Università della Svizzera italiana

Institute of Computing
CI

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

Student: Lorenzo Varese Discussed with: None

Solution for Project 3

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

1. Task: Implementation of Linear Algebra Functions and Stencil Operators [35 Points]

1.1. Implementation of Linear Algebra Functions

In the `linalg.cpp` file, I am tasked with implementing functions that perform basic linear algebra operations.

1.1.1. 2-norm of a vector

`hpc_norm2()` is a function designed to compute the 2-norm (also known as the Euclidean norm) of a given vector. It accepts a vector field `x` and its size `N` as input arguments. The function iterates over each element in the vector, squares it, and then adds it to a running total. The square root of this total is then returned as the 2-norm of the vector.

```
double hpc_norm2(Field const &x, const int N) {
    double result = 0;
    for (int i = 0; i < N; i++) {
        result += x[i] * x[i];
    }
    return sqrt(result);
}
```

1.1.2. Sets entries in a vector to value

`hpc_fill()` is a function designed to populate a given vector field `x` with a specified scalar value. It takes the vector field `x`, the scalar value `value`, and the size `N` of the vector as input parameters. The function iterates over each element in the vector and sets it to the specified scalar 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. Blas level 1 vector-vector operations: $y = alpha * x + y$

`hpc_axpy()` is a function that performs a vector-vector operation, specifically designed to compute $y = \alpha \times x + y$. It takes as input the vector field `y`, the scalar α, the vector field `x`, and the size N of the vectors. The function iterates through each element of the vectors `x` and `y`, and updates the elements of `y` according to the formula $y[i] = \alpha \times x[i] + y[i]$.

```
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];
    }
}
```

1.1.4. Blas level 1 vector-vector operations: $y = x + alpha * (l - r)$

`hpc_add_scaled_diff()` is a function designed to perform a specific vector-vector operation, which calculates $y = x + \alpha \times (l - r)$. The function accepts six arguments: two vector fields `y` and `x`, a scalar α, two more vector fields `l` and `r`, and the size N of these vectors. The function iterates through each element of the vectors, and updates the elements of `y` based on the formula $y[i] = x[i] + \alpha \times (l[i] - r[i])$.

```
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]);
    }
}
```

1.1.5. Blas level 1 vector-vector operations: $y = alpha * (l - r)$

`hpc_scaled_diff()` is a function designed to execute a specialized vector-vector operation, computing $y = \alpha \times (l - r)$. The function takes five parameters: the vector field `y`, a scalar α, and two additional vector fields `l` and `r`, along with the size N of these vectors. The function iterates through each element in the vectors `l` and `r`, computing the scaled difference according to the formula $y[i] = \alpha \times (l[i] - r[i])$, and stores the result in the vector `y`.

```
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]);
    }
}
```

1.1.6. Blas level 1 vector-vector operations: $y = alpha * x$

`hpc_scale()` is a function designed to perform a scaling operation on a vector, effectively computing $y = \alpha \times x$. The function takes four parameters: the vector field `y` where the result will be stored, a scalar α, the original vector field `x`, and the size N of these vectors. The function iterates over each element in the vector `x`, multiplies it by the scalar α, and stores the result in the corresponding entry of the vector `y`.

```
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]; } }
```

1.1.7. Linear combination of two vectors: $y = alpha * x + beta * z$

`hpc_lcomb()` is a function that performs a linear combination of two vectors x and z, scaled by the coefficients α and β, respectively. The result is stored in the vector y. Specifically, the function computes $y = \alpha \times x + \beta \times z$. It takes six parameters: the vector field `y` for storing the result, the scalar α, the original vector field `x`, the scalar β, the second vector field `z`, and the size N of these vectors. The function iterates through each element in vectors x and z, performs the linear combination as specified, and stores the result in the corresponding element of vector y.

```
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];
    }
}
```

1.1.8. Copy one vector into another

`hpc_copy()` is a function designed to copy the elements of one vector x into another vector y. It iterates over each element in the source vector x and assigns its value to the corresponding element in the destination vector y. The function takes three parameters: the target vector field `y`, the source vector field `x`, and the size N of these vectors.

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

1.2. Implementation of Stencil Operations

The core logic of the implementation primarily consists of a function named `diffusion`, which performs stencil operations on a 2D grid.

1.2.1. Core Logic

The stencil operation involves computing values for the interior points as well as the boundary points of the mesh. These are detailed as follows:

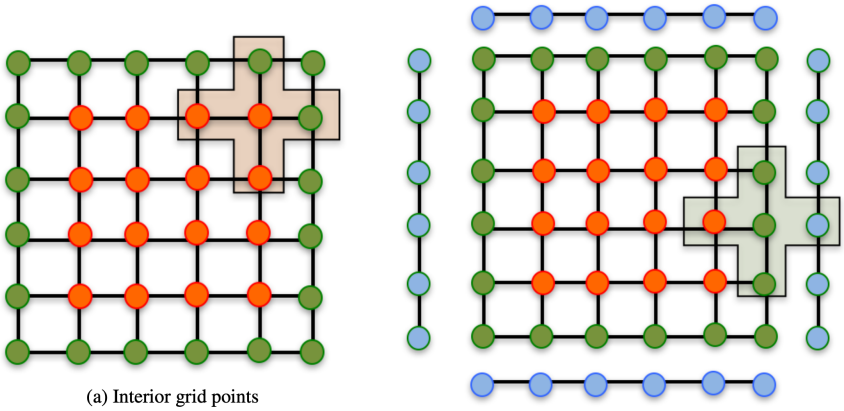
1. **Interior Points**: The inner loop iterates over the grid indices (i, j) for the mesh, updating the $f(i, j)$ field based on the surrounding values of s and additional parameters like α and β.
2. **Boundary Points**: The boundary points are calculated separately for each side (East, West, North, and South) of the grid. The calculations are similar to the interior points but include additional terms for boundary conditions (e.g., `bndE[j]` for the east boundary).

A visual representation can be found in Figure 1. The main difference between the two cases lies in the handling of boundary points. In the formula for inner grid points, all neighboring points contribute to the new value of $f(i, j)$. For boundary points, however, one or more neighboring points are outside the grid. These are removed from the formula and are replaced by fixed boundary values.

1.2.2. Statistics Collection

Lastly, the function updates the `stats::flops_diff` variable to accumulate the number of floating-point operations performed.

Overall, this implementation focuses on updating the field f by leveraging both the immediate neighboring points and some global parameters. This makes it a crucial part for simulations that involve spatial discretization, like solving partial differential equations on a grid.



(a) Interior grid points

(b) Boundary grid points

Figure 1: Representation of stencil operations on a 2D mesh.

1.3. Comparison of Conjugate Gradient and Newton Methods vs Reference

Upon completion of the mini-app implementation, it was requested to compare the results with a given reference output to validate the correctness of the implementation.

The output at the end of the implementation reports the following:

```
======================================================================
                      Welcome to mini-stencil!
version   :: Serial C++
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 200, Newton 50, tolerance 1e-06
======================================================================
----------------------------------------------------------------------
simulation took 0.131906 seconds
1510 conjugate gradient iterations, at rate of 11447.5 iters/second
300 newton iterations
----------------------------------------------------------------------
Goodbye!
```

To consider the implementation as correct, the number of iterations for both the Conjugate Gradient and Newton methods should closely align with those reported in the reference output. Specifically, the observed output indicates 1510 iterations for the Conjugate Gradient method and 300 iterations for the Newton method, which are consistent with the reference values.

1.4. Result Visualization

Before plotting the solution, it was necessary to modify the `plotting.py` script to address certain issues. However, after resolving these, the plot generated closely resembled the reference plot.

The plot, as displayed in Figure 2, shows the heatmap visualization of the solution and confirms the accuracy of our implementation.

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

2.1. Replace Welcome Message in `main.cpp` [2 Points]

The welcome message in the `main.cpp` file has been updated to inform the user about two main aspects:

- That they are running the OpenMP version of the application.

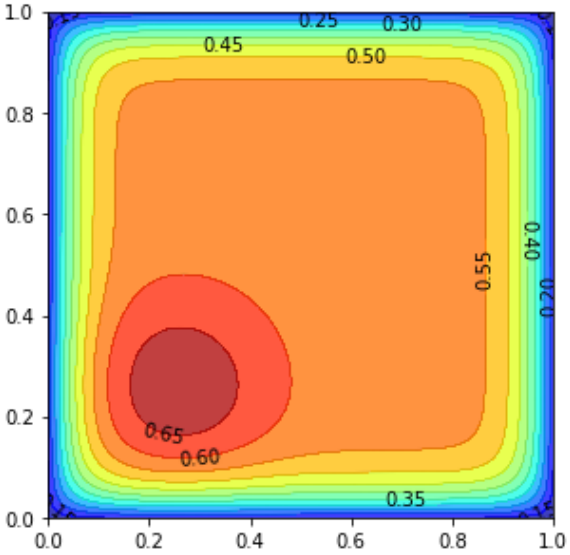


Figure 2: Visualization of the solution.

- The number of threads that the application is using.

```
#pragma omp parallel
{
    num_threads = omp_get_num_threads();
}

std::cout << "============================" << std::endl;
std::cout << "    Welcome to mini-stencil!" << std::endl;
std::cout << "version   :: C++ OpenMP"      << std::endl;
std::cout << "threads   :: " << num_threads << std::endl;
```

This enhancement aims to make the application more informative. Specifically, the number of threads is acquired using the OpenMP function `omp_get_num_threads()` within a parallel region.

2.2. Linear Algebra Kernel [15 Points]

The `linalg.cpp` for linear algebra operations underwent several optimizations by leveraging OpenMP parallelization techniques. Three major points regarding these improvements are discussed below.

2.2.1. Parallelization with Reduction Clause

The functions `hpc_dot` and `hpc_norm2` both employ OpenMP parallelization with reduction clauses. The OpenMP directive `#pragma omp parallel for reduction(+: result)` allows each thread to compute a partial sum, which is eventually reduced to the final result in a thread-safe manner. This optimization is particularly beneficial in improving the computational speed of inner product and norm calculations. The core of the parallelized dot product is encapsulated in the following code snippet.

```
#pragma omp parallel for reduction(+ : result) shared(x, y, N)
for (int i = 0; i < N; i++)
    result += x[i] * y[i];
```

2.2.2. Straightforward Parallelization

Several functions, such as `hpc_fill`, `hpc_axpy`, and `hpc_lcomb`, are straightforwardly parallelized using OpenMP. These optimizations ensure that each core is utilized effectively, thereby speeding up these elementary operations. Here the example related to `hpc_filling`.

```
#pragma omp parallel for default(none) shared(x, N, value)
for (int i = 0; i < N; i++)
{
    x[i] = value;
}
```

2.2.3. Performance Limitations Due to Small Problem Size

Despite the implemented optimizations, it's important to emphasize that any performance gains are potentially offset by the overhead of initializing and managing threads. This is especially true for smaller problem sizes, measured by the number of grid points.

For the mesh sizes required in the assignment, the overhead costs appear to outweigh the advantages of parallelization, leading to negligible or even negative performance impact, as evidenced by Figure 3.

In particular, for a 128x128 grid size, increasing the thread count consistently results in increased runtime. This is likely due to the time required for thread handling outweighing the benefits gained from parallelizing the calculations.

Regarding a 256x256 grid size, a modest performance improvement is observed when employing 2 and 3 threads. However, this gain disappears when a greater number of threads are used. It's noteworthy that even when performance improvements are achieved, they are marginally above 10%. This raises questions about the utility of adding such complexity for a minimal performance increment.

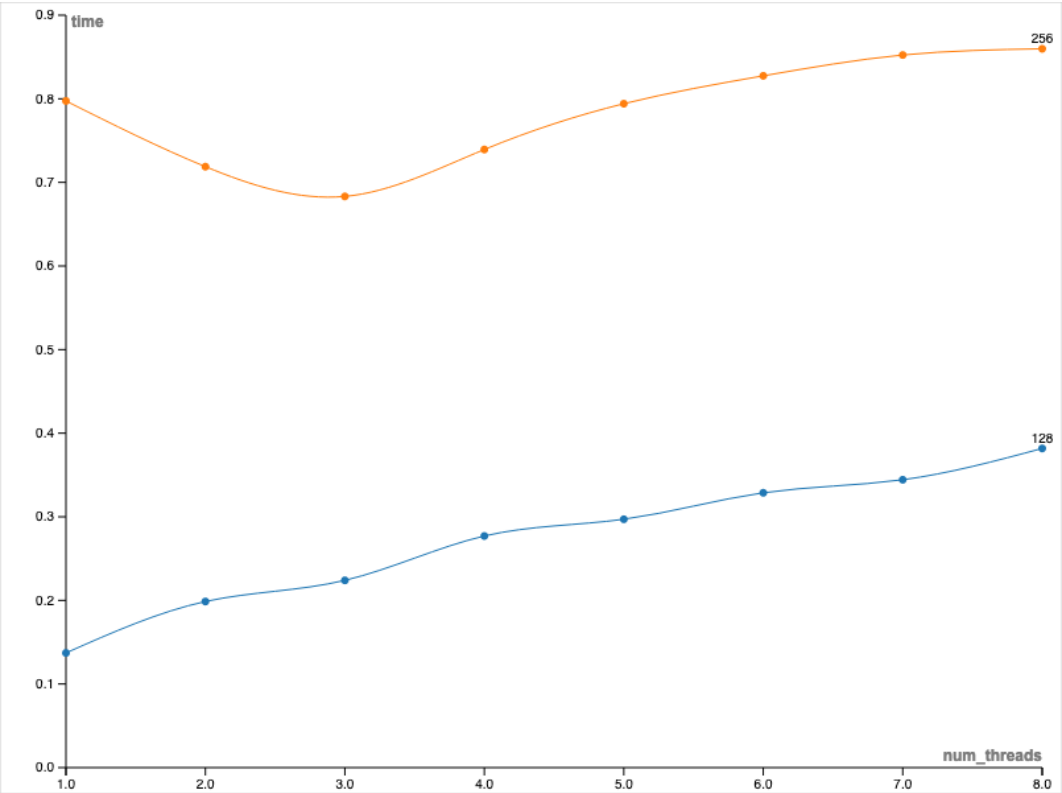


Figure 3: Performance benchmarks indicating the limitations of parallelization for the two required problem sizes.

2.3. The Diffusion Stencil [10 Points]

The nested for loops and the inner grid points serve as primary candidates for optimization in the diffusion stencil. These elements constitute the majority of computational work, thereby offering the highest potential for performance gains.

In regard to the parallelization of the inner grid, OpenMP was utilized to accelerate the computations. The directive `pragma omp parallel for collapse(2) default(shared)` was employed to allow the concurrent execution of the nested loops. By collapsing the loops, task granularity was improved, thereby making it possible to distribute smaller tasks across threads more efficiently. The inner grid points were updated based on a predefined equation, allowing for effective parallelization due to the parallel nature of the computations.

As for the parallelization of the boundary, its computational weight is considerably less than that of the inner grid. Initial efforts were made to parallelize these calculations as well, using dynamic choice of thread number in OpenMP. This approach to dynamic thread allocation was intentionally chosen to balance the benefits of parallelization against the overhead incurred. While the inner grid was parallelized using a predefined value for `OMP_NUM_THREADS`, the parallelization of the boundary points was deliberately restricted to a maximum of two threads. This limitation was imposed to mitigate the risk of excessive overhead, which could potentially negate the advantages gained from parallelizing this less computationally intensive section of the code.

However, these attempts were ultimately omitted from the final implementation due to their marginal impact on performance. To provide quantification, consider a square grid with side length n. The boundary then consists of $4 \times (n-2)$ points, in contrast to $(n-2)^2$ inner grid points. Therefore, the boundary operations are a minor portion of the computational load, justifying their exclusion from the parallelization strategy.

2.4. Strong Scaling [10+3 Points]

The concept of strong scaling is primarily concerned with the solution time for a fixed total problem size, as the number of processing elements (threads, in our context) increases. A strong scaling study, therefore, measures the impact on performance due to distributing a fixed problem across an increasing number of threads.

From the Figure 4, which showcases the time spent on different resolutions as the number of threads increases, several observations can be made:

- For the grid size of 1024, there is a sharp decrease in time spent as the number of threads increases from 1 to 8. Beyond that, the performance plateaus, indicating that the problem reaches the runtime minimum with around 8 threads for this resolution.
- The grid size of 512 depicts an irregular pattern, with time fluctuating as the number of threads increases. This suggests that for this resolution, there might be an overhead or a contention issue as threads increase, causing the non-linear pattern. However, we can observe a good runtime decreasing from 1 to 4 threads.
- For the 256 grid size, the graph indicates a general trend of increasing time as threads increase. This could signify that the overhead of parallelizing such a small grid size is not compensated by the performance gains, leading to diminished returns as more threads are utilized.
- The 128 and 64 grid sizes exhibit similar behavior to the 256 grid size. The graphs for these resolutions indicate that there's an increasing trend in time as the number of threads increases, further underscoring the overhead implications when parallelizing smaller problems.

It's important to note that for smaller grid sizes, the overhead of parallelization (like thread creation and synchronization) may not be offset by the potential performance gains. This might be why we see a trend of increasing time for smaller grid sizes as threads increase.

Another key observation regarding the chart series is that both axes remain consistent across all five size measurements. The x-axis represents the number of threads, while the y-axis indicates the

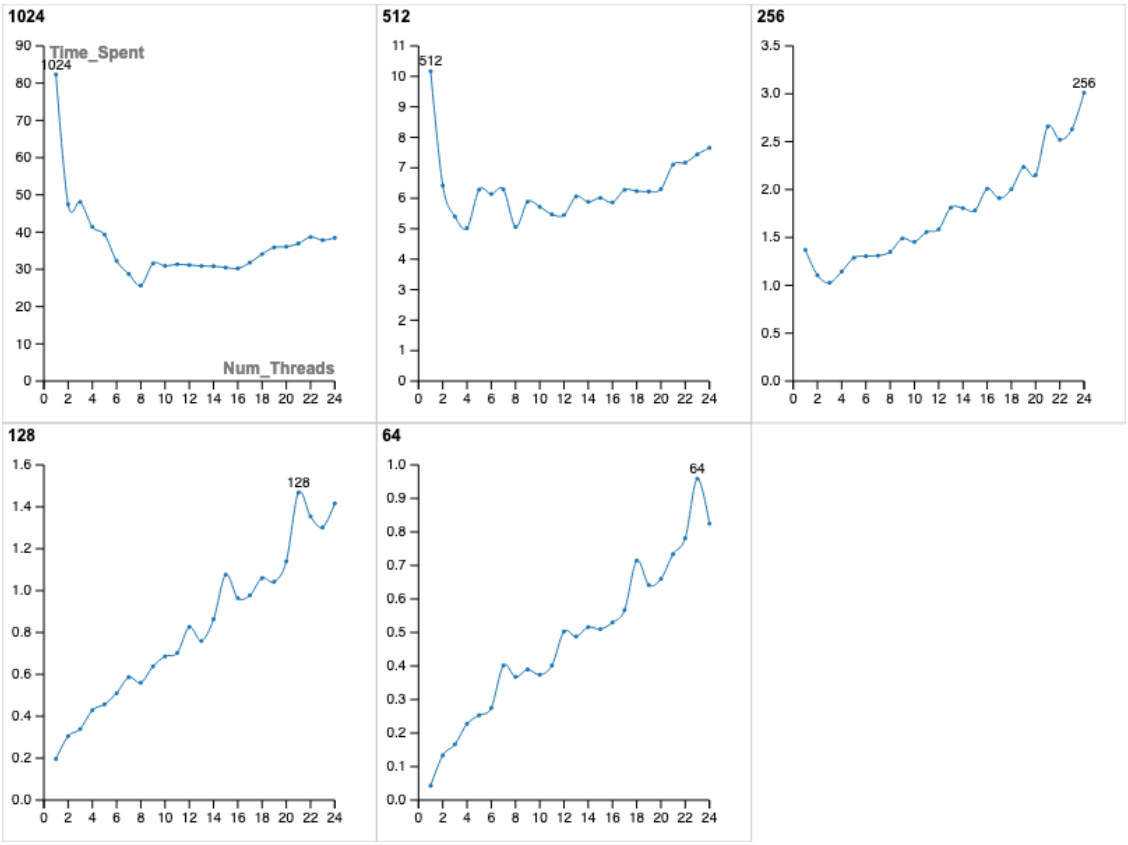


Figure 4: Time taken for different grid sizes against number of threads.

time elapsed in seconds. Nonetheless, the y-axis scale has been adjusted, spanning from 0 to the peak runtime, to clearly highlight the runtime variations across benchmarks with varying thread counts.

An important detail to note is that my local machine is equipped with 8 performance cores. This insight is pivotal in comprehending why, for notably large problem sizes, we observe a parallelization plateau when utilizing a number of threads greater than 8.

In conclusion, strong scaling reveals the limits of parallel efficiency for different problem sizes. As the number of threads increases, there's a point beyond which performance gains diminish, especially for smaller problem sizes. This insight is crucial for deciding the optimal number of threads to employ for varying problem dimensions.

2.4.1. Achieving Bitwise-Identical Results in a Threaded OpenMP PDE Solver

It's theoretically challenging to guarantee bitwise-identical results with a parallel OpenMP PDE solver due to the inherent complexities related to floating-point arithmetic and compiler optimizations.

The primary challenge in achieving bitwise-identical results lies in the presence of functions that perform arithmetic operations, specifically additions and multiplications, on double-precision numbers. Examples of these functions include `hpc_dot()` and `hpc_norm2()`. These arithmetic operations introduce approximation errors that can vary depending on the sequence in which they are executed. One potential solution for obtaining bitwise identical results is to keep these particular functions serial. However, doing so would compromise the goal of fully parallelizing the solver.

2.5. Weak Scaling [10 Points]

In weak scaling studies, the primary focus is the observation of the system's performance when increasing the problem size in proportion to the number of cores. It provides a way to understand

how efficiently a system can handle increased workloads without altering the work per core.

While plots offer a visual representation of performance scaling, the decision to employ a table in this context is driven by the need to concisely highlight critical results and facilitate straightforward comparisons.

The methodology adopted here maintains a consistent work-per-core ratio. As the number of threads increases, the grid size is also enlarged to ensure each thread consistently handles a fixed number of grid points. This approach can be represented mathematically by the equation:

$$n_{pth} = \frac{size \times size}{n_{threads}} \tag{1}$$

Results from the weak scaling tests reveal interesting trends in the system's behavior as both the problem size and thread count grow. The Table 1 showcases the computational time taken for different grid sizes and thread counts while ensuring the number of core-points remains constant.

After having conducted the strong scaling analysis, I understood the critical aspects of executing a multi-threaded analysis on my local machine. Notably, the M2's ARM architecture and the parallelization over small problem sizes presented challenges, as discussed in previous sections. However, the focus of this analysis was to select data that was relevant considering the following constraints: the maximum number of performance cores is 8, and when doubling the grid side size (given that it's a square grid), I would need to quadruple the number of threads to maintain the same number of core-points. Owing to the thread count limitation (max 8), the overhead of parallelization for problems smaller than 256x256, and the fixed core-points constraint, I analyzed the runtime of the four scenarios presented in Table 1.

Grid Size	**Number of Threads**	**Time**	**Number of core-points**
128x128	1	0.151	16384
256x256	4	0.783	16384
256x256	2	0.775	32768
512x512	8	3.567	32768
512x512	2	4.472	131072
1024x1024	8	17.215	131072
1024x1024	2	6.810	524288
2048x2048	8	22.203	524288

Table 1: Weak scaling results highlighting computational time for varying grid sizes and thread counts while maintaining a constant work-per-core ratio.

From the table, it's evident that while the number of points per core remains consistent across different scenarios, there is a significant performance drop observed. This performance decrease causes the run-time to amplify by a factor ranging from 3 to 5 when compared with fewer threads.

In conclusion, the weak scaling analysis revealed the inherent challenges and limitations of multi-threading on the M2's ARM architecture, especially with smaller problem sizes.

3. Faster Implementation Using SIMD Instructions [Bonus 5-10]

Incorporating SIMD (Single Instruction, Multiple Data) instructions can significantly accelerate the computational components of the code. SIMD enables the same operation to be performed on multiple data points simultaneously, thereby enhancing data-level parallelism.

Here's an example of SIMD optimization using OpenMP's `#pragma omp simd` directive:

```
#pragma omp simd
for(int i = 0; i < N; i++) {
  C[i] = A[i] + B[i];
}
```

In cases where the compiler supports Intel's intrinsics, the optimization can be made even more explicit. For instance, using Intel's SSE (Streaming SIMD Extensions):

```
#include <xmmintrin.h>

// Note: A, B, and C are arrays: float A[4], B[4], and C[4]
__m128 vectorA = _mm_loadu_ps(A);
__m128 vectorB = _mm_loadu_ps(B);
__m128 vectorC = _mm_add_ps(vectorA, vectorB);
_mm_storeu_ps(C, vectorC);
```

In the example using Intel intrinsics, `__m128` is a data type that holds four single-precision floating-point numbers. Functions like `_mm_loadu_ps` and `_mm_add_ps` load data into these 128-bit variables and perform addition, respectively.

While SIMD instructions offer significant performance gains, it's important to note that my development environment poses limitations to direct implementation and testing of these optimizations. My local machine, a MacBook Pro with an M2 ARM processor, does not support Intel's intrinsics or SSE extensions. Although ARM architectures do have their own equivalent of SIMD, known as NEON, the syntax and behavior differ from Intel's SIMD instructions, and therefore, porting code or using existing libraries that rely on Intel's intrinsics would be challenging.

However, this limitation does not preclude the theoretical understanding or algorithmic design that would take advantage of SIMD instructions. I can provide a possible implementation of a dot product using `#pragma omp simd` for a vector of floats:

```
float dot_product_simd(float* A, float* B, int N) {
  float sum = 0.0;
  #pragma omp simd reduction(+:sum)
  for(int i = 0; i < N; i++) {
    sum += A[i] * B[i];
  }
  return sum;
}
```