App Evaluation: High Performance Computing (HPC)

By

Carlos Arbonés *46480960F*
Benet Ramió *41580669G*

Departament de Ciències de la Computació
Universitat Politècnica de Catalunya

Professor:
Julita Corbalan

A project submitted for the course of

Paral·lelisme i Sistemes Distribuits

Barcelona, April 14, 2023

Contents

Introduction

High Performance Computing (HPC) has become a critical tool in various scientific research and data analysis domains. In this project, we aim to generate the *D class* of three applications, namely *LU-mZ*, *SPMZ*, and *BT-mZ*, for the MZ-MPI version of NAS Parallel Benchmarks (*NPB3.3.1-MZ/NPB3.3-MZ-MPI*). Our project focuses on evaluating the performance and energy consumption of these applications with different configurations of nodes, threads, and processes, using a combination of MPI and OpenMP parallelization techniques.

A key objective of our project is to determine the most efficient level of parallelism for these applications, comparing the performance of MPI (using many processes per node and fewer threads per process) and OpenMP (using fewer processes per node and more threads per process). To achieve this, we will execute the applications with various configurations and analyze the results. We will include justification for the selection of experiments, along with graphical representations of the time taken by the applications in relation to the total number of cores used. We will also analyze the speedup and efficiency of the applications with respect to the total number of cores used, to assess the performance of different parallelization techniques.

In the second section of our project, we will gather more information about the jobs executed using the *sacct* command of the Slurm workload manager. We will explore how to extract additional information from *sacct* and perform a simple analysis of the energy consumed by some of the jobs executed.

Additionally, we will compare the performance and energy consumption of the applications with different configurations of nodes, threads, and processes. We will analyze the results obtained from the previous sections and draw conclusions on the most efficient level of parallelism for these applications, considering factors such as speedup, efficiency, and energy consumption.

Finally, we will compare the impact of changing the OpenMP schedule in one of the programs, specifically the *LU-mZ* program.

Through this project, we aim to contribute to the understanding of the performance and energy efficiency of HPC applications, specifically focusing on the *LU-mZ*, *SPMZ*, and *BT-mZ* programs.

Method

The project utilized *Marenostrum*, a high-performance computing technology that is composed of multiple nodes, each containing a large number of central processing units (CPUs). For this project, we used a total of 8 nodes (maximum), each equipped with 48 CPUs, resulting in a total of 384 CPUs available for our computations.

To effectively harness the parallel computing capabilities of Marenostrum, we employed two parallelization techniques: *OpenMP* and *MPI*. OpenMP is a shared-memory parallelization technique that employs multiple threads within a single node. In contrast, MPI is a distributed-memory parallelization technique that facilitates communication and coordination between multiple nodes.

The *sbatch* and *srun* commands were used to submit and execute our programs on the Marenostrum supercomputer. We submitted batch jobs using the *sbatch* command, while the *srun* command was used to execute parallel programs interactively.

We used three parallel programs in our project: *bt*, *sp*, and *lu*. The *bt* program was used to solve a system of linear equations, while the *sp* program was utilized for sparse matrix multiplication. The *lu* program computed the LU decomposition of a matrix.

To evaluate the performance of our parallel programs, we utilized the speedup and efficiency metrics. Speedup is the ratio of the time taken by a serial program to the time taken by a parallel program. It is used to measure how much faster a parallel program is compared to a serial program. Efficiency is a measure of how effectively a parallel program utilizes the available processors. Efficiency ranges from 0 to 1, with 1 indicating perfect efficiency, where the parallel program achieves a speedup equal to the number of processors used. The formulas for speedup and efficiency are given as:

$$\textit{Speedup} = \frac{T(1)}{T(p)} \tag{2.1}$$

$$\textit{Efficiency} = \frac{\textit{Speedup(p)}}{p} \tag{2.2}$$

In our project, we utilized dynamic and static scheduling techniques. OpenMP offers three types of schedulers: static, dynamic, and runtime. The static scheduler divides loop iterations among threads at compile-time, providing a predictable workload distribution. The dynamic scheduler, on the other hand, distributes iterations during runtime, allowing for load balancing but introducing some overhead. Lastly, the runtime scheduler offers the flexibility to choose between static and dynamic scheduling at runtime.

Before starting, it is necessary to present the commands used through the project and their utility:

1. `make suite`
   Compiles and executes a suite of automated test cases for a program or library.

2. `vim {script}`
   Opens the Vim text editor with the specified script file for editing.

3. `sbatch {script}`
   Submits a job script to a Slurm job scheduler.

4. `scancel {jobid}`
   Cancels a job that has been submitted to the Slurm job scheduler.

5. `sacct -j {jobid} --format=jobid,jobname,elapsed,state`
   Displays the status and accounting information of a job that has been submitted to the Slurm job scheduler.

6. `sacct -p -j {jobid} --format="JobID,TotalCPU,AllocCPUS, ReqCPUS,ConsumedEnergy,AveCPUFreq,JobName,Elapsed" --units=K > output.csv`[1]
   Generates a CSV file containing the accounting information of a job that has been submitted to the Slurm job scheduler.

7. `export OMP_SCHEDULE="DYNAMIC/STATIC"`
   Sets the scheduling policy for OpenMP threads in parallel regions of a program.

[1] The command was searched in web page 5

Results

After executing the three programs, namely bt-mz, sp-mz, and lu-mz, we aim to investigate the impact of executing the programs with more of an MPI-like approach (i.e., more tasks and fewer threads) versus an OpenMP-like approach (i.e., more threads and fewer tasks) on their overall performance.

To ensure that our experiments were both representative and efficient, we exhaustively tested all possible combinations of nodes and tasks, where the number of threads per task was an integer. This approach was taken to avoid any inaccuracies in the results, which could have led to an erroneous interpretation of the data, had we used non-integer values.

We ran the programs with different process counts, ranging from 8 to 384, in order to evaluate their performance and scalability. However, it's worth noting that LU-MZ only accepts up to 16 processes, so we ran it with process counts of 8, 12, and 16 only.

3.1 Analysis of the level of parallelism

3.1.1 Analysis of the Execution Time

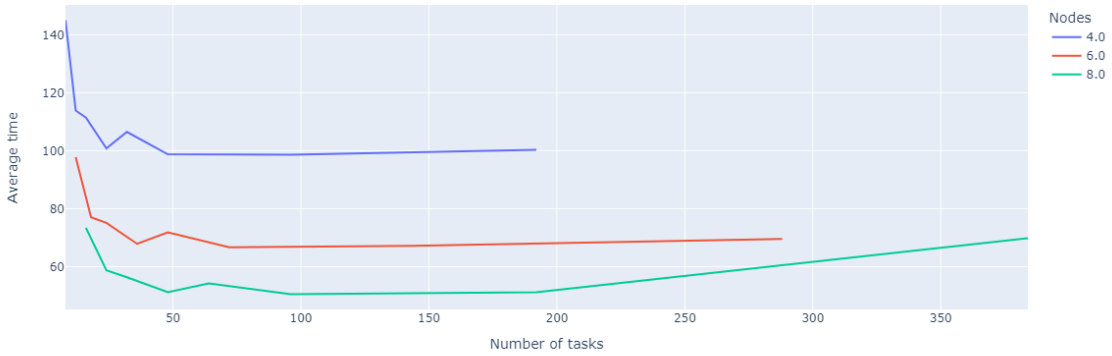

Figure 3.1: **Execution Time vs Tasks: Relationship between average execution time and number of tasks. (bt-mz)**

Our results indicate a decreasing trend in execution time as the number of tasks increases. Interestingly, for each number of nodes tested, we observed a peak in execution time when the number of tasks was at its minimum. However, as we increased the number of tasks, the average execution time decreased until it reached a certain point (32 tasks for 4 nodes, 48 tasks for 6 nodes, and 64

tasks for 8 nodes) where a slight peak was observed (see Figure 6.1) . Beyond this point, the average execution time remained relatively constant, even as we further increased the number of tasks.

We also observed a significant difference in average execution time based on the number of nodes used. Specifically, when using 4 nodes, the average time was significantly higher than when using 6 or 8 nodes. . However, we also noted that execution time was slightly slower with 6 nodes than with 8 nodes, which may be attributed to the allocation of additional resources in the latter scenario.

The peak in execution time when the number of tasks is at its minimum is likely due to the overhead associated with parallelization. As the number of tasks increases, the overhead becomes less significant, resulting in a decrease in the average execution time.

The difference in average execution time based on the number of nodes used is logical, as more nodes provide more resources for the application to utilize.

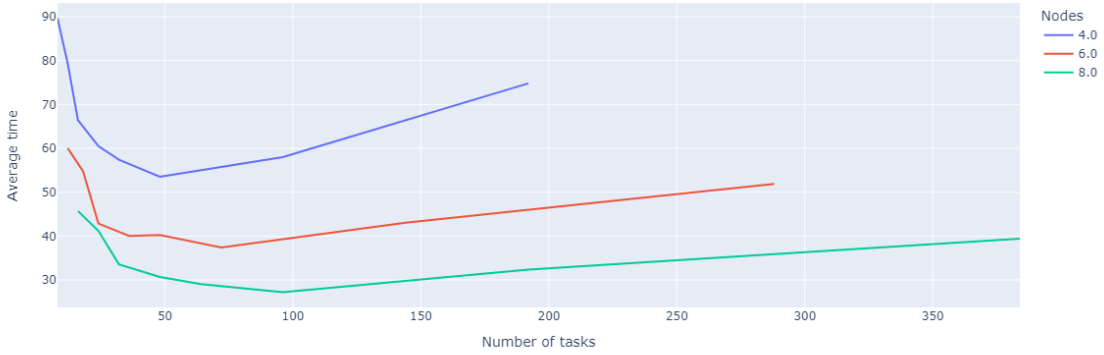

Figure 3.2: **Execution Time vs Tasks: Relationship between average execution time and number of tasks. (sp-mz)**

Our analysis of the sp program indicates a clear quadratic pattern in the average execution time. Similar to our observations with the bt program, we noted that a lower number of tasks resulted in a higher average time. As we increased the number of tasks, the average time began to decrease until reaching a minimum at approximately 48, 72, and 96 tasks for 4, 6, and 8 nodes, respectively (as illustrated in Figure 6.2). Beyond this point, increasing the number of tasks resulted in an increase in average time. Notably, we observed that this increase was more pronounced when using 4 nodes, while it was more gradual with 6 and 8 nodes.

The clear quadratic pattern observed in the average execution time of the sp program can be attributed to the same reasons as the bt program. As the number

of tasks increases, the workload is divided into smaller tasks that can be executed simultaneously on multiple nodes, resulting in faster execution times overall. However, the overhead associated with task distribution, communication, and synchronization becomes more significant as the number of tasks increases beyond a certain point.

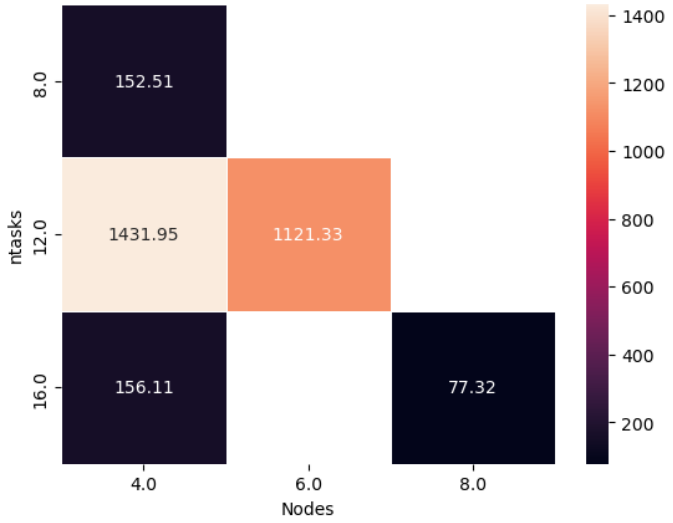

Figure 3.3: **Execution Time Heatmap: Color-coded representation of average execution times for varying numbers of tasks and threads. (lu-mz)**

For the lu program, we have fewer observations compared to the other programs, and we present the values we have in Figure 3.3. We can observe that with 4 nodes, the execution time for 8 and 16 tasks is similar, but when using 12 tasks, the time increases significantly, reaching almost 10 times the previous value. Comparing the results with other numbers of nodes, we can see that with 6 nodes, the average time with 12 tasks is lower than with 4 nodes. And with 8 nodes, the average time for 16 tasks is approximately half of the time required with 4 nodes.

3.1.2 Analysis of the SpeedUp

To analyze the SpeedUp (2.1), we first need to calculate it. To do so, we require the T_{serial} value, which is obtained from the time calculated during the execution of each program with 1 node, 8 tasks, and 1 thread per task.

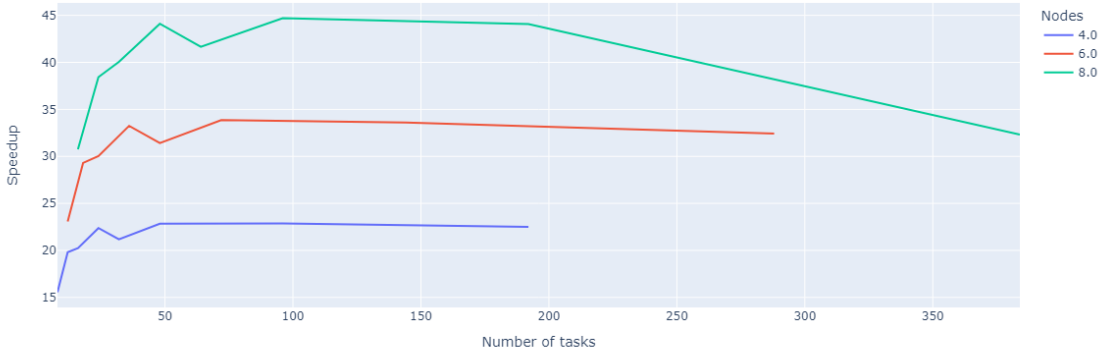

Figure 3.4: **SpeedUp vs Tasks: Relationship between speedUp and number of tasks. (bt-mz)**

We observe that for all nodes, the speedup is low with a low number of tasks. As we increase the number of tasks, the speedup also increases and reaches a maximum value for each node. After reaching the maximum value, the speedup stabilizes as we continue to increase the number of tasks. It is noteworthy that with 8 nodes, we achieved a speedup of 32 at 384 number of tasks, a notorious decrease if we compare with the other values. We can see that the Speedup increases also with the number of nodes Figure 6.3.

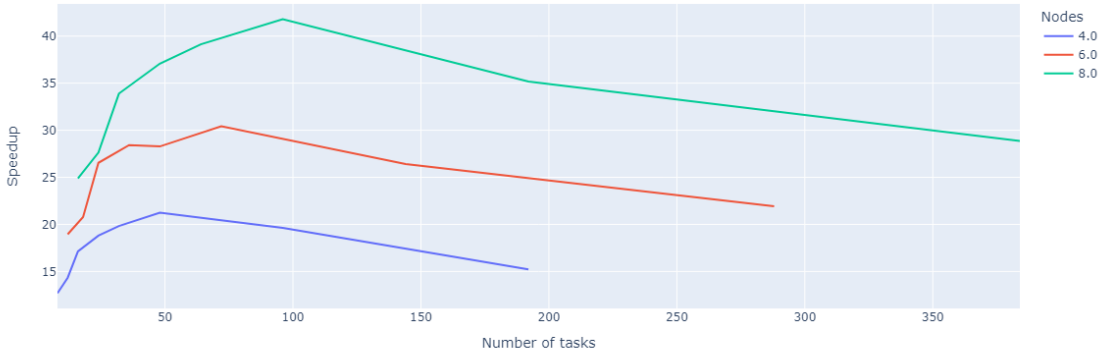

Figure 3.5: **SpeedUp vs Tasks: Relationship between speedUp and number of tasks. (sp-mz)**

It is evident that the program exhibits a discernible pattern in its performance. Specifically, the speedup of the program increases proportionally with the number of tasks until it reaches a maximum value. Subsequently, the speedup begins to decrease as the number of tasks increases further.

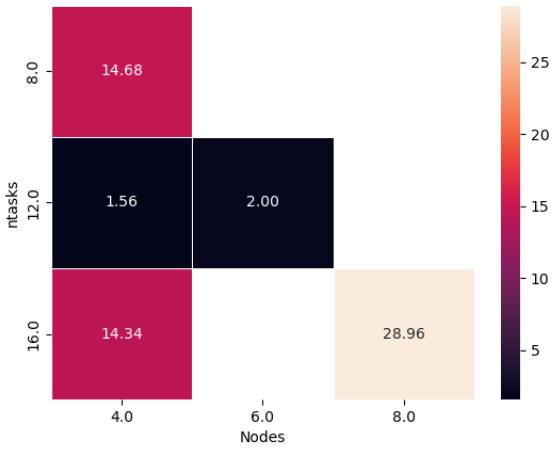

Figure 3.6: **SpeedUp Heatmap: Color-coded representation of SpeedUp for varying numbers of tasks and threads. (lu-mz)**

The results show that the program attains its maximum speedup when executed on 8 nodes and 16 tasks. It is important to note that this optimal configuration is dependent on the specific characteristics of the program and the hardware platform used.

Additionally, we observe a general trend of increasing speedup with an increase in the number of nodes employed. This trend can be attributed to the ability to divide the workload among multiple nodes and execute them simultaneously, thus reducing the execution time. However, it is important to note that this trend may not always hold true and may be limited by other factors like communication and parallelism overhead.

3.1.3 Analysis of the Efficiency

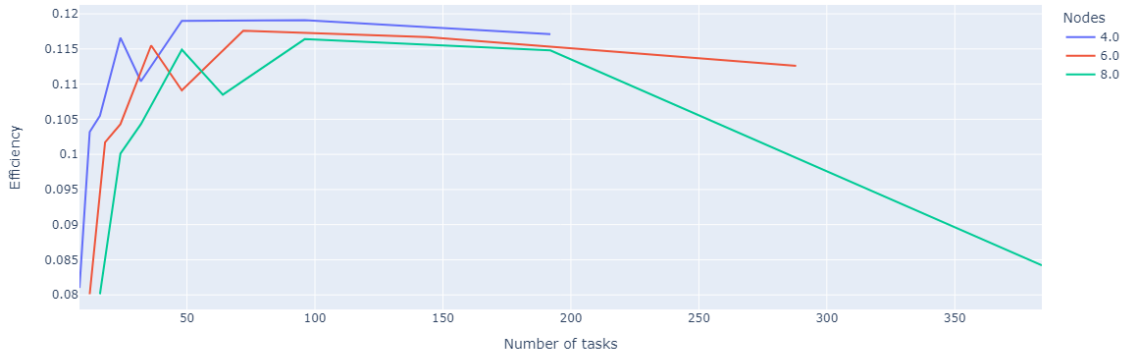

Figure 3.7: **Efficiency vs Tasks: Relationship between efficiency and number of tasks. (bt-mz)**

The analysis reveals that the efficiency of the program follows a consistent pattern across the different numbers of nodes tested, with values that are relatively close to each other. Specifically, we observe an initial low efficiency, which gradually increases until it reaches a maximum value at approximately 80 tasks, before subsequently decreasing.

The efficiency values obtained are generally low, with a maximum value of 0.12. However, it is desirable for the efficiency to approach 1, as this indicates that the resources are being optimally utilized. A low efficiency value suggests that there may be issues with load balancing, communication overhead, or other factors that are hindering the performance of the program. Therefore, further optimization of the program is necessary to achieve higher efficiency values.

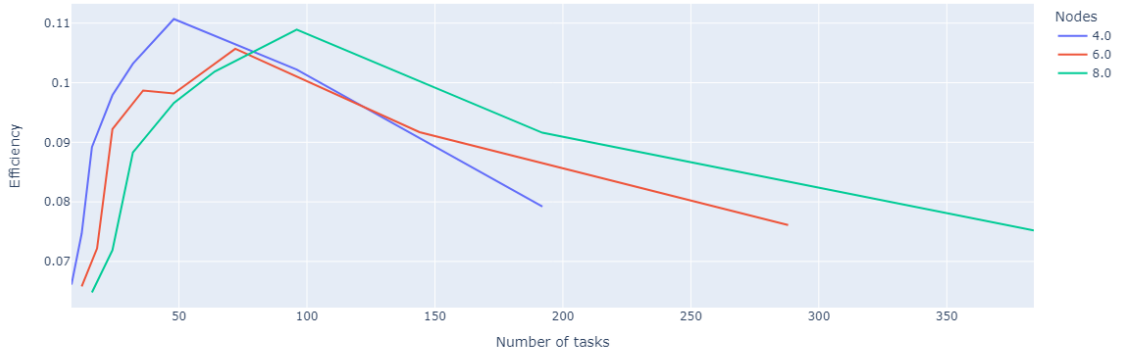

Figure 3.8: **Efficiency vs Tasks: Relationship between efficiency and number of tasks. (sp-mz)**

Upon analyzing the results, it becomes apparent that the program's pattern is more evident than in the previous experiment. We observe a substantial decrease in efficiency when the number of tasks exceeds 120. Similar to the previous program, we observe peaks with a maximum efficiency value of 0.11, which is achieved with four nodes.

Notably, the efficiency trend is different from the previous experiment. We observe that the efficiency with fewer nodes is higher than with more nodes. Further analysis is necessary to understand the reasons behind this trend, and to optimize the program for higher efficiency values.

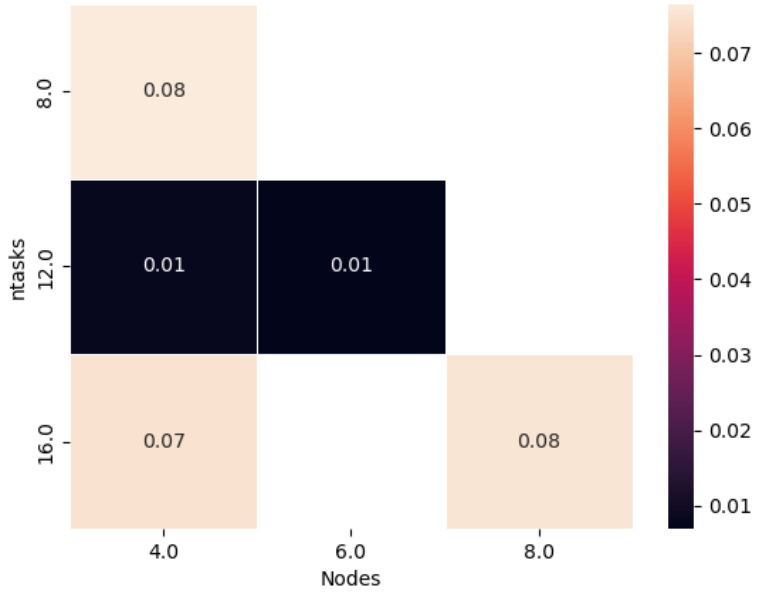

Figure 3.9: **Efficiency Heatmap: Color-coded representation of Efficiency for varying numbers of tasks and threads. (lu-mz)**

Upon observation, it has been noted that the optimal efficiency is achieved using four nodes and eight tasks, as well as with eight nodes and 16 tasks. However, the efficiency value in the latter case is significantly low, measuring only 0.08.

3.1.4 Conclusions

From the perspective of average execution time, increasing the number of tasks generally leads to a decrease in the time required to complete them. However, beyond a certain point, the decrease in average time is no longer worth the ad-

ditional computational resources expended. In such cases, a combination of MPI and OpenMP is the most effective solution.

Similarly, when considering speedup, the relationship between the number of tasks and the resulting speedup is generally consistent regardless of whether the number of tasks is low or high. In either case, a combination of MPI and OpenMP is the optimal approach.

This same pattern holds true for efficiency as well. Regardless of the number of tasks, a combination of MPI and OpenMP results in the highest level of efficiency.

3.2 Consumed Energy and Total Cost

This section will focus on analyzing and discussing the energy consumption of various programs when run on different resources, as well as calculating the total cost of execution. To obtain information on the energy consumption and other relevant parameters of the programs, the 'sacct' command was utilized.

As a reference, a cost of 0.2 € per kilowatt-hour ($\frac{KWatts}{h}$) will be used. The formula for calculating energy consumption is as follows:

$$Energy(J) = AveragePower(Watts) \cdot Time(s) \tag{3.1}$$

After analyzing the programs, the following results were obtained:

Figure 3.10: **Relationship between elpased time and Consumed Energy (bt-mz)**

We observe that with more elapsed time, there is an increase in the consumed energy in the majority of cases when we separate by the number of nodes. We notice that the energy consumed, when separated for each number of nodes, is more or less similar, although with 4 nodes the elapsed time is superior. This is because although the power used with more nodes is higher, it is also faster, which compensates for the increased energy consumption.

As we can see in Figure 6.5, for the *sp* program, we observe the same pattern. For the *lu* program (Figure 6.6), we notice that the most consumed energy was with 4 nodes.

Figure 3.11: **Relationship between Average Power and Consumed Energy(bt-mz)**

Here, we observe a quadratic tendency, where although programs with fewer processes have lower Average Power, as they have more consumed time, the consumed energy will be higher, as energy is calculated using Equation (3.1). The executions with a moderate number of processes seem to be the ones with the least consumed energy.

The pattern in the *sp* program (Figure 6.7) is similar but more distributed. For the *lu* program, the consumed energy is much higher compared to *bt* and *sp*.

3.2.1 Cost

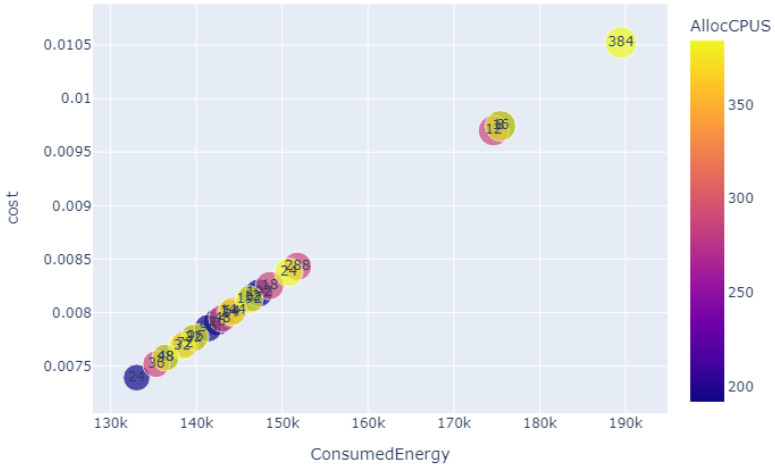

Figure 3.12: **Relationship between Cost and Consumed Energy. (bt-mz)**

Cost and consumed energy are linearly dependent, as higher consumed energy results in higher costs. The highest cost is obtained with 8 nodes and 384 processes, with a cost of 0.0105 €, which is obviously the program that wasted the most consumed energy.

The same trend can be observed for *sp* (Figure 6.9) and *lu* (Figure 6.10). Although the maximum cost for *sp* is 0.0075 €and for *lu* is 0.08 €(with 4 nodes).

Figure 3.13: **Relationship between Cost and Elapsed time. (bt-mz)**

As we can see in the images, executions with longer elapsed time tend to have

higher costs, as observed in the case of *sp* (Figure 6.11) and *lu* (Figure 6.12). This indicates that as the elapsed time increases, the cost of the programs also tends to increase.

3.2.2 Average Power

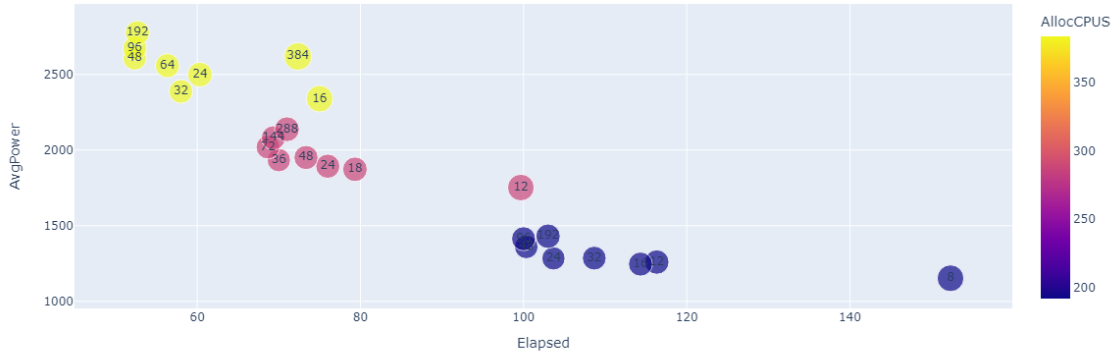

Figure 3.14: **Relationship between elapsed time and Average Power. (bt-mz)**

After conducting our analysis, we discovered a pattern indicating that the average power consumption tends to decrease as the number of nodes decreases. Specifically, we observed that the average power consumption was significantly higher with eight nodes than with six nodes, and six nodes consumed more power on average than four nodes. Furthermore, we found that within each cluster of the same number of nodes, programs with a higher number of tasks generally consumed more energy. However, it's worth noting that these programs tended to have shorter elapsed execution times.

A similar conclusion can be drawn from the analysis of the 'sp' (Figure 6.13) and 'lu' (Figure 6.14) programs. In the case of 'sp', we also observe the pattern of decreasing average power consumption with decreasing number of nodes, and programs with a higher number of tasks generally consume more energy. This pattern is also evident when analyzing the data separately for each number of nodes.

When examining the 'lu' program, we note that the maximum consumed energy is achieved with eight nodes and 16 tasks, which is significantly different from other configurations.

While adding more nodes can improve performance, it requires careful consideration of energy and resource requirements to ensure that the system remains efficient and cost-effective.

3.2.3 Conclusions

The findings indicate that the highest power consumption is observed for a greater number of nodes, while the cost exhibits a linear relationship with consumed energy. Upon analyzing the consumption patterns with respect to the number of nodes, it is observed that the energy consumption reaches a minimum when there are 24, 36, and 48 tasks assigned to 4, 6, and 8 nodes, respectively. Hence, the optimal approach for minimizing energy consumption and overall cost of execution would involve a combination of MPI and OpenMP.

3.3 Analysis of Schedules

The final analysis that has been carried out pertains to the scheduling methods employed by the programs. Initially, we will provide a concise description of the parallelism exploited in each application. Subsequently, we will proceed to modify the code in the 'lu' program, implementing a runtime schedule, and executing it using the OMP_SCHEDULE STATIC and DYNAMIC.

Both the 'bt' and 'sp' programs employ static parallel loops, some of which contain the 'nowait' directive while others do not. Additionally, certain parallel loops in these programs use both private and shared variables. Notably, the 'error.f' file contains an 'atomic' clause, which ensures mutual exclusion when accessing shared variables and in the 'rhs.f' file, a master construct is used, which designates a single thread to execute the enclosed block of code.

On the other hand, the 'lu' program includes, in addition to the details explaind before, several other OpenMP directives such as barriers, 'omp single', 'omp do reduction' and 'omp flush'. Specifically, the use of barriers allows for synchronization of the threads at specific points in the program, ensuring that they reach the same point before continuing to execute the subsequent code. The 'omp single' directive designates a single thread to execute the associated code block, while the 'omp do reduction' directive enables the efficient calculation of the sum of a series of values. Lastly, the 'omp flush' directive ensures that any changes made to memory by a thread are visible to other threads in the program.

In terms of time of execution, in Figure 3.15 we can see the heatmaps of the execution time for both schedules. Upon analysis of both schedules, it is evident that the case with 12 tasks requires the most time for the application to execute. However, the time difference between schedules is not uniform. While the case with 12 tasks takes approximately 0.1 times more, the other cases necessitate twice as much time to execute the program. It is important to note that the

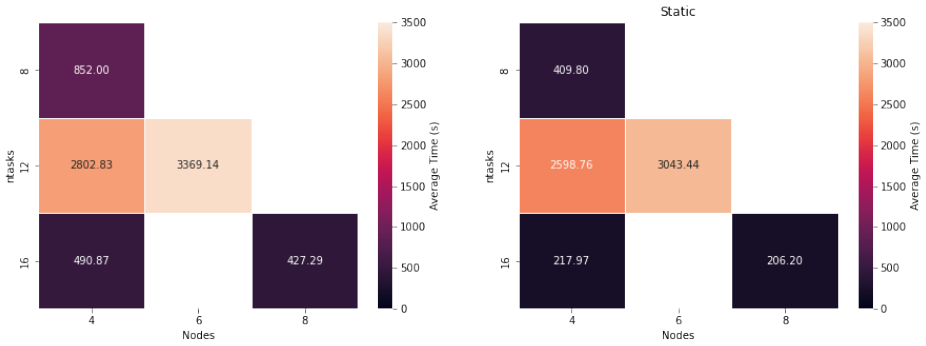

Figure 3.15: **Heatmap of execution time for dynamic and static schedule. (lu-mz)**

dynamic scheduling method may introduce overhead due to the need for run-time decision-making. In contrast, the static scheduling method may result in load imbalance if the workload is not evenly distributed among the iterations of the loop. For instances where less time is required, using a dynamic schedule it is not practical due to the added overhead. However, when additional time is needed, the overhead is compensated for the imbalance of the iterations, and the type of schedule used does not matter that much because the difference between schedules is much less.

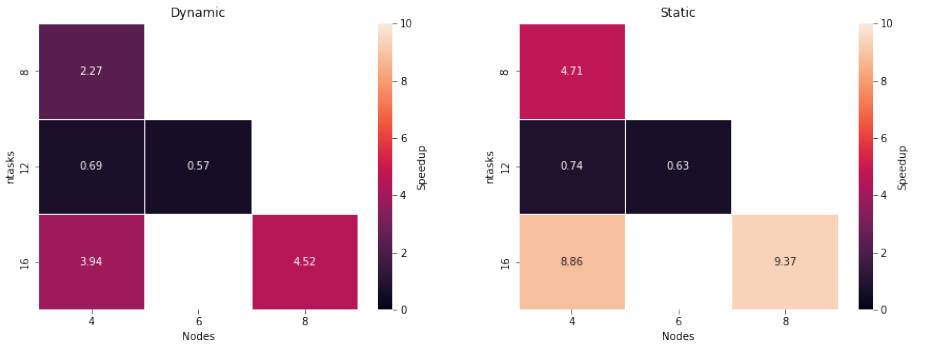

Figure 3.16: **Heatmap of speedup for dynamic and static schedule. (lu-mz)**

In terms of speedup and efficiency, plotted in Figures 3.16 and 3.17, we can see the same as before. With 12 tasks, the speedup and efficiency barley changes between the schedules. However, in other executions, the static schedule outperforms the dynamic schedule, showing a doubling effect. Notably, the execution involving 8 nodes and 16 tasks yields the best speedup performance, while the execution involving 4 nodes and 16 tasks achieves the highest level of efficiency.

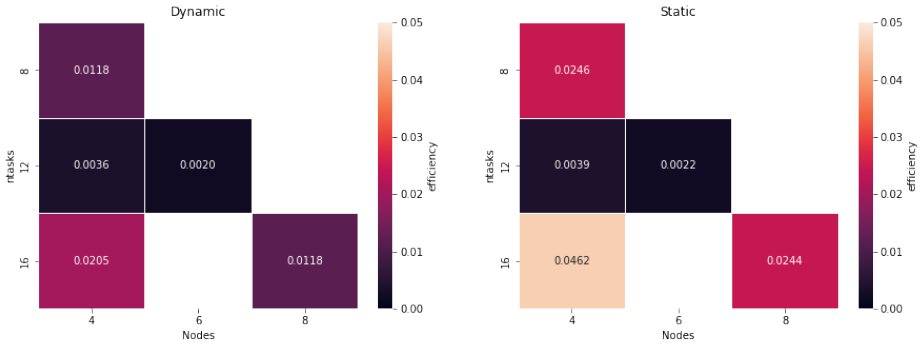

Figure 3.17: **Heatmap of efficiency for dynamic and static schedule. (lu-mz)**

3.3.1 Conclusions

In summary, it is important to note that for extended execution times (12 tasks per node), the utilization of varying schedule types does not yield a significant impact. Nonetheless, in the case of shorter execution times, employing static schedules is highly recommended due to the overheating concerns associated with dynamic schedules. The utilization of static schedules results in a 50% reduction in execution time, while simultaneously doubling both speedup and efficiency, thereby proving to be a superior alternative.

Conclusions and Organization

Based on the results of our experiments, we can deduce that the most efficient methodology for executing applications, considering execution time, speedup, efficiency, consumed energy, and cost, involves a combination of MPI and OpenMP, although the specifics of such a combination may vary depending on the application in question. Regarding scheduling, we have determined that in the experiments conducted, a static schedule is the most optimal approach, given that the tasks are not excessively unbalanced.

In order to optimize our efforts, we adopted a division of labor approach. To conduct the analysis of execution time, speedup, efficiency as well as the consumed energy and total cost of each execution, we convened for several days and worked together. During this time, we made all necessary modifications to the applications and scripts, generated executions, produced corresponding graphs, and recorded our observations in a raw format. Subsequently, Carlos was responsible for drafting the report based on this data, while Benet took charge of the final section, which dealt with the effects of modifying the schedule of OpenMP. It is noteworthy that Carlos took charge of the report's introduction and method sections, while Benet contributed to the conclusion and organizational components.

References

- ☐ SchedMD. (n.d.). sacct - Slurm documentation. Retrieved April 1, 2023, from `https://slurm.schedmd.com/sacct.html`
- ☐ Eijkhout, V. (2015). "Introduction to Parallel Computing: Second Edition". Available online at: `https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/EijkhoutParallelProgramming.pdf`

Appendix

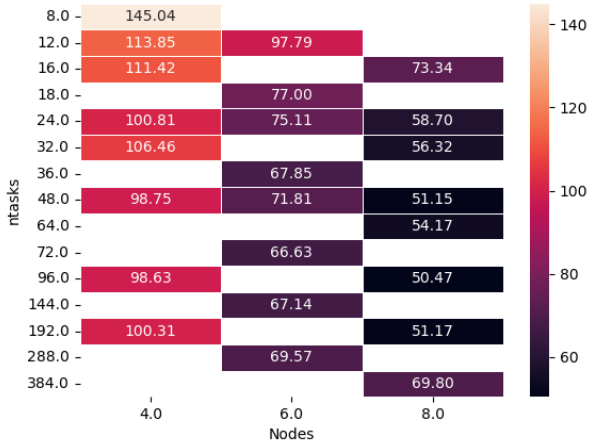

Figure 6.1: **Execution Time Heatmap: Color-coded representation of average execution times for varying numbers of tasks and threads. (bt-mz)**

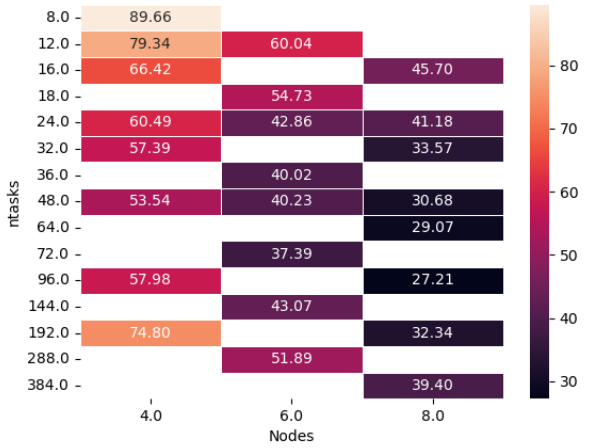

Figure 6.2: **Execution Time Heatmap: Color-coded representation of average execution times for varying numbers of tasks and threads. (sp-mz)**

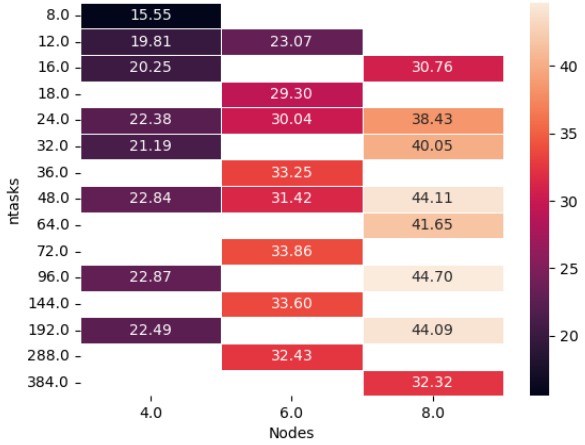

Figure 6.3: **SpeedUp Heatmap: Color-coded representation of SpeedUp for varying numbers of tasks and threads. (bt-mz)**

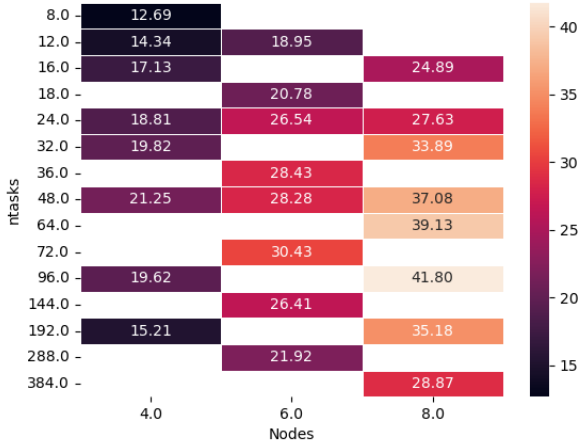

Figure 6.4: **SpeedUp Heatmap: Color-coded representation of SpeedUp for varying numbers of tasks and threads. (sp-mz)**

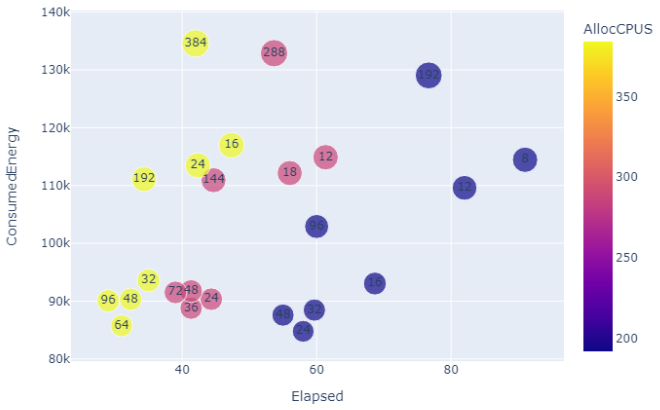

Figure 6.5: **Relationship between elapsed time and Consumed Energy. (sp-mz)**

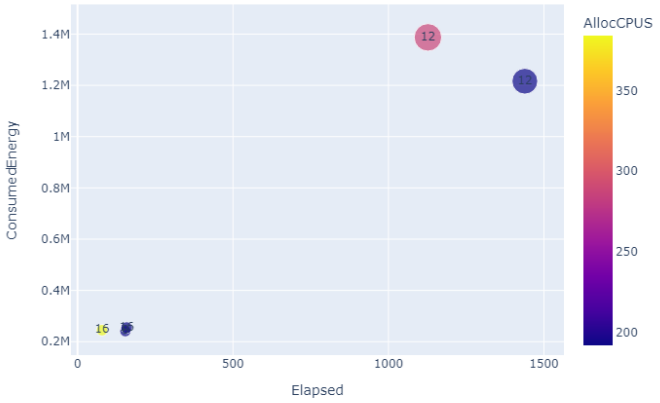

Figure 6.6: **Relationship between elapsed time and Consumed Energy. (lu-mz)**

Figure 6.7: **Relationship between Consumed Energy time and Average Power. (sp-mz)**

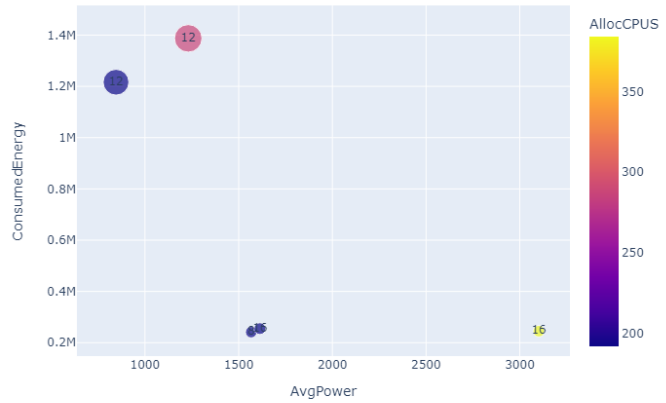

Figure 6.8: **Relationship between Consumed Energy time and Average Power. (lu-mz)**

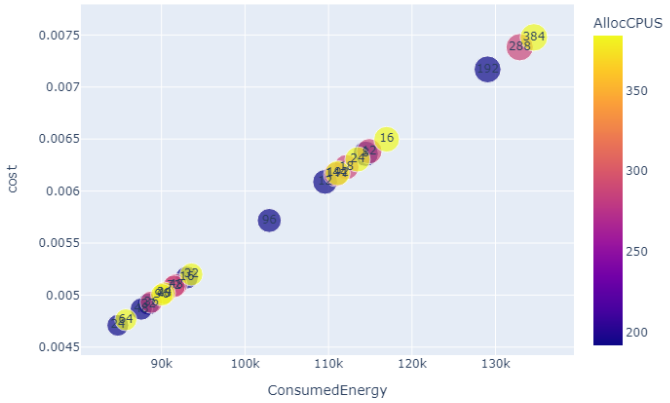

Figure 6.9: **Relationship between Consumed Energy time and cost. (sp-mz)**

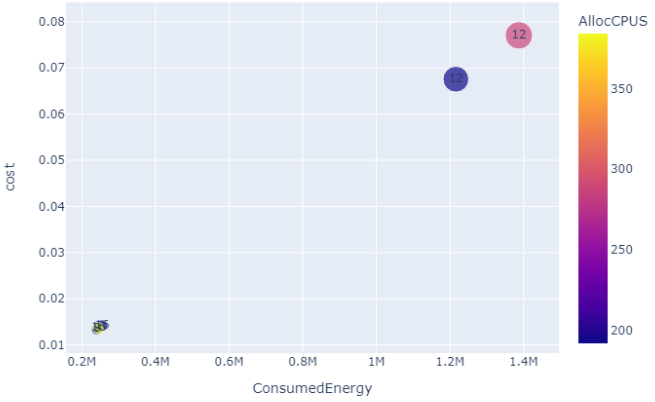

Figure 6.10: **Relationship between Consumed Energy time and cost. (lu-mz)**

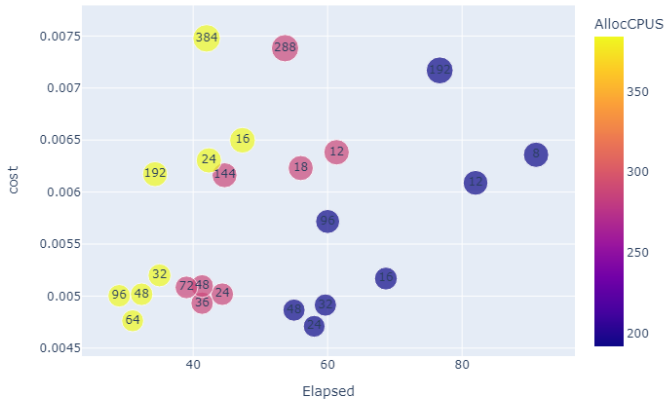

Figure 6.11: **Relationship between elapsed time and cost. (sp-mz)**

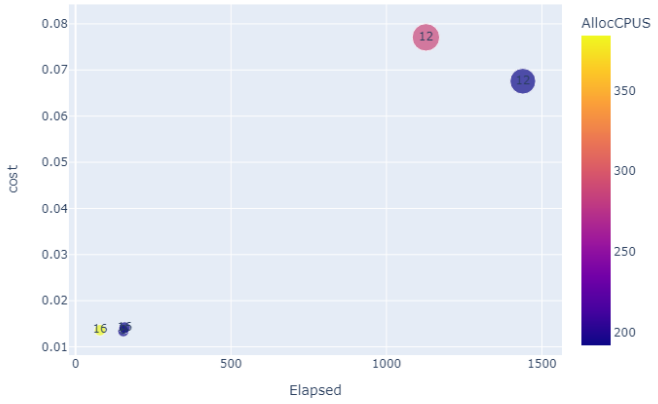

Figure 6.12: **Relationship between elapsed time and cost. (lu-mz)**

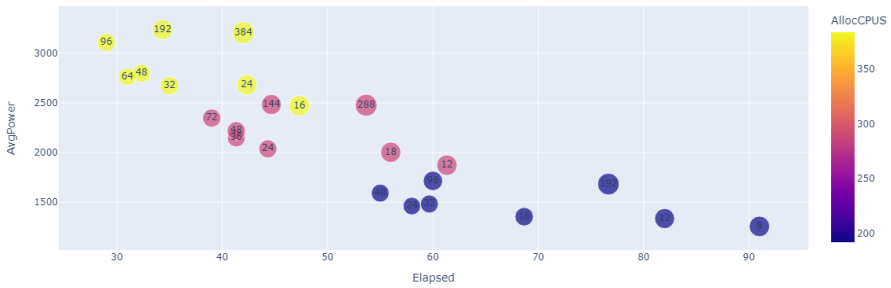

Figure 6.13: **Relationship between elapsed time and Average Power. (sp-mz)**

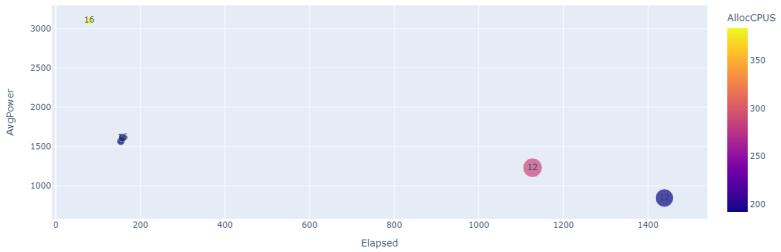

Figure 6.14: **Relationship between elapsed time and Average Power. (lu-mz)**

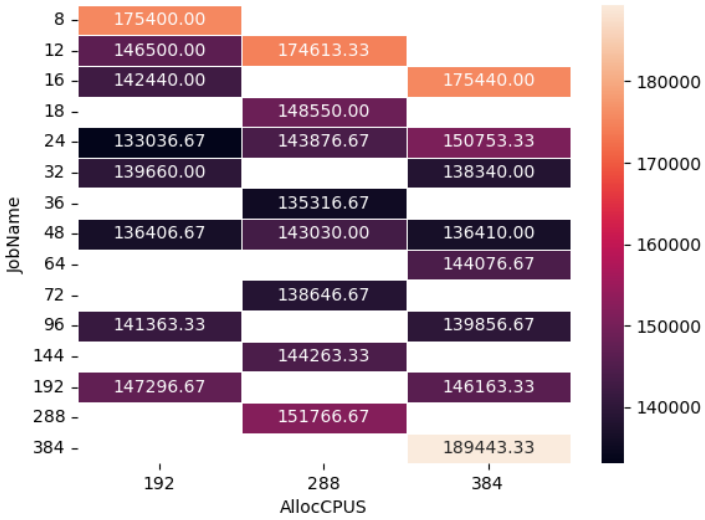

Figure 6.15: **Color-coded representation of Consumed Energy for varying numbers of tasks and threads. (bt-mz)**

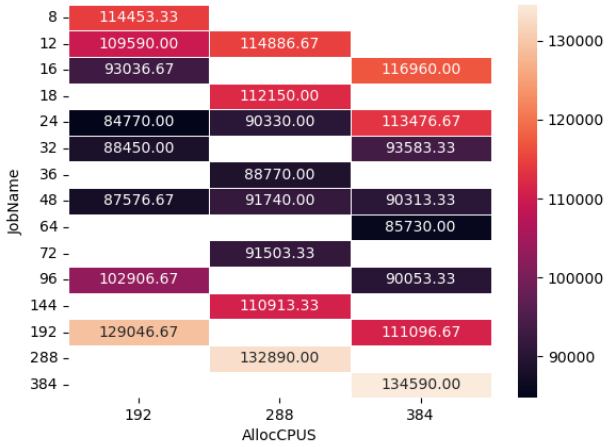

Figure 6.16: **Color-coded representation of Consumed Energy for varying numbers of tasks and threads. (sp-mz)**

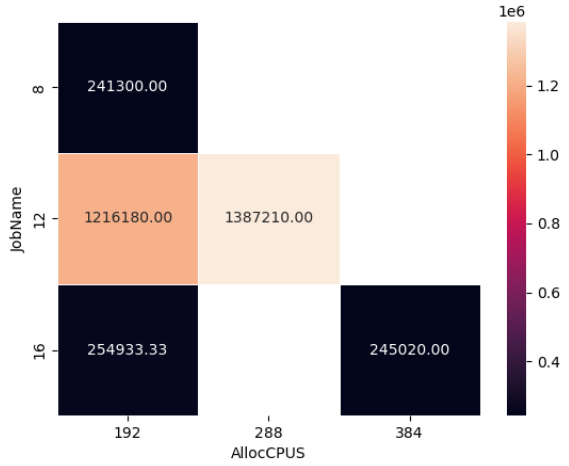

Figure 6.17: **Color-coded representation of Consumed Energy for varying numbers of tasks and threads. (lu-mz)**