Parallelization Strategies for Numerical Simulation of Heat Transfer in CPU Heatsinks

BOUJIDA Nezar

18/11/2023

This study focuses on the parallelization of a numerical simulation that determines the temperature dynamics of a recent CPU model, specifically the AMD EPYC "Rome", using a heatsink.
We explore different parallelization methodologies to optimize simulation time and computational resource usage.

Microprocessors and Metal Heatsink

In the realm of computing, the evolution of microprocessors has been marked by significant advancements in processing power and energy efficiency. However, this progress comes with its own set of challenges, among which overheating remains a critical issue. As of 2023, the quest for more powerful computational capabilities has led to the development of microprocessors that consume in excess of 200 Watts of power. This substantial energy consumption translates almost entirely into heat, posing a significant threat to the reliability and performance of these devices. The phenomenon of overheating not only jeopardizes the operational stability of microprocessors but also limits the scope for further advancements in processor design and functionality. The criticality of managing and mitigating heat generation in microprocessors has, therefore, become a paramount concern for researchers and engineers.

Metal heatsinks serve as a fundamental component in thermal management systems for microprocessors. Their primary function is to efficiently transfer heat away from the processor to the external environment. This process is critical for maintaining optimal operating temperatures.

Contents

1 Introduction

Our research focuses on parallelizing a numerical simulation to analyze the temperature dynamics of the AMD EPYC "Rome" CPU, with the addition of a heatsink. This simulation requires a spatial discretization process where the heatsink is divided into a finite three-dimensional grid of cuboidal cells, enabling precise modeling of its thermal behavior.

The mesh's granularity is vital for the accuracy of the simulation. Although finer meshes provide more detailed insights, they significantly increase computational resource demands (for more information, see the referenced simulation paper).

To facilitate our study, we utilize Grid5000, a dedicated research infrastructure for large-scale parallel and distributed computing experiments. It acts as a supercomputer, hosting numerous CPUs. This allows us to employ finer meshes for more accurate thermal behavior modeling while parallelizing the process to minimize execution time.

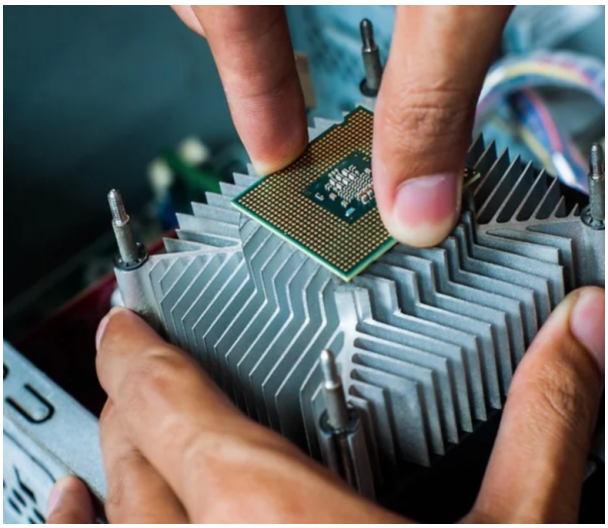


Figure 1: A fan-cooled heat sink on a processor.

2 Achieved Results

By employing parallelization techniques, we have significantly reduced the execution time. In the Normal mode with finer meshes, utilizing a 2D slicing method, we observed a reduction from 760 seconds to 154, resulting in a speedup factor of approximately 4.94x.

Additionally, implementing non-blocking communication further decreased the execution time to 135 seconds, achieving a speedup factor of approximately 5.63x.

3 Parallelization Techniques

The heatsink is partitioned into parallelepipeds, which are then distributed across multiple processors. This parallel approach enables simultaneous processing of distinct sections of the heatsink.

A crucial aspect of our simulation is the Temperature Update Mechanism, responsible for calculating temperature changes in each cell based on the temperatures of its neighboring cells. For cells at the edges of each parallelepiped, communication with adjacent parallelepipeds is necessary to obtain the required data. This communication is facilitated through the Message Passing Interface (MPI).

To determine the convergence, indicating whether the steady state has been reached, we calculate the sum of temperature differences across all cells from time t to $t+1$. With our parallelization approach, this operation is performed on each processor responsible for a portion of the heatsink.

Finally, MPI is used to gather these values onto a chosen processor, which determines whether the steady state has been achieved.

3.1 First Technique: Slicing Along the Z-Axis

The initial parallelization technique we employed involves slicing the heatsink along the z-axis. This method divides the heatsink into several rectangular parallelepipeds, which are then distributed across different processes.

This strategy benefits from ensuring memory contiguity across the planes, facilitating the aggregation of results upon completion of the computations.

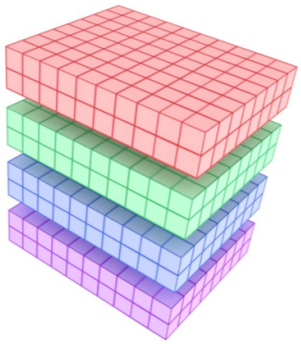


Figure 2: Slicing along the z-axis.

3.2 Second Technique: 2D Data Slicing Along Z and Y Axes

Our second approach to parallelization incorporates 2D data slicing along both the z and y axes. This technique further decomposes the heatsink into a larger number of parallelepipeds, with each assigned to a specific processor, maintaining contiguity along the x-axis. Similar to the first method, this approach requires increased communications between processes to accurately compute thermal transfers for cells on the edges.

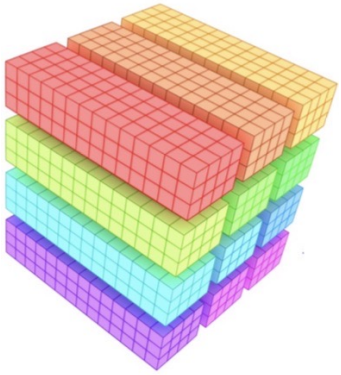


Figure 3: 2D data slicing along z and y axes.

4 Results Analysis

In our investigation of parallelization efficiency, we employed both 1D and 2D partitioning techniques across two distinct operational modes: Fast and Normal. The "Fast" mode is characterized by comparatively larger slices, leading to a reduced computational load and, consequently, quicker execution times. Conversely, the "Normal" mode uses finer meshes for slicing, aiming for higher detail at the cost of increased computational demand. This distinction is crucial, as finer meshes, despite their potential for yielding more detailed results, significantly elevate the required computational resources, thus extending the execution time.

4.1 Performance in Fast Mode

Optimal performance was achieved using a 1D partition with four processors, resulting in an execution time of 2 seconds. Compared to the sequential code, this represents a speedup factor of 2x. However, further partitioning smaller ensembles across more processors led to excessive, time-consuming communications, causing a decline in performance beyond this configuration.

In contrast, 2D partitioning provided acceptable performance improvements over the sequential code, with less variation compared to 1D slicing.

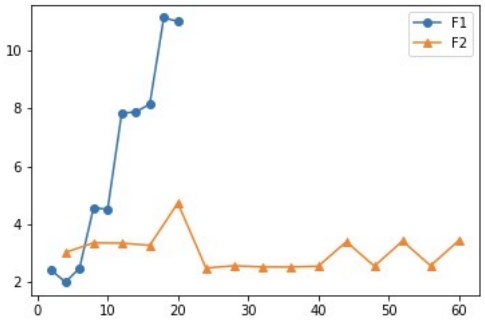


Figure 4: Performance of 1D and 2D Partitioning in Fast Mode

4.2 Performance in Normal Mode

2D partitioning demonstrates higher efficiency, achieving an execution time of 154 seconds and speeding up the sequential code execution by a factor of 4.94x. Moreover, 2D partitioning maintains consistency, reaching optimal results with 32 processors and showing minimal variance compared to 1D.

On the other hand, 1D partitioning achieves its best performance with 12 processors, resulting in an execution time of 168 seconds, representing a speedup factor of 4.52x compared to the sequential code. However, performance begins to decline beyond 20 processors.

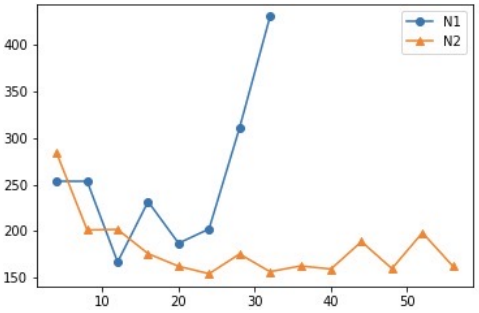


Figure 5: Performance of 1D and 2D Partitioning in Normal Mode.

5 Non-Blocking Communications

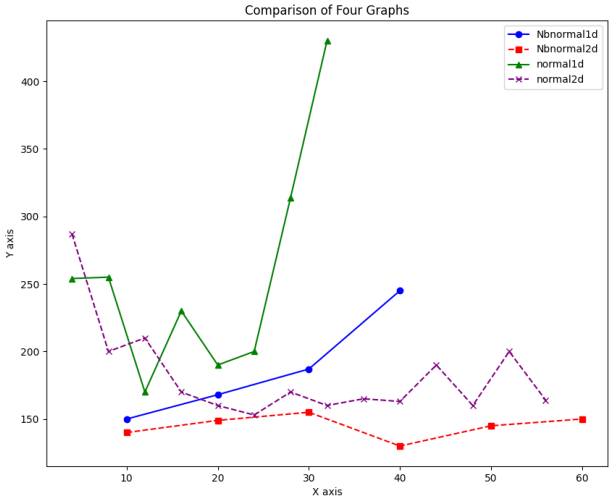


Figure 6: Performance of 1D and 2D Partitioning in Normal Mode using Non-Blocking Communications.

Continuing our study, we explored the impact of employing non-blocking communication in our two parallelization approaches. Non-blocking communication allows processes to initiate data transfer operations and proceed with their execution without having to wait for the transfer to complete. This technique can significantly enhance the efficiency of parallel applications by reducing time and potentially lowering the overall execution time within a distributed computing environment. In both the 1D and 2D slicing methods, the non-blocking communication mode performed better, achieving optimal results. Specifically, utilizing 2D non-blocking communication with over 40 processors resulted in a speedup of the sequential code execution time by a factor of 5.63x.

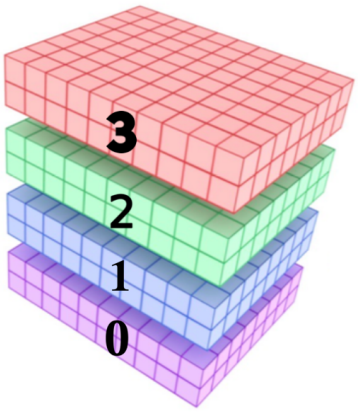
6 Checkpointing for Enhanced Efficiency and Reliability

For computations extending over long periods, the integration of a checkpointing system into our codes emerges as a highly beneficial strategy. This system, designated for codes marked with the suffix $*cp.c$, enables the process to periodically save its current state.
Such a mechanism ensures that, in the face of any failure or interruption, computations are not reset but can instead resume from the most recent checkpoint. This capability not only significantly boosts efficiency by eliminating the need to restart processes from the beginning but also substantially increases the reliability of long-running operations.

7 Q&A Section

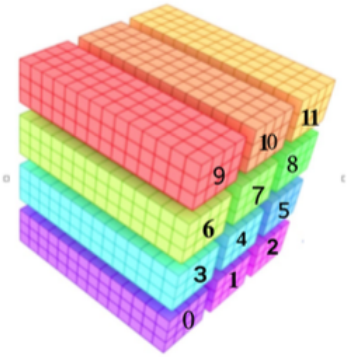
How did we distribute the processors in the case of 1-dimensional slicing?
We determined the number of XY planes in a heatsink by dividing its height (Z) by the slicing thickness ΔZ. For example, with $Z = 0.15$ and $\Delta Z = 0.01$, we identified 15 XY planes. These planes are distributed among K processors by dividing the total XY planes by K. If the division result is less than 1, it indicates more processors than XY planes, assigning one plane per processor. Otherwise, XY planes are evenly distributed, with the last processor handling any leftovers.

How did we calculate the thermal transfers for 1D slicing?
The calculation depends on neighboring cells. We introduced two additional planes on either side for each processor, except the first and last, which received one each. These extra planes facilitate information exchange between processors at each iteration.



How did we facilitate communications in the case of 1D decomposition?
Communication between neighboring processors is essential. Processors 1 to $k-2$ make two communications, while 0 and $k-1$ make one. We employed MPI's mpi_send_recv function within if-else statements to manage this, repeating the process at every iteration.

How did we distribute the processors in the case of 2-dimensional slicing?
We used a function to divide k processes into i, j such that $k = ij + r$, then partitioned the z-axis by i and the y-axis by j. The ij processes were distributed accordingly.



How did we calculate the thermal transfers for 2D slicing?
A similar approach to 1D slicing was used, introducing 4 additional planes for necessary information.

How did we facilitate communications in the case of 2D decomposition?
We divided communications along the z-axis and y-axis, allowing processes to communicate with their direct neighbors. This was implemented using arithmetic operations to identify sending and receiving processors.

What is non-blocking communication?
Non-blocking communication allows processes to continue without waiting for data transfers to complete, overlapping communication and computation for enhanced efficiency and reduced execution time.

How did we implement non-blocking communication?
We adapted our 1D and 2D codes to use non-blocking techniques, integrating MPI_Wait and MPI_Test to monitor communication completion without disrupting computations.

Why did non-blocking communication yield good results even with the use of MPI_Test and MPI_Wait?
Using MPI_Test and MPI_Wait allows efficient checking of communication completion, minimizing delays associated with waiting for data transfers and improving overall execution time.

How does the checkpointing system work?
The system uses load_checkpoint and save_checkpoint methods to manage the continuity of computations. It saves and reloads critical variables at specified intervals, ensuring process continuity after interruptions by storing data in unique text files per processor.