# High Performance Computing: Parallelise deqn with OpenMP

Stella Tee

June 27, 2024

## 1 Introduction

Using parallel programming frameworks such as OpenMP is a critical approach to take advantage of the processing power of contemporary multi-core platforms. This paper explores the complexities of using OpenMP alone for the optimisation of the deqn programme, a simulation tool for solving partial differential equations. The main goals are to analyse the current codebase, add parallelism to important areas, and improve the iteration process strategically. Through an examination of how different thread counts affect the execution time of critical programme elements, the study offers insightful information about how well OpenMP parallelizes particular loops and operations. Moreover, an in-depth examination of OpenMP overhead and conversations regarding the employed scheduling schemes enhance our comprehension of the implemented optimisations.

## 2 Detailed explanation of optimisation performed

### 2.1 Parallelization Strategy

Our chosen parallelization strategy is to use the static scheduling policy for parallelizing loops in the program. This strategy involves dividing the loop iterations into equal-sized chunks at compile-time and assigning each thread a fixed chunk of work. Static scheduling is particularly useful when the workload is evenly distributed across loop iterations, ensuring a balanced distribution of tasks among threads. This approach minimizes the overhead associated with dynamic load balancing and can be beneficial when the loop iterations have roughly uniform execution times. It simplifies the scheduling process and can lead to improved cache locality. The paper discussed for loops with uniformly distributed iterations with static scheduling provide the best performance with the least overhead [3].

### 2.2 Memory access pattern

The memory access pattern of the program is crucial for its overall performance, especially in parallelized applications. In our program, the primary data structures being accessed are the arrays u0 and u1 within the Mesh class. These arrays represent the temperature distribution across the 2D mesh. The nested loops within the ExplicitScheme class, particularly in the diffuse and reset methods, iterate over these arrays.

## 3 Discussion on the iteration scheduling used and Optimizing Workload Distribution with OpenMP Directives

### 3.1 Schedule clause - STATIC

The program is designed in a way that the OpenMP schedule clause is utilized to control the workload distribution among threads for parallel loops. Specifically, a schedule(static) policy is employed in both the diffuse and reset methods within the ExplicitScheme class. The choice of static scheduling aligns well with the relatively uniform workload across iterations in these loops. Since each iteration involves a consistent amount of computation, static scheduling, which divides the iterations into contiguous chunks assigned to each thread at the beginning of the parallel region, is appropriate. This helps achieve load balance and minimizes the overhead associated with dynamic scheduling. The decision to use static scheduling is supported by research indicating its effectiveness in scenarios with a predictable and balanced workload [4].

### 3.2 Collapse clause

The OpenMP collapse clause is employed in the diffuse and reset methods within the ExplicitScheme class, collapsing the nested loops to enhance workload distribution and cache efficiency. The decision to collapse these loops is justified by the fact that the workload within each iteration is relatively uniform, and collapsing nested loops can lead to improved data locality. This can be especially beneficial in scenarios where adjacent elements in the 2D arrays are accessed, as it helps exploit spatial locality and reduces potential cache misses. The impact of loop collapsing on workload distribution and performance aligns with the underlying mesh structure and the nature of heat transfer computations. Research on loop collapsing in parallel programming supports its utilization in cases where the workload within iterations is well-balanced and conducive to enhanced cache efficiency [1].

## 4 Analysis

### 4.1 Time Measurement

To accurately measure the execution time of pertinent loops and functions, OpenMP timers, specifically `omp_get_wtime()`, were employed. The crucial loops for measurement comprised the diffusion (`diffuse`), resetting (`reset`), boundary updates (`updateBoundaries`), and the temperature calculation loop within the `Mesh` class. These timings were recorded for subsequent analysis.

### 4.2 Benchmark Over Varying Number of Threads

In the first figure($F$igure 1), using the collapse clause, the optimal performance was achieved with 16 threads, resulting in a total runtime of 0.222747 seconds. The diffuse and reset operations displayed a significant reduction in time, contributing to the overall efficiency. However, for 32 threads, a slight performance degradation was observed due to potential overhead from managing a higher number of threads.

In the second figure ($F$igure 2), utilizing the schedule clause with a static policy, similar trends were observed. The optimum performance occurred with 16 threads, achieving a total runtime of 0.211269 seconds. This configuration demonstrated the most balanced distribution of workload among threads, leading to enhanced cache efficiency and reduced contention.

Thread count significantly impacts the performance of parallelized code. This influence is due to several factors, including workload distribution, communication overhead, and inherent application parallelism. The author mentioned that in OpenMP parallelization, varying thread count directly affects computation distribution across processors, impacting load balancing and resource contention [2]. This becomes especially crucial in complex simulations like heat transfer, where computations have intricate dependencies. An article highlighted the optimal thread count aligns with the underlying algorithm and data dependencies [5]. Underutilizing resources occurs with too few threads, while excessive overhead arises with too many. Therefore, selecting the appropriate thread count is crucial for optimal parallel efficiency. This is evident in benchmark results, where runtime variations across simulation components are observed with different thread counts.

| Thread count | Diffuse | Reset | Boundary | Temperature | Total Run Time |
|---|---|---|---|---|---|
| 2 | 5.567 | 2.0096 | 1.6606 | 6.819 | 0.240790 |
| 4 | 3.1063 | 1.31 | 1.311 | 6.926 | 0.240612 |
| 6 | 2.207 | 1.0703 | 1.3032 | 6.909 | 0.242452 |
| 8 | 2.005 | 9.97999 | 1.5243 | 7.3222 | 0.232203 |
| 16 | 1.6697 | 2.4176 | 1.9527 | 6.9213 | ***0.222747*** |
| 32 | 2.049 | 1.9034 | 3.4709 | 12.8479 | 0.313837 |

Figure 1: Breakdown of Running Time for square.in using collapse clause ($Diffuse, Reset, Boundary, Temperature \times 10^{-5}$)

| Thread count | Diffuse | Reset | Boundary | Temperature | Total Run Time |
|---|---|---|---|---|---|
| 2 | 1.8193 | 1.0526 | 1.5523 | 4.4713 | 0.252529 |
| 4 | 1.1767 | 0.8523 | 1.9997 | 5.33 | 0.331347 |
| 6 | 1.0794 | 0.9207 | 1.686 | 6.278 | 0.324506 |
| 8 | 1.8092 | 0.6708 | 1.2189 | 7.1592 | 0.223464 |
| 16 | 0.8989 | 0.9309 | 1.5529 | 8.4017 | ***0.211269*** |
| 32 | 1.6277 | 2.7979 | 2.018 | 11.11228 | 0.233457 |

Figure 2: Breakdown of Running Time for square.in using schedule clause with static ($Diffuse, Reset, Boundary, Temperature \times 10^{-5}$)

## 5 Analysis of OpenMP Overhead

### 5.1 Measurement

The assessment of OpenMP overhead involved meticulous comparisons between serial and parallel executions of the code. To obtain a comprehensive understanding, the program was executed under various scenarios. First, a single-threaded run provided a baseline for the inherent serial performance, allowing for the isolation of OpenMP-induced overhead. Subsequently, the `omp_set_num_threads` function was employed to control the thread count for multi-threaded executions. The focus was directed towards the explicit scheme's key methods—specifically, the diffuse and reset operations—and the boundary reflection process, where the impact of OpenMP constructs was anticipated. The results of these comparisons are visually presented in ($F$igures 1, 2 and 3).

### 5.2 Analysis

In the ExplicitScheme class's diffuse and reset methods, which are the core computational components, the overhead introduced by OpenMP was noticeable when running with a limited number of iterations per thread. The explicit scheme's reflective boundary updates also exhibited overhead, particularly when the number of threads increased. The primary source of overhead was attributed to thread creation and synchronization mechanisms. With a coarse-grained computation like the one in the diffuse and reset methods, the overhead became more pronounced, impacting overall parallel performance. As the number of threads increased, the overhead tended to escalate, outweighing the performance gains.

#### 5.2.1 One Thread Per Core

The data *(*Figure 3) illustrates the impact of having one thread per core on the performance of the programme. In the case of one thread assigned to each core, the parallelization benefits are not fully realized. The diffuse method, which involves fine-grained computation, experiences a substantial increase in execution time. This is evident in the "Thread count: 1, Core: 40" scenario, where the diffuse time increases from 0.4575 to 2.3199, indicating a notable overhead introduced by managing multiple threads.

The reset method, characterized by a similar fine-grained nature, also shows an increase in execution time, emphasizing the overhead associated with parallelization. On the other hand, the boundary and temperature loops, with potentially coarser granularity, exhibit varying impacts. The boundary loop sees a slight increase, while the temperature loop experiences a reduction in execution time. The overall runtime shows a moderate increase, emphasizing the importance of

| Core | Thread count | Diffuse | Reset | Boundary | Temperature | Total Run Time |
|---|---|---|---|---|---|---|
| 1 | 1 | 0.457501 | 0.416501 | 0.784201 | 6.7912 | 0.227781 |
| 40 | 1 | 2.3199 | 5.19 | 0.622901 | 3.7824 | 0.212153 |

Figure 3: Breakdown of Running Time for square.in using schedule clause with static based on one thread per X core ($Diffuse, Reset, Boundary, Temperature \times 10^{-5}$)

carefully considering the balance between computational workload and thread management overhead.

#### 5.2.2 Overall analysis

Upon analysis, it became evident that the diffuse and reset methods, integral to the explicit scheme, demonstrated varying degrees of sensitivity to OpenMP overhead. The diffuse operation, characterized by fine-grained computations, showcased a notable impact on performance when parallelized. This effect was particularly pronounced when the computational workload was insufficient to offset the thread management and synchronization overhead introduced by OpenMP. Conversely, based on the result *(*Figure 3) the reset method, with its potentially coarser granularity, exhibited a more balanced relationship between computation and OpenMP overhead. The boundary reflection process, crucial for maintaining the integrity of the simulation, also displayed sensitivity to the number of threads employed.

### 5.3 Recommendation

The observations suggest nuanced recommendations for optimizing thread configuration. In scenarios where fine-grained computations dominate, it is essential to carefully balance the number of threads to mitigate excessive overhead. One thread per core might be optimal, but this should be validated empirically. For coarser computations, a moderate number of threads could provide a favorable trade-off [2]. Additionally, the single-threaded baseline proved valuable in gauging the extent of OpenMP-induced overhead, emphasizing the importance of considering the inherent characteristics of the computational tasks when configuring thread counts.

## 6 Conclusion

In conclusion, this investigation successfully demonstrated the potential performance gains achievable through OpenMP parallelization of the deqn program. Key optimizations included employing static scheduling, collapsing loops, and carefully selecting thread counts. The optimal performance was attained with 16 threads, achieving a minimum runtime of 0.211269 seconds. Additionally, the analysis revealed the crucial role of thread count in balancing workload distribution and minimizing overhead. While OpenMP effectively accelerated certain components, overhead became evident in fine-grained computations, necessitating a balanced thread configuration. Optimizing thread count based on computation granularity is essential for maximizing parallel efficiency.

## References

[1] Toolify AI. Optimize openmp schedule with automated algorithm selection. 2024. Accessed on 05 February 2024.

[2] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, Cambridge, MA, 2008.

[3] Florina M. Ciorba, Christian Iwainsky, and Philipp Buder. *OpenMP Loop Scheduling Revisited: Making a Case for More Schedules.* Springer, 2018. https://link.springer.com/chapter/10.1007/978-3-319-98521-$3_2Sec3$.

[4] M. S. Müller D. Terboven and R. van Nieuwpoort, editors. *Parallel Processing Proceedings Vol 4128.* Springer, Berlin, 2006.

[5] Oliver Weidner, Malcolm Atkinson, and Adam Barker. Towards a comprehensive framework for telemetry data in hpc environments. *arXiv.org*, 2017. Accessed on 06 February 2024.