Deliverable

1. Introduction

2. Task Decomposition Strategies

2.1 Analysis of task granularities and dependencies

*Analysis of task granularities and dependencies.*

*Explain the original task dependence graph that is obtained as well as the new graphs that you obtained for Jacobi and Gauss-Seidel, reasoning about the causes of the data dependencies that appear and how you will protect them in your parallel OpenMP code. Include the timelines with the simulated execution for 4 threads to support your explanations about potential scalability.*

For the deliverable: Include the task dependency graph shown by Tareador. Is there any parallelism that can be exploited at this granularity level?

For the deliverable: Include the excerpt of the code that you have modify in order to specify

one task per block.



For the deliverable: Which variable was causing the serialisation of all the tasks? Are you

obtaining more parallelism? How will you protect the access to this variable in your OpenMP implementation? Simulate the execution when using 4 threads and extract your conclusions. Is there any other part of the code that can also be parallelised?. If so, modify again the instrumentation to parallelise it.

Variable sum is provoking this parallelisation in both cases, Gauss and Jacobi.

3. Implementation in OpenMP and performance analysis

3.1 OpenMP parallelization and execution analysis: Jacobi

3.1.1 First Implementation

*Describe how did you implement in OpenMP the proposed data decomposition strategy for the heat equation when applying the Jacobi solver, commenting how did you address any detected performance bottleneck (serialisation, load balancing, ...). You should include the output generated by modelfactors.py and any captures of Paraver windows to justify your explanations. You should also analyse the speed–up (strong scalability) results that have been obtained for the different numbers of threads, reasoning about the performance that is obtained.*

For the deliverable: Is the execution time of the OpenMP version of heat using 8 threads reduced compare to the sequential execution or execution using 1 thread? if not, you should reconsider your implementaton. For instance, what kind of synchronization are you using? Review different strategies to avoid and/or reduce the amount of sychronization overheads per interation. Include an excerpt of the code to show the OpenMP annotations you have added to the code.



1 thread jacobi (heat-omp-jacobi-1-boada-11.txt):

Iterations : 25000

Resolution : 254

Residual : 0.000050

Solver : 0 (Jacobi)

Num. Heat sources : 2

1: (0.00, 0.00) 1.00 2.50

2: (0.50, 1.00) 1.00 2.50

Time: 2.247

Flops and Flops per second: (11.182 GFlop => 4975.90 MFlop/s)

Convergence to residual=0.000050: 15756 iterations

8 threads jacobi (heat-omp-jacobi-8-boada-11.txt):

Iterations : 25000

Resolution : 254

Residual : 0.000050

Solver : 0 (Jacobi)

Num. Heat sources : 2

1: (0.00, 0.00) 1.00 2.50

2: (0.50, 1.00) 1.00 2.50

Time: 2.238

Flops and Flops per second: (11.182 GFlop => 4995.31 MFlop/s)

Convergence to residual=0.000050: 15756 iterations

3.1.2 Overall Analysis

For the deliverable: Include the Modelfactor tables. Is the scalability that is obtained with this initial parallelisation appropriate? Which is the metric reported by modelfactors.py that you should

address first?

(overall\_scalability.csv):



3.1.3 Detailed Analysis

For the deliverable: Include the window timelines or paraver Hints that you consider necessary. What is the region of the code that is provoking the low value for the parallel fraction in your parallelisation?

Overhead, shown in image (yellow).

3.1.4 Optimization

For the deliverable: Include an excerpt of the code to show the OpenMP annotations you have added to the code.

Fem que també es paral·lelitzi.



Overall Analysis of the Optimized Code

For the deliverable: Include the Modelfactor tables and the plot of scalability. Is the execution time reduced?. Have you increased the scalability? Is the parallel fraction larger than before?

Detailed Analysis of the Optimized Code

For the deliverable: Include the window timelines or paraver Hints that you consider necessary. Has the execution time for the invocations to function solve changed? Why the new code that you have parallelised makes the difference in the performance results? Reason about the scalability that is obtained.

3.2 OpenMP parallelization and execution analysis: Gauss-Seidel

*Describe how did you enforce the dependences that appear when using the Gauss-Seidel solver, focussing on the mechanism used to guarantee the proper synchronization between implicit tasks. Analyse the speed–up (strong scalability) plot that has been obtained for the different numbers of threads, reasoning about the performance that is obtained, discussing the results returned by modelfactors.py and including captures of Paraver windows to justify your explanations. Finally include your conclusions about the optimum value for the ratio computation/synchronization in the parallelization of this solver for 16 threads.*

3.2.1 First Implementation

3.2.2 Overall Analysis

3.2.3 Detailed Analysis

For the deliverable: Include the Modelfactor tables, the plot of scalability, and the window timelines or paraver Hints that you consider necessary. Is the scalability observed appropriate? Is there any metric reported by modelfactors.py that you should further investigate? Do you think we can increase the parallelism?

3.2.4 Optimization

For the deliverable: Include an excerpt of the code to show the modifications done. Reason why changing the number of blocks in the j dimension changes the ratio between computation and synchronization.

Number of blocks tune

For the deliverable: Include the plots obtained with submit-userparam-omp.sh and explain the plot that is obtained. Also, include the scalability plot and explain the performance obtained.

3.3 Optionals

*If you have done the optional part in this laboratory assignment, please include and comment in your*

*report what have you done, the relevant portions of the code, performance plots, relevant information*

*provided by modelfactors.py or Paraver windows that have been obtained.*

Optional 1

Implement the Jacobi solver using explicit tasks and following an iterative task decomposition. Compare the performance results that are obtained with the ones obtained with the data decomposition strategy. Perform an overall analysis using Modelfactors and analyse the execution time of the explicit tasks and solve instances using paraver.

Optional 2

Implement the Gauss-Seidel solver using explicit tasks and task dependences, following an iterative task decomposition. Compare the performance results that are obtained with the ones obtained with the data decomposition strategy.

4. Conclusions

5. Final Survey

*We would like to get some feedback from you so that we can continue improving the practical part of the course. In particular, we are interested in your opinion about the usage of modelfactors. Can you please tell us briefly your opinion about it? Was modelfactors useful for you in order to understand the performance of your parallel application? Would you advice its use in the laboratory assignments of this course in future editions?*

*From 0 to 10, how would you rate:*

* *modelfactors;*
* *Tareador;*
* *Extrae + Paraver.*

*Feel free to include any other comment that you want to add about the practical sessions.*