize*. Finally, both the meshes are written to the output files, together with the error between them.

After analyzing the application, we conclude that the algorithm has a very high workload in the *LUFactorize* function, comprising of more than 90% of the execution time. This is mostly because LUFactorize is a matrix implementation of the algorithm presented in the previous section. While it provides accurate results, its memory usage, for example, makes it unusable for big meshes. As an example, a mesh with more than fifty thousand cells will easily consume more than 10 GB of RAM.

After dissecting the code and understanding the problem at hand, we began to notice several implementation errors, these errors, such as reading the same variable repeatedly from a file and long chains of calculation with a heavy division at the end, were easy to spot, and could clearly been avoided. We changed all those trivial aspects of the application, which required minimal effort. That being said, this simple optimizations paid results. The computation time has been greatly reduced, with the aforementioned *LUFactorize* function taking an even more prominent role in our profile. Since *LUFactorize* is very time-consuming and could overshadow the part of the program we were focusing on, we removed it.

Below is table of results we got from PAPI, displaying the difference between the original version and the best sequential version. These results show the improvements our small changes got. By not computing the same value in every iteration of the loop and making those values constant, we greatly reduced the number of loads and stores and, consequently, the number of cache accesses. This is because the *const* keyword tells the compiler that a particular variable will not change during runtime.

	original version	optimized sequential version
Total instructions	2.517.584	285.551
Load instructions	630.156	86.532
Store instructions	326.459	39.208
FP operations	55.673	44.019
L1 data accesses	1.061.761	153.593
L2 data accesses	22.914	17.467

Table 1: PAPI comparison

4 Shared Memory Parallel Optimization(OpenMP)

After optimizing the sequential code, we turned our efforts to parallelizing the code. The two loops responsible for the matrix free calculations were ideal candidates. We parallelized both this loops. We had some struggles with data-races in these, but we overcame the problems rather easily. The data-races exist because the mesh is traversed by the edges, however, as we found out, if they are traversed by cell, these data-races no longer exist. Also, the library that was provided includes some iterator style structures. These were also a problem, because, while OpenMP as no problem in paralleliz-ing STL iterators, this doesn't hold for *FVlib*'s iterators. So, we had to convert those to a standard for loop. The code was successfully parallelized, however, results were disappointing, execution time didn't decrease at all, hinting at a memory bound application.

Below is a graph detailing the results we got for the parallel portion of the program. We ran this test on an Intel Xeon 5650, which has 12 physical cores and HyperThreading. As this processor has HT it got us wondering if it would scale any better in a processor with 24 physical cores, so we also ran the tests

on an AMD Opteron 6174. Also, as most of the optimizations were made in a personal machine, we included those results as well. The faster times can be explained the the use of a recent compiler. The processor is an Intel Ivy Bridge i7 (2.3 GHz) with 4 physical cores. The slower times on the AMD machine can be explained by bad space locality. One of the strengths of the AMD processor is it's big cache, however, logically, if the program as bad locality, it will perform worse than in it's Intel counterpart. As it can be seen, the application doesn't scale at all as expected, this is because of the high number of memory accesses. Also, the deep indirection chain in *FVLib* plays it's part, with every structure being a collection of pointers to other structures.

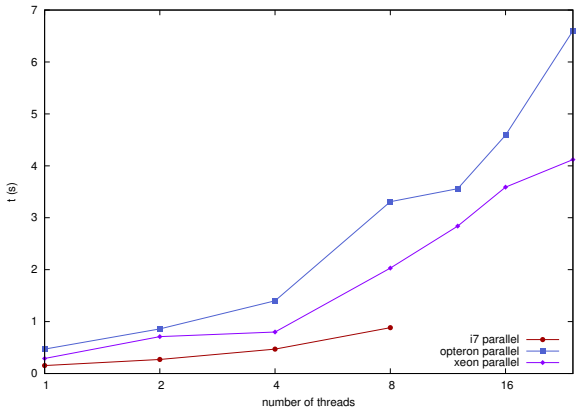

Figure 2: Scalability of the parallel region

Below is a plot of the total execution times.

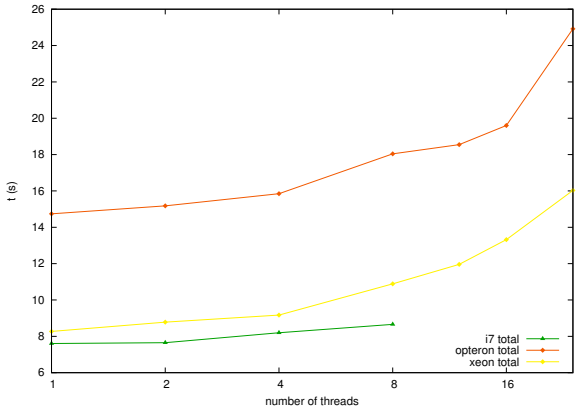

Figure 3: Total execution time

5 Distributed Memory Optimization(MPI)

Another approach we tried for optimizing the code was an MPI version. The strategy was to have a master process coordinating some slave processes. Each slave process computed a pre-assigned part of the mesh, an then, the results were colected by the master thread. Below is an illustration of what would typically happen.

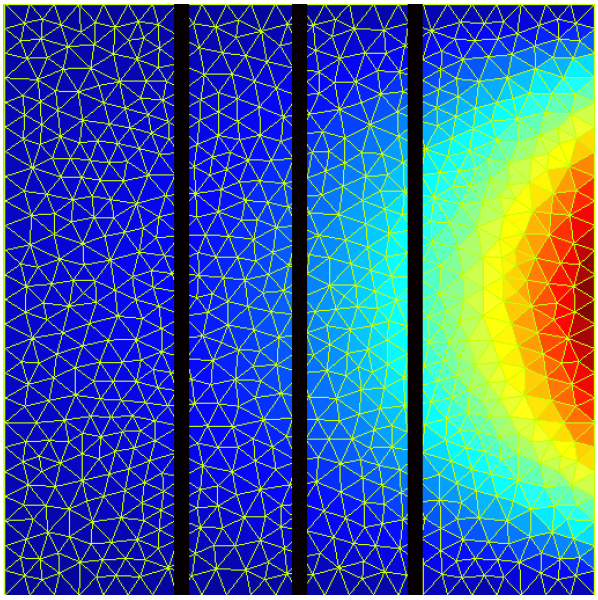

Figure 4: Total execution time

As it can be seen, the way we partitioned the mesh, is, at best, problematic. Since the fluxes are computed from right to left, one can easily see the high level of barrier synchronization and communication needed where the mesh was partitioned. Another problem with this implementation is because of the way the mesh is stored. Since we don't know where a particular cell is, another level of synchronization arises, because we can't compute the flux of a cell in a section, before the previous section is computed. Some of FVLib's templates were hard to serialize and some balancing problems were also encountered. This version proved to be very problematic, causing some error spikes, almost the final result.

Below are two plots portraying both the speedups achieved and the communication rate.

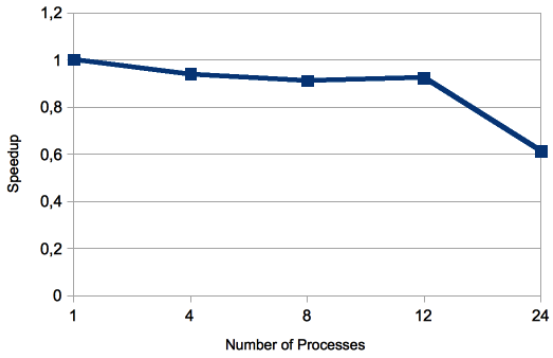

Figure 5: Total execution time

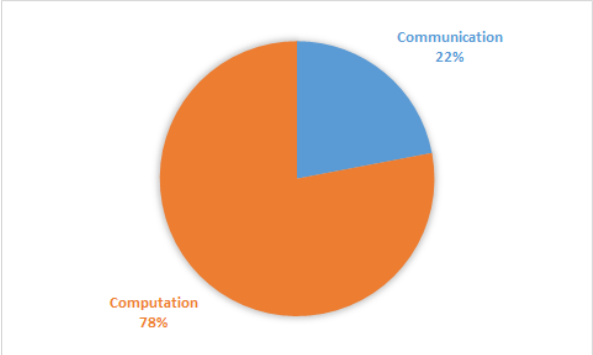

Figure 6: Total execution time

6 Conclusion

This extended abstract serves as a introduction to the study here presented. The initial results from the implementations of optimized versions, sequential and parallel, shows small improvements in the computed time. Future iterations of the solutions increase the improvements.

Through the development of the solutions some problems were presented, such as the mesh being disperse and the structures implemented with extensive use of pointers. This problems delayed the development of solutions. The decision of maintaining the abstraction of the system while favorable in terms of comprehension, proves punishing in terms of performance.

In the future released paper a deep analysis of the results will be made, showing the performance improvements obtained.