PARAL·LELISME: LABORATORI 5

# 

Pol Casacuberta Gil i Salvador Grimalt BInimelis

Quadrimestre 1, curs 22-23

Paral·lelisme, grup 21

[**1 Introduction**](#_rw87h54t1c2c) **2**

[**2 Sequential heat diffusion program and analysis with Tareador**](#_xwwfhmbsp568) **3**

[**3 Parallelisation of the heat equation solvers**](#_b4fns9bns7pi) **8**

[3.1 Jacobi solver](#_fzk71gvncf22) 8

[3.2 Gauss-Seidel solver](#_baftncotycd0) 13

[**Optional 1:**](#_8hdfxeumr7ek) **20**

[**Optional 2:**](#_gpjuuu76r3ve) **23**

[**4 Conclusion**](#_p6oez2z87mcr) **26**

# 1 Introduction

For the first session we have been introduced to the use of a heat diffusion algorithm and its two solvers, Jacobi and Gauss-Seidel, to study their parallelization.

First of starting any task, we readed and tried to understand the codes “heat.c” and “solver.c” because we thought it’s important to know accurately how the algorithms work to achieve a decent parallelization of its operations.

The code of heat.c consists in a gateway to choose the solver we want to execute. Using a switch, when we introduce a 0 as a parametre we’ll use the Jacobi solver and if we introduce a 1 we are going to use the Gauss-Seider solver.

It also gives some data about those solvers’ execution about their results. That data will only be obtained in the execution of the while(1) loop, the copy\_mat and solve functions are being executed in ended. This will only happen when two conditions are met: the return value of the matrix is lower than the param.residual value and the number of iterations (that references to the number of iterations of the while(1) loop) reaches the param.maxiter (by default, 25000).

These values can be changed using some specific arguments.

The configuration file test.dat was responsible for specifying the number of heat sources on the 2D solid.

The implementation of “solver.c” was quite trickier and we will not try to understand the mathematics behind the heat diffusion simulation. We will investigate the copy\_mat function and the solve function.

The copy\_mat function’s purpose is to copy a matrix into another. In order to do this, it traverses the matrices in blocks, analogically to the one used in solve when first storing the calculated values of the heat diffusion simulation.

First, we have to determine the number of blocks the code is going to generate to go through the matrices, using the two outermost “for” loops. The integer variables nblocksi and nblocksj set the division to 8 i blocks by 1 j block, which means that, at first, a matrix will be cut into what we can call 8 stripes.

Now, we have two new variables, i\_start and i\_end. They use logical propositions for specifical results. The formula used to calculate the starting value of a block is:

lowerb(id, p, n) (id \* (n/p) + (id < (n%p) ? id : n%p) )

So, i\_start is the result of dividing the number of blocks by the size of the matrix plus the minimum value between the number of blocks and the residue of the aforementioned division. In conclusion, it calculates the first position of the i block.

The same can be applied to i\_end, which marks the end of the given i block traverses each iteration -only now it gives the last value of the block:

upperb(id, p, n) ( lowerb(id, p, n) + (n/p) + (id < (n%p)) -1 )

It basically uses the i\_start value as a stepping stone (lowerb(id, p, n)) and adds the size of the i block to it. Before the last block has been reached, the id < (n%p) conditional statement will return a 1 (true), effectively canceling out the - 1. Otherwise, it returns 0 (false), and the total value i\_end will be decremented by one.

It can be inferred that the j\_start and j\_end values in the second outermost loop (regarding the 1 j block each i block contained) follow the same pattern.

The copy\_mat function is only executed when the Jacobi solver is called upon, because it needs to copy the results from the solve calculation to the result matrix. That is the only observable difference between the sequential implementations of both solvers —the use of an auxiliary matrix in Jacobi versus the direct rewriting of the original matrix as the computations are completed in Gauss-Seidel.

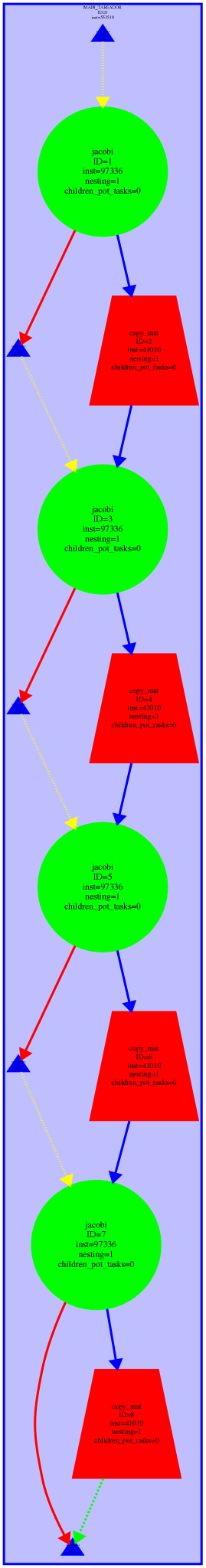
Both solvers use the solve function, which traverses the result matrix in the same way as copy\_mat. The Laboratory Assignment’s instructions regard irrelevant the mathematical properties of the calculations of the heat diffusion simulation —for that reason, we will not explain them in this Report.

Once compiled and executed (see commands in Annex II) selecting the Jacobi solver option (-a 0) we obtain a set of data of its performance. It includes the execution time in seconds, the number of floating point operations (Flops), the average number of Flops per second, the residual value and the number of iterations performed before reaching said residual value.

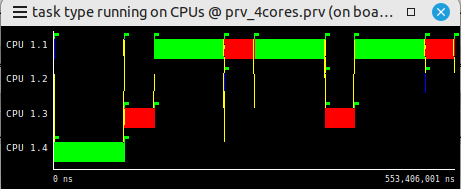
Being that we have not changed the default values of most of the parameters set by the algorithm, the execution runs with a maximum of 25,000 iterations, a resolution of 254 and residual value of 0.000050.

# 2 Sequential heat diffusion program and analysis with Tareador

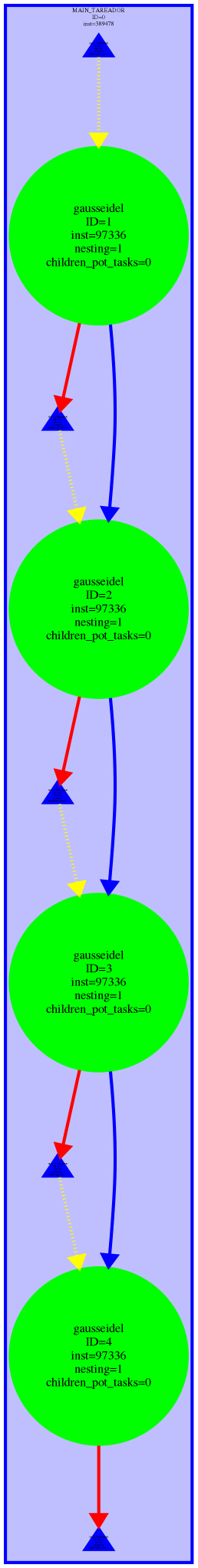
For the deliverable: Include the task dependency graph shown by Tareador. Is there any

parallelism that can be exploited at this granularity level?

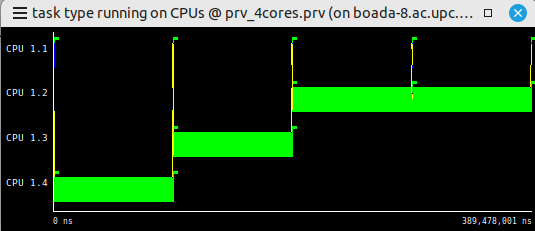
Jacobi



We can see from both the dependancy graph and from the paraver window that there isn’t any parallelism to be exploited at this current granularity level. with the Jacobi method



Gauss-Seidel



Now we take a look at the Gauss-Seidel method, which we also cannot parallelize 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.*

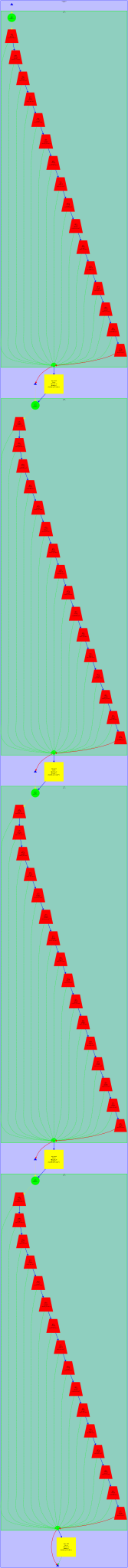
First, we have to understand the code to know how to specify one task per block. Sizex: number of rows, sizey: number of columns.

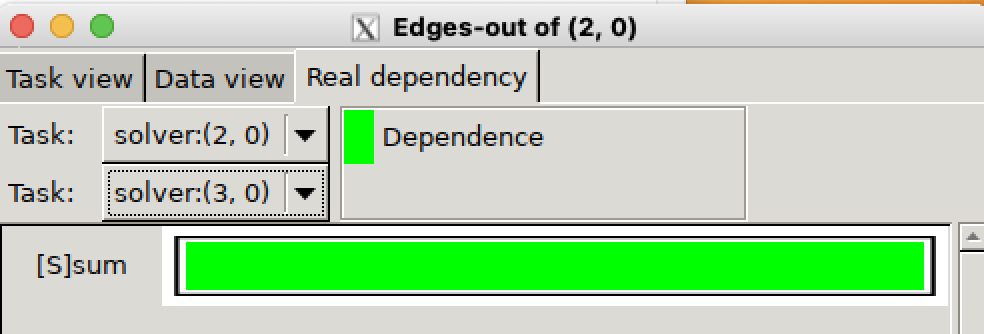
If we take a look at the lowerb function, what it is actually doing is just:

lowerb: blocki \* nblocksi/ sizex + min(blocksi, nblocksi%sizex)

in other words, it calculated the first position of the i block.

As for the upperb: basically it uses the lowerb or “i\_start” as a starting point, and then it adds the size of the i block to it and if id < (n%p) evaluates to true, then it cancels out the -1. Otherwise, it returned 0 and the total value i\_end would be decremented by one.



****

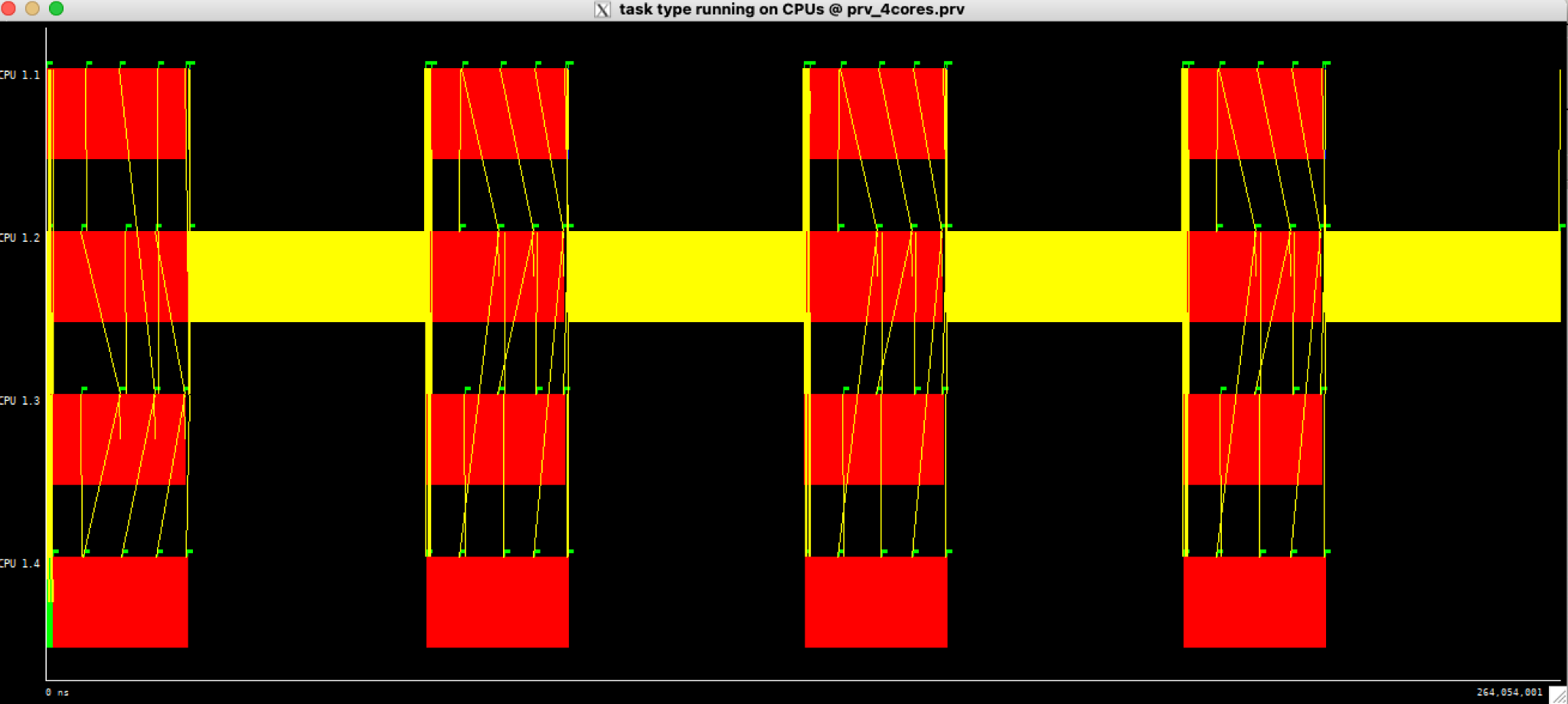
*For the deliverable: Which variable was causing the serialisation of all the tasks? Are you*

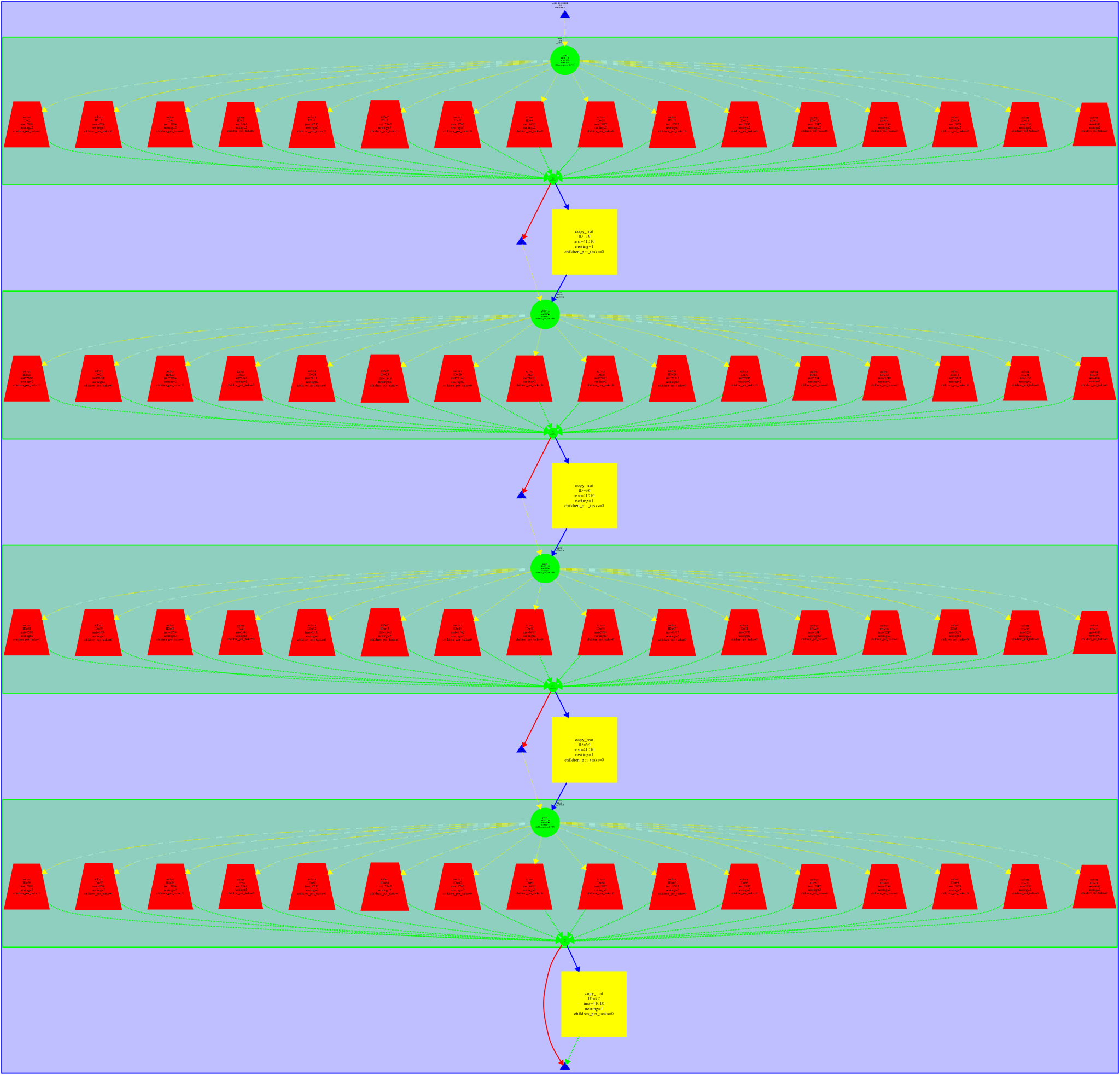
*obtaining more parallelism? How will you protect the access to this variable in your OpenMP im-plementation? Simulate the execution when using 4 threads and extract your conclusions. Is there any other part of the code that can also be parallelised?. If so, modify again the instrumentation to parallelise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

There is clearly one problem that arises from trying to parallelise by block, and that is the sum variable, which is preventing us from fully parallelizing our application. We will tackle that problem by performing a reduction on that particular variable (reduction(+, sum)).

We are now getting some actual parallelism going on, which has to do with the sum variable that we were able to parallelise properly.

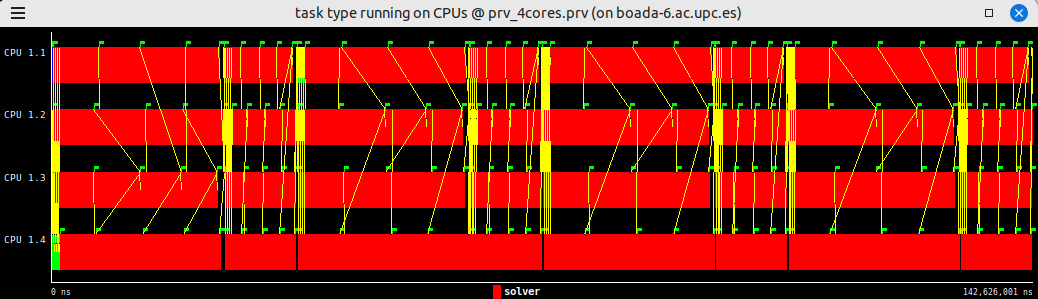
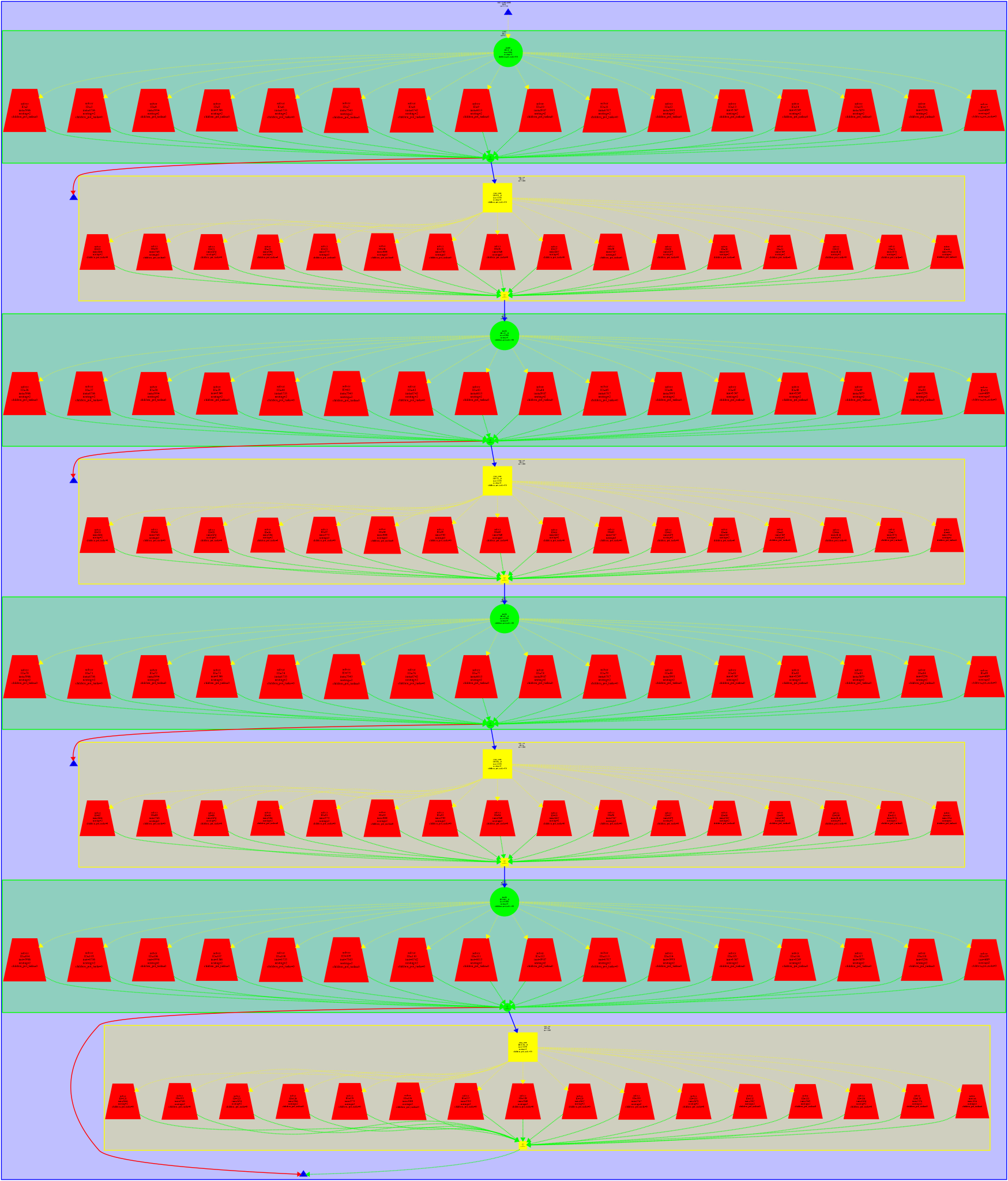
There are some parts of the code that are still not yet parallelised

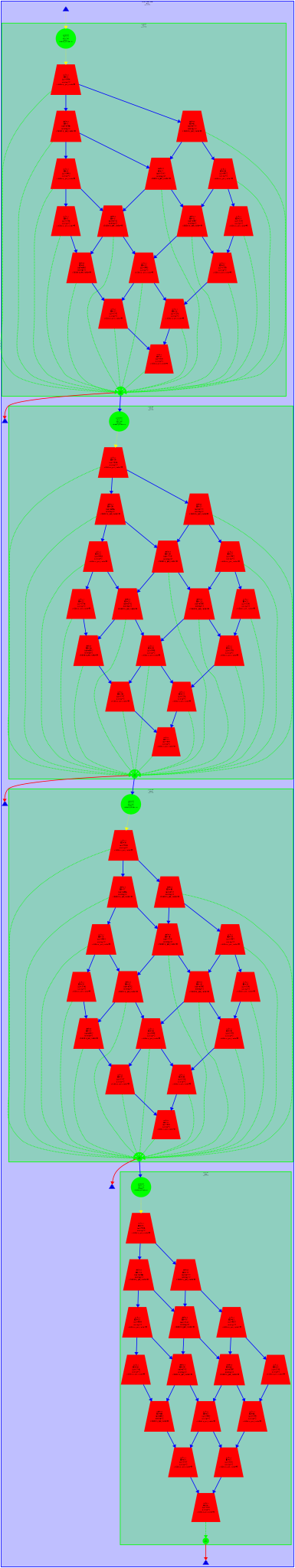


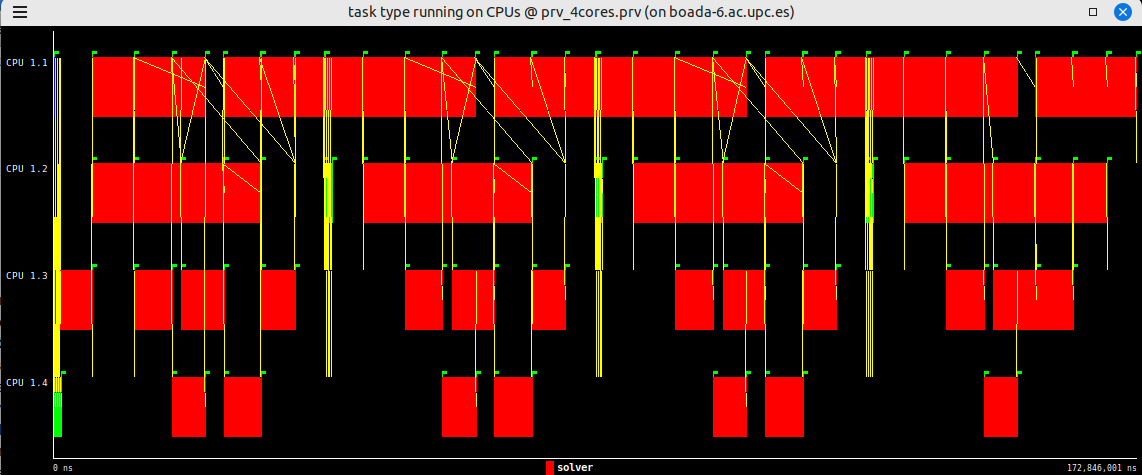


Knowing that Jacobi uses two functions solve, and copy\_mat we could apply the same decomposition to try and parallelise further our application.

This is the result:



Now we see that the code is much, much more parallel than it was before.

Now we take a look at the fully parallel Gauss algorithm with both solve and copy\_mat functions both parallelized.

The Gauss-Seidel solver was not as easily parallelizable as Jacobi, which we already expected. Gauss-Seidel does not use an auxiliary matrix and instead rewrites the original given matrix as it goes on, which obviously takes up less space at the cost of some synchronization and dependency issues. We have seen some writing dependency in diff. The parallelization goes from 1 to 2 to 3 to 4 processors and then it goes down to 3, 2 and 1 parallel task. This is due to how we traverse the matrix.

These dependencies are caused because tmp depends on the matrix of values and some of those values might have been recalculated by the time it gets to load them (In the Jacobi this would not happen as unew points to a different matrix, so there would not be an overlap between the values we need to calculate with and the results of the calculation). So we will need to implement a synchronization object that ensures that we do not read values that have been modified when performing the

# 3 Parallelisation of the heat equation solvers

## 3.1 Jacobi solver

*3.1.1*

*FirstImplementation*

*For the deliverable: Include an excerpt of the code to show the OpenMP annotations you have added to the code. Is the execution time reduced compare to the sequential execution? (if not, you should reconsider your implementaton).*

Jacobi time: 2.219

Jacobi sequential time: 4.492

We performed a diff command on both the sequential Jacobi generated image and the parallel one, and they were exactly the same (no output in the diff command). In this parallel version, the time was halved from 4.5 s to 2.2 s.

| double solve (double \*u, double \*unew, unsigned sizex, unsigned sizey) {  double tmp, diff, sum=0.0;   int nblocksi=omp\_get\_max\_threads();  int nblocksj=1;   #pragma omp parallel reduction(+:sum) private(tmp,diff) firstprivate(nblocksi, nblocksj)  // complete data sharing constructs here  {  int blocki = omp\_get\_thread\_num();  int i\_start = lowerb(blocki, nblocksi, sizex);  int i\_end = upperb(blocki, nblocksi, sizex);  for (int blockj=0; blockj<nblocksj; ++blockj) {  int j\_start = lowerb(blockj, nblocksj, sizey);  int j\_end = upperb(blockj, nblocksj, sizey);  for (int i=max(1, i\_start); i<=min(sizex-2, i\_end); i++) {  for (int j=max(1, j\_start); j<=min(sizey-2, j\_end); j++) {  tmp = 0.25 \* ( u[ i\*sizey + (j-1) ] + // left  u[ i\*sizey + (j+1) ] + // right  u[ (i-1)\*sizey + j ] + // top  u[ (i+1)\*sizey + j ] ); // bottom  diff = tmp - u[i\*sizey+ j];  sum += diff \* diff;  unew[i\*sizey+j] = tmp;  }  }  }  }   return sum; } |
| --- |

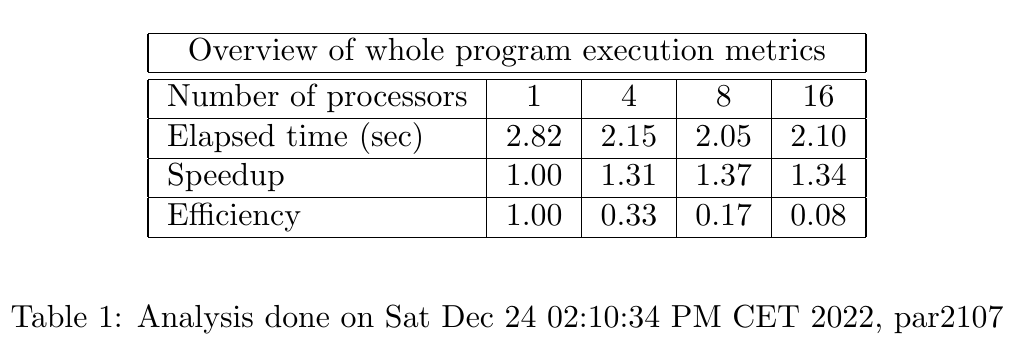
*3.1.2*

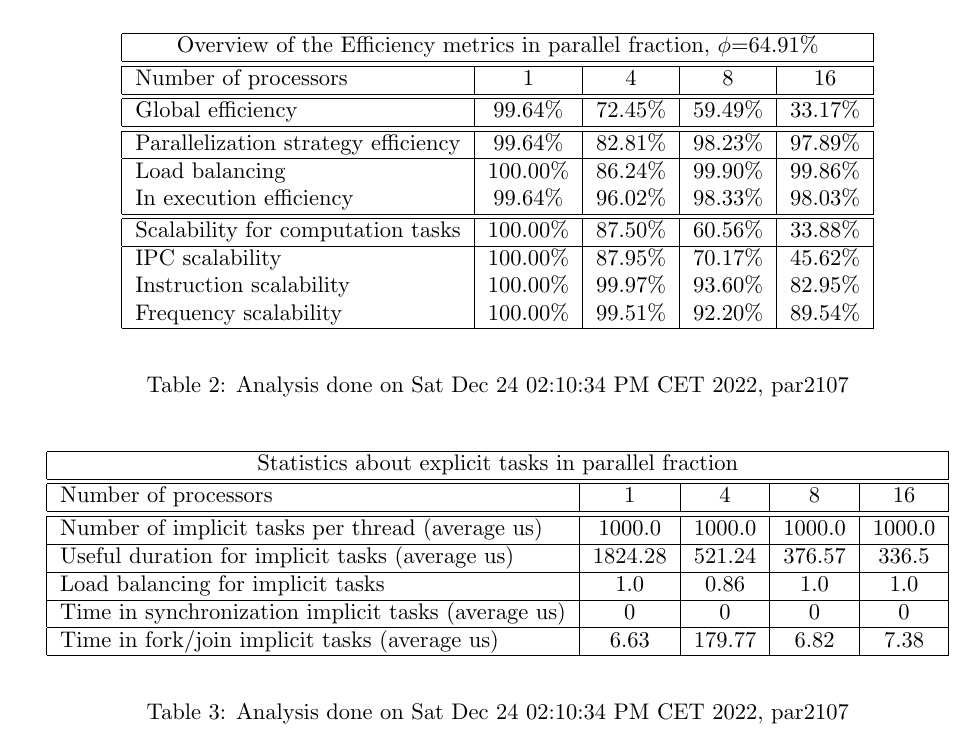
*Overall Analysis*

*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?*

For the analysis of our code we will be using the 16 threads because for some reason the script is not generating the file with 12 threads, and for the sake of simplicity we decided to just use the 16 threads instead.



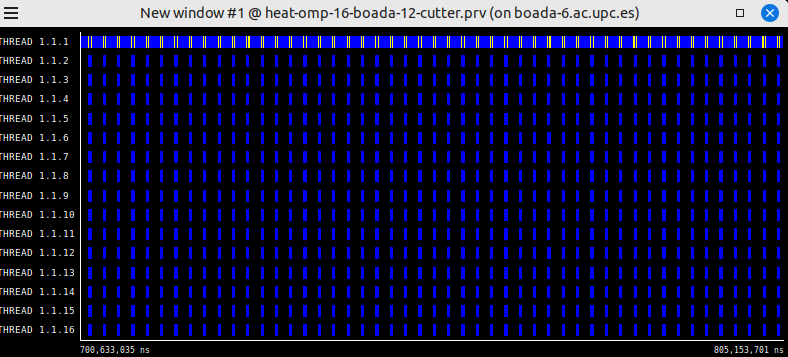


*3.1.3*

*Detailed Analysis*

*For the deliverable: Include the window timelines or paraver Hints that you consider neces-*

*sary. What is the region of the code that is provoking the low value for the parallel fraction in your parallelisation?*



The cause of the low value for the parallel fraction of our parallelisation is most likely due tone big part of the execution not being parallelized that is, copy\_mat is not parallel yet, in the following excerpt of code we’ll give a shot at parallelizing this function and see how it performs

*3.1.4*

*Optimization*

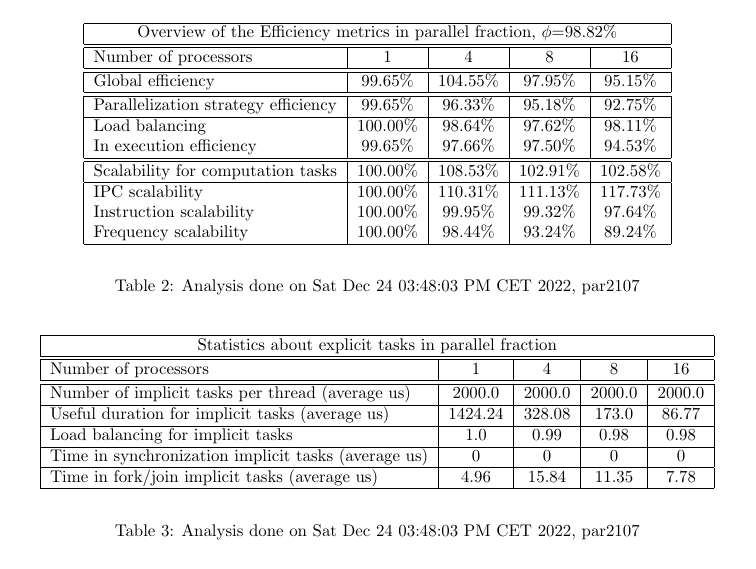
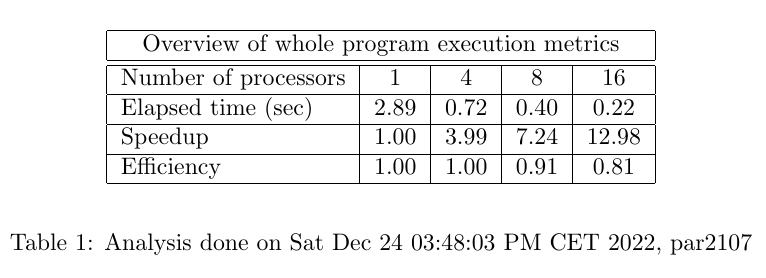
*For the deliverable: Include an excerpt of the code to show the OpenMP annotations you have added to the code.*

| *void copy\_mat (double \*u, double \*v, unsigned sizex, unsigned sizey) {   int nblocksi=omp\_get\_max\_threads();  int nblocksj=1;  #pragma omp parallel firstprivate(nblocksi, nblocksj)  {  int blocki = omp\_get\_thread\_num();  int i\_start = lowerb(blocki, nblocksi, sizex);  int i\_end = upperb(blocki, nblocksi, sizex);  for (int blockj=0; blockj<nblocksj; ++blockj) {  int j\_start = lowerb(blockj, nblocksj, sizey);  int j\_end = upperb(blockj, nblocksj, sizey);  for (int i=max(1, i\_start); i<=min(sizex-2, i\_end); i++)  for (int j=max(1, j\_start); j<=min(sizey-2, j\_end); j++)  v[i\*sizey+j] = u[i\*sizey+j];  }  } }* |
| --- |

*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?*



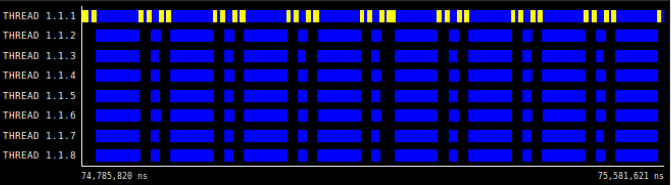
After parallelizing that, we get the following results:

But we see now that the parallel fraction has increased dramatically, from 64 to 98%.

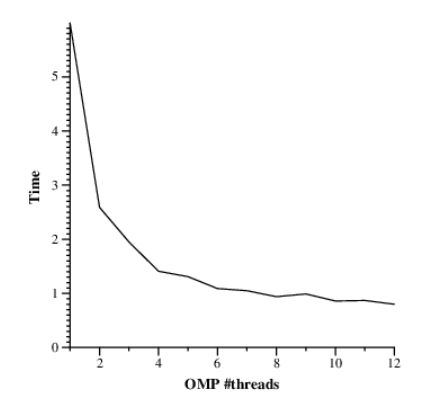
These are very promising results since with 16 cores we get a speedup of 13 which is much, much better than we saw on the previous version of our code.

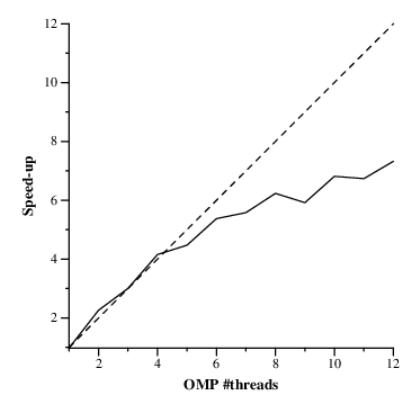
*Detailed Analysis of the Optimized Code*

*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.*



Upon revisiting the paraver execution we could see that zooming into the gaps that we saw previously, these were much, smaller, so we interpreted that as a sign that we were on track with our guess. Obviously being able to parallelise more chunks of our code without adding excessive syncrhonization overheads is going to be beneficial for the overall performance of our code, which is why our optimization worked so well.





The Scalability on this graph looks very promising on the first 4–5 threads, but then as threads increment, the speed-up starts performing poorly.

## 3.2 Gauss-Seidel solver

3.2.1

First Implementation

For the deliverable: Include an excerpt of the code to show the OpenMP annotations you have added to the code. Is the execution time reduced compare to the sequential execution? (if not, you should reconsider your implementation).

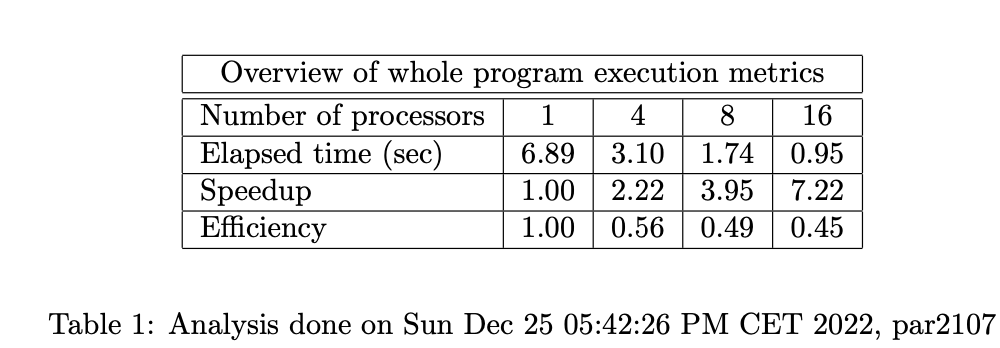
| Type | Time | Flops | Flops per second |
| --- | --- | --- | --- |
| Sequential | 5.163s | 8.806 GFlop | 1705.64 MFlop/s |
| Parallel 16 cores | 1.029s | 8806 GFlop | 8556.88 MFlop/s |

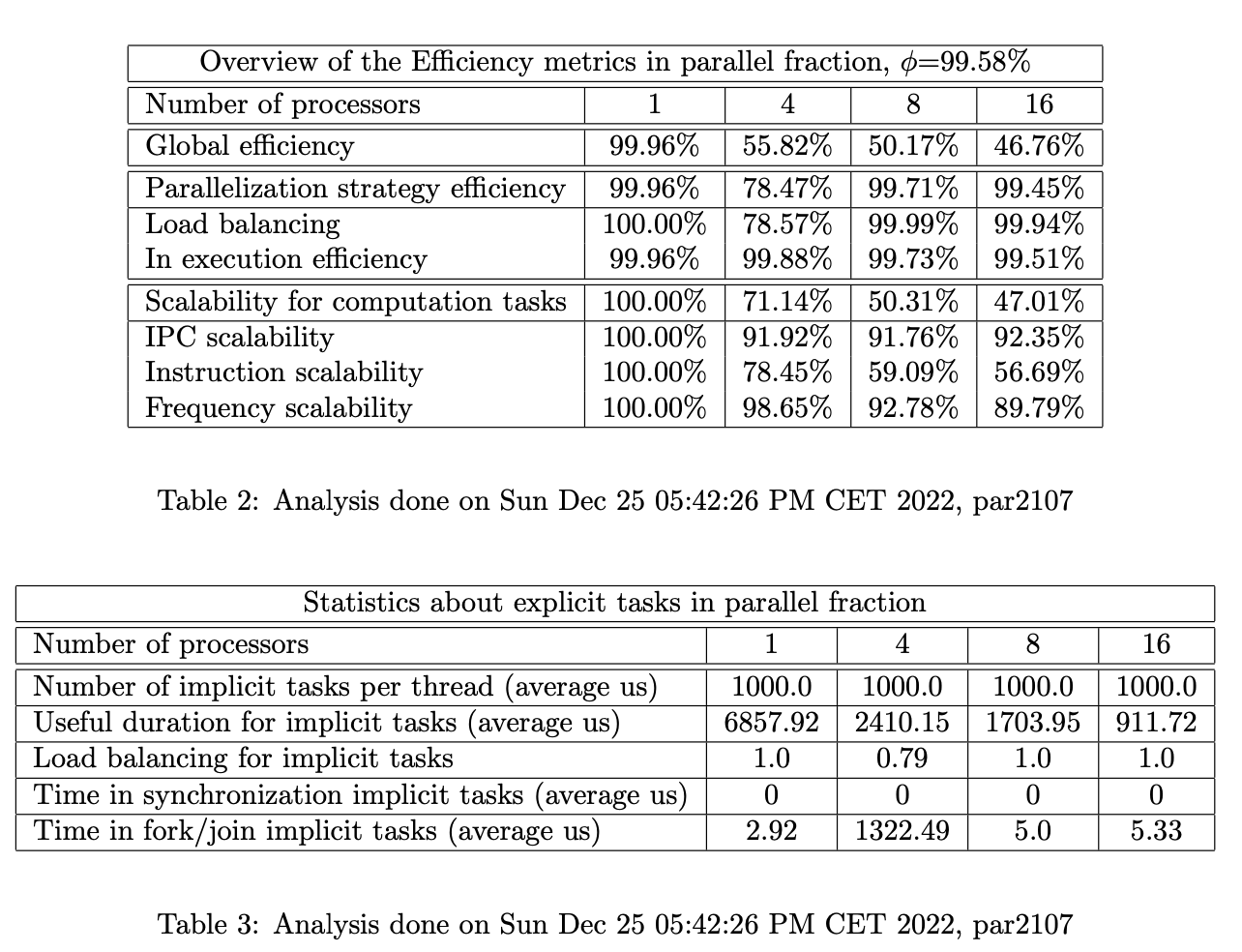
| // 2D-blocked solver: one iteration step double solve (double \*u, double \*unew, unsigned sizex, unsigned sizey) {  double tmp, diff, sum=0.0;   int nblocksi=omp\_get\_max\_threads();  int nblocksj=1;  // We use this if to check whether the function is called by the Jacobi method or the Gauss  // If u == unew then it's Gauss otherwise we are using the Jacobi method  if (u == unew) {  int nblocksj = nblocksi;  int next[nblocksi][16];  next[0][0] = nblocksj;  for (int i = 1; i < nblocksi; i++) next[i][0] = 0;  #pragma omp parallel reduction(+:sum) firstprivate(nblocksi, nblocksj) private(tmp, diff) shared(next)   {  int order;  int blocki = omp\_get\_thread\_num();  int i\_start = lowerb(blocki, nblocksi, sizex);  int i\_end = upperb(blocki, nblocksi, sizex);  for (int blockj=0; blockj<nblocksj; ++blockj) {  int j\_start = lowerb(blockj, nblocksj, sizey);  int j\_end = upperb(blockj, nblocksj, sizey);    do {  #pragma omp atomic read  order = next[blocki][0];  }while(order < blockj + 1);   for (int i=max(1, i\_start); i<=min(sizex-2, i\_end); i++) {  for (int j=max(1, j\_start); j<=min(sizey-2, j\_end); j++) {  tmp = 0.25 \* ( u[ i\*sizey + (j-1) ] + // left  u[ i\*sizey + (j+1) ] + // right  u[ (i-1)\*sizey + j ] + // top  u[ (i+1)\*sizey + j ] ); // bottom  diff = tmp - u[i\*sizey+ j];  sum += diff \* diff;  unew[i\*sizey+j] = tmp;  }  }  if (blocki < nblocksi-1){  #pragma omp atomic update  next[blocki+1][0]++;  }  }  }  }  else {  #pragma omp parallel reduction(+:sum) private(tmp,diff) firstprivate(nblocksi, nblocksj)  {  …  }  return sum; } |
| --- |

The execution is clearly much faster than the sequential version 5-fold better than it used to be, We’ll analyze why that is, looking at the modelfactors tables, and also tackle why Jacobi seems like a faster implementation.

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?





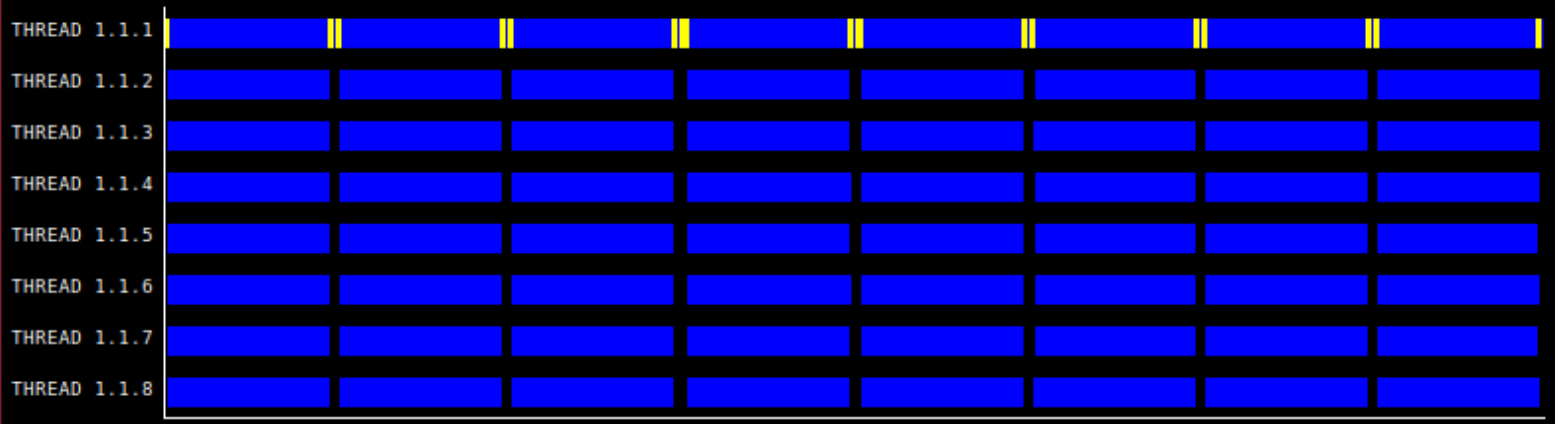
We can see that the parallel fraction is very good, at 99.58%, load balancing also seems to be performing quite well, Execution efficiency also seems to be correct, but scalability for computation tasks seems to be quite the problem here, specifically Instruction scalability.

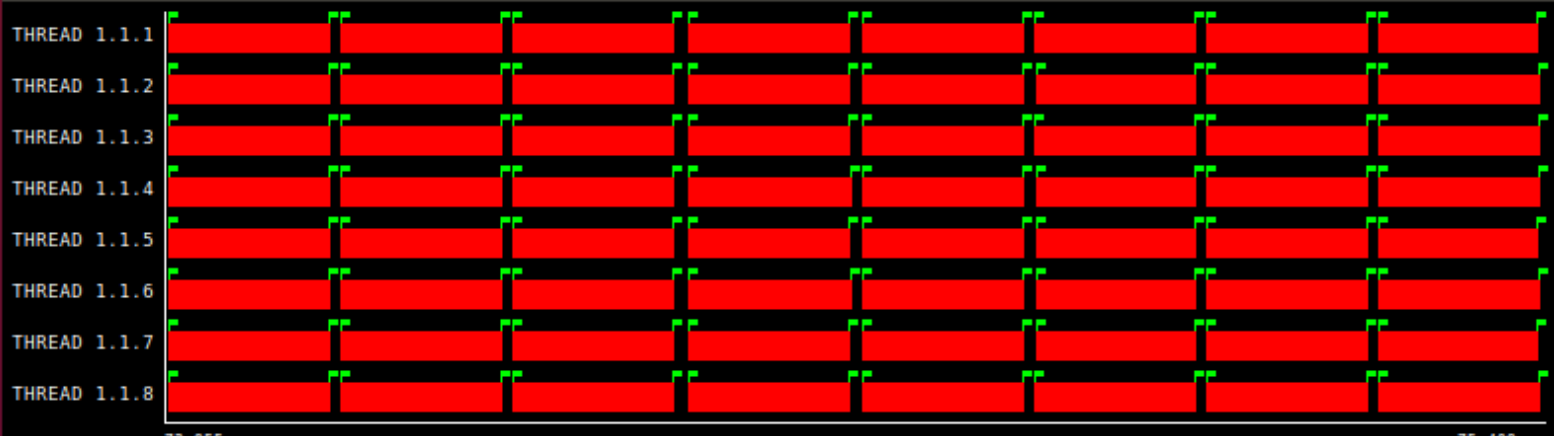
By no means is this bad parallelization since we are obviously seeing some results, but on the Jacobi, it was much better, we’ll take a look and see if we can improve the performance a little bit.

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 synchronisation.*

**



After further inspecting the problem through paraver hints, we found some sequential execution that was affecting the performance of our code.

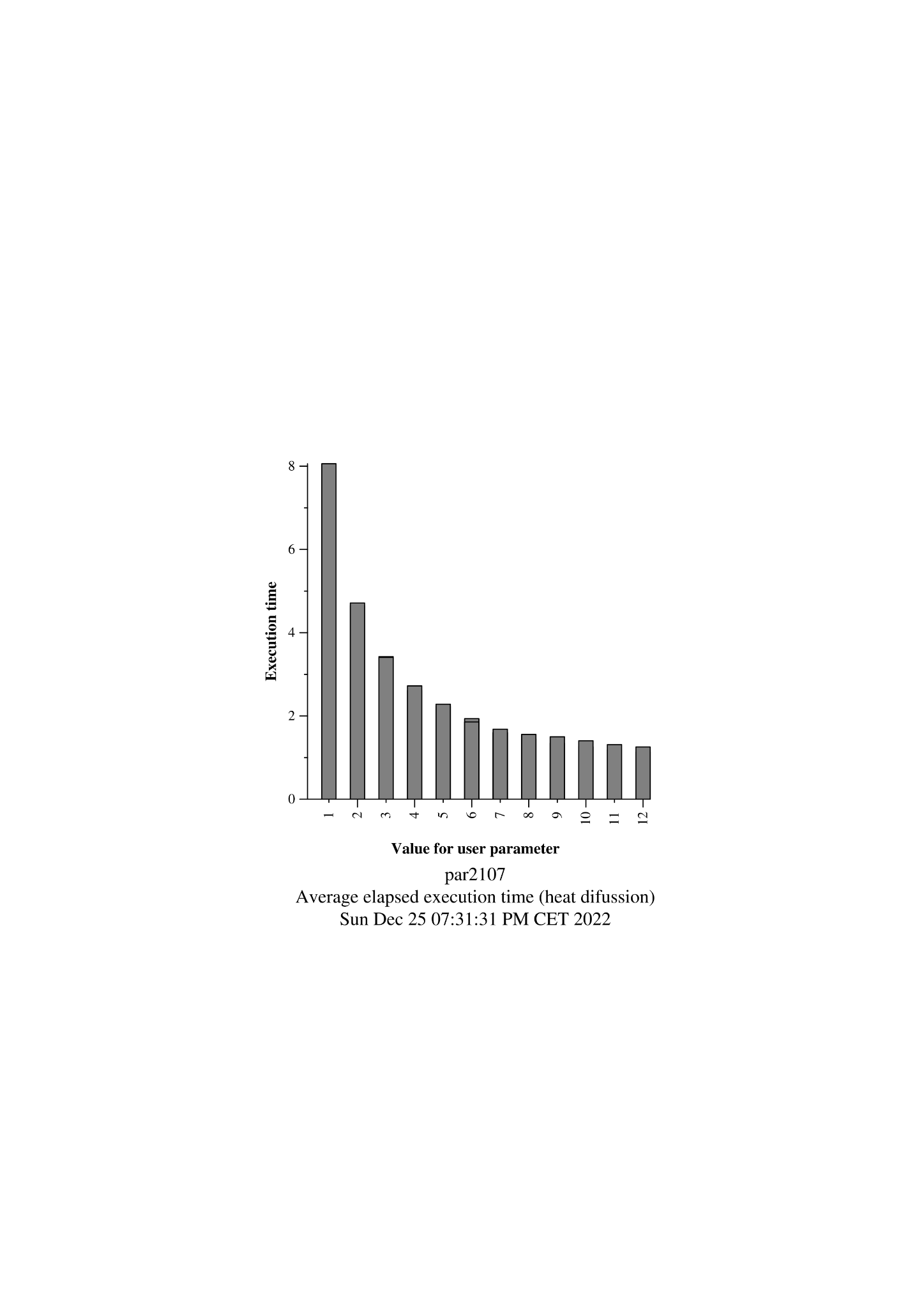
With that said, we thought about how to improve the performance of our code further, one possible culprit of that decrease in performance could be the number of blocks in the j dimension since right now njblocks= 1. Modifying the ratio between computation and synchronisation because the number of cells to be calculated per task would change accordingly. If we have smaller and smaller tasks then the synchronisation should be greater, but the computation is more parallelizable, being able to use more threads. The balance between these two factors is key to obtaining good performance.

*Number of blocks tune*

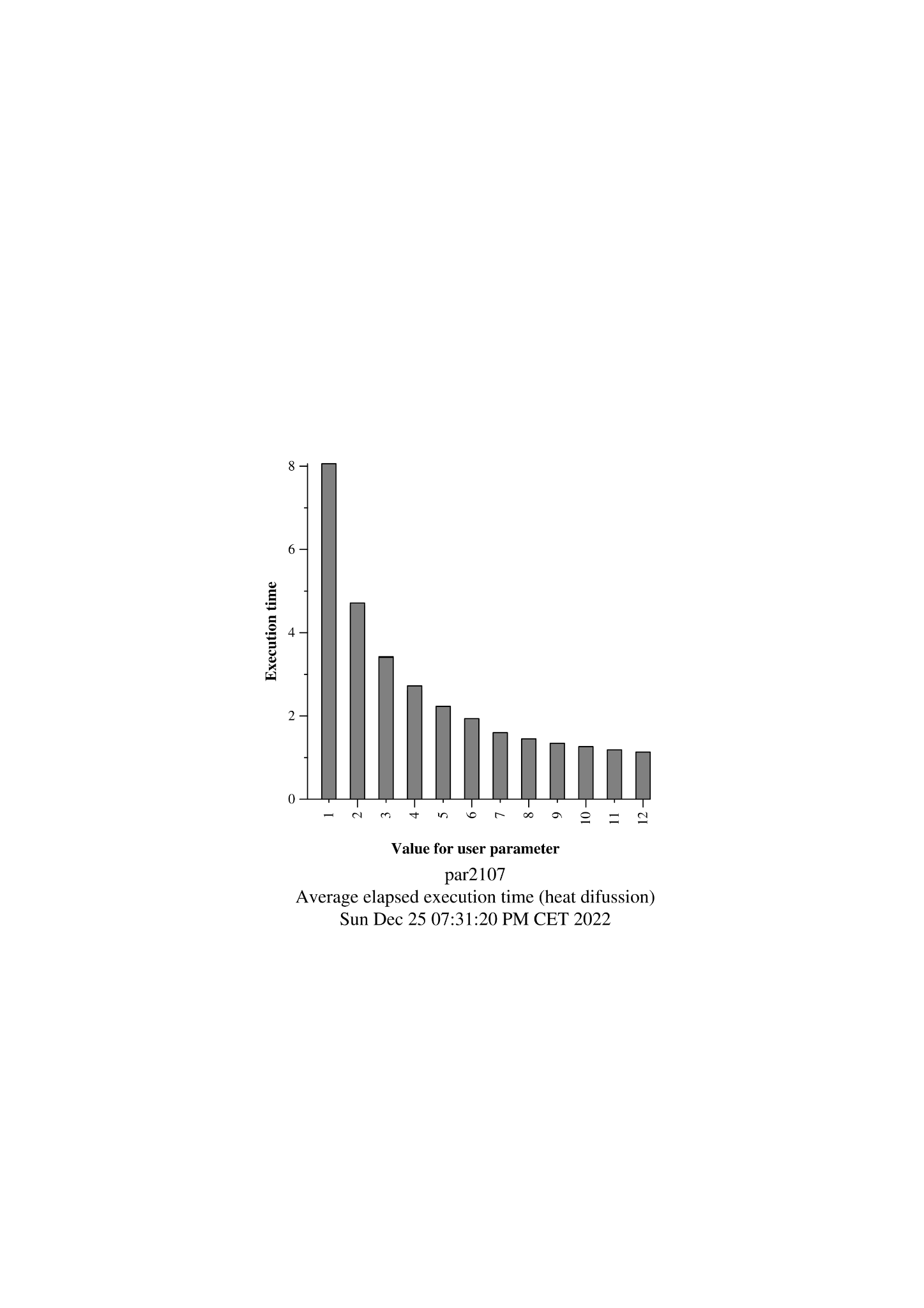
*For the deliverable: Include the plots obtained with submit-userparam-omp.sh and explain the plot that is obtained. Also, include the scalability plot and explain the performance obtained.*

nblocksj = userparam

12 threads:

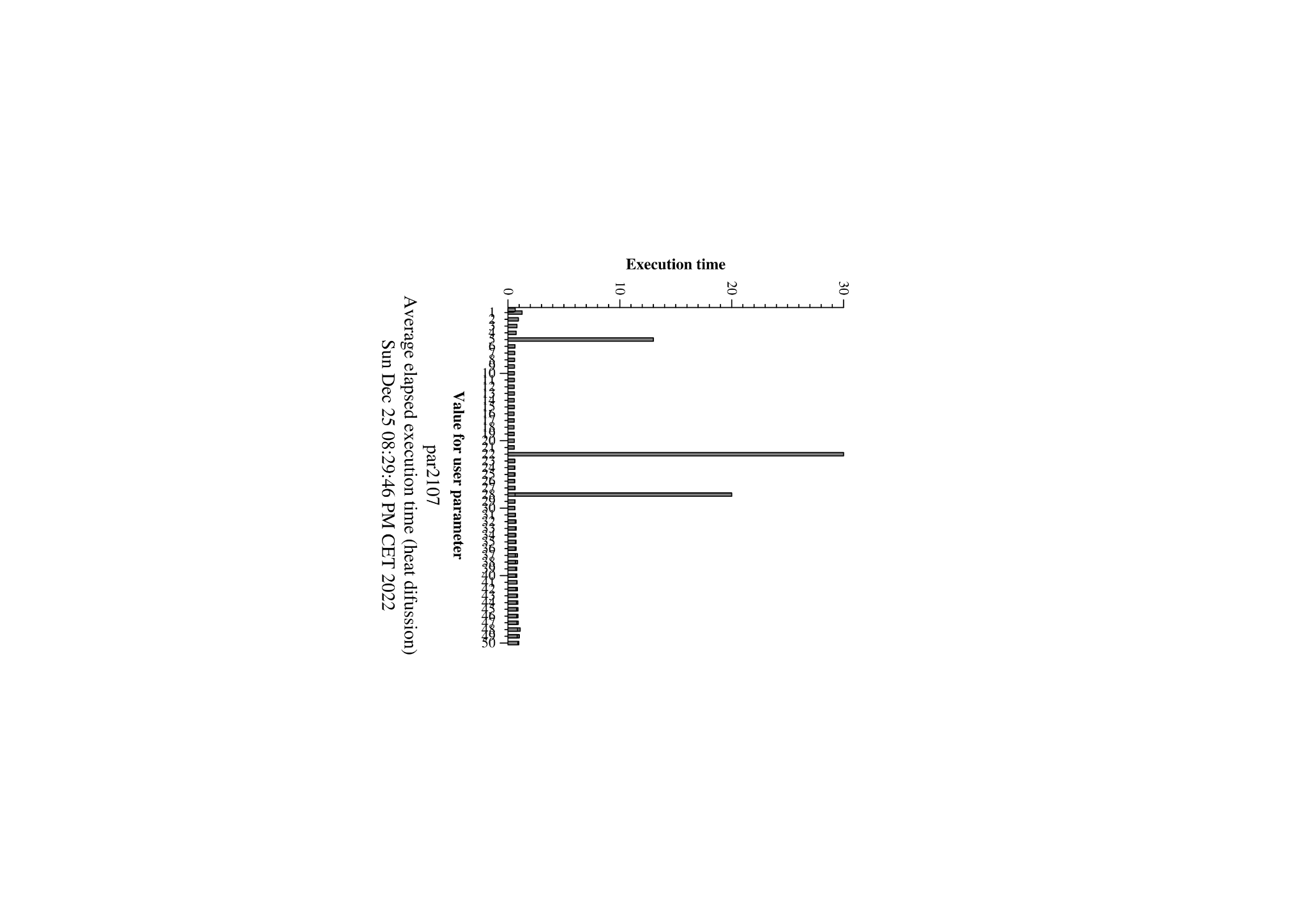


16 threads

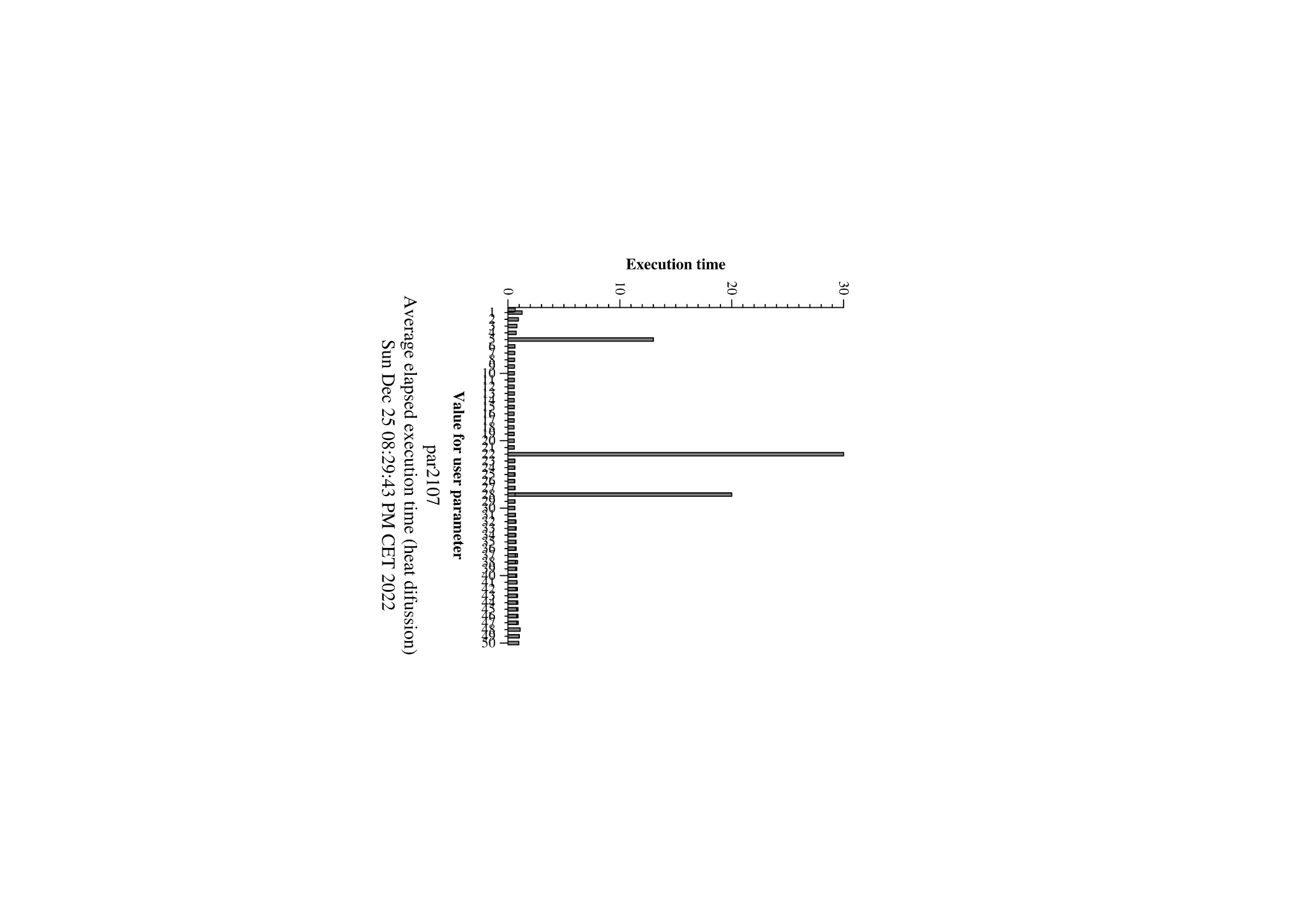


nblocksj = userparam\*

12 threads



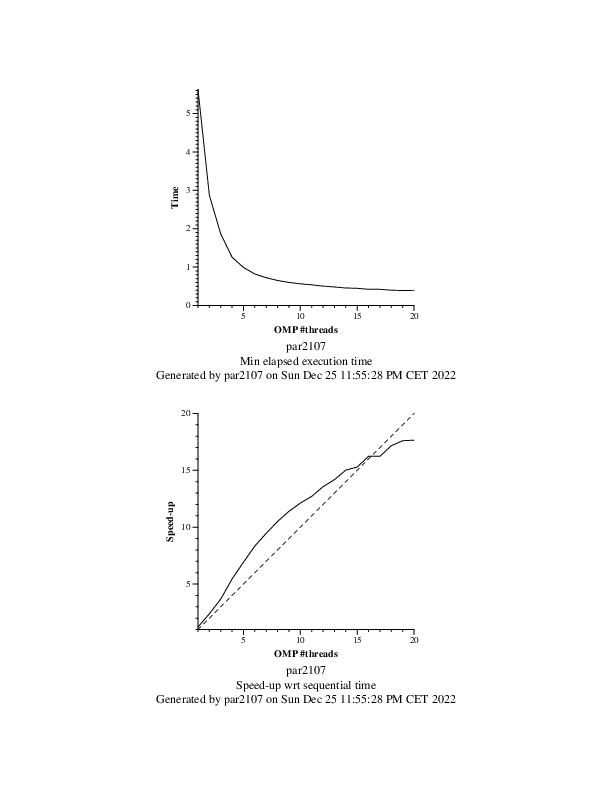
16 threads:



Using userparam, directly does not seem to be a good choice, at least, it could be much better to use bigger values for userparam since we did not seem to find a valley in our graph and instead the time kept going down.

So we decided to change the range, 1 through 50, we could roughly see that 18 would be a potential good choice, but to be sure we will zoom further to be sure.



This is for 16 threads, and we see that roughly the best choice is 10. So for 16 threads, that would make for 16\*10 = 160 nblocksj.

These are the results with nblocksj = 160

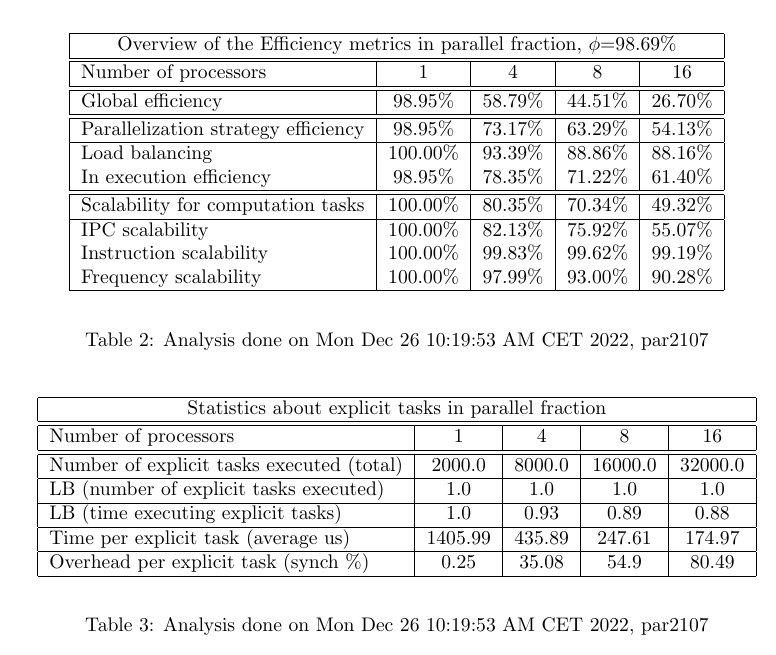
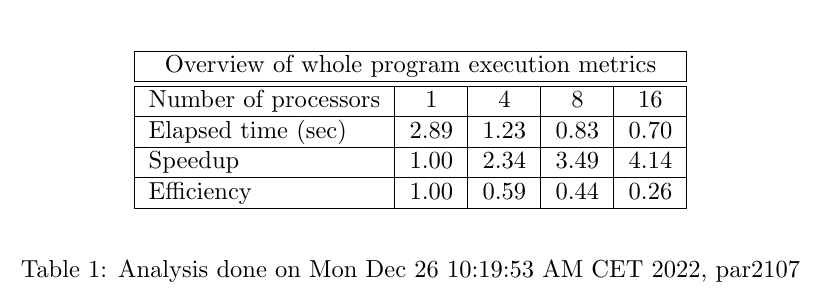
We can see that we get pretty promising results for threads up to 15 with speed-ups above their corresponding number of threads which is surprising to see and after the that, adding more threads does not equal a 1 on 1 gain for the speed-up but still, not bad at all.

# Optional 1:

After confirming that the image generated by our code was correct through a diff, we realized that our code was much, much slower than that of the generated by the implicit tasks, the overhead of the explicit tasks was causing a big hit to the performance of our code, here’s an excerpt of the code used:

| void copy\_mat (double \*u, double \*v, unsigned sizex, unsigned sizey) {   int nblocksi=omp\_get\_max\_threads();  int nblocksj=1;  #pragma omp parallel  #pragma omp single  {  #pragma omp taskloop firstprivate(nblocksi, nblocksj)  for (int blocki=0; blocki <nblocksi; ++blocki) {  int i\_start = lowerb(blocki, nblocksi, sizex);  int i\_end = upperb(blocki, nblocksi, sizex);  for (int blockj=0; blockj<nblocksj; ++blockj) {  int j\_start = lowerb(blockj, nblocksj, sizey);  int j\_end = upperb(blockj, nblocksj, sizey);  for (int i=max(1, i\_start); i<=min(sizex-2, i\_end); i++)  for (int j=max(1, j\_start); j<=min(sizey-2, j\_end); j++)  v[i\*sizey+j] = u[i\*sizey+j];  }  }  } } |
| --- |

| double solve (double \*u, double \*unew, unsigned sizex, unsigned sizey) {  double tmp, diff, sum=0.0;   int nblocksi=omp\_get\_max\_threads();  int nblocksj=1;  // We use this if to check whether the function is called by the Jacobi method or the Gauss  // If u == unew then it's Gauss otherwise we are using the Jacobi method  if (u == unew) {  …  }  else {  int nblocksj = 1;  #pragma omp parallel  #pragma omp single  // complete data sharing constructs here  {  #pragma omp taskloop reduction(+:sum) firstprivate(nblocksi, nblocksj) private(tmp, diff)  for(int blocki = 0; blocki < nblocksi; ++blocki) {  int i\_start = lowerb(blocki, nblocksi, sizex);  int i\_end = upperb(blocki, nblocksi, sizex);  for (int blockj=0; blockj<nblocksj; ++blockj) {  int j\_start = lowerb(blockj, nblocksj, sizey);  int j\_end = upperb(blockj, nblocksj, sizey);  for (int i=max(1, i\_start); i<=min(sizex-2, i\_end); i++) {  for (int j=max(1, j\_start); j<=min(sizey-2, j\_end); j++) {  tmp = 0.25 \* ( u[ i\*sizey + (j-1) ] + // left  u[ i\*sizey + (j+1) ] + // right  u[ (i-1)\*sizey + j ] + // top  u[ (i+1)\*sizey + j ] ); // bottom  diff = tmp - u[i\*sizey+ j];  sum += diff \* diff;  unew[i\*sizey+j] = tmp;  }  }  }  }  }  }   return sum; } |
| --- |

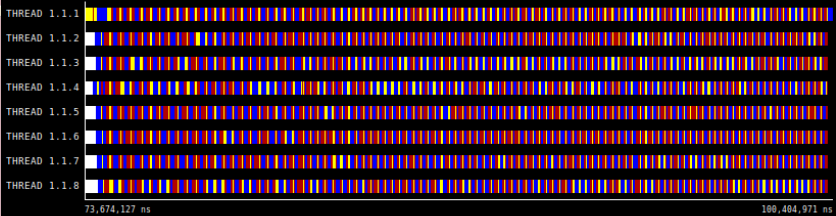


As we can see, these results are far from what we saw on the implicit task decomposition, but we did not see how we could improve upon this with explicit task decomposition, the overheads from synchronization and explicit tasks, hurts the execution time of the application.

# Optional 2:

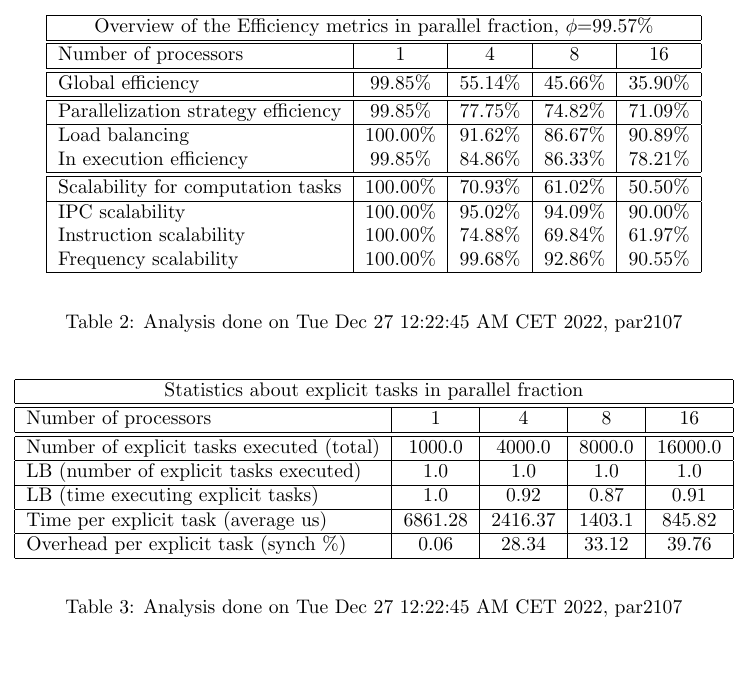
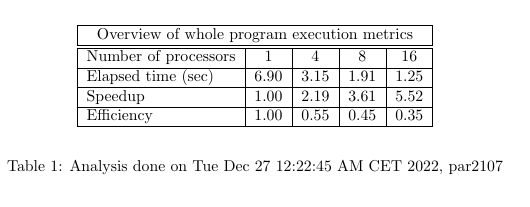
| double solve (double \*u, double \*unew, unsigned sizex, unsigned sizey) {  double tmp, diff, sum=0.0;   int nblocksi=omp\_get\_max\_threads();  int nblocksj=1;  // We use this if to check whether the function is called by the Jacobi method or the Gauss  // If u == unew then it's Gauss otherwise we are using the Jacobi method  if (u == unew) {  int nblocksj = 160;  int next[nblocksi][16];   next[0][0] = nblocksj;  for (int i = 1; i < nblocksi; i++) next[i][0] = 0;  #pragma omp parallel reduction(+:sum) firstprivate(nblocksi, nblocksj) private(tmp, diff) shared(next)   {  int order;  int blocki = omp\_get\_thread\_num();  int i\_start = lowerb(blocki, nblocksi, sizex);  int i\_end = upperb(blocki, nblocksi, sizex);  for (int blockj=0; blockj<nblocksj; ++blockj) {  int j\_start = lowerb(blockj, nblocksj, sizey);  int j\_end = upperb(blockj, nblocksj, sizey);    do {  #pragma omp atomic read  order = next[blocki][0];  }while(order < blockj + 1);   for (int i=max(1, i\_start); i<=min(sizex-2, i\_end); i++) {  for (int j=max(1, j\_start); j<=min(sizey-2, j\_end); j++) {  tmp = 0.25 \* ( u[ i\*sizey + (j-1) ] + // left  u[ i\*sizey + (j+1) ] + // right  u[ (i-1)\*sizey + j ] + // top  u[ (i+1)\*sizey + j ] ); // bottom  diff = tmp - u[i\*sizey+ j];  sum += diff \* diff;  unew[i\*sizey+j] = tmp;  }  }  if (blocki < nblocksi-1){  #pragma omp atomic update  next[blocki+1][0]++;  }  }  }  }  else {  …  }   return sum; } |
| --- |

We thought that we could use the same synchronization object to synchronize our explicit tasks just for the sake of simplicity and performance, adding to it a taskloop with the proper visibility for each variable firstprivate, private and shared we are able to create a piece of code which implements the gaussel-Seidel with explicit tasks. We made sure it worked by using diff on the generated images.



We can see just by a quick look that we have many more red spots, that means synchronization is a big problem in our code which we would have to solve, we know now that by using an implicit task implementation we could improve upon that.

After using modelfactor on the results, we obtained the following tables



Overall, not terrible results, but then again, they were much better with the use of implicit tasks, and yes, it is a much simpler implementation, but if the application requires it, it could be useful to use such an implementation. Overall we see a huge chunk of overhead per explicit task which seems to increase per each thread we add, so it doesn’t seem like it will scale very nicely if we further increase our number of threads, it’s not terrible as we are getting better results than using a sequential implementation, but it is far from ideal.

# 4 Conclusion

All in all, with this in depth dive into parallelization through data decomposition, we were able to understand the power and raw performance of using implicit tasks.

As for the Jacobi method, by just removing one variable out of the equation with a simple reduction, we were able to make the entire code parallel. After that, realizing that we could do that same parallelization with a similar function that Jacobi used, increased the performance of the code by a huge chunk.

The Gauss-Seidel solver was personally a bit of a harder nut to crack than Jacobi, just because of the complexity of the dependencies that were naturally introduced by not using an auxiliary matrix, which made everything that much more complicated. We had to create our own synchronization object that made sure the order of the necessary tasks was always respected. This needed deep understanding of how the Gauss method accesses the matrix and how to implement such synchronization.

This need to synchronize tasks made it impossible for gauss-Seidel to reach the pure performance of Jacobi, but we managed to get closer to it by realizing that finding out the balance between computation and synchronization that comes from selecting the right amount of blocks to be traversed. We found that a good enough value would be 160.

With Jacobi, we got down to 0.24 s. As for Gauss-Seidel we got it down to near 0.9s which is honestly much less than the sequential (for 16 threads), so we are pretty happy about those results.

Also, after doing the optional assignments we realized that for this specific purpose the use of explicit tasks brought big synchronization problems and the performance of those solutions rendered mediocre at best, which is why we found it so eye-opening to see the contrast between using implicit and explicit tasks. While harder to implement or at least, less straightforward, in this case, very rewarding in terms of performance results.

The decision of using a synchronization object of a matrix instead of just a vector was just to try and prevent false sharing, since a block holds only 16 ints. That vector was accessed by all threads, so we felt the need to prevent that by using a matrix.