Cross-platform development technologies

MEI: Topics of Software Development, Universidade do Minho

1st David Machado
University of Minho
Braga, Portugal
pg54468@alunos.uminho.pt

2nd José Ferreira
University of Minho
Braga, Portugal
pg52690@alunos.uminho.pt

3rd Simão Quintela
University of Minho
Braga, Portugal
pg52257@alunos.uminho.pt

***Abstract*—In this document we present our findings in the attempt to optimise a single threaded program execution time. The challenge was proposed by our professors in the course of Parallel Computing and the program provided was the simulation of a *3d fluid*.**

***Index Terms*—performance, optimization, timing**

I. Introduction

This document describes the development of the Work Assignment - Phase 1 for the Parallel Computing course. In this project, the aim was to use instruction-level optimization techniques applied to a single-threaded program and code analysis/profiling tools.

II. Report Organization

The report is organized the following way. First, we explain how to project works and its use. Then, we show the profiling made to the code. We exhibit the graph generated and comment about possible optimizations to the code and the respective improvement. Also, if the improvement is not what we were expecting we try to explain the reason for that.

III. Original Code

A. Analysis

In this first phase, we aimed to understand how the code works. After a detailed analysis of the code, we realized that it was simulating the dynamics of a fluid. After understanding what it was about, we decided to run the program in its initial state and measure its execution time and other parameters.

#CC	#I	CPI	Time (s)
34,637,883,423	15,639,701,788	2.22	12.1235

TABLE I
Initial Code Profiling

After obtaining this data, we continue the profiling process and generate the call graph to understand which functions are taking the longest to execute. We found out that the critical function was *lin_solve* so we started working on that function.

IV. Optimizations

In the list below we've listed all the optimizations we've carried out in function *lin_solve*, explaining them in detail later on.

- Multiplication for the inverse
- Temporal and Spatial Locality
- Index calculation
- Paralelize the sum of the neighbors
- Temporal and Spatial Locality in other functions

A. Multiplication for the inverse

In the inner cycle, in every iteration, we were doing a division, which is computationally heavy, by a constant. We decided to create a variable outside which is the inverse of the constant and then multiply that new variable inside the loop instead of dividing.

B. Temporal and Spatial Locality

We also found that the locality of the program was not being well explored. To optimize the function in that fields we changed the order of the loops. Instead of going through i, j, k, we changed it to k, j, i. This optimization has advantages because the array is organized in the following order: This

i...i+M, j, k	i, j...j+N, k	i, j, k...k+O

makes caching more efficient because the values we are accessing are closer to each other.

C. Index calculation

In the inner cycle, in every iteration, we were calculating indexes using the macro provided. Those operations were generating a lot of instructions so we destructored the macro in rows and slices. We calculate the first i index which is I(1,j,k) and to access the next i values we sum 1 to that index. To access j and k we created variables slice and row that have the value M+2 and row*(N+2), respectively. With this optimization, we reduced the number of instructions.

D. Paralelize the sum of the neighbors

When it comes to calculating the new x[I(i,j,k)] we have one big problem which is a data dependence. To calculate the new value on position i,j,k we need to know the result of the previous iteration, which makes it hard to vectorize the code.

The solution we implemented was to delay as much as possible this dependence. So, we split the inner loop into 2 sequential loops, and in the first loop, we operate the neighbors without the dependent one which makes this section parallelizable. After this loop, we add the operation that has the dependence to every missing position.

E. Temporal and Spatial Locality in other functions

After optimizing the lin_solve function, we looked for other small optimizations that could be made and we found out that locality was not well explored in functions *project* and *advect*. We applied the same process as explained before.

V. Results

In this section, we will display what effect each optimization had on our code. In order to better understand the values we splitted the data into two tables the first one having the information regarding the number of cycles, instructions, CPI and time.
The latter having the amount of

Optimization	#CC	#I	CPI	Time (s)
A	24,998,266,516	17,445,968,190	1.42	8.7546
B	19,399,215,120	14,559,359,179	1.33	6.7986
C	19,263,985,072	14,346,497,126	1.35	6.7521
D	12,178,818,309	10,220,469,264	1.19	4.2764
E	10,330,790,720	10,155,434,690	1.02	3.6306

TABLE II
Initial Code Profiling

Optimization	#L1_Cache_Misses	%Cache hits	Time (s)
A	2,599,217,137	34.41	8.7546
B	2,600,109,628	34.42	6.7986
C	472,817,147	7.44	6.7521
D	474,667,954	13.13	4.2764
E	221,099,807	6.13	3.6306

TABLE III
Initial Code Profiling

VI. Conclusion

In conclusion, we are happy about the overall optimizations we made to the code. It was possible to reduce the overall execution time in 70%, because of all the changes previously mentioned.
However we believe it's still possible to further reduce the overall execution time, for multiple reasons. First of all, the **CPI** is still quite high, knowing that the processing unit could reach up to 0.25 (in best the scenario), having 1.02 is still not optimal. The reason is the way we approach the **loop dependence** present of the inner loop of the function **lin_solve**.

We also explored a tiling/blocking strategy to leverage better data **locality**. However, the results were not as favorable as anticipated. Although this approach reduced memory access, it led to an increase in both the number of clock cycles (#CC) and instructions (#I), offsetting the potential gains from improved memory efficiency.