Dynamic Load Balancing and Scalability Analysis of the Mandelbrot Set in a Multi-Threaded HPC Application

Francisco Pegoraro Etcheverria[1], Rayan Raddatz de Matos[1], Kenichi Brumati[1], Lucas Mello Schnorr[1]

[1]Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

Abstract. *This work investigates load balancing and scalability in High-Performance Computing (HPC) systems using the Mandelbrot Set as a benchmark. Due to its irregular and computationally intensive workload, the Mandelbrot Set is ideal for evaluating dynamic workload distribution strategies. We present a custom MPI-based implementation employing a client-server model with dynamic load balancing, where idle workers request new work from a coordinator. Our contributions include implementing this parallel application and a performance analysis highlighting load imbalance and system scalability. The main results indicate that the granularity factor significantly influences load balance, and its choice depends on the selected fractal region.*

1. Introduction

High-Performance Computing (HPC) systems are increasingly complex and extensive [Dongarra and Keyes 2024]. They accommodate applications that frequently have irregular workloads and are computationally demanding. One of the significant concerns when dealing with irregular workloads is achieving a balanced workload distribution among the computational resources available to maximize performance and resource usage. The load balancing task has a significant role in the application performance when executing it in the cluster, ensuring that each process in the system does approximately the same work during the program execution.

A common activity to study the correct usage of an HPC system is to verify if the application achieves adequate load balancing amongst its processes. This is particularly challenging for irregular workloads, as their dynamic nature makes standard static partitioning techniques unsuitable. The Mandelbrot Set [Mandelbrot 1980] is a typical application with an irregular workload because different points require varying numbers of iterations to compute. For each point c in the complex plane, the application iteratively applies the function $f(z) = z^2 + c$, starting from $z = 0$. c belongs to the Mandelbrot Set if the computation remains bounded, that is, if its magnitude does not exceed a fixed radius within a limited number of iterations. To generate a 2D image of the set, the application performs this computation for every pixel corresponding to a point in the complex plane.

The Mandelbrot Set is a well-known compute-bound application and is considered embarrassingly parallel, as all point calculations are independent. Therefore, being highly parallelizable and computationally intensive, it often serves as a benchmark for HPC to measure the performance of a computational system. In this paper, we implement an MPI version of this application from scratch with a dynamic load balancer where workers, when idle, contact the coordinator for a new set of points to work on. We then

study the load balancing and scalability of this HPC application. Our significant contributions include: (1) a multi-threaded application that implements the Mandelbrot Set using a client-server architecture where the server is an MPI application with dynamic load-balancing, (2) a performance analysis with metrics to provide an overview and load imbalance perspectives. Our main results indicate that the granularity factor significantly influences load balance, and its choice depends on the selected fractal region and the severity of how unbalanced the load is.

This paper has the following organization. Section 2 presents other load balancing performance analysis of the Mandelbrot set. Section 3 discusses materials and methods of our work, including a detailed description of the observed system. Section 4 presents the experimental results. Finally, Section 5 concludes this text with future work.

Software and Data Availability. We endeavor to make our analysis reproducible for a better science. We made available a companion material hosted in a public GitHub repository at `https://github.com/schnorr/fractal_pcad/tree/main/papers/2025_SSCAD-WIC/companion`. Our companion contains the source code of this article and the software necessary to handle the created datasets. We also include instructions to run the experiment and figures.

2. Related Work

Load balancing is a widely studied field. Chenzhong and Francis [Xu and Lau 2013] provide a general overview of load balancing techniques for parallel computers. Sandeep [Sharma et al. 2008] more specifically analyzes different load balancing algorithms and concludes that while dynamic distributed load balancing algorithms are generally considered better, the static algorithms are more stable and predictable. Another work [Mohammed et al. 2020] proposes a two-level dynamic load balancing to scientific applications that operate with MPI+OpenMP and uses the Mandelbrot application as one of its test cases because of the application's irregularity. The Mandelbrot Set as an HPC application has also been vastly studied and used as a benchmark in various works for its computationally intensive, parallelizable, and irregular nature. For example, considering the Mandelbrot Set, Gomez [Gómez 2020] has conducted a case study comparing MPI against OpenMP. Another work [Ozen et al. 2015] has specifically explored dynamic parallelism in OpenMP. Yang [Yang et al. 2011] also studies specific parameters to control load balancing among application processes. The related work demonstrates how pertinent the Mandelbrot Set is as a benchmarking application. Again, our work employs this application with a strong focus on performance analysis, general metrics (mean client time, speedup, and efficiency), and a metric specific to quantifying load imbalance (imbalance percentage), as we detail next.

3. Materials and Methods

3.1. The multi-threaded client–server MPI application for the Mandelbrot Set

The implemented multi-threaded system adopts a Client–Server architecture designed to parallelize the computation of the Mandelbrot Set while enabling efficient load balancing across multiple computing nodes. Figure 1 illustrates the overall architecture, highlighting the main threads, communication queues, and data flow between components. The Client is responsible for managing user interactions and rendering the fractal image produced by the Server. When the user requests a new region, the client issues a *payload* to the Server.

The Server comprises a central MPI coordinator which receives *payloads* from the client, discretizes the workload into smaller problems, and dynamically distributes these smaller problems to a pool of MPI workers by demand. When the Server starts, workers approach the coordinator to request work. Upon receiving a smaller problem, the workers carry out the numerical computations for them (Mandelbrot Set) before sending the *responses* to the coordinator, which forwards them to the Client.

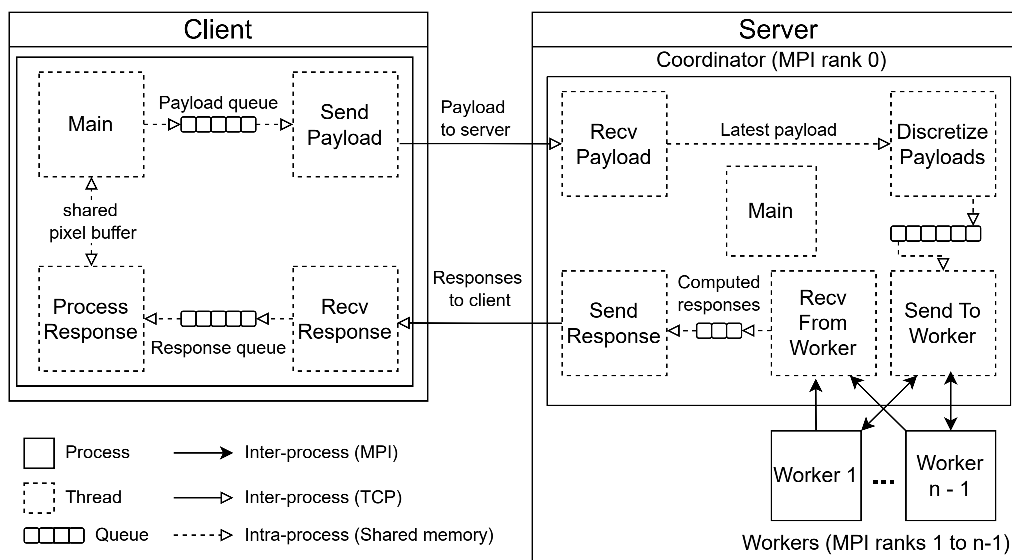


Figure 1. Multi-threaded system overview with processes, threads, and queues.

Each interaction between the Client and the Server consists of the exchange of *payload* and *response* objects. A *payload* is a data structure that specifies the region of the Mandelbrot Set to be computed, including the bounds in the complex plane (given in `long double` precision), the corresponding screen coordinates, the *depth*, which is the maximum number of iterations to apply in the Mandelbrot algorithm, and the *granularity*, which determines the size of the square blocks into which the workload is partitioned. For example, a granularity of 5 means that the Server will split the fractal space into several 5×5 square blocks. Each payload also includes an increasing generation number to identify it in the case the Client sends several *payloads* one after another. The Server replies to a single *payload* with several *response* objects, each carrying its corresponding payload, as well as the calculated depth count at each pixel position. In addition to the depth counts, the responses also include some metadata, such as the id of the worker that computed it. By delivering results block by block, the Server enables the Client to view the partial fractal regions without waiting for the entire computation to complete.

We designed the Client to be responsive and highly interactive. As shown in Figure 1, the Client contains four concurrent threads. The `Main` thread manages both rendering of the fractal image and collection of user mouse and keyboard input. When a new region is selected, `Main` constructs the corresponding payload and pushes it to a dedicated queue. The `SendPayload` thread dequeues payloads from this queue and transmits them to the Server over a TCP connection. Meanwhile, `RecvResponse` listens for incoming responses, and enqueues them into a response queue. Finally, the `ProcessResponse` thread retrieves these responses and integrates them into the dis-

played image by applying a coloring function to the calculated depth count for each pixel, updating the pixel buffer incrementally as results arrive.

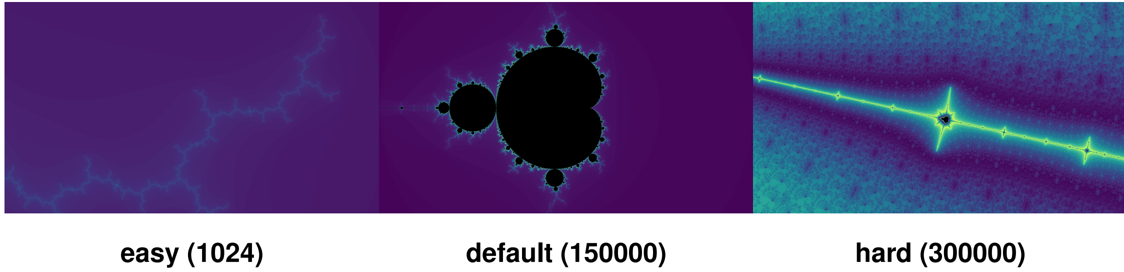
On the Server side, the `RecvPayload` thread listens for Client payloads, forwarding them to the `DiscretizePayloads` thread, which divides the requested region into several payloads sized according to the specified granularity. These are then placed into a queue, with outdated payloads being discarded to prevent workers from computing regions that are no longer relevant. As workers become available, they request a new payload from the coordinator. The `SendToWorker` thread dynamically assigns them payloads from the queue. Each worker independently computes a response, producing the depth counts for all pixels in that subregion. Once the response becomes ready, it is sent to the `RecvFromWorker` thread, which enqueues it to a response queue. These responses are then collected by `SendResponse`, which sends them back to the Client.

3.2. Hardware & Software configuration

We run all experiments at the *Parque Computacional de Alto Desempenho* (PCAD) at INF/UFRGS. The Client executes on a single *draco* node, while the server executes on one to six *cei* nodes. The draco node on which the Client executes has two Intel Xeon E5-2640 v2 processors at 2.00 GHz. Each cei node, used for the compute-bound part, has two Intel Xeon Silver 4116 processors at 2.10 GHz, providing 24 physical cores each for a total of 144 physical cores. In all experiments, we have exclusive access to the machines without any type of virtualization. We also use the *performance* frequency governor of the `acpi-cpufreq`. The MPI implementation was OpenMPI version 4.1.4 and the Linux Kernel 6.1.0 with SMP support as released by the Debian 12 distribution. The Client–Server Ethernet network is 1Gbps, while the MPI application executes in a 10Gbps Ethernet switch.

3.3. Experimental Project

We designed a set of experiments with various input parameters to evaluate the performance, scalability and load balancing of the application on the target system. These parameters were chosen to test different computational characteristics of the application, enabling us to assess how the system behaves under different workloads. The experiments consisted of rendering fractal images with a resolution of 1920×1080 pixels. Each execution is the combination of a value of the following factors: Granularity, Number of Nodes, and Fractal Cases. The **Granularity** factor has the six levels: [5×5, 10×10, 20×20, 40×40, 60×60, 120×120], respectively resulting in [82944, 20736, 5184, 1296, 576, 144] tasks for workers. Smaller blocks improve load balancing but increase communication overhead. Larger blocks may lead to severe load imbalance. The **Number of Nodes** factor varies from 1 to 6, with each *node* contributing 24 physical cores to the server. This corresponds to a total of 24 to 144 MPI ranks (steps of 24), enabling the evaluation of scalability. Finally, **Fractal Cases** has three levels: [easy, default, hard]. Figure 2 illustrates representative images of each region. The *easy* (maximum depth of 1024) depicts a region where most points escape in only a few iterations, testing the communication overhead, rather than computational speed. The *default* (150000) depicts a typical unbalanced Mandelbrot fractal region, containing both points that are computationally intensive, as well as many points that escape quickly, stressing load balancing. Finally, the *hard* (300000) depicts a deep region that is computationally intensive but balanced, to assess throughput. Max depth values were selected to keep execution time bounded.



easy (1024) **default (150000)** **hard (300000)**

Figure 2. The three fractal cases, with the corresponding maximum depth values.

With these factors, we adopt a Full Factorial Design [Jain 1990], enabling the exploration of all possible combinations of factors, resulting in 108 distinct configurations (6×6×3). Each configuration has been executed ten times so we can assess the experimental variability, and the execution order has been randomized to avoid potential bias. All experiments consider a simplified Client as we executed everything in the cluster without a graphical interface. Our textual Client instead receives parameters through the command line. The `ProcessResponse` thread is therefore absent, and the `Main` thread enqueues the payload and dequeues responses from the `RecvResponse` thread.

3.4. Observability

We manually instrument the code of the Client and Server to collect and combine specific events and derive both execution time and load balancing metrics. In the Client, we register the elapsed time between the creation of each payload and the arrival of the first response, as well as the last response. These metrics enable us to verify the latency of the application as well as total perceived time from the user perspective. In the server, we measured the time between a payload being received and its discretization, the first and last responses being received by the `RecvFromWorker` thread, and the moments these responses are sent to the Client in the `SendResponse` thread. This information allow us to verify the discretization cost, and the amount of compute time from the perspective of the coordinator. Finally, in each MPI worker, we measured the individual times to compute each payload, their pixel and depth counts, as well as the aggregate of these values, allowing us to assess load balancing.

4. Results

We present the performance evaluation of our multi-threaded MPI application based on the experiments described earlier. We focus on four key metrics: the mean client time, speedup, efficiency, and imbalance percentage. The **Mean Client Time** represents the total time taken for the Client to receive the fully computed fractal for each case (*payload*). The **Speedup** measures the ratio of the mean Client time with a single node for a given case and granularity setting to the mean Client time with another number of nodes for that same setting. That is, for a given number of nodes n, $S(n) = \frac{T(1)}{T(n)}$. We emphasize that our speedup metric is relative to the number of nodes rather than processors. Our server architecture is asymmetric, which necessitates a careful definition of ideal performance and efficiency. The baseline configuration on a single node uses 23 workers and one coordinator, while each additional node contributes 24 workers. This results in a worker count for n nodes of $24n - 1$. Standard efficiency calculations using node count would

yield misleading values above 1.0 due to this uneven worker distribution. Therefore, we normalize our metrics based on worker count rather than node count. We define the ideal speedup as $S_{ideal}(n) = \frac{24n-1}{23}$, and **Efficiency** as $E(n) = \frac{S(n)}{S_{ideal}(n)}$. This method of computing S and E ensures that perfect linear scaling as workers are added results in efficiency = 1.0, enabling fair comparison across configurations. Finally, the **Imbalance Percentage** [DeRose et al. 2007] depicts how unevenly the computational workload is distributed among workers. Lower values are better. It is calculated as:

$$\text{Imbalance Percentage} = \frac{L_{\text{max}} - L_{\text{avg}}}{L_{\text{max}}} \times \frac{n}{n-1} \quad (1)$$

where L_{max} is the computation time of the slowest worker, L_{avg} is the average computation time across all workers, and n is the number of workers. We report the mean value across the 10 trials. In our analysis we focus solely on Client times, which directly reflect user-perceived performance, as the coordinator metrics closely mirror client-side values. We also focus on worker-level timings, which reveal the degree of load balancing achieved.

Figures 3, 4 and 5 depict the time, speedup and efficiency results. We see that performance appears to scale well with the addition of nodes for the *default* and *hard* cases, provided an adequate granularity (nor low nor high). The granularity 20 appears to be the best, with an efficiency of around 0.98 with 6 nodes in the *hard* case, and approximately 0.85 in the *default* case. This is likely due to it presenting a good trade-off between the payload size and the number of payloads, with small communication overhead while providing good load balancing.

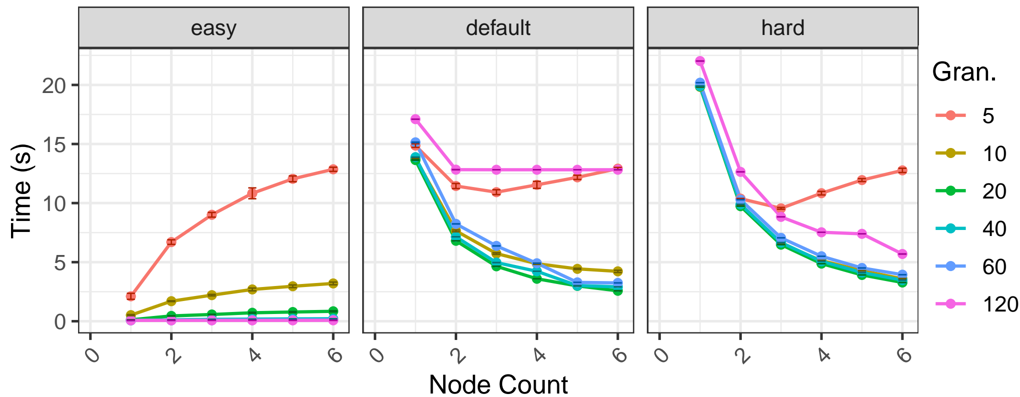


Figure 3. Mean Client time for each case, with 99% CI error bars.

This interpretation can be confirmed in Figure 6, which shows generally better load balancing for lower granularities. The load balancing at higher granularity values tends to degrade as the number of nodes increases. The *default* case in particular seems to suffer from more worker imbalance than the *hard* case, due to the fractal region having a mix of very easy and very hard regions.

In contrast, the *easy* case shows a different trend: higher granularities perform better, and increasing node counts worsen performance. Because most points in this region escape in only a few iterations, computation is inexpensive, and the bottleneck is communication. As such, lower granularities increase overhead, which seems to worsen as more nodes are added. This effect is especially visible at granularity 5 (see Figure 3 for

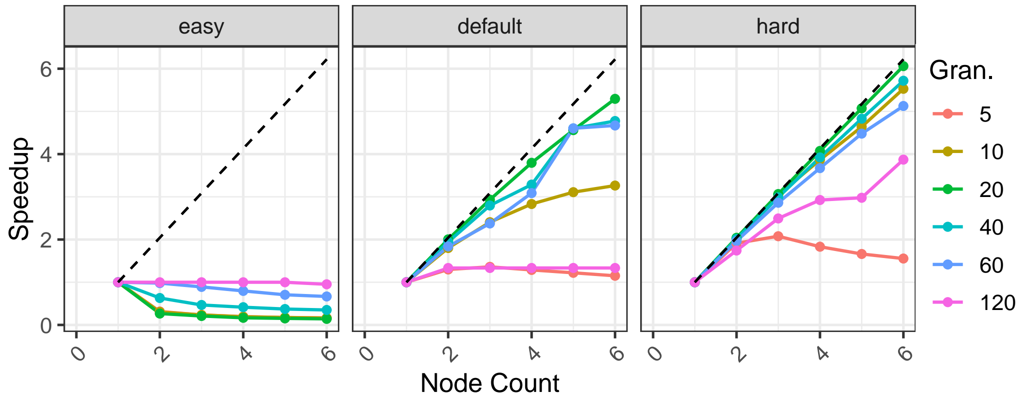


Figure 4. Speedup and ideal speedup for each case.

instance): in the *default* and *hard* cases, performance worsens past 3 nodes, nearly matching the times observed in the *easy* case. We conclude that the performance is limited by communication rather than computation time at such low granularities.

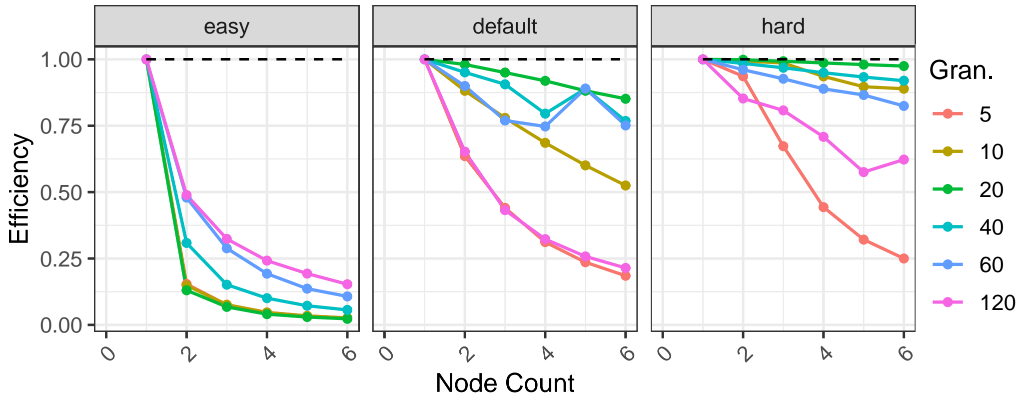


Figure 5. Efficiency for each case.

Imbalance is also high across granularities in the *easy* case, as the work is so light that some workers can finish a payload and request another, while other workers are still waiting for their next payload.

5. Conclusion

This work presented a dynamic, multi-threaded MPI-based implementation of the Mandelbrot Set to study load balancing and scalability in HPC systems. Through extensive experimentation, we demonstrated that workload granularity plays a crucial role in performance, with optimal values depending on the computational characteristics of the fractal region. These results show that scaling depends on the balance between computation and communication costs. For harder fractal regions, the system scales very well with additional nodes when granularity is appropriately chosen, with a granularity of 20 striking the best balance. However, for simpler regions, communication overhead dominates and additional nodes can even reduce performance. These insights highlight the importance of tuning granularity based on workload characteristics to achieve efficient parallel execution. As future work, we plan to investigate varying granularity values based on neighborhood fractal depth and its impact on performance and load balance.

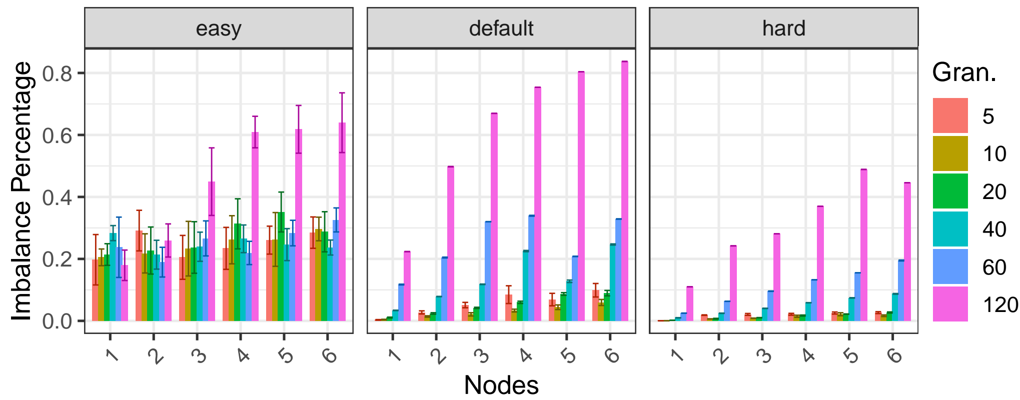


Figure 6. Mean Imbalance Percentage for each case, with 99% CI error bars.

Acknowledgments. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. We thank FAPERGS and CNPq for financial support, including scientific initiation scholarships from FAPERGS (PROBIC) and CNPq (PBIC). We also thank UFRGS for all institutional support, and the Parallel and Distributed Processing Research Group for cluster resources.

References

DeRose, L., Homer, B., and Johnson, D. (2007). Detecting application load imbalance on high end massively parallel systems. In *European Conference on Parallel Processing*, pages 150–159. Springer.

Dongarra, J. and Keyes, D. E. (2024). The co-evolution of computational physics and high-performance computing. *Nature Reviews Physics*.

Gómez, E. S. (2020). Mpi vs openmp: A case study on parallel generation of mandelbrot set. *Innovation and Software*, 1(2):12–26.

Jain, R. (1990). *The art of computer systems performance analysis*. john wiley & sons.

Mandelbrot, B. B. (1980). "fractal aspects of the iteration of z $\rightarrow$ z λ(1-z) for complex λ and z". *Annals of the New York Academy of Sciences*, 357(1):249–259.

Mohammed, A., Cavelan, A., Ciorba, F. M., Cabezón, R. M., and Banicescu, I. (2020). Two-level dynamic load balancing for high performance scientific applications. In *SIAM Conference on Parallel Processing for Scientific Computing*.

Ozen, G., Ayguade, E., and Labarta, J. (2015). Exploring dynamic parallelism in openmp. In *Proceedings of the Second Workshop on Accelerator Programming Using Directives*, WACCPD '15, New York, NY, USA. Association for Computing Machinery.

Sharma, S., Singh, S., and Sharma, M. (2008). Performance analysis of load balancing algorithms. *International Journal of Civil and Environmental Engineering*, 2(2):367.

Xu, C. and Lau, F. C. M. (2013). *Load Balancing in Parallel Computers: Theory and Practice*. Springer Publishing Company, Incorporated, 1st edition.

Yang, C.-T., Wu, C.-C., and Chang, J.-H. (2011). Performance-based parallel loop self-scheduling using hybrid openmp and mpi programming on multicore smp clusters. *Concurrency and Computation: Practice and Experience*, 23(8):721–744.