Università della Svizzera italiana

Institute of Computing
CI

High-Performance Computing Lab **Institute of Computing**

Student: Fabian Flores Gobet Discussed with: Ekkenhard Steinmacher

Solution for Project 3

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

This project will introduce you a parallel space solution of a nonlinear PDE using OpenMP.

1. Implementing the linear algebra functions and the stencil operators [35 Points]

1.1. Linear Algebra

1.1.1. hpc_norm2()

```
double hpc_norm2(Field const& x, const int N)
{
    double result = hpc_dot(x,x,N);
    return sqrt(result);
}
```

The L2 norm in 1D vectors can be calculated with the use of an inner product. As such, I employed the prior defined function hpc_dot(), returning the square root of its result.

1.1.2. hpc_fill()

```
void hpc_fill(Field& x, const double value, const int N)
{
    for (int i = 0; i < N; i++)
        x[i] = value;
}
```

This function iterates through all indices of x and sets its value to '*value*'.

1.1.3. hpc_axpy()

```
void hpc_axpy(Field& y, const double alpha, Field const& x, const int N)
{
    for (int i = 0; i < N; i++)
        y[i] += alpha * x[i];
}
```

This function iterates through all indices i of y, and sets its value as the sum of itself with the product of x indice i by alpha.

1.1.4. hpc_add_scaled_diff()

```
void hpc_add_scaled_diff(Field& y, Field const& x, const double alpha,
    Field const& l, Field const& r, const int N)
{
    for (int i = 0; i < N; i++)
        y[i] = x[i] + alpha * (l[i]-r[i]);
}
```

This function iterates through all indices i of y, and sets its value as the sum of x and indice i by alpha times l minus r indice i.

1.1.5. hpc_scaled_diff()

```
void hpc_scaled_diff(Field& y, const double alpha,
    Field const& l, Field const& r, const int N)
{
    for (int i = 0; i < N; i++)
        y[i] = alpha * (l[i]-r[i]);
}
```

This function iterates through all indices i of y, and sets its value as the the product of alpha by l minus r indice i.

1.1.6. hpc_scale()

```
void hpc_scale(Field& y, const double alpha, Field const& x, const int N)
{
    for (int i = 0; i < N; i++)
        y[i] = alpha * x[i];
}
```

This function iterates through all indices i of y, and sets its value as x indice i times a constant alpha.

1.1.7. hpc_lcomb()

```
void hpc_lcomb(Field& y, const double alpha, Field const& x, const double beta
    ,
    Field const& z, const int N)
{
    for (int i = 0; i < N; i++)
        y[i] = alpha * x[i] + beta * z[i];
}
```

This function iterates through all indices i of y, and sets its value as x indice i times a constant alpha, plus z indice i times a constant beta.

1.1.8. hpc_copy()

```
void hpc_copy(Field& y, Field const& x, const int N)
{
    for (int i = 0; i < N; i++)
        y[i] = x[i];
}
```

This function iterates through all indices i of y, and sets its value to the same value given by x at indice i.

1.2. Stencil operators

1.2.1. Interior grid points

```
for (int j=1; j < jend; j++) {
    for (int i=1; i < iend; i++) {
        f(i,j) = -(4. + alpha) * s(i,j)
                + s(i-1,j) + s(i+1,j) + s(i,j-1)
                + s(i,j+1) + beta * s(i,j) * (1.0 - s(i,j))
                + alpha * y_old(i,j);
    }
}
```

This function computes the inner grid points according to the formula stated on the exercise sheet. Because we are not in the boundaries we can directly use all adjacent s(i,j).

1.2.2. West boundary

```
int i = 0;
for (int j = 1; j < jend; j++)
{
    f(i,j) = -(4. + alpha) * s(i,j)
                + s(i,j-1) + s(i,j+1) + s(i+1,j)
                + alpha*y_old(i,j) + bndW[j]
                + beta * s(i,j) * (1.0 - s(i,j));
}
```

This function computes the west boundary points according to the formula stated on the exercise sheet. Because we are in the boundary, points adjacent to the right have to use the values ate bndW for each j. Notice that we aren't adressing corners as these will be addressed separately.

1.2.3. Inner north boundary

```
for (int i = 1; i < iend; i++)
{
    f(i,j) = -(4. + alpha) * s(i,j)
               + s(i,j-1) + s(i-1,j) + s(i+1,j)
               + alpha*y_old(i,j) + bndN[i]
               + beta * s(i,j) * (1.0 - s(i,j));
}
```

This function computes the inner north points according to the formula stated on the exercise sheet. We are not taking into account the corners as these will are addressed individually. Therefore we just have to employ the use of bndN at position j to account for the node upper vertically adjacent.

1.2.4. Inner south boundary

```
for (int i = 1; i < iend; i++)
{
    f(i,j) = -(4. + alpha) * s(i,j)
               + s(i+1,j) + s(i-1,j) + s(i,j+1)
               + alpha*y_old(i,j) + bndS[i]
               + beta * s(i,j) * (1.0 - s(i,j));
}
```

This function computes the inner south points according to the formula stated on the exercise sheet. We are not taking into account the corners as these will are addressed individually. Therefore we just have to employ the use of bndS at position j to account for the node under vertically adjacent.

1.3. Results

After implementing all the prior functions regarding operators.cpp and linalg.cpp, by running the commands

```
$ ./main 128 100 0.005
$ ./plotting.py
```

the resulting image computed by the algorithm is

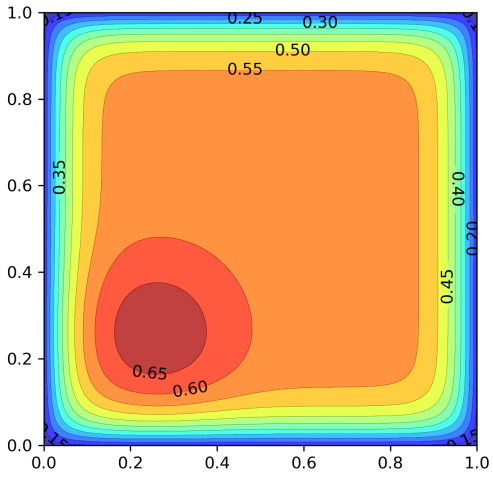

Figure 1: Exercise 1 output.png

2. Adding OpenMP to the nonlinear PDE mini-app [50 Points]

2.0.1. Replace welcome message in main.cpp [2 Points]

In order to retrieve the number of threads at current use I added the following code (lines 111-112 of main.cpp)

```
const char* omp_num_threads_p = std::getenv("OMP_NUM_THREADS");
int numt = (omp_num_threads_p!=nullptr) ? std::atoi(omp_num_threads_p):1;
```

The prior code gets retrieves the pointer to the environment variable $OMP\_NUM\_THREADS$, and in case the pointer is a null pointer sets the *numt* variable to 1. If it's not a null pointer, then it retrieves the value of the environment variable.

The variable *numt* is then used in the initializing message (line 117 of main.cpp) as such

```
std::cout << "threads---::-" << numt << std::endl;
```

2.0.2. Linear algebra kernel [15 Points]

Implementation of OpenMP directives in this part is straight forward. There are no potential race conditions, for the exception of *hpc_dot()*. As such, we can employ the use of the $\#pragma\ omp\ parallel\ for$ directive in all of these in a similar fashion.

The general idea for each of them (except *hpc_dot()*) is to declare the shared parameters between all running threads, whereas none of them will be private for the exception of the index regarding the for loop. This explanation covers all cases and only particularities will be addressed in any of the cases.

Because it is unsuitable to repeat lines of code previously shown, I will refer to the implementation of the omp directives and their location in linalg.cpp.

2.0.2.1. hpc_dot() - line 57

```
double hpc_dot(Field const& x, Field const& y, const int N)
{
    double result = 0;

    #pragma omp parallel for default(none) shared(x,y,N) reduction(+:result)
    for (int i = 0; i < N; i++)
        result += x[i] * y[i];

    return result;
}
```

This function wasn't previously implemented, but omp directives were applied to it. Unlike the rest of the cases, the variable *result* has a race condition composed of additions. Therefore, the use of the reduction clause is applied.

2.0.2.2. hpc_norm2()

The function *hpc_norm2()* makes use of *hpc_dot()*, so all uses of the function are directed to a block of code that already uses parallel programming. Besides that, there's nothing else we can do on this function in terms of implementing OpenMP.

2.0.2.3. hpc_fill() - line 79

```
#pragma omp parallel for default(none) shared(x,value,N)
```

2.0.2.4. hpc_axpy() - line 95

```
#pragma omp parallel for default(none) shared(alpha,x,y,N)
```

2.0.2.5. hpc_add_scaled_diff() - line 107

```
#pragma omp parallel for default(none) shared(y,x,alpha,l,r,N)
```

2.0.2.6. hpc_scaled_diff() - line 119

```
#pragma omp parallel for default(none) shared(y,alpha,l,r,N)
```

2.0.2.7. hpc_scale() - line 130

```
#pragma omp parallel for default(none) shared(y,alpha,x,N)
```

2.0.2.8. hpc_lcomb() - line 142

```
#pragma omp parallel for default(none) shared(y,alpha,x,beta,z,N)
```

2.0.2.9. hpc_copy() - line 152

```
#pragma omp parallel for default(none) shared(x,y,N)
```

2.0.3. The diffusion stencil [10 Points]

As before, implementation of OpenMP directives in this part is also straight forward. There are race conditions, only some slight considerations to be taken in nested for loops. As such, we can employ the use of the *#pragma omp parallel for* directive in all of these in a similar fashion.

The general idea for each of them is to declare the shared parameters between all running threads, whereas none of them will be private for the exception of the indices regarding the for loops. This explanation covers all cases and only particularities will be addressed in any of the cases.

2.0.3.1. Interior grid points - line 39

```
#pragma omp parallel for default(none) shared(f,s,alpha,beta,y_old,jend,iend)
for (int j=1; j < jend; j++) {
    for (int i=1; i < iend; i++) {
        ...
    }
}
```

In this function we have nested for loops, so we have to decide which one of them we want to employ the use of OpenMP directives. We know that there's computational overhead associated with the management of threads(i.e. creating, scheduling and eliminating threads). Therefore, because there are no other constraints it makes sense that we perform paralleling in the outter for loop to reduce thread overhead.

2.0.3.2. Inner east boundary - line 53

```
int i = nx - 1;
#pragma omp parallel for default(none) shared(f,s,alpha,beta,y_old,jend,bndE,i)
for (int j = 1; j < jend; j++)
{
...
}
```

The inner east boundary computation code was given and therefore it wasn't addressed before. We see that its just composed of a for loop where no race conditions are present. Therefore, we can use a simple *omp parallel for* directive.

2.0.3.3. West boundary - line 67

```
#pragma omp parallel for default(none) shared(f,s,alpha,beta,y_old,jend,bndW,i)
for (int j = 1; j < jend; j++)
{   ...
}
```

2.0.3.4. Inner north boundary - line 92

```
 #pragma omp parallel for default(none) shared(f,s,alpha,beta,y_old,iend,bndN,j)
for (int i = 1; i < iend; i++)
{   ...
}
```

2.0.3.5. Inner south boundary - line 124

```
#pragma omp parallel for default(none) shared(f,s,alpha,beta,y_old,iend,bndS,j)
for (int i = 1; i < iend; i++)
{   ...
}
```

This function computes the inner south points according to the formula stated on the exercise sheet. We are not taking into account the corners as these will are addressed individually. Therefore we just have to employ the use of bndS at position j to account for the node under vertically adjacent.

2.1. Strong scaling [10+3 Points]

For this section of the exercise I will expose 5 plots for mesh sizes of 64x64, 128x128, 256x256, 512x512 and 1024x1024, where each one has the running times for 1 to 24 threads compared to the serial time.

I will then proceed in discussing and comparing results in-between the various plots.

2.1.1. Mesh of 64x64

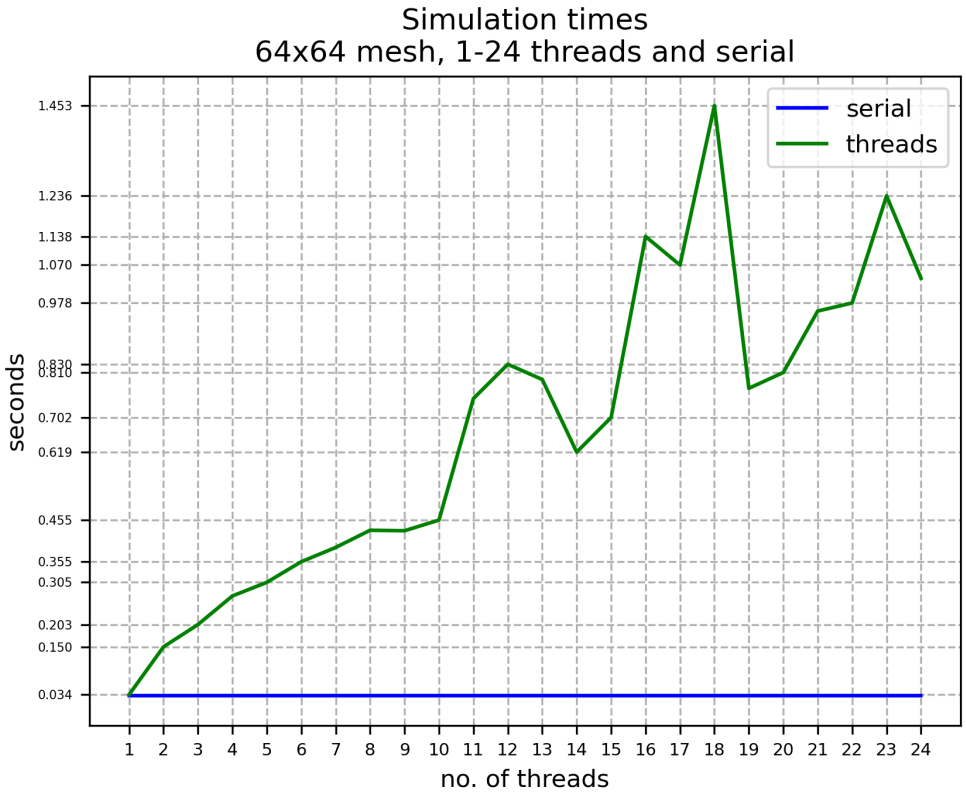

Figure 2: Exercise 2.1, mesh 64x64

2.1.2. Mesh of 128x128

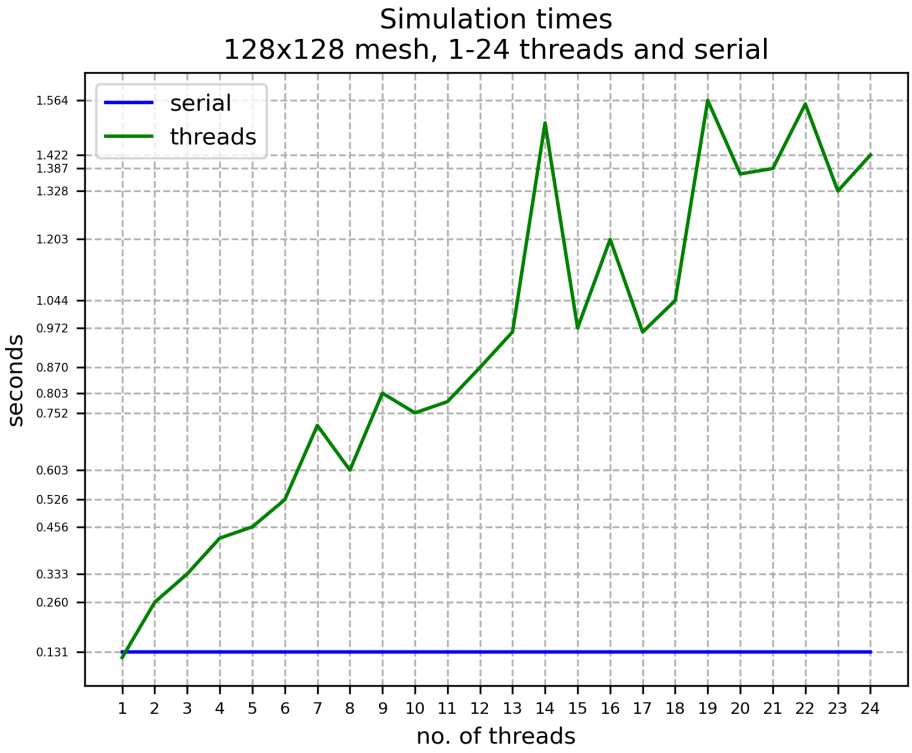

Figure 3: Exercise 2.1, mesh 128x128

2.1.3. Mesh of 256x256

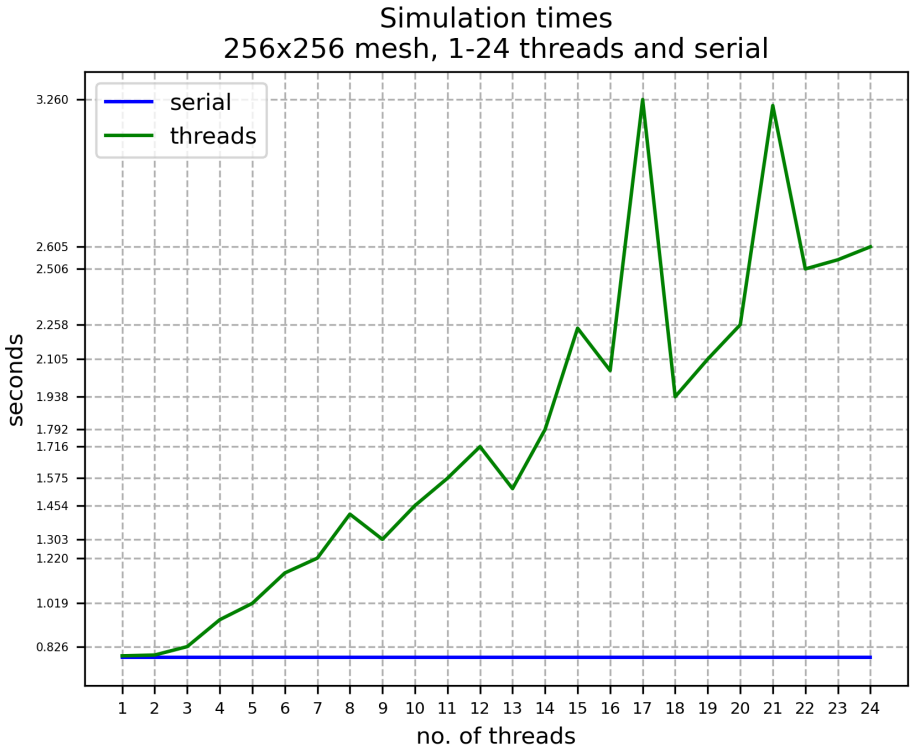

Figure 4: Exercise 2.1, mesh 256x256

2.1.4. Mesh of 512x512

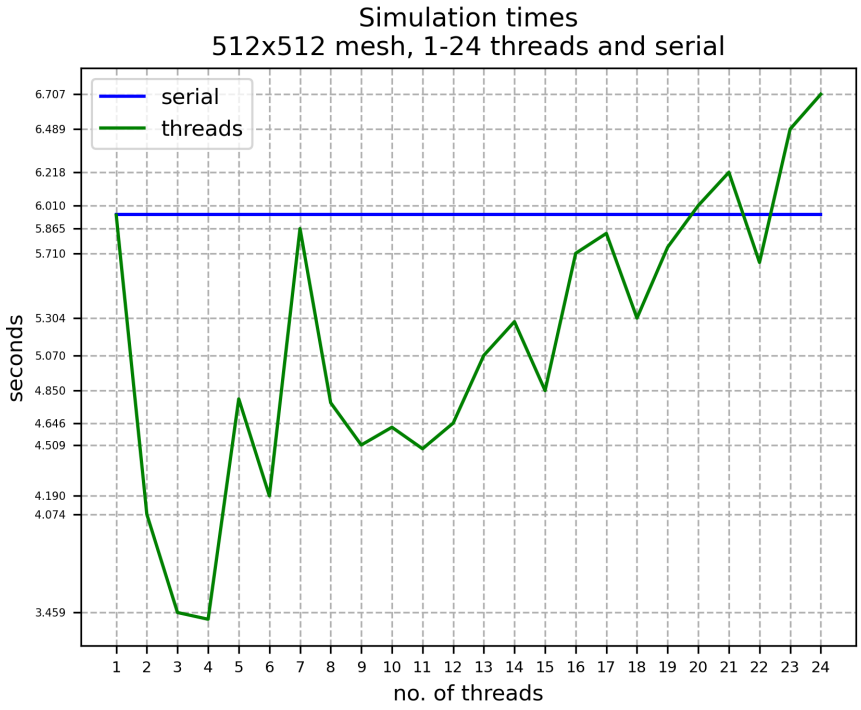

Figure 5: Exercise 2.1, mesh 512x512

2.1.5. Mesh of 1024x1024

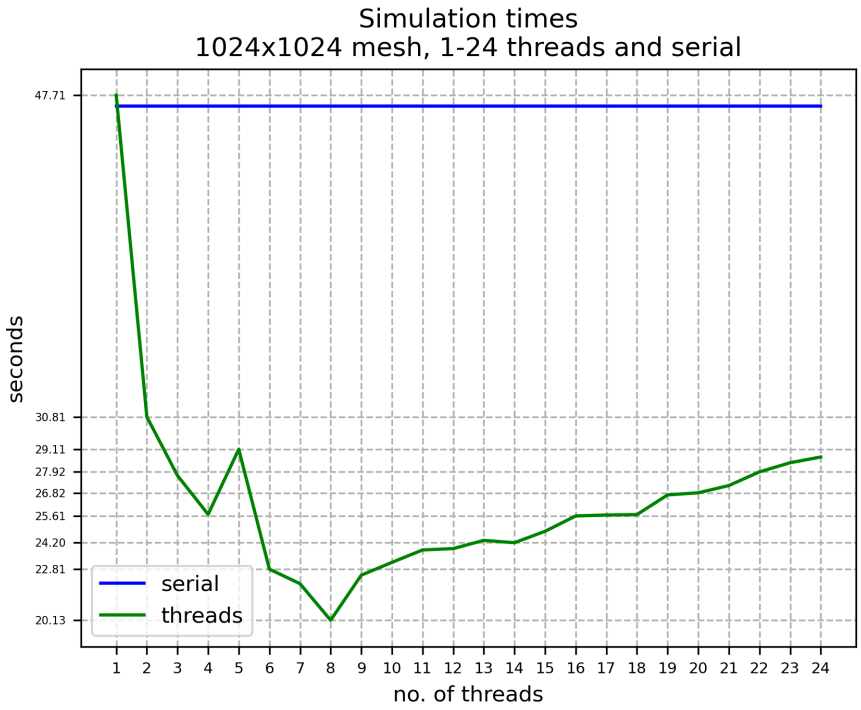

Figure 6: Exercise 2.1, mesh 1024x1024

2.1.6. Discussion

Attending to the implementation in sections 2.0.2 and 2.0.3, we can see in the running tests of 64x64, 128x128 and 256x256 that the general tendency of times increase along with the number of threads, having running times above the serial version.

This happens because for small mesh sizes, thread overhead is such that the number of conjugate gradient iterations (CGi/s) per second is always lower compared to the serial version. In other words, There are far greater number of unit processing cycles employed for the management of threads, and therefore it is detrimental against the serial version.

The following images show that this is the case regarding conjugate number of iterations per second.

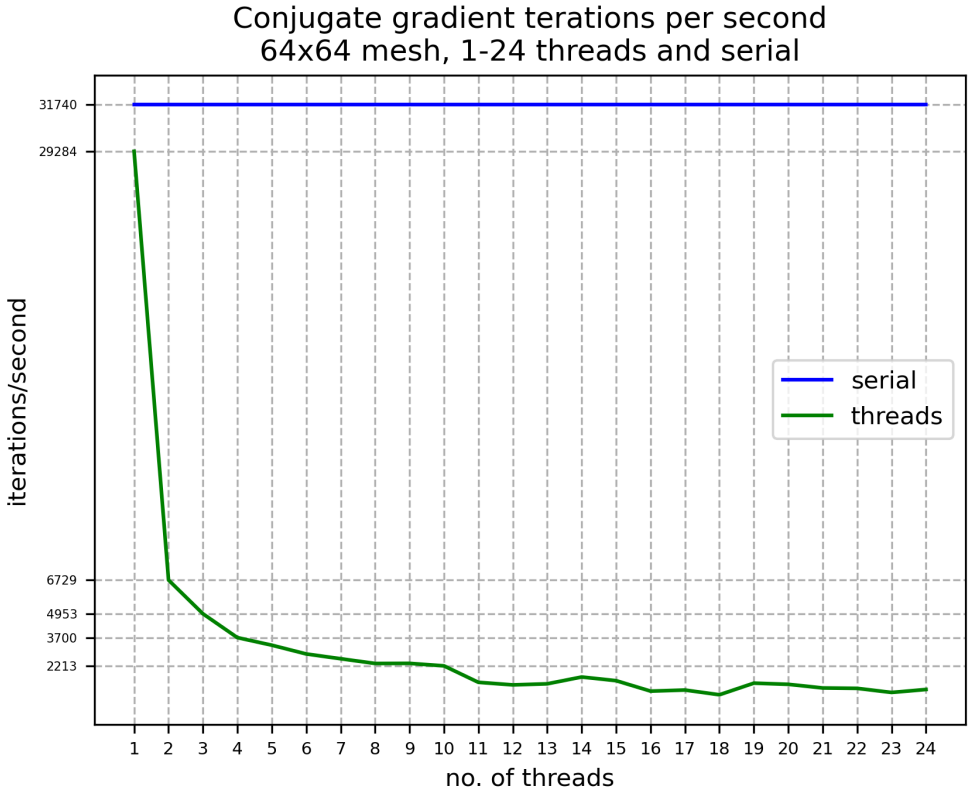

Figure 7: Exercise 2.1, CGi/s 64x64

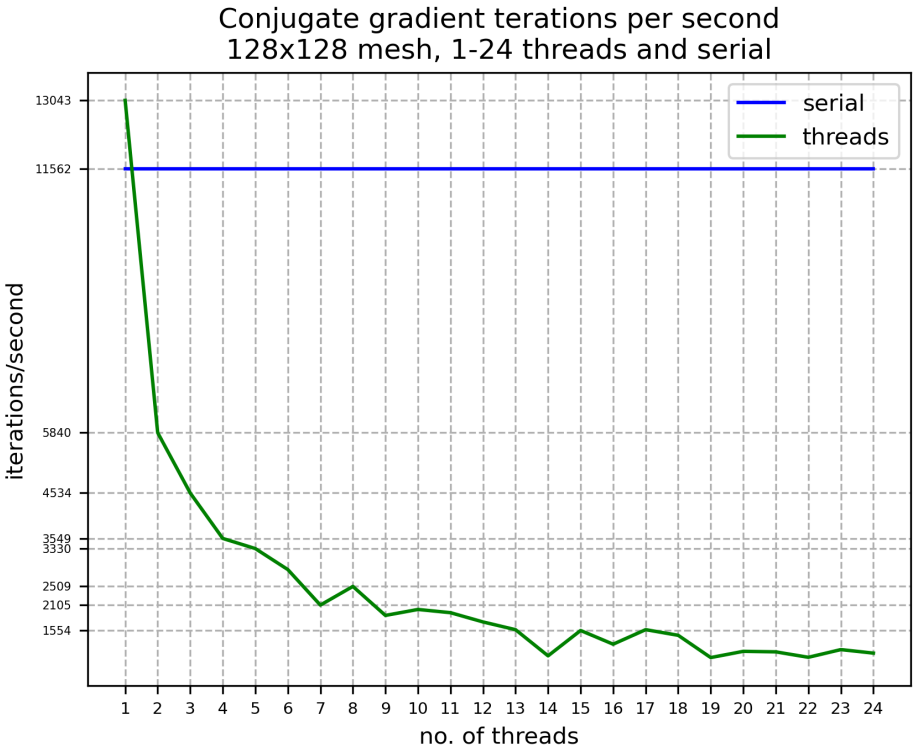

Figure 8: Exercise 2.1, CGi/s 128x128

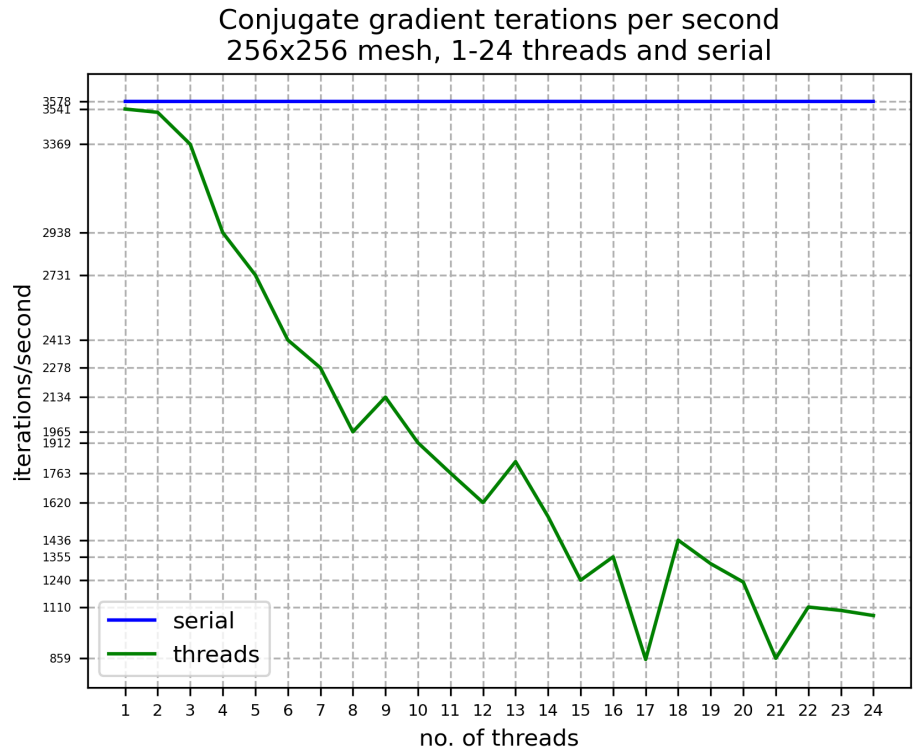

Figure 9: Exercise 2.1, CGi/s 256x256

In contrast, for the test of 512x512 we can already see that up to 20 threads, computation times tend to be much smaller, having 4 as the optimal number of threads. We can also see that for a number of threads above 20, thread overhead tends to follow a greater computation load in comparison to the serial.

When looking the CGi/s for this test, we can understand that this is the case, having greater of CGi/s in comparison to the serial version up to 20 threads.

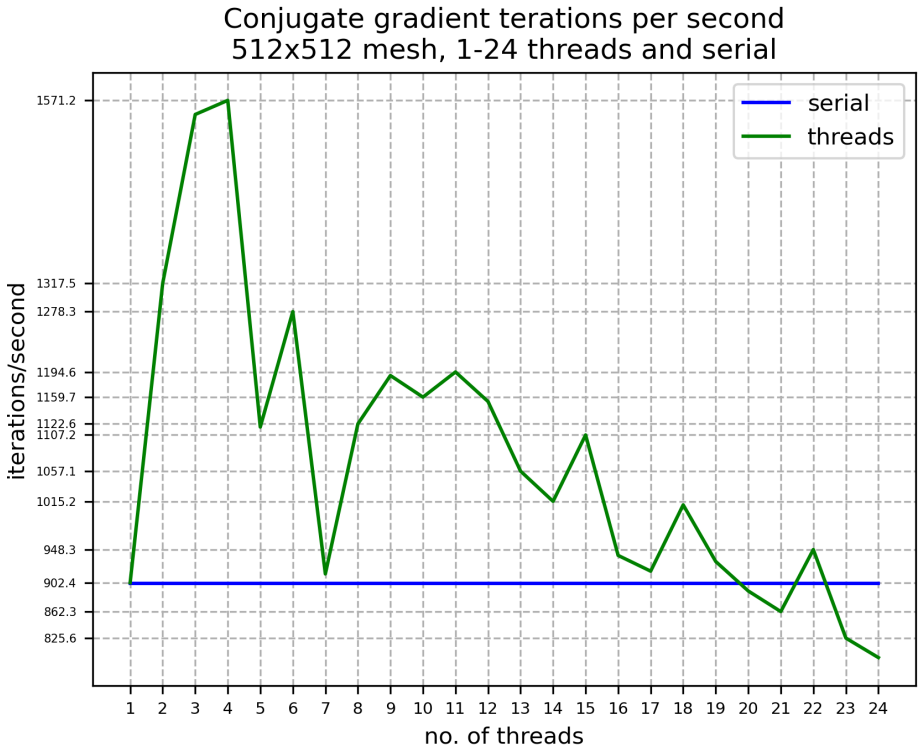

Figure 10: Exercise 2.1, CGi/s 512x512

Finally, for the 1024x1024 test, thread overhead hardly exceeds the computational weight in the serial version. In fact, the benefit of paralleling is such that even for higher number of threads (running on a 12 core device), the optimized version still has far lower running times.

With the same reasoning applied to prior cases, we can guess that the number of CGi/s is always greater when compared to the serial version.

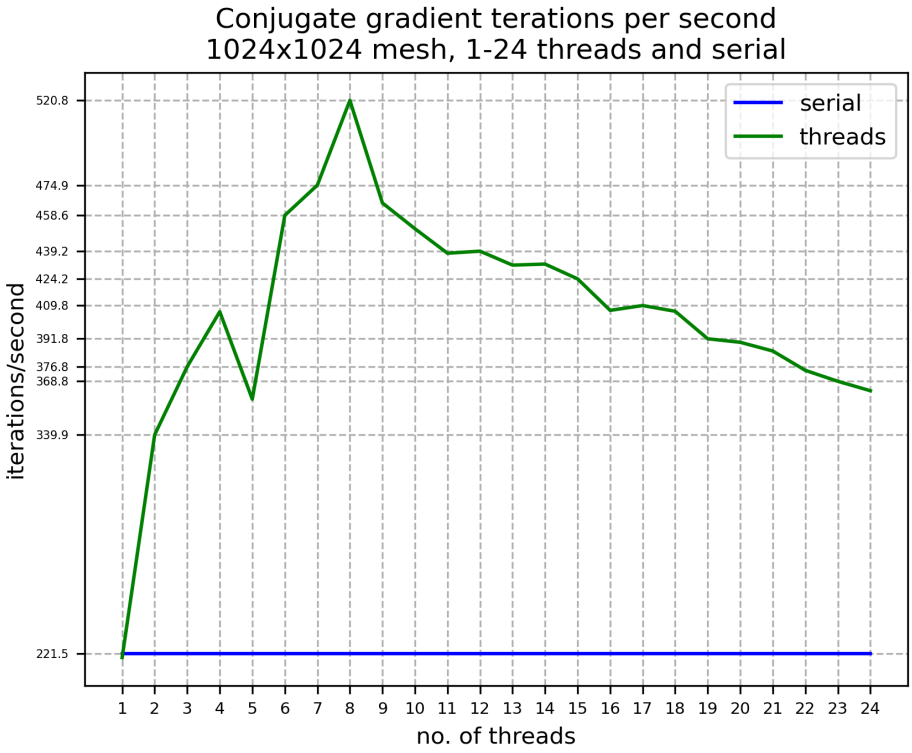

Figure 11: Exercise 2.1, CGi/s 1024x1024

Regarding the last argument proposel for a threaded OpenMP PDE solver producing bitwise-identical results, we should first understand in what case they might not.

Computational floating point arithmetic, in general, lacks commutative and associative properties. This happens because the finite representation of a decimal number (the precision) is limited.

Taking the prior statement into account, if there's blocks of code to which an OpenMP uses a reduction clause, we can understand how the final result can be different from execution to execution with the same starting conditions. The reason why this can happen it's because the nature of thread scheduling is nondeterministic.

Hence, such implementations where cross dependencies are present, reassuring bitwise-identical results implies some sort of parallel side effect.

2.2. Weak scaling [10 Points]

We have been using at most 24 threads and mesh sizes in a square fashion of 64, 128, 256, 512 and 1024. When we double the mesh size in each dimension, this equates to multiplying the total size by for 2^2. In turn, for every growing dimension in terms of double, the number of threads should be multiplied by 2^2. This ensures that the workload for each thread is the same across every step.

For this section of the exercise I performed an analysis beginning in a 64x64 mesh, with starting numbers of threads 1, 2 and 4. I limited the initial number of threads by before because the computational burden of thread overhead for initial thread number above 4 gets too high. For the same reason a limiting mesh size of 512x12 had to be imposed.

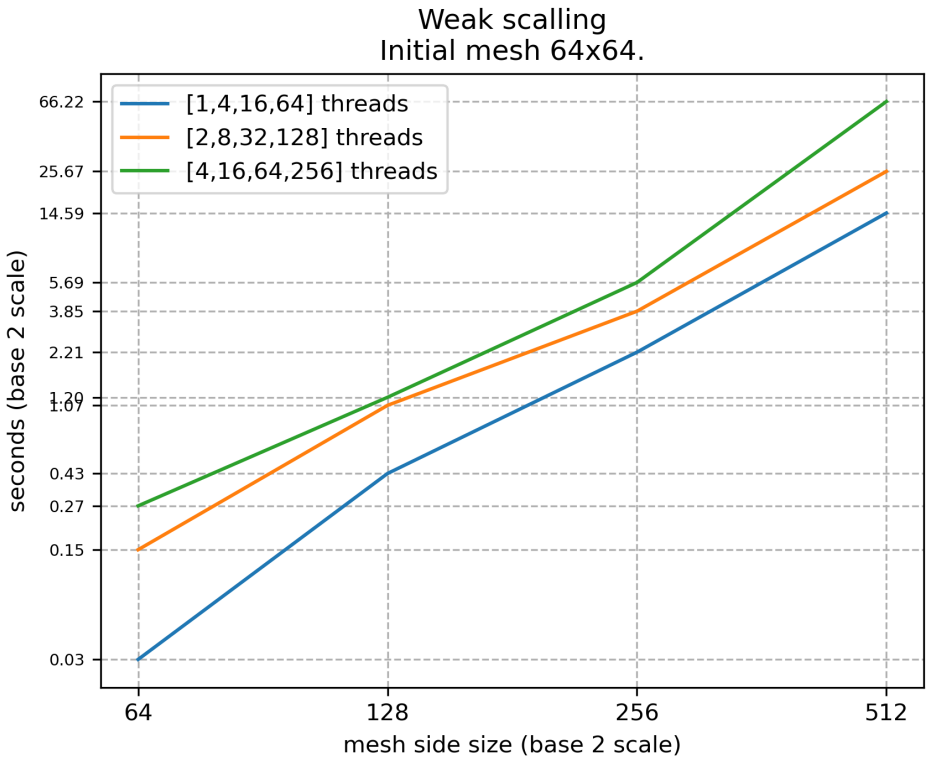

Figure 12: Exercise 2.2, initial mesh 64x64

It is note worthy that the plot in Figure 12 has a base 2 logarithmic scale in both the x-axis and y-axis. By analyzing the plot we can infer that in any of the initial number of threads case, the plot follows a growth in linear fashion (in the log scale). This means that the computation time, with the same work load per thread, tends to grow exponentially in the range of 64 to 512 squared mesh size.

One could still argue that this wouldn't be the case with mesh size raging from 512x512 to 1024x1024 because the paralleled version had better performance in these cases on the strong scaling analysis. So I computed the same analysis only now starting at a mesh size of 256x256.

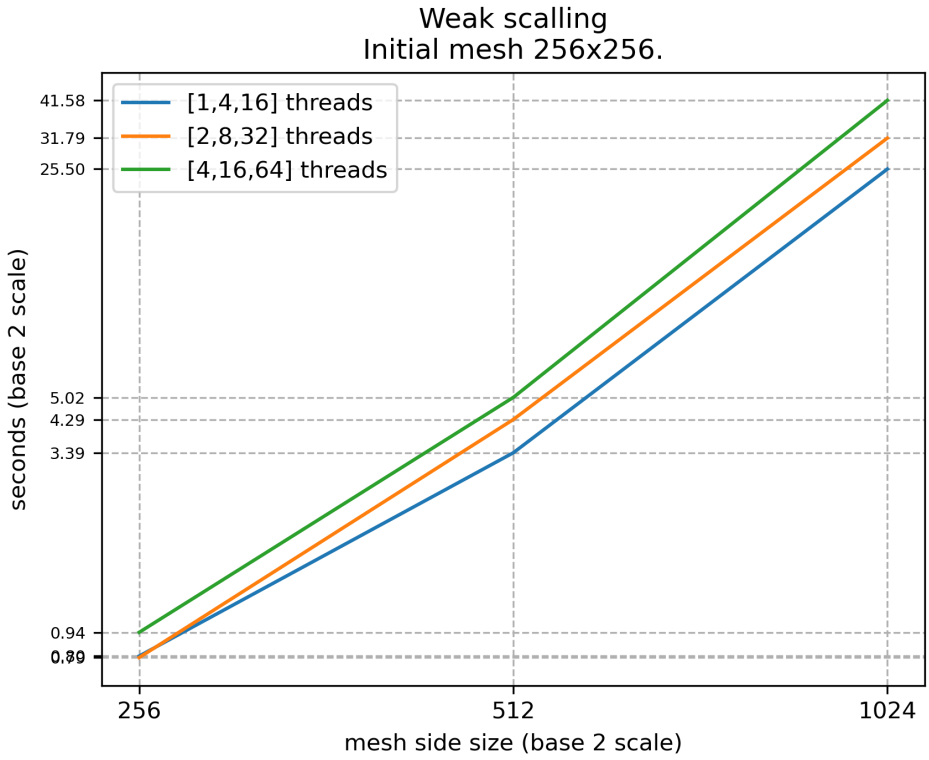

Figure 13: Exercise 2.2, initial mesh 256x256

Just like in Figure 12, the paralleled program still grows in an exponential fashion. Attending to this analysis we can infer that the implementation as it is doesn't comply to a good scaling on an equal work load per thread basis, on a machine of 12 cores, namely the MacBook 16.

3. Bonus Question [5-10 Points]

SIMD instructions, or Single Instruction Multiple Data, leverage the optimization of operations over vectors/matrices over CPU cycles.

If we were to employ the use of SIMD instructions, vectoring all necessary data structures, we could potentially make the implementation faster. Furthermore, it could be the case that using OpenMP for multi-threading and SIMD instructions could greatly increase computational and time efficiency. These factors are, however, dependant on a deeper analysis of the implementation at subject and continuous code profiling with performance analysis tools.