Università della Svizzera italiana

Institute of Computing
CI

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

Student: Jonatan Bella Discussed with: None

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

1.1. Implement the missing linear algebra functions in linalg.cpp:

1.1.1. Norm 2

For this implementation, I took advantage of the dot product function already implemented hpc_dot.

```
double hpc_norm2(Field const& x, const int N) {
  double result = 0;
  result = hpc_dot(x,x,N);
  return sqrt(result);
}
```

Since we can think of the L2 norm as the inner product of a vector with itself.

1.1.2. Fill

I used a simple for loop to set the entries in a vector to a value.

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

1.1.3. 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 over the vectors and adds the scaled vector x to y.

1.1.4. add 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 over the vectors and adds the scaled difference of l and r to x and stores the result in y.

1.1.5. 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];
    }
}
```

1.1.6. Linear Combination

```
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 over the vectors and adds the scaled vector x (by alpha) to the scaled vector z (by beta) and stores the result in y.

1.1.7. 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 over the vectors and copies the vector x to y.

1.2. Implement the missing stencil kernel in operators.cpp:

The stencil kernel fragment for the interior points is implemented as follows:

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

Following the equation as presented in the assignment. Notice that since we are not in the boundary, we can use directly the adjacent s values.

1.3. Results

Finally, I ran the program with the functions implemented before with the provided parameters:

```
./main 128 100 0.005
```

Using the node in an exclusive manner, with the following results:

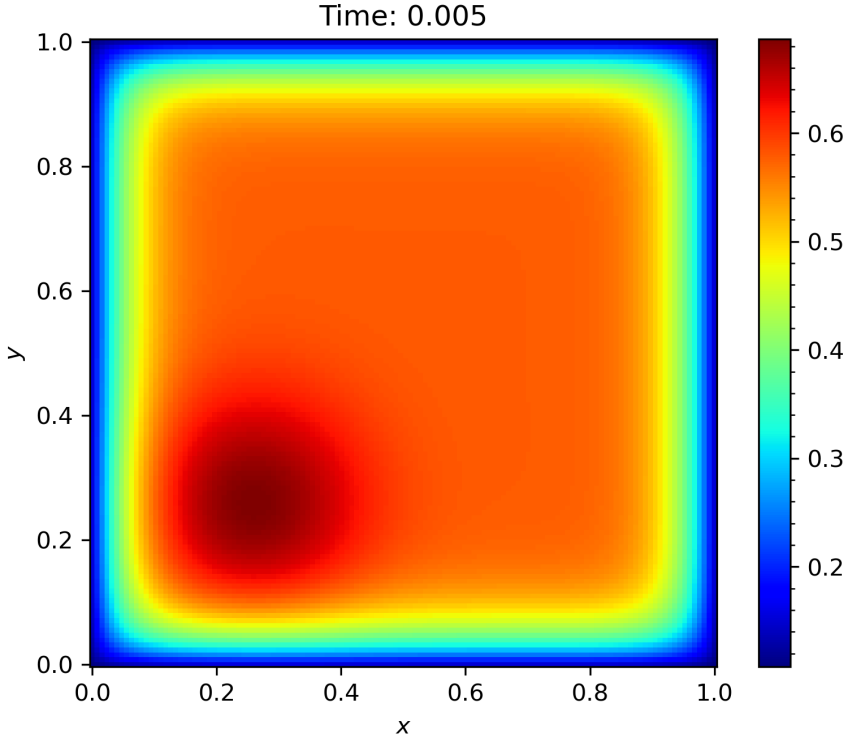

Figure 1: Contour plot of our simulation

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

2.1. Welcome message OpenMP

To print out the number of threads used if OpenMP is enabled, I implemented the following code:

```
#ifdef _OPENMP
threads = omp_get_max_threads();
std::cout << "version --- :: C++ OpenMP" << std::endl;
std::cout << "threads --- :: " << threads << std::endl;
#else
std::cout << "version --- :: C++ Serial" << std::endl;
#endif
```

Following the suggestions of the assignment with the use of macros. Such that we get the number of maximum threads with the function $omp_get_max_threads()$ if OPENMP is enabled. Otherwise, we print out that the program is running in serial.

2.2. Parallelize the linear algebra functions

In general, Almost all the function can be parallelized straightforwardly with the use of OpenMP in this case. With the exception of the dot product one since it can lead to race conditions given that the variable *result* is updated in parallel and accumulated. To solve this, we use the reduction clause in the parallel region.

2.2.1. Dot product OPENMP directive

```
#pragma omp parallel for default(none) shared(x,y,N) reduction(+:result)
```

As mentioned before, the dot product implementation can lead to a race condition if we do not manage the shared variable *result* properly. To solve this, we use the reduction clause. In addition, as suggested in class, I use the default(none) clause to make sure that all the variables are declared in the parallel region as a good practice. Therefore, we need to share the read variables x, y and N.

2.2.2. L2 norm

No changes since the function calls for hpc_dot which is already parallelized.

2.2.3. Fill

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

There is no possible raze condition since we are not updating the same variable in parallel. I share the corresponding used variables.

2.2.4. axpy

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

Same reason as the fill function, no race condition since the "accumulation" corresponds to the same iteration. Such that the updates are independent. I share the corresponding used variables.

2.2.5. add scaled diff

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

2.2.6. scale diff

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

2.2.7. Linear Combination

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

2.2.8. Copy

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

2.3. Operators.cpp parallelization

The parallelization of the operators.cpp file is also straightforward since we can parallelize each of the stencil kernel fragments (that have a loop) without any risk of a race condition since there are no accumulation ongoing for the defined variables. Therefore, we can share all variables that are needed on each for loop, except the indices of the loop that are managed by the pragma directive.

2.3.1. Interior points

```
#pragma omp parallel for default(none) shared(f,s,alpha,beta,y_old,jend,iend)
collapse(2)
```

In this case, we can use the collapse clause to collapse the two loops into one. [1] Which allow us to parallelize both loops, outer and inner ones.

2.3.2. East boundary

```
#pragma omp parallel for default(none)
shared(alpha,beta,s_old,s_new,f,jend,bndE,i)
```

In this case, and the same for the next ones, is a simple for loop parallelization without potential race conditions. We need to share the respective used variables given that we are using default(none) clause.

2.3.3. West boundary

```
#pragma omp parallel for default(none)
shared(alpha,beta,s_old,s_new,f,jend,bndW,i)
```

2.3.4. North boundary

```
#pragma omp parallel for default(none)
shared(alpha,beta,s_old,s_new,f,iend,bndN,j)
```

2.3.5. South boundary

```
#pragma omp parallel for default(none)
shared(alpha,beta,s_old,s_new,f,iend,bndS,j)
```

[1] https://stackoverflow.com/questions/13357065/how-does-openmp-handle-nested-loops

2.4. Bitwise identical results

In relation to this, and following the openmp specification [2],we can say that it is practically challenging to obtain bitwise identical results. This happens since the order on which the values are combined are unspecified and different rounding errors may be introduced when dissimilar-sized values are joined in a different order. Therefore, there is a trade-off in enforcing some ordering against parallel efficiency. However, slight variations provided by floating point operations and ordering of operations can be acceptable, even more if the difference on running time is substantial.

2.5. Strong Scaling

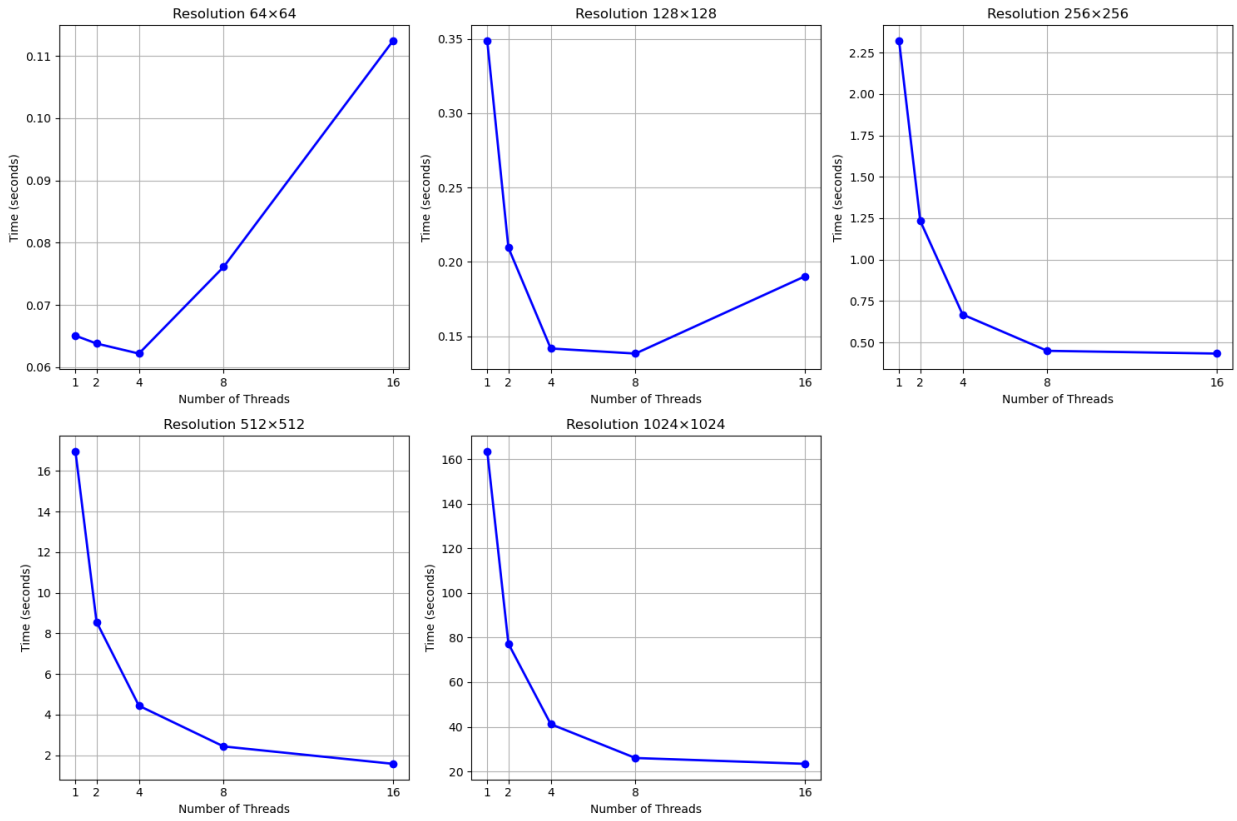

Figure 2: Strong scaling plot for each resolution and thread number

As we can see, there a substantial and increasing improvement relative to the amount of data processed. For example, the parallelization can have a small benefit or even negative effect if the size of the problem is small enough to not compensate for the overhead costs of the parallelization as it happen with 64x64 example.

Table 1: Scaling Analysis for Resolution 64×64

Threads	Time (s)	Speedup	Efficiency
1	0.065	1.000	1.000
2	0.064	1.020	0.510
4	0.062	1.046	0.262
8	0.076	0.855	0.107
16	0.112	0.579	0.036

[2] `https://www.openmp.org/spec-html/5.2/openmpsu50.html`

However, the benefits of the parallelization increased with the size. "Diluting" more the overhead costs. Justifying higher threads numbers. You can notice more in detail the improvement in efficiency and Speedup:

Table 2: Scaling Analysis for Resolution 128×128

Threads	Time (s)	Speedup	Efficiency
1	0.349	1.000	1.000
2	0.210	1.663	0.832
4	0.142	2.458	0.614
8	0.138	2.519	0.315
16	0.190	1.832	0.114

Table 3: Scaling Analysis for Resolution 256×256

Threads	Time (s)	Speedup	Efficiency
1	2.320	1.000	1.000
2	1.233	1.881	0.941
4	0.668	3.471	0.868
8	0.449	5.168	0.646
16	0.433	5.365	0.335

Table 4: Scaling Analysis for Resolution 512×512

Threads	Time (s)	Speedup	Efficiency
1	16.974	1.000	1.000
2	8.560	1.983	0.991
4	4.436	3.826	0.957
8	2.432	6.980	0.873
16	1.575	10.775	0.673

Table 5: Scaling Analysis for Resolution 1024×1024

Threads	Time (s)	Speedup	Efficiency
1	163.546	1.000	1.000
2	77.248	2.117	1.059
4	41.202	3.969	0.992
8	26.020	6.285	0.786
16	23.414	6.985	0.437

Notice that for obtaining this results I constructed a bash file (*strong_scaling.sh*) to automate the process of running the program with the different resolutions and thread numbers. An important remark is that I used the exclusive node in the *sbatch* file to ensure that the whole node is used by the program, otherwise, the results would seem as that the parallelization would not be helping.

2.6. Weak Scaling

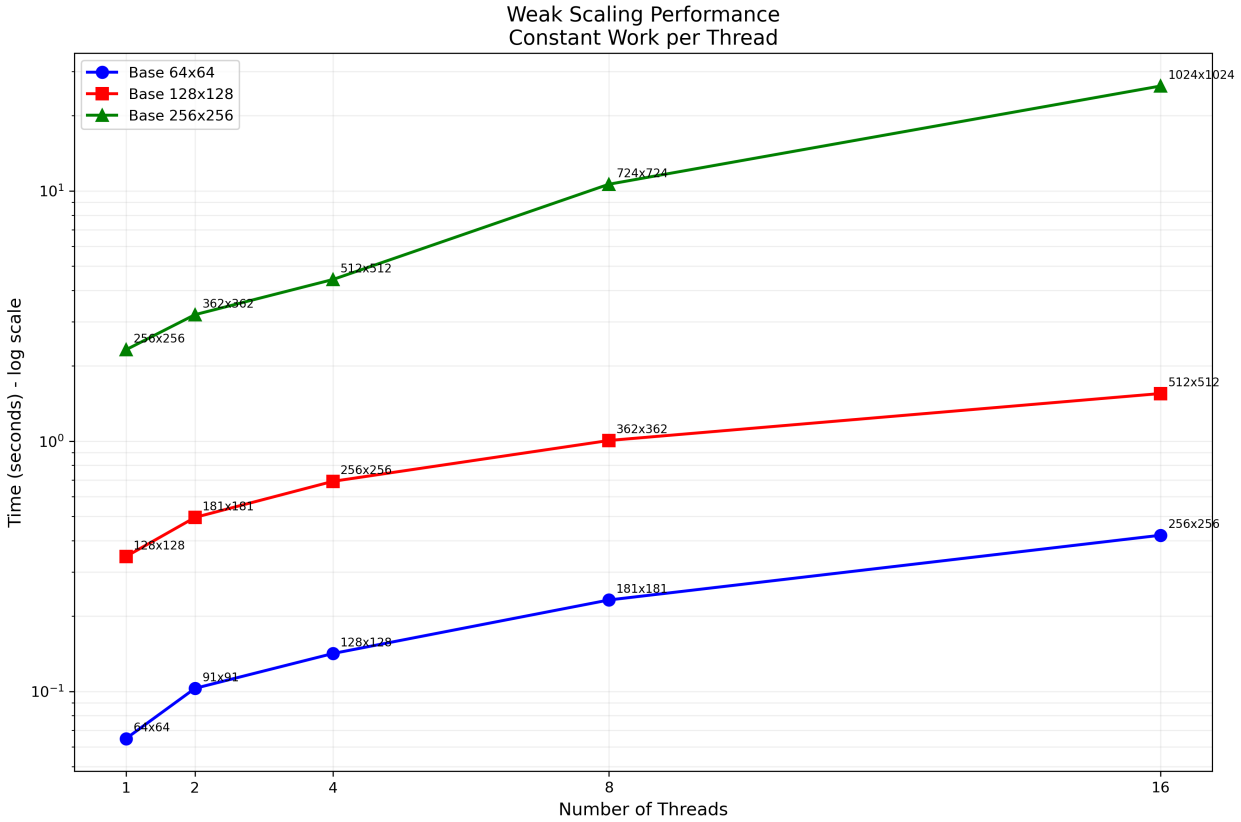

Figure 3: Weak scaling plot for a constant work per thread

Given a base size, we compute the size that keep the work per thread, with the increased number of thread, constant:

- Base 64: 4100 work per thread
- Base 128: 16400 work per thread
- Base 256: 65600 work per thread

In an idea situation we would expect that doubling the threads and the grid size would stay roughly constant because each thread would handle the extra work. However, each thread adds an overhead cost since each thread share variables, needs to synchronize in reductions, etc.

To implement this I modify the previous batch file (*weak_scaling.sh*) to compute the grid size based on the work per thread and the base initial grid size such that it keeps the work per thread constant with the provided expression. Initially I tried to use the *sqrt* of the number of threads, but it was not working as expected. [3] Therefore, I made use of cases with precomputed square root values [4]

[3] `https://www.unix.com/unix-for-dummies-questions-and-answers/152680-sqrt-bash.html`
`https://stackoverflow.com/questions/12722095/how-do-i-use-floating-point-arithmetic-in-bash`
`https://ryanstutorials.net/bash-scripting-tutorial/bash-functions.php`

[4] `https://linuxize.com/post/bash-case-statement/`

3. Task: Quality of the Report [15 Points]

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your OpenMP solutions.
  - your write-up with your name project_number_lastname_firstname.pdf,
- Submit your .tgz through Icorsi.