Università della Svizzera italiana

Institute of Computing CI

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

Student: Zitian Wang Discussed with: Ning Ding; Xiaorui Wang

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

1.1. Implement the functions

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

    // TODO
    if(N == 0){
        return 0.0;
    }
    for(int i = 0; i < N; i++){
        result += x[i] * x[i];
    }

    return sqrt(result);
}
```

Listing 1: HPC Norm Calculation

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

Listing 2: HPC Fill Function

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

Listing 3: HPC AXPY Function

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

Listing 4: HPC Add Scaled Difference Function

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

Listing 5: HPC Scaled Difference Function

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

Listing 6: HPC Scale Function

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

Listing 7: HPC Linear Combination Function

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

Listing 8: HPC Copy Function

1.2. Implement the kernel

```
// the interior grid points
for (int j = 1; j < jend; j++) {
    for (int i = 1; i < iend; i++) {
        // TODO
        // f(i,j) = ...
        f(i, j) = -(4.0 + 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)
              + alpha * s_old(i, j)
              + beta * s_new(i, j) * (1.0 - s_new(i, j));
    }
}
```

Listing 9: Interior Grid Points Calculation

1.3. Plot the solution

```
[wangzi@icsnode30 mini_app]$ ./main 128 100 0.005
======================================================================
                      Welcome to mini-stencil!
version   :: C++ Serial
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
======================================================================
----------------------------------------------------------------------
simulation took 0.223021 seconds
1514 conjugate gradient iterations, at rate of 6788.61 iters/second
300 newton iterations
----------------------------------------------------------------------
### 1, 128, 100, 1514, 300, 0.223021 ###
Goodbye!
```

Listing 10: Mini-Stencil Program Output

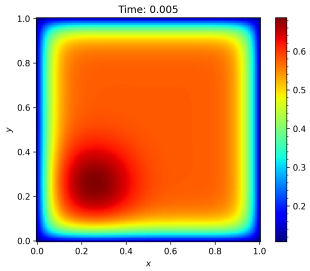

Figure 1: The population concentration at time t = 0.005

2. Task: Adding OpenMP to the nonlinear PDE mini-app

2.1. Replace the welcome message

```
[wangzi@icsnode33 mini_app]$ ./main 128 100 0.005
================================================================================
                        Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 4
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.145261 seconds
1515 conjugate gradient iterations, at rate of 10429.5 iters/second
300 newton iterations
--------------------------------------------------------------------------------
### 4, 128, 100, 1515, 300, 0.145261 ###
Goodbye!
```

Listing 11: Mini-Stencil Program Output with OpenMP

2.2. Linear algebra kernel

To avoid conflict between different threads updating the same value at the same time, the parallel function for `hpc_dot` and `hpc_norm2` uses:

`# pragma omp parallel for reduction (+: result)`

Other functions use:

`# pragma omp parallel for.`

```
double hpc_dot(Field const& x, Field const& y, const int N) {
    double result = 0;
    #pragma omp parallel for reduction(+:result)
    for (int i = 0; i < N; i++) {
        result += x[i] * y[i];
    }

    return result;
}
```

Listing 12: HPC Dot Product Function with Parallelization

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

    // TODO
    if(N == 0){
        return 0.0;
    }
    #pragma omp parallel for reduction(+:result)
    for(int i = 0; i < N; i++){
        result += x[i] * x[i];
    }

    return sqrt(result);
}
```

Listing 13: HPC Norm Calculation Function with Parallelization

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

Listing 14: HPC Fill Function with Parallelization

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

Listing 15: HPC AXPY Function with Parallelization

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

Listing 16: HPC Add Scaled Difference Function with Parallelization

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

Listing 17: HPC Scaled Difference Function with Parallelization

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

Listing 18: HPC Scale Function with Parallelization

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

```
}
```

Listing 19: HPC Linear Combination Function with Parallelization

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

Listing 20: HPC Copy Function with Parallelization

2.3. The diffusion stencil

For code with double loop, should use `#pragma omp parallel for collapse(2)` to parallel.

```
  // the interior grid points
  #pragma omp parallel for collapse(2)
  for (int j = 1; j < jend; j++) {
      for (int i = 1; i < iend; i++) {
          // TODO
          // f(i,j) = ...
          f(i, j) = -(4.0 + 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)
                + alpha * s_old(i, j)
                + beta * s_new(i, j) * (1.0 - s_new(i, j));
      }
  }
```

Listing 21: Parallelized Interior Grid Points Calculation with Collapse

For code with independence loop, use `#pragma omp parallel for` for parallel. The role of boundary loops is to impose these boundary conditions on the boundaries of the mesh so that the entire simulation conforms to the real physical boundaries.

```
  // east boundary
  {
      int i = nx - 1;
      #pragma omp parallel for
      for (int j = 1; j < jend; j++) {
          f(i, j) = -(4.0 + alpha) * s_new(i, j)
                  + s_new(i - 1, j) + bndE[j]
                  + s_new(i, j - 1) + s_new(i, j + 1)
                  + alpha * s_old(i, j)
                  + beta * s_new(i, j) * (1.0 - s_new(i, j));
      }
  }

  // west boundary
  {
      int i = 0;
      #pragma omp parallel for
      for (int j = 1; j < jend; j++) {
          f(i, j) = -(4.0 + alpha) * s_new(i, j)
                  + bndW[j]       + s_new(i + 1, j)
                  + s_new(i, j - 1) + s_new(i, j + 1)
                  + alpha * s_old(i, j)
                  + beta * s_new(i, j) * (1.0 - s_new(i, j));
```

```
    }
}
```

Listing 22: Parallelized Boundary Calculation for East and West Boundaries

Since each corner point is a different, independent computation, use `#pragma omp parallel sections` to compute each corner point separately in parallel.

```
// north boundary (plus NE and NW corners)
#pragma omp parallel sections
{
    #pragma omp section
    {
        int j = nx - 1;
        int i = 0; // NW corner
        f(i, j) = -(4.0 + alpha) * s_new(i, j)
                + bndW[j]      + s_new(i + 1, j)
                + s_new(i, j - 1) + bndN[i]
                + alpha * s_old(i, j)
                + beta * s_new(i, j) * (1.0 - s_new(i, j));
    }

    #pragma omp section
    {
        int j = nx - 1;
        for (int i = 1; i < iend; i++) {
            f(i, j) = -(4.0 + alpha) * s_new(i, j)
                    + s_new(i - 1, j) + s_new(i + 1, j)
                    + s_new(i, j - 1) + bndN[i]
                    + alpha * s_old(i, j)
                    + beta * s_new(i, j) * (1.0 - s_new(i, j));
        }
    }

    #pragma omp section
    {
        int j = nx - 1;
        int i = nx - 1; // NE corner
        f(i, j) = -(4.0 + alpha) * s_new(i, j)
                + s_new(i - 1, j) + bndE[j]
                + s_new(i, j - 1) + bndN[i]
                + alpha * s_old(i, j)
                + beta * s_new(i, j) * (1.0 - s_new(i, j));
    }
}

// south boundary (plus SW and SE corners)
#pragma omp parallel sections
{
    #pragma omp section
    {
        int j = 0;
        int i = 0; // SW corner
        f(i, j) = -(4.0 + alpha) * s_new(i, j)
                + bndW[j] + s_new(i + 1, j)
                + bndS[i] + s_new(i, j + 1)
                + alpha * s_old(i, j)
                + beta * s_new(i, j) * (1.0 - s_new(i, j));
    }

    #pragma omp section
```

```
54        {
55            int j = 0;
56            for (int i = 1; i < iend; i++) {
57                f(i, j) = -(4.0 + alpha) * s_new(i, j)
58                        + s_new(i - 1, j) + s_new(i + 1, j)
59                        + bndS[i]       + s_new(i, j + 1)
60                        + alpha * s_old(i, j)
61                        + beta * s_new(i, j) * (1.0 - s_new(i, j));
62            }
63        }
64
65        #pragma omp section
66        {
67            int j = 0;
68            int i = nx - 1; // SE corner
69            f(i, j) = -(4.0 + alpha) * s_new(i, j)
70                    + s_new(i - 1, j) + bndE[j]
71                    + bndS[i]       + s_new(i, j + 1)
72                    + alpha * s_old(i, j)
73                    + beta * s_new(i, j) * (1.0 - s_new(i, j));
74        }
75    }
```

Listing 23: Parallelized North and South Boundaries Calculation with Sections

2.4. Bitwise Identical Results in OpenMP PDE Solvers

In general, it is difficult to guarantee bitwise-identical results when using a threaded OpenMP PDE solver. The primary reason lies in the **non-associativity of floating-point arithmetic**:

$$(a + b) + c \neq a + (b + c) \tag{1}$$

In sequential execution, floating-point operations follow a fixed order, whereas in parallel execution, the order of operations can change depending on thread scheduling. When using OpenMP, the `reduction` clause creates local copies of the variable for each thread, leading to discrepancies in the order of summation. The final combination of these local results is non-deterministic, which can lead to small variations due to rounding.

Moreover, the use of **parallel reduction in OpenMP** makes achieving identical results particularly challenging:

- **Thread Scheduling:** OpenMP schedules threads differently across runs, leading to variations in floating-point addition.
- **Hardware and Compiler Effects:** Different compilers, compiler options, and hardware architectures can produce different rounding behaviors.

In conclusion, it is generally not feasible to achieve bitwise-identical results across parallel runs of an OpenMP-based PDE solver due to floating-point arithmetic limitations. Ensuring strict ordering would require serial reduction, significantly reducing parallel efficiency, thus making it impractical for performance-critical applications.

2.5. Strong scaling

Strong scaling refers to the study of the performance improvement when increasing the number of threads while keeping the problem size constant. In our analysis, we considered different grid resolutions, specifically 64×64, 128×128, 256×256, 512×512, and 1024×1024. We ran the simulation using $N_{CPU} = 1, 2, 4, 8, 16$ threads and recorded the solution time for each configuration.

The figure below shows the relationship between the number of threads and the time to solution for different resolutions. Ideally, as the number of threads increases, the time to solution should decrease proportionally, demonstrating good scalability.

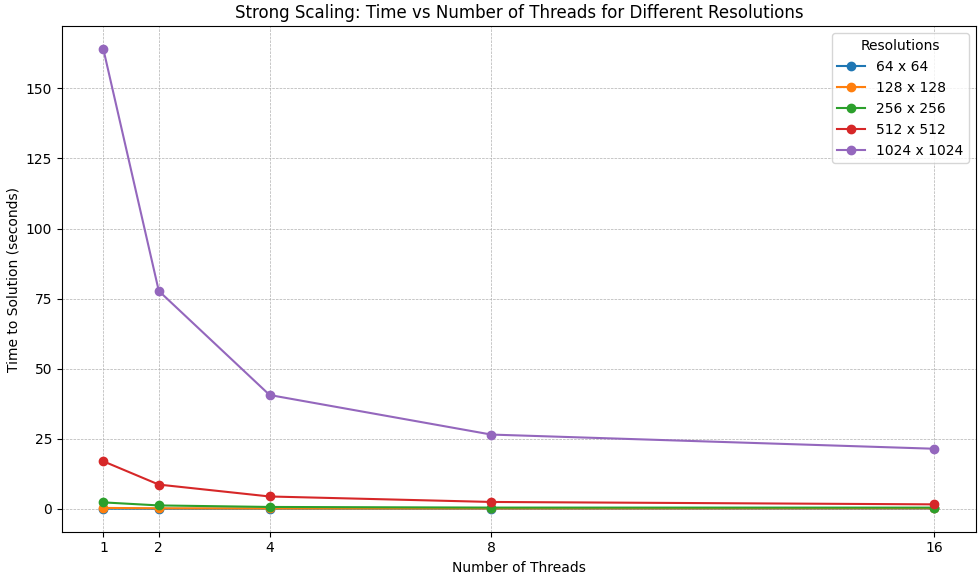

Figure 2: Strong Scaling: Time vs Number of Threads for Different Resolutions. The plot shows how the time to solution decreases with the increasing number of threads, at various resolutions from 64×64 to 1024×1024.

The strong scaling results indicate that as the number of threads increases, the time to solution does decrease; however, the scaling efficiency diminishes beyond a certain point. For lower resolutions (e.g., 64×64), the overhead of managing multiple threads leads to suboptimal performance, particularly when more threads are used. This effect is less pronounced at higher resolutions (e.g., 1024×1024), where there is enough workload to justify the usage of additional threads.

Moreover, the diminishing returns observed as the number of threads increases can be attributed to the overheads associated with thread synchronization and the limited parallelism available in smaller problem sizes. This is consistent with Amdahl's Law, which states that the speedup of a parallel program is limited by the fraction of the code that cannot be parallelized.

The performance bottlenecks observed suggest that careful consideration must be given to the balance between problem size and the number of threads. For small problem sizes, increasing the number of threads beyond a certain point may yield little to no benefit and can even degrade performance due to increased communication overhead.

2.6. Weak scaling

Weak scaling refers to the study of performance as both the number of threads and the problem size increase proportionally, thereby maintaining a constant workload per thread. In this analysis, we explored different base resolutions (64×64, 128×128, 256×256) and measured the time to solution using $N_{CPU} = 1, 2, 4, 8, 16$ threads, while proportionally increasing the problem size to maintain a constant load per thread.

The figure below presents the weak scaling results, showing the relationship between the number of threads and the time to solution while maintaining the workload per thread constant.

The weak scaling results demonstrate that, as expected, the time to solution increases gradually with the number of threads and corresponding increase in problem size. Ideally, with perfect weak scaling, the time to solution would remain constant; however, the results reveal a slight increase, especially for larger thread counts.

This behavior can be attributed to several factors:

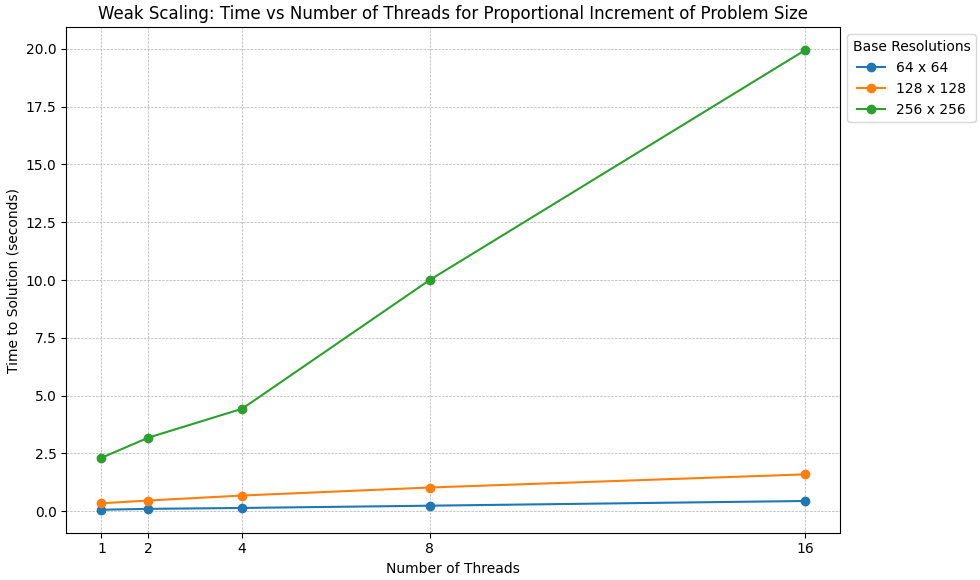

Figure 3: Weak Scaling: Time vs Number of Threads for Proportional Increment of Problem Size. This plot illustrates the effect of increasing both the number of threads and problem size in a balanced manner to maintain constant workload per thread.

- **Communication Overheads**: As the number of threads increases, the communication between threads also increases, leading to overhead that affects overall performance.
- **Memory Access Contention**: More threads can result in increased contention for memory access, leading to performance bottlenecks, particularly for larger resolutions.
- **Synchronization Costs**: Maintaining a consistent workload across multiple threads requires synchronization, which adds overhead as the number of threads increases.

Despite these challenges, the overall weak scaling performance indicates reasonable efficiency, especially for moderate numbers of threads. The gradual increase in runtime with more threads suggests that the solver is able to manage larger workloads effectively, though improvements in minimizing communication overhead could further enhance scalability.

In conclusion, the weak scaling analysis highlights the importance of balancing workload distribution with efficient communication and memory access strategies to achieve optimal performance as the problem size and thread count increase in parallel systems.