OpenMP Heat Dissipation Simulation

Victoria Peterson - 15476758

***Repository**: https://github.com/VictorianHues/HeatDissipationSim*

I. Introduction

PARALLEL computing is essential to optimizing performance in scientific computing processes and simulations. Open Multi-processing (OpenMP) is an API that can be used for shared-memory multiprocessing in C, allowing for faster execution of tasks using parallel processing on multiple CPU cores.

It is important then to understand how this multi-threading process works and how one might take advantage of the tools to implement efficient parallelization with limited overhead from the API. By selecting different parameters in a scientific computing implementation (in this case Heat Dissipation), analysis may be done on different performance measures such as runtime and Floating-point operations per second (FLOPs). This may also extend to previous Cache Coherency implementations by tracking the simulated Read and Write requests made by the parallel CPUs and using the resulting address trace to analyze the parallel performance on different cache systems.

The following report details the methods used to parallelize the Heat Dissipation model, benchmark performance across different parameters, and evaluate the affects of Cache Coherency Protocols on resulting hit rates.

II. Theory

Parallel computing aims to improve performance by executing multiple operations concurrently using the available CPU cores on a machine, making it useful for processing large-scale simulations of real-world systems such as diffusion.

OpenMP is a shared-memory parallel programming model that provides an API in C and C++ to create multi-threaded tasks that share a single address space. This simplification of parallelization makes it easy to implement, but difficult to optimize due to uneven work load between processors, synchronization overhead from barriers and critical section, and scalability of task implementations.

Heat dissipation is simulated in this case, which is a mathematical model based on a weighted finite difference approximation of the diffusion equation:

$$\frac{\partial u}{\partial t} = D\nabla^2 u = D(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}) \tag{1}$$

This mathematical model defines a system in which heat dissipates in an initially-defined grid of temperatures, allowing the temperature to diffuse through the system until some equilibrium temperature is reached.

III. Method

The mathematical diffusion model in equation 1is discretized into an algorithmic form which allows for an iterative solution of the system using the grid of temperatures input into the algorithm. A conductivity grid of the same size is used to define the rate at which heat passes through the surface of the system, and the system is defined as a cylinder where-in the horizontal boundaries are periodic and the vertical boundaries are fixed (Dirichlet boundaries) as per Figure 1.

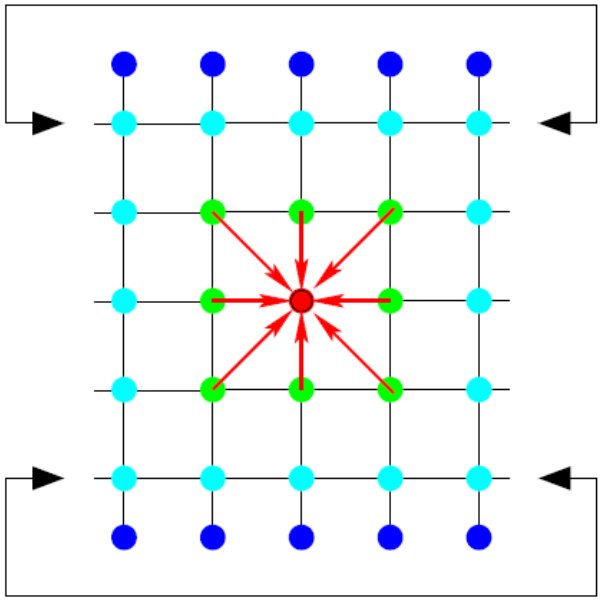


Fig. 1: Illustration of fixed value upper and lower boundaries with periodic horizontal boundaries

The discretized algorithm also uses Moore's Neighborhood such that each cell calculates its current value

based on the last time steps adjacent and corner neighbors, where the joint weight of the direct neighbors is $\frac{\sqrt{2}}{\sqrt{2}+1}$ times the conducted temperature and the diagonal neighbors are $\frac{1}{\sqrt{2}+1}$ times the conducted temperature.

The iterative calculations are described in Algorithm 1 in which the temperature grid is T, the new temperature grid is T', the conductivity grid is C, the grid size is $N \times M$. The process iterates through the grid in row-major order starting from the second row and ending on the second-to-last row in order to retain the fixed upper and lower boundaries. Additionally, whenever a left or right neighboring index is checked, the algorithm uses a modulo to ensure periodic horizontal boundaries. The final resulting temperature of the calculated cell is then compared to the last time-steps temperature to get the difference and maintain the maximum difference of the entire grid.

Algorithm 1 Update Temperature Grid

$max\_diff \leftarrow 0.0$
▷ Parallelized loop over interior rows, ghost cells excluded
for $i = 1$ to $N - 2$ **do**
  for $j = 0$ to $M - 1$ **do**
    $up \leftarrow i - 1$
    $down \leftarrow i + 1$
    $left \leftarrow (j - 1) \bmod M$
    $right \leftarrow (j + 1) \bmod M$
    $c \leftarrow C[i][j]$
    $adjacent\_sum \leftarrow$
    $T[i][left] + T[i][right] + T[up][j] + T[down][j]$
    $diagonal\_sum \leftarrow$
    $T[up][left]+T[up][right]+T[down][left]+T[down][right]$
    $weighted\_sum \leftarrow$
    $\frac{\sqrt{2}}{\sqrt{2}+1} \times adjacent\_sum + \frac{1}{\sqrt{2}+1} \times diagonal\_sum$
    $T'[i][j] \leftarrow c \times T[i][j] + (1 - c) \times weighted\_sum$
    $diff \leftarrow |T'[i][j] - T[i][j]|$
    $max\_diff \leftarrow \max(max\_diff, diff)$
  end for
end for

The maximum difference of the grid is checked against a convergence threshold after each time step, continuing if the convergence threshold has not been met or stopping if it has. The algorithm may also stop if a specified maximum number of iterations/time-steps is met.

The implementation applies the OpenMP API specification `#pragma omp parallel for reduction(max:local_max_diff) schedule(static) num_threads(p->nthreads)` directly before the initial row-major for-loop to divide the temperature grid calculations into groups of rows depending on the number of processors defined by *p-¿nthreads*. Additionally, the *schedule(static)* clause specified if the parallel processing uses a static load balance, forcing equal task allocation, or dynamic load balancing, allocating tasks based on workload completion within each thread. The clause `reduction(max:local_max_diff)` ensures that no race condition occurs when multiple processors need to update the maximum difference simultaneously by initializing a local maximum difference variable locally in in thread.

To specify the number of threads for the OpenMP implementation as well as other parameters, the implementation takes the command-line arguments:

- -n N, the number of rows
- -m M, the number of columns
- -i maxiter, the masimum number of iterations
- -k k, the reporting period
- -e ϵ, the convergence threshold
- -c FILE, the input file to read conductivities from
- -t FILE, the input file to read inital temperature points from
- -H H, the highest temperature in the input file
- -L L, the lowest temperature in the input file
- -p P, specifies the number of threads to be used
- -r enables result processing

A number of experiments are run over a set of predefined parameters to best analyze the performance of the OpenMP API framework for this Heat dissipation application, and some parameters will remain the same throughout to best allow for comparison between results:

- Grids will be of $N \times M$ size and include experiments on dimensions $50 \times 50, 50 \times 100, 100 \times 50, 100 \times 100, 250 \times 250, 1000 \times 1000$
- Conductivity grids will be uniformly distributed with conductivity rates of 0.5
- Initial temperature grids will generally use the ”areas” initial template in Figure 2 as a semi-random unbiased initial condition
- Cache Coherency Protocol tests will use address traces from single-iteration heat dissipation simulations

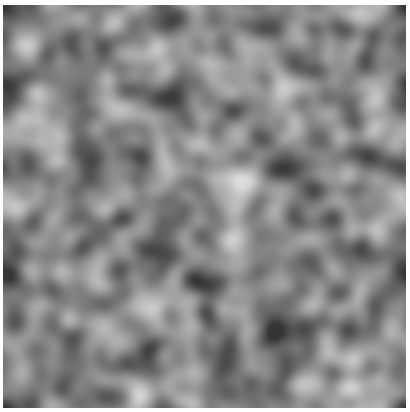
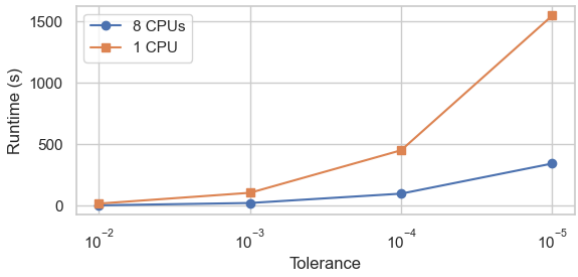


Fig. 2: The initial gradient of the 1000×1000 template grid

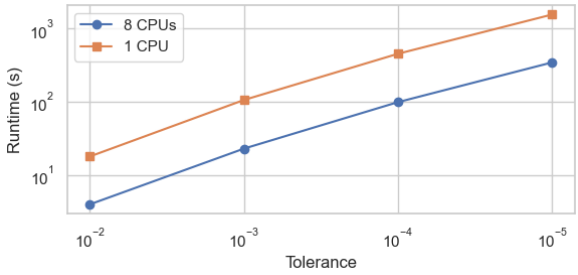
IV. Results

1) OpenMP Parallelization Performance: An initial look at OpenMP parallelization should start with a CPU count analysis looking at the extremes of possible parallelization on a local machine with 1 CPU core and 8 CPU cores. Figure 3a shows how each decreasing logarithmic step of tolerance sees a considerable improvement on the runtime of the system.

So where tolerances become more strict, the single CPU runtime explodes upwards and diverges considerably from the 8 CPU runtime, exposing a logarithmic difference between the two in Figure 3b.



(a) Logarithmic x axis, linear y axis



(b) Logarithmic x, y axes

Fig. 3: OpenMP run-time over a range of tolerances for 1 CPU and 8 CPUs on a 1000×1000 initial "areas" grid.

The magnitude of difference between the single CPU and multi-CPU run-times means that a lower convergence threshold selection will express the magnitude of change between run-times for different amounts of CPUs running in parallel. This allows for both an observable trend between CPU number selections and also reflects actual diffusion equilibrium better, but requires much more computation on larger grids as per Figure 9. As such, we choose a smaller grid of 250×250 to explore the run-time trend over a range of CPU counts in Figure 4.

The resulting trend in Figure 4 shows a quick decrease in the runtime between one and two CPUs, with a more regular decrease with further CPU inclusions up to 14 total CPUs in parallel.

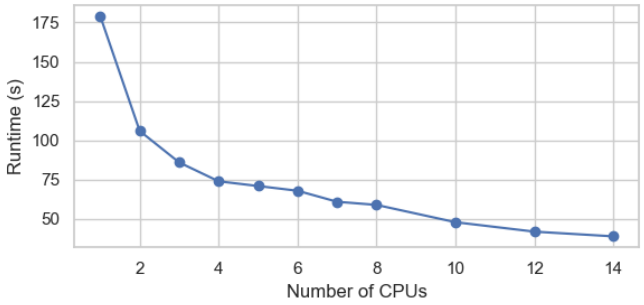


Fig. 4: OpenMP run-time over a range of parallel CPUs on a 250×250 initial "areas" grid and a convergence threshold of `1E-12`.

Returning to the comparison between the single CPU and 8 CPU set-up, it becomes possible then to explore how the initial condition might influence run-time between different CPU counts. Figure 5 shows how the regularity and boundary conditions influences how quickly the algorithm converges when all other factors are equal.

A gradient temperature grid from Figure 6a reaches the convergence threshold quickly as the diffusion of the system is already relatively steady since the upper and lower boundaries are also gradients. The temperature ranges around the periodic boundaries are the main locations where diffusion actually occurs since the right and left side of the system must find some equilibrium, but this ultimately leads to a quick overall diffusion of the system.

Interestingly, the plasma initial condition from Figure 6b matches closely with the "areas" temperature grid at 8 CPUs, but not at 1 CPU, indicating that the scaling of run-time with CPU number in Figure 4 is not one-to-one across different initial conditions.

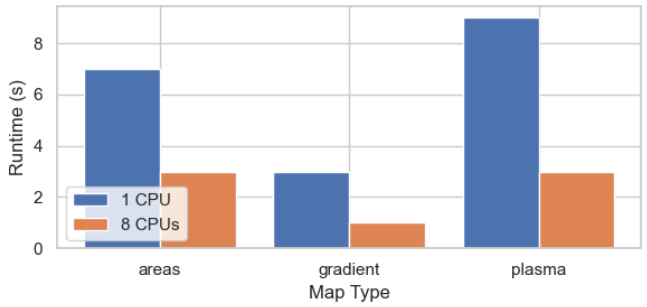


Fig. 5: Comparison of CPU count run-times between different initial gradients on a 100×100 grid with a convergence threshold of `1E-12`

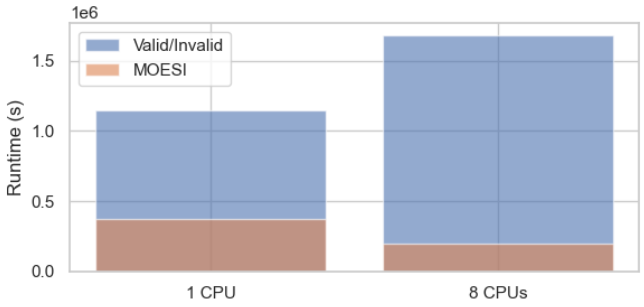
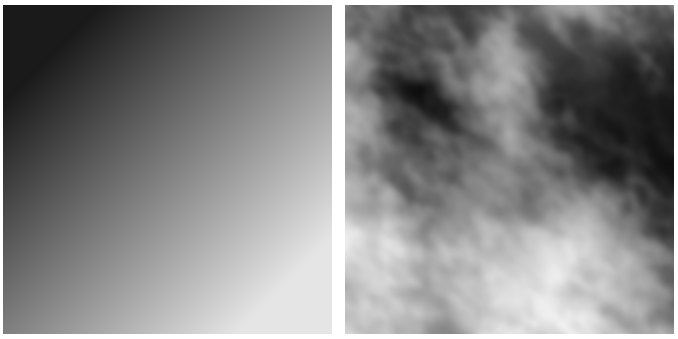
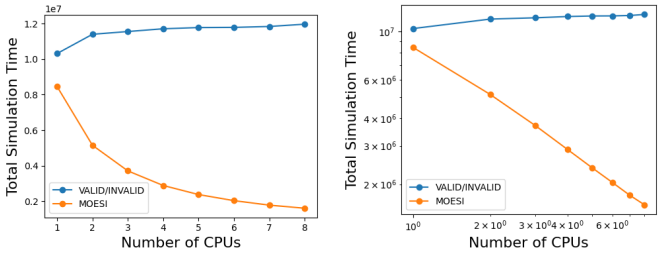


Fig. 7: CPU-Cache-Bus-Memory system run-time for one iteration on an "area" temperature grid of size 100×100



(a) Gradient (b) Plasma

Fig. 6: Initial Temperature Grids



(a) Linear axes (b) Log-Log Axes

Fig. 8: Total Simulation time in $1 - 8$ CPU systems for VALID-INVALID protocol and MOESI protocol

2) Cache Coherency Protocol Performance using OpenMP Address Traces: The CPU number exploration can be expanded further by using the address traces from the OpenMP parallel implementation in a simple VALID-INVALID and MOESI Cache Coherency protocol. In these experiments, all address traces will use the "areas" initial temperature grid and run over a single iteration to explore the influence of OpenMP parallelization on Cache coherency when completing tasks with repeated READS and WRITES to the same locations in memory.

With an initial look at the CPU-Cache-Bus-Memory system with both 1 CPU and 8 CPUs in Figure 7, the same general trend appears when compared to the findings from assignment 3 in Figure 8, where runtime trends upward for the VALID-INVALID protocol and lgoarithmically declines for the MOESI protocol.

The run-time of the system also scale with the increasing grid size in Figure 9 as the number of calculations for each grid cell creates additional Read and Write requests to addresses in memory. The algorithm reads from all 8 neighboring cells and writes to itself at each grid location when calculating, making approximately $8(N \times M)^2$ reads and $(N \times M)^2$, scaling a worst-case system at a rate of N^2.

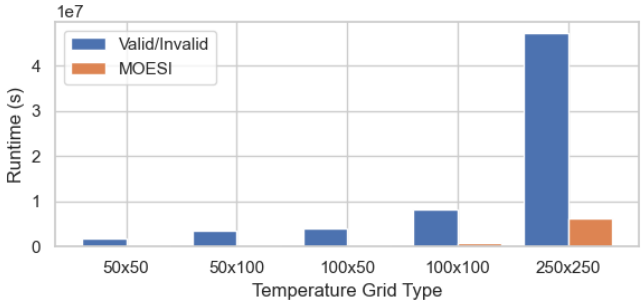


Fig. 9: CPU-Cache-Bus-Memory system run-time for different grid-sizes for 1 iteration averaged from 8 parallel CPUs

With an effective Cache Coherency protocol, fewer

accesses to main memory are required, thus reducing runtime considerably as can be observed with the MOESI protocol.

In the OpenMP implementation, a key factor affecting performance is how memory accesses occur in `C`. Since memory is accessed in a row-major order, cache accesses are more localized within a row than across columns and as a result, iterating over columns is expected to take longer than iterating over rows due to reduced spatial locality in cache access.

Looking more closely at the 50×100 and 100×50 grid sizes run-times in Table I shows this assumption seemingly holds true for the MOESI protocol, but not for the VALID-INVALID protocol, possibly due to how the VALID-INVALID protocol accesses memory after completing write-through processes.

Grid Size	Valid/Invalid (ns)	MOESI (ns)
50×100	3,545,810	400,268
100×50	3,978,898	387,985

TABLE I: Runtime comparison of cache protocols for selected grid sizes

To try to better confirm the theory, the algorithm itself can be changed so that iterations occur in column-major order instead of row-major, thus going against the row-major memory accesses of C. The resulting data in Table II unfortunately fails to support the idea as the trend reverses with the VALID-INVALID protocol sees considerable run-time advantages with the row-major implementation, while the MOESI sees a considerable disadvantage.

Cache Protocol	Column Major (ns)	Row Major (ns)
Valid/Invalid	8,242,714	6,577,224
MOESI	779,429	926,822

TABLE II: Runtime comparison of cache protocols in column-major and row-major order

While the result of the row-major/column-major hypothesis may not be conclusive, the results of the experiments still express the same results from assignment 3, showing that the MOESI Cache Coherency Protocol nearly universally improves upon the VALID-INVALID protocol in both Read and Write Hit rates as per Table III, Table IV, and Table V.

Grid Type	Valid/Invalid Hitrate	MOESI Hitrate
50x50	0.8924	0.9668
50x100	0.8873	0.9505
100x50	0.8625	0.9684
100x100	0.8625	0.9687
250x250	0.8828	0.9689

TABLE III: Hitrate comparison between Valid/Invalid and MOESI protocols.

Grid Type	Valid/Invalid Read Hitrate	MOESI Read Hitrate
50x50	0.9147	0.9745
50x100	0.9086	0.9555
100x50	0.8830	0.9756
100x100	0.8810	0.9760
250x250	0.8967	0.9761

TABLE IV: Read hitrate comparison between Valid/Invalid and MOESI protocols.

Grid Type	Valid/Invalid Write Hitrate	MOESI Write Hitrate
50x50	0.7286	0.9051
50x100	0.7312	0.9133
100x50	0.7125	0.9150
100x100	0.7273	0.9150
250x250	0.7808	0.9160

TABLE V: Write hitrate comparison between Valid/Invalid and MOESI protocols.

These hit rate results match closely with the real-world algorithmic traces in Figure 10 from assignment 3 in the form of Matrix Multiplication, Matrix Vector Multiplication, and Fast Fourier Transform, all of which achieved high hit rates due to repeated accesses to the same locations in memory. The heat dissipation algorithm similarly accesses similar locations in memory, resulting in high read and write hit rates when using the MOESI protocol.

Interestingly, the heat dissipation algorithm also closely matches the ”all reads on random address” trace experiment in Figure 10, especially for the VALID-INVALID protocol, possibly due to how invalidations occur during Write requests coupled with the larger number of reads on neighboring cells compared to local writes.

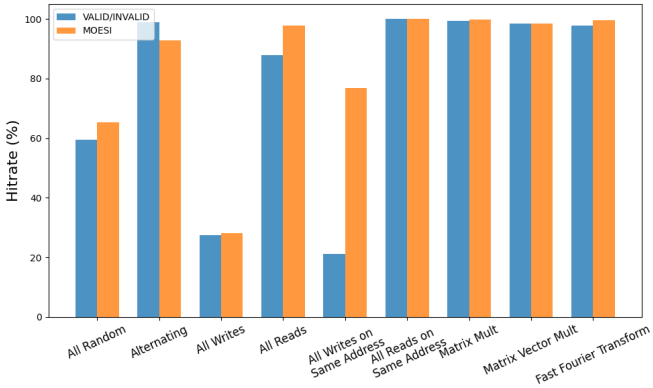


Fig. 10: Hit rate averaged over 8 CPUs for different trace file experiments over 100000 requests on both VALID-INVALID protocol and MOESI protocol

V. Conclusion

It becomes difficult to make any solid conclusions based on the results of the experiments beyond the previously explored fact that the MOESI protocol almost always improves the performance of the system compared to the VALID-INVALID protocol due to a reduction in the number of read and write accesses to main memory. When testing the traces produced by the OpenMP parallelization of the Heat Dissipation algorithm, there is little more to be gained without further testing of expanded parameter sets.

What is interesting are the logarithmic improvements to run-time observed with the initial OpenMP implementation when performing complex calculations, as well as the relatively continuous improvement to run-time with the addition of more CPUs. The shared-memory multiprocessor system with simple synchronization clauses seemingly limits the overhead and allows the rate of improvements to run-time to reduce at a slower rate than what was observed with additional CPUs in the Cache Coherency protocol run-time tests.

The parallelization process for simple algorithms like the Heat diffusion discretization allows for improvements to computation that could not be gained easily with a faster processor, making OpenMP a key to successful complex system implementations.