Exploiting future Exascale systems with Partitioned Global Address Spaces

Bruno Manuel Mendes Amorim

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. João Pedro Faria Mendonça Barreto
Prof. José Carlos Alves Pereira Monteiro

Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. José Carlos Alves Pereira Monteiro
Member of the Committee: Prof. Vasco Miguel Gomes Nunes Manquinho

January 2021

Abstract

The high-performance computing (HPC) industry is determinedly building towards next-generation exascale supercomputers. With this big leap in performance, the number of cores present in these future systems will be immense. Current state-of-the-art bulk synchronous two-sided communication models might not provide the massive performance and scalability required to exploit the power of these future systems. A paradigm shift towards an asynchronous communication and execution model to support the increasing number of nodes present in future supercomputers seems to be unavoidable. GASPI (Global Address Space Programming Interface) offers a Partitioned Global Address Space (PGAS) and allows for zero-copy data transfers that are completely asynchronous and one-sided, enabling a true overlap of communication and computation. Although promising, the PGAS model is still immature. Industrial-level HPC applications have yet to be developed with this model, which generates uncertainty about the model's effectiveness with real-world applications. The goal of this thesis is to contribute to a better understanding of the actual strengths and limitations of the GASPI programming model when applied to HPC applications that will benefit from future exascale systems. To achieve that, we focused on the parallelization of a representative method from the domain of plasma physics, the Particle-in-Cell (PIC) method. Departing from an existing sequential implementation (ZPIC), we evaluated the performance and programming productivity of GASPI when used to parallelize this implementation. After a thorough performance evaluation on the MareNostrum 4 supercomputer we concluded that, while GASPI might fall behind the industry standard in terms of usability, its performance and scalability reliably outperformed an MPI implementation of the same application.

Keywords: Exascale, High-performance Computing, GASPI, PGAS

Resumo

A indústria de computação de alto desempenho está a evoluir rapidamente, alimentando a possibilidade de se construírem os primeiros supercomputadores exascale num futuro próximo. Com este grande salto em desempenho, o número de núcleos de CPU presentes nestes sistemas será colossal. Os modelos de comunicação síncrona usados atualmente podem não fornecer o desempenho e a escalabilidade necessários para explorar por completo o potencial dos supercomputadores do futuro. Uma mudança de paradigma para um modelo de comunicação e execução assíncrona pode ser necessário para suportar o enorme número de núcleos de CPU presentes nos supercomputadores do futuro. GASPI (Global Address Space Programming Interface) implementa um espaço de endereço global particionado (PGAS) e permite transferências de dados completamente assíncronas e unilaterais, permitindo uma sobreposição de comunicação com computação. Embora promissor, o modelo PGAS ainda é imaturo. A falta de aplicações de nível industrial com este modelo gera incerteza sobre a eficácia do paradigma. O objetivo desta dissertação é contribuir para uma melhor compreensão das vantagens e desvantagens do modelo de programação do GASPI quando aplicado a problemas do mundo real. Com esse objetivo, desenvolvemos uma versão distribuída de uma implementação de um método representativo do domínio da física de plasmas, o método Particle-in-Cell (PIC). Partindo de uma implementação sequencial existente (ZPIC) avaliámos a produtividade e o desempenho do GASPI. Após uma profunda avaliação de desempenho efetuada no supercomputador MareNostrum 4, concluímos que, embora o GASPI possa ficar para trás do padrão da indústria em termos de usabilidade, seu desempenho e escalabilidade superaram uma implementação da mesma aplicação em MPI.

Palavras-chave: Exascale, Computação de alto desempenho, GASPI, PGAS

Contents

List of Figures

Acronyms

API	Application Programming Interface
COMA	Cache-Only Memory Architecture
CPU	Central Processing Unit
DIMM	Dual In-line Memory Module
DSM	Distributed Shared Memory
GASPI	Global Address Space Programming Interface
HPC	High-Performance Computing
IDL	Interactive Data Language
MN4	MareNostrum 4
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi-Processing
PGAS	Partitioned Global Address Space
PIC	Particle-in-Cell
POSIX	Portable Operating System Interface
RAM	Random-Access Memory
RDMA	Remote Direct Memory Access
SIMD	Single Instruction Multiple Data
SMP	Shared Memory Processor
UMA	Uniform Memory Access

Chapter 1

Introduction

As we approach the creation of the first exascale supercomputers, those capable of operating at a rate in the order of 10^{18} floating-point operations per second, there is a great deal of uncertainty in the high-performance computing community regarding the programming models that will be used in these future systems.

Current state-of-the-art two-sided communication technologies offer no guarantee that they will be able to power next-generation supercomputers. Shared memory models are, in comparison to other paradigms, not very scalable [1, 2]. Additionally, shared memory cache-coherent hardware is expensive and hard to design [3]. Distributed shared memory systems, in theory, offer the ease of programming of shared memory communication models while offering the scalability of distributed memory architectures, with the possibility of truly heterogeneous systems [4, 5]. But in reality, the paradigm requires more research and development before their promise of state-of-the-art message-passing performance can be delivered [6]. Even then, there is reason to believe that the synchronous two-sided communication model used by MPI is not prepared to be implemented on the magnitude of an exascale system. A paradigm shift towards an asynchronous communication and execution model to support the increasing number of nodes present in future supercomputers seems to be unavoidable.

This is where GASPI [7] (Global Address Space Programming Interface) becomes relevant. GASPI implements a Partitioned Global Address Space (PGAS) and offers the possibility of a true overlap of communication and computation. This is possible by means of a one-sided remote direct memory access (RDMA) communication model. This model allows for zero-copy data transfers that are completely asynchronous and one-sided. A notification mechanism is employed to inform the receiving side on the completion of a data transfer. If used correctly, this model allows for communication routines with no inherent synchronization between tasks.

The novelty of the technology introduces inevitable challenges. Mainly the lack of proper documentation and debugging tools, which severely hinders the development of GASPI applications.

This is further complicated by the novel parallelization and communication mechanisms introduced. Programmers are usually more comfortable and experienced with synchronous communication, the "extremely" asynchronous communication model present in GASPI may require some preparation by the programmers before they can use it effectively. The inability to incrementally parallelize existing serial code further complicates software development.

While many of these issues are common in adopting new and innovative programming models, we are faced with a question. Do the performance benefits that GASPI promises outweigh the described setbacks? Answering this question will be the main focus of this thesis.

1.1 Contributions

To assess the beneficial impact that GASPI might offer, we developed a distributed version of an existing plasma simulation tool, called ZPIC [8], using GPI-2[1], the latest implementation of the GASPI standard. Plasma simulations are a very pertinent class of high-performance computing applications and are employed on many major research subjects ranging from laser wakefield acceleration [9, 10] to thermonuclear fusion [8]. ZPIC implements a Particle-in-Cell algorithm. This technique simulates the motion of each particle individually in continuous space, but current and charge densities are weighted onto a stationary computational grid. Parallel Particle-in-Cell simulation using traditional two-sided communication models is a well-studied field with various existing implementations like OSIRIS [11]. However, to the best of our knowledge, parallel Particle-in-Cell implementations using a PGAS are not well investigated.

The parallelization effort followed the common architecture of parallel Particle-in-Cell implementations. The simulation space is divided in a checkerboard-like fashion and distributed throughout the participating nodes. The implementation is heavily focused on maximizing asynchronous communication and minimizing synchronization. This focus allows for overlapping communication and computation, crucial for an efficient parallel implementation.

The obtained results were quite satisfactory. Our GASPI implementation managed to stay competitive, and even outperform, an optimized implementation powered by the current industry standard. Although GASPI's usability is currently inferior to the current state of the art, GASPI provides a significant advancement for the high-performance computing industry.

1.2 Outline

This document is organized as follows: In Chapter 2, we cover some parallel programming models in use today and describe some implementations of each paradigm. Additionally, we give an overview of the GASPI standard and present a simple code example using both GASPI and MPI to compare basic communication routines offered by the two standards.

In Chapter 3, we address the use of computers in simulating scientific plasma experiments, also covering the Particle-in-Cell method. Next, we describe some software implementations of the

[1] http://www.gpi-site.com/

Particle-in-Cell, with a focus on the ZPIC plasma simulation tool. Furthermore, we cover the plasma experiments that we will use to test our distributed implementation.

Chapter 4 gives a comprehensive description of our distributed ZPIC implementation. We go over the partitioning of the simulation space, the distributed particle generation, and the implemented communication routines. This chapter ends with an overview of our GASPI/OpenMP hybrid code.

In Chapter 5, we start by explaining the experimental conditions for the performance evaluation of our implementations. Next, we briefly explain how the GASPI code was validated. Then, we thoroughly analyze the performance of our GASPI implementation, comparing it to a reference MPI implementation of the same plasma simulation tool. Additionally, we share our findings regarding the performance bottlenecks of our implementation. Finally, we analyze the execution performance of our GASPI/OpenMP code.

Lastly, in Chapter 6, we conclude by sharing our user experience with GASPI and give our final remarks about GASPI in the nearing exascale era.

Chapter 2

Parallel Programming

In this chapter, we cover some parallel programming paradigms in use today, along with some implementations of each programming model. Additionally, we give a brief description of the GASPI standard and present a small code comparison between GASPI and MPI. In conclusion, we present a brief overview of how to evaluate the performance of parallel applications.

2.1 Shared Memory

In parallel computing, shared memory is an architecture where memory is shared by multiple processing units, as they share a single address space. These processing units can be individual cores inside a multi-core CPU or multiple Shared Memory Processors (SMPs) all linked to the same logical memory [3].

Shared memory systems can be categorized in the following categories [1]:

- **Uniform Memory Access (UMA)**: shared memory is accessible by all processors uniformly, the same way a single processor accesses its memory.

- **Non-Uniform Memory Access (NUMA)**: each processor has part of the shared memory attached. The memory still has a single address space so any processor can access any memory location directly using its real address. However, the access time depends on the distance between the processor and the physical memory. Also, if a memory hierarchy is present memory accesses will have a non-uniform latency and throughput.

- **Cache-Only Memory Architecture (COMA)**: Similar to NUMA, but the local memories of each processor are used only as caches. There is no memory hierarchy and the address space is made of all the caches. This architecture requires that data needs to be migrated to the processor using it.

In shared memory systems, communication between tasks (usually threads) is implicit and as simple as writing to and reading from memory.

Shared memory architectures suffer from some drawbacks: Performance degradation is likely to happen when several processors try to access the shared memory due to bus contention. The usual solution to this problem is to resort to caches.

However, having several copies of data spread throughout multiple caches is probable to result in a memory coherence problem [1]. The use of caches might also introduce false sharing. False sharing arises when two processors repeatedly write to two different words of the same cache block in an interleaved fashion. This causes the cache block to bounce back and forth between the two caches, thus ruining cache hit rate [12].

2.1.1 POSIX Threads

Portable Operating System Interface (POSIX) Threads [13], usually known as pthreads, is a set of procedure calls and types defined in the POSIX.1c standard. Pthreads allows the programmer to explicitly manage threads in all aspects. Thread creation and destruction, as well as specifying critical regions of code by employing mutexes, and access control by semaphores are all explicitly handled by the programmer.

The pthreads model is cumbersome, it is lower level than necessary for most scientific applications and is aimed at providing task parallelism, not data parallelism [14]. This level of control over threads makes programming time consuming and error-prone since all synchronization, thread lifetime management, workload distribution, and assignment fall to the programmer.

2.1.2 OpenMP

OpenMP is an application programming interface (API) that supports multi-platform shared-memory parallel programming [15]. OpenMP's compiler directives and callable runtime library routines extend well-established programming languages like C, C++, and Fortran [13].

Thread lifetime management is implicit. Compiler directives are used to specify which sections of code are to be run in parallel. OpenMP provides numerous constructs to support implicit synchronization which programmers can use to specify the region of code where synchronization is necessary. The actual synchronization process is thus relieved from the programmer's responsibility [13]. Workload distribution and assignment are also implicit and handled by OpenMP.

Additionally, OpenMP makes incremental parallelization possible, allowing for easy parallelization of existing code [14].

2.2 Distributed Memory

Distributed memory is an architecture where each task (usually processes) has its own private address space. Since there is no global memory, communication between processing units is required for accessing remote data [1, 3].

Communication between workers is explicit and very frequently handled by a message passing interface, a well-established paradigm for parallel programming [2, 3]. The number of processors on modern massive parallel systems can reach the hundreds of thousands [1].

Computer hardware aimed at distributed computation is much easier to design than cache-coherent shared memory systems. Also, as all communication is explicitly handled by the programmer, there are fewer unforeseen performance issues than with implicit communication [3].

In distributed memory systems the interconnection network between nodes plays a significant role in determining the communication speed. There are many known network topologies that favor different communication patterns [1].

Thanks to simpler hardware [3], higher efficiency [2] and scalability [1] (owing to nearly no performance loss to memory contention) distributed memory systems coupled with message passing interfaces are a very attractive choice for large-scale high-performance parallel applications.

2.2.1 MPI

MPI (Message Passing Interface) is a specification for a message passing programming interface standard [2]. MPI offers a rich collection of point-to-point communication procedures and collective operations for data movement, global computation, and synchronization.

The standard Receive routine offered by MPI (*MPI_Recv*) blocks until the communication is complete. The standard Send (*MPI_Send*), depending on the implementation, either blocks the sender until the data has been transferred into a matching receive buffer or copied into a temporary system buffer. The specification also defines asynchronous communication routines (*MPI_Isend* and *MPI_Irecv*) that block neither the sender nor the receiver. Since these procedures are asynchronous, there is a need to check if the data transfer has been completed. To achieve this the standard defines the functions *MPI_Test* and *MPI_Wait* to check the status of a data transfer and wait for a transmission to be complete respectively [1].

These non-blocking communication procedures allow for overlapping communication and computation, which is crucial for high-efficiency parallel computation. However, communication in MPI requires data to be copied to and from temporary buffers, resulting in misused CPU time. This overhead can be significant if the messages are large in nature.

Collective operations are also defined in the specification. Collective operations are procedures that involve a group of tasks, rather than just a sender and a receiver. Some of these are *MPI_Barrier* that blocks tasks in a group from proceeding until all tasks have reached the barrier; *MPI_Reduce* that applies a commutative user-defined operation to data provided by each task in a group, only one of the tasks gets the result; And *MPI_Allreduce*, same as *MPI_Reduce* but all tasks in the group get the final result [1].

Currently, MPI is extensively used in clusters and other distributed memory systems due to its rich functionality [1] and is considered the defacto standard for developing high-performance computing applications on distributed memory architectures [16]. It provides language bindings for C, C++, and Fortran [13]. Some well-known MPI implementations are MPICH [2] and OpenMPI [17].

2.2.2 Hybrid Programming

SMP clusters are becoming increasingly more prominent in the high-performance computing industry. While message passing programs can be easily ported to these systems, the paradigm is not well suited for intra-node communication. To this end, a combination of message passing and shared memory models into a hybrid programming approach [18] could be employed to better exploit the cost-to-performance efficiency of these systems. For example, employing both MPI and OpenMP for inter and intra-node communication, respectively.

Although hybrid programming approaches might make use of other programming languages, mixed MPI and OpenMP implementations are likely to represent the most prevalent use of hybrid programming on SMP clusters due to their portability and status as industry standards in their respective paradigms.

While the distributed memory architecture of MPI allows for high scalability, it suffers from some drawbacks. Communication latency is often the dominating overhead, forcing a coarse code granularity. On the other hand, OpenMP makes better use of shared memory, its run time scheduling allows for both fine and course-grained parallelism. However, OpenMP code can only run on a shared memory context, and with limited scalability.

Combining both paradigms allows the programmer to take advantage of the benefits of both models. With MPI handling high-level parallelism and communication, while OpenMP deals with the lower level parallelization responsibilities. Situations where the implementation is plagued by poorly performing inter-node communication latency will likely see improvement with this approach as it reduces number of inter-node messages. This is achieved by increasing the size of said messages as a result of the increased number of threads per communicating task. If load balancing poses a problem to application performance, this approach allows MPI to implement a more coarse-grained view of the problem while OpenMP implicitly handles much of the load balancing between tasks on a node.

However, while effective, this approach is not always the most indicated and cannot be regarded as the ideal technique for all problems. In reality, significant attention must be allocated to the nature of the problem at hand before employing a hybrid parallelization approach. Circumstances where node memory contention or false sharing are introduced or situations where the OpenMP code suffers from poor memory access patterns will prevent this approach from performing effectively.

2.2.3 GASPI

GASPI [7] (Global Address Space Programming Interface) is a specification for a Partitioned Global Address Space (PGAS) API that aims to provide a scalable and efficient alternative for bulk synchronous two-sided communication patterns, with a one-sided asynchronous communication and execution model.

To achieve this GASPI makes use of RDMA (Remote Direct Memory Access) driven communication [19]. RDMA [20, 21] is a common component of high-performance networks that allows for one-sided data transfers. Unlike usual Send/Receive routines of message passing interfaces, RDMA operations allow machines to read from (and write to) pre-defined memory regions of remote machines, with no participation from the CPU on the remote side. Additionally, since the data is being read from and written directly to pre-determined memory regions, there is no need to copy the data to and from temporary buffers. These zero-copy transfers reduce CPU overhead (to zero) and latency compared to traditional message passing.

GASPI introduces memory segments, which are parts of the global address space that can be hosted by the GASPI processes. Local segments are accessible by standard load/stores operations, just like ordinary allocated memory. Remote segments can be accessed by every other process through the GASPI read and write operations. Modern hardware often employs a variety of memory levels that vary in bandwidth and latency of read and write operations. GASPI memory segments are able to represent and map many varieties of memory hardware in software as a contiguous block of virtual memory.

GASPI communication requests are posted to queues. These queues allow for higher scalability and for separation of concerns. Communication requests that concern distinct parts of the program can be posted to different queues. These requests are then synchronized with requests on the same queue, but different queues are independent from each other and are not synchronized, improving scalability. The specification guarantees fairness for requests posted to different queues, so no queue will have its requests delayed indefinitely.

Remote operations that are completely handled by the calling side allow for communication with no inherent synchronization between the local and the remote process. However, at some point, the receiving process needs confirmation as to whether the data was received and if it is ready for use. To this end, GASPI introduces notifications. Notifications allow processes to inform each other of completed data transfers.

In order to develop fault-tolerant parallel programs, GASPI offers a timeout mechanism for all potentially blocking procedures. Timeouts can be specified in milliseconds or may be set block the procedure call until completion.

The API offers both synchronous and asynchronous collective communication operations. These calls may be interrupted by the timeout mechanism. If so, the operation is then continued in the next call of the routine. If a collective operation is interrupted, it will need several invocations until it is complete. Some of the collective procedures offered by GASPI are *gaspi_barrier* and *gaspi_allreduce*. It is important to note that GASPI does not define a "reduce" operation equivalent to the one offered by MPI [19].

2.2.4 GASPI/MPI Code Example

To illustrate the main differences in basic communication procedures in MPI and GASPI we will present a small code example. In the following programs, each process does a simple data transfer to another process in a round-robin fashion. Firstly, an MPI version:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define RIGHT(proc_id,num_proc) ((proc_id + num_proc + 1) % num_proc)
#define LEFT(proc_id,num_proc) ((proc_id + num_proc - 1) % num_proc)

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int proc_id, num_proc;
    MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);

    static const int VLEN = 1 << 2;

    double* src_array = (double *) malloc(VLEN * sizeof(double) * 2);
    double* rcv_array = src_array + VLEN;

    for (int j = 0; j < VLEN; ++j)
    {
        src_array[j] = (double) (proc_id * VLEN + j);
    }

    MPI_Request requests[2];
    MPI_Status statuses[2];

    MPI_Irecv(rcv_array, VLEN, MPI_DOUBLE, LEFT(proc_id, num_proc),
        proc_id, MPI_COMM_WORLD, &requests[0]);

    MPI_Isend(src_array, VLEN, MPI_DOUBLE, RIGHT(proc_id, num_proc),
        RIGHT(proc_id, num_proc), MPI_COMM_WORLD, &requests[1]);

    MPI_Waitall(2, requests, statuses);

    for (int j = 0; j < VLEN; ++j)
    {
        printf ("rank␣%d␣rcv␣elem␣%d:␣%f␣\n", proc_id, j, rcv_array[j]);
    }

    MPI_Finalize();
    return 0;
}
```

The code is relatively simple and easy to understand. It uses simple asynchronous send and receive routines to allow for an eventual overlap of communication and computation. After both the send and receive are completed, guaranteed by the *MPI_Waitall* function call, the process prints the floats it received from the process to its “left” in a round-robin topology.

In the following code snippet we present the GASPI version (this code is a slightly modified version of the tutorial code provided on the GASPI Forum website [7]):

```
#include <GASPI.h>
#include <stdlib.h>

#define RIGHT(proc_id,num_proc)((proc_id + num_proc + 1) % num_proc)
#define LEFT(proc_id,num_proc)((proc_id + num_proc - 1) % num_proc)

int main()
{
    static const int VLEN = 1 << 2;

    gaspi_proc_init(GASPI_BLOCK);

    gaspi_rank_t proc_id, num_proc;
    gaspi_proc_rank(&proc_id);
    gaspi_proc_num(&num_proc);

    gaspi_segment_id_t const segment_id = 0;
    gaspi_size_t const segment_size = 2 * VLEN * sizeof(double);

    gaspi_segment_create(
    segment_id,             // The segment id to identify the segment.
    segment_size,           // The size of the segment(in bytes).
    GASPI_GROUP_ALL,        // The group where segment will be registered.
    GASPI_BLOCK,            // Timeout in milliseconds.
    GASPI_MEM_UNINITIALIZED // Memory allocation policy.
    );

    gaspi_pointer_t array;
    gaspi_segment_ptr(segment_id, &array);

    double* src_array =(double *)(array);
    double* rcv_array = src_array + VLEN;

    for(int j = 0; j < VLEN; ++j)
    {
        src_array[j] = (double)(proc_id * VLEN + j);
    }

    gaspi_notification_id_t data_available = 0; // notification id
    gaspi_queue_id_t queue_id = 0;
41
42     gaspi_offset_t loc_off = 0;
43     gaspi_offset_t rem_off = VLEN * sizeof(double);
44
45
46     gaspi_wait(queue_id, GASPI_BLOCK);
47
48     gaspi_write_notify(
49     segment_id,                 // The segment id where data is located.
50     loc_off,                    // The offset where the data is located.
51     RIGHT(proc_id, num_proc),   // The rank where to write and notify.
52     segment_id,                 // The remote segment id to write the data to
53     rem_off,                    // The remote offset where to write to.
54     VLEN * sizeof(double),      // The size of the data to write.
55     data_available,             // The notification id to use.
56     1 + proc_id,                // The notification value used.
57     queue_id,                   // The queue where to post the request.
58     GASPI_BLOCK                 // Timeout in milliseconds.
59     );
60
61     gaspi_notification_id_t id;
62
63     gaspi_notify_waitsome(
64     segment_id,      // The segment id.
65     data_available,  // The notification id where to start to wait.
66     1,               // The number of notifications to wait for.
67     &id,             // Output parameter with the id of a received notification.
68     GASPI_BLOCK      // Timeout in milliseconds.
69     );
70
71     gaspi_notification_t value;
72     gaspi_notify_reset(segment_id, id, &value);
73
74     for(int j = 0; j < VLEN; ++j)
75     {
76         printf("rank␣%d␣rcv␣elem␣%d:␣%f␣\n", proc_id, j, rcv_array[j]);
77     }
78
79     gaspi_wait(queue_id, GASPI_BLOCK);
80
81     gaspi_proc_term(GASPI_BLOCK);
82
83     return EXIT_SUCCESS;
84 }
```

The GASPI version is, to a certain degree, more complex. Additional concepts are necessary to achieve one-sided asynchronous communication. Here we see the notification concept in action, they are used to alert the receiving side that a data transfer was completed. The *gaspi_write_notify* (line 48) routine writes to a remote segment and then notifies the remote process on the completion of the data transfer. Then, by calling *gaspi_notify_waitsome* (line 63), the process waits for a notification to arrive at one of its queues, signifying that a remote data transfer from another process is complete. Then, by calling *gaspi_notify_reset* (line 72), the notification value is retrieved, and the notification is reset. These notification values can be used by the programmer to carry additional information regarding data transfers. Only then the process waits for its outgoing data transfer to complete, by calling *gaspi_wait* (line 79). The *GASPI_BLOCK* macro is used as the time-out argument for the GASPI routines. This macro extends to an extremely large number, blocking the process until the routine is completed.

2.3 Distributed Shared Memory

In Distributed shared memory (DSM) systems the address space is, just like in distributed memory architectures, composed of the local private memory of several inter-connected processors. But their address space is, either partially or entirely, shared with other workers in the network, just like on a shared memory architecture.

Just like operating systems, distributed shared memory systems abstract physical memory accesses by mapping the address space of the available memory on to a contiguous virtual memory block. This virtual address space is then divided into virtual memory pages that can then be individually requested or modified by the participating nodes, with no knowledge of their real address or physical location within the system. This facilitates programming by completely abstracting communication, just like in shared memory architectures, but can lead to costly unwanted data transfers that are not immediately explicit to the programmer.

Lacking affordable dedicated hardware for a distributed shared address space on a large scale [6], DSM systems are generally implemented as a software abstraction over a standard message passing interface. These systems offer the portability and, thanks to the abstraction of remote data transfers, the ease of programming of shared memory systems [4].

These DSM systems inherit the scalability of the underlying distributed memory architecture while maintaining the same programming methodology of implicit communication present in shared memory systems, allowing for easy porting of existing shared memory applications while still ensuring acceptable performance.

In theory, DSM architectures would allow for truly heterogeneous parallel systems. The participating machines could even have diverse internal representations for basic types, like integers or floats. The DSM compiler and/or runtime environment could add the necessary conversion routines to data being transferred. However, the conversion overhead seems to outweigh the benefits from the potential increase in the number of participating machines [5].

DSM architectures often employ directories to keep track of the state of virtual memory pages. So, in the event of a read, the processor would have to ask the directory where the most recent copy of the data is located (known as an indirection operation). In contrast, in the event of a write, the node that executes the write operation must, if needed, update the directory (known as an invalidation operation). This centralized approach introduces a potential bottleneck in both data access and data manipulation operations [6].

DSM systems have many objectives and limitations similar to the ones present in shared memory architectures. Much of the research put into multiprocessor caches, NUMA architectures and replicated databases can be adapted to fit DSM systems [5].

An example of such systems is Argo [6]. Argo is a software DSM system that is currently implemented over MPI. With the aim of providing an alternative to traditional long-latency centralized directory coherence control and synchronization.

Typically, in previous DSM systems, a directory would be used to enforce coherence. Argo aims to eliminate this centralized bottleneck and achieve a truly distributed solution. This is perhaps the most significant optimization proposed for such DSM systems.

These improvements, along with RDMA communication, eliminate the need for both a centralized directory and for code execution on the receiving side of data transfers. Improving both execution performance and communication latency, compared to existing DSM systems.

2.4 Performance Evaluation of Parallel Code

The most straightforward metric for evaluating the performance of parallel code is speedup. Speedup measures the performance of the parallel version (with p tasks) relative to its serial counterpart [22]. Speedup can be experimentally obtained by the ratio between the execution time of the serial and parallel versions.

$$Speedup(p) = \frac{Sequential\ Execution\ Time}{Parallel\ Execution\ Time} \tag{2.1}$$

Although speedup is a very direct way to tell how much faster (or slower) the parallel version of an application is, it does not immediately reveal how efficiently the extra computing power is being used. To this end, we can compute the efficiency of the parallel implementation (running on p tasks):

$$Efficiency(p) = \frac{Sequential\ Execution\ Time}{p \times Parallel\ Execution\ Time} = \frac{Speedup(p)}{p} \tag{2.2}$$

Chapter 3

Particle Simulation of Plasmas

Computer simulations have often been employed as a substitute for real-life scientific experiments. Either because these are too difficult/expensive, or outright impossible to be performed with current technology. Particle simulation of plasmas is a good example of these difficult experiments. With an estimated 99% of matter in the universe in a plasma-like state and several significant research subjects including controlled thermonuclear fusion and laser wakefield acceleration [8, 9], plasma simulation is a valuable and challenging research subject [23].

The roots of particle simulation go back as early as the late 1950s. Early simulations modeled about 10^2 to 10^3 particles, and their interactions, on the rudimentary computers of the time. Thanks to hardware advancements and algorithm improvements, modern massively parallel computers allow for particle counts in the order of 10^{10} [23, 24].

The actual number of particles involved in real plasma interactions greatly exceeds, by several orders of magnitude, the limit number of particles a modern supercomputer can practically simulate. To overcome this, computer models often employ a crucial optimization known as *superparticles*. *Superparticles* are groups of real particles modeled as a single *superparticle*. Assuming simulation parameters are correctly set and the charge/mass ratio of these *superparticles* is in line with real particles, the simulation result is very accurate [24].

3.1 Particle-in-cell Method

In early physics models, space charge forces of individual particles were computed by Coulomb's law, a $O(n^2)$ complexity operation for n particles [23]. Recognizing the need for a more scalable model, the first Particle-in-Cell algorithms where introduced. Particle-in-Cell algorithms are a much more scalable solution, closer to $O(n)$ complexity operation for n particles [8].

The Particle-in-Cell (PIC) concept was formalized during the 1970s [23] and is well suited to study complex systems with a great degree of freedom and accuracy [11]. This technique simulates the motion of each particle individually in continuous space, but current and charge densities are computed by weighting the discrete particles onto a stationary computational grid. The state of a PIC simulation can be roughly defined by the state of three key components: particles, electric current, and electromagnetic field (composed of electric and magnetic fields) [8]. In each iteration, these elements interact with each other to advance the simulation state. The way these elements interact can be summarized by Figure 3.1.

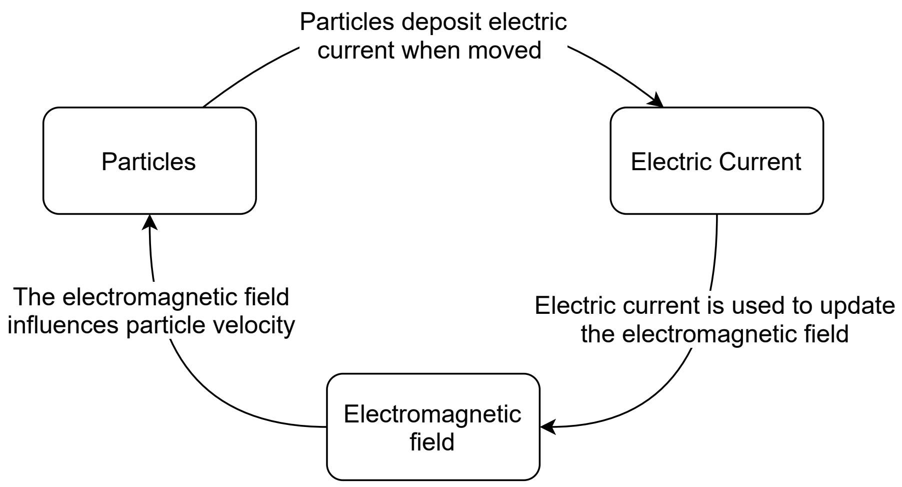

Particles deposit electric current when moved
Particles
Electric Current
The electromagnetic field influences particle velocity
Electric current is used to update the electromagnetic field
Electromagnetic field

Figure 3.1: Sequence of computations in the PIC method.

Each iteration, every time a particle is moved it deposits electric current on the simulation space. This electric current is then used to update the electromagnetic field, that are then used to modify each particle's velocity.

3.1.1 OSIRIS

OSIRIS [11] is a three-dimensional, relativistic, massively parallel, object-oriented PIC implementation written in Fortran 90. Although Fortran 90 is not an object-oriented programming language, those concepts can be implemented.

Regarding parallelization, the implementation allows for an arbitrary decomposition on all spatial coordinates of the simulation space with effective load balancing. When defining the problem input, it is only necessary to define the global physical problem to be simulated and its domain decomposition. The user does not need to specify the parallelization details. The parallelization is implemented in MPI and is done by dividing each spatial direction into a fixed number of blocks that are then assigned to nodes.

Communication between nodes follows the usual procedure for PIC algorithms. Grids are updated by exchanging quantities like electric and magnetic fields. During particle movement, if a particle crosses the boundary between node's cells, it is relocated to the correct node.

To facilitate visualization of simulation datasets, a user-friendly visualization and analysis infrastructure based on Interactive Data Language[1] (IDL) was developed. This infrastructure includes visualization routines for one, two and three-dimensional scalar data, and two and three-dimensional vector data.

3.1.2 ZPIC

ZPIC [8] is a fully relativistic PIC implementation focused on educational plasma physics simulations based on OSIRIS. Unlike OSIRIS, ZPIC is not implemented to benefit from parallel execution. This was a deliberate design decision with the intention of reducing code complexity. Also, ZPIC is meant to be run on common laptops and personal computers while OSIRIS was designed with supercomputers in mind.

ZPIC was written using the C programming language. Implementing in such a low-level programming language allows for better memory management and code optimization.

In order to provide appropriate data analysis and visualization tools to examine simulation results and diagnostics, the developers opted for Jupyter Notebooks[2] paired with source code interfaces. Both the notebooks and interfaces were written in Python. This high-level language offers many libraries with mathematical modules and plotting routines that aid in data manipulation and visualization. Also, the high abstraction level of the language simplifies the adjustment of the notebooks and corresponding functionalities to new projects. The role of the source code interfaces is to reduce the amount of code show to the user on the Jupyter Notebooks. This improves the clarity of the notebooks, especially to the non-expert target audience of the implementation.

Like other Particle-in-cell implementations, the simulation space in ZPIC is divided into cells. Each cell has its own electric current and electromagnetic field values while particles roam freely through the simulation space. Each of the three cell quantities (electric current, electric and magnetic fields) are internally represented as an one-dimensional array, but are accessed in a way that mimics a 2-dimensional matrix. It is important to note that, even on a 2-dimensional simulation space, ZPIC's particle velocity vectors are 3-dimensional. This is also true for electric current and the electromagnetic field; their values are also tracked on three dimensions.

Simulations can be configured to have a moving window. This means that every few iterations, the simulation window will be moved one cell to the right. An example of a moving window simulation is laser wakefield acceleration [9, 10], where the window tracks the front end of a laser pulse moving through an electron plasma, energizing the electrons. In practice, to achieve the moving window effect, all ZPIC does is shift all particles and cell data one cell to the left every time the window is moved. Particles that leave the simulation space (through the left or right borders) are deleted and, to resemble an endless plasma, new particles on are instantiated on the right side of the simulation space.

[1] https://www.harrisgeospatial.com/Software-Technology/IDL
[2] https://jupyter.org/index.html

Particle species can be configured to use custom density profiles. One of the available options only instantiates starting particles on a specified region of the simulation space. This is used in laser wakefield acceleration simulations to only create starting particles on the right side of the simulation space, if at all. Additional particles are then introduced to the simulation on moving window iterations. The goal is to ensure that the simulation is able to capture the instant the laser pulse collides with the particle plasma.

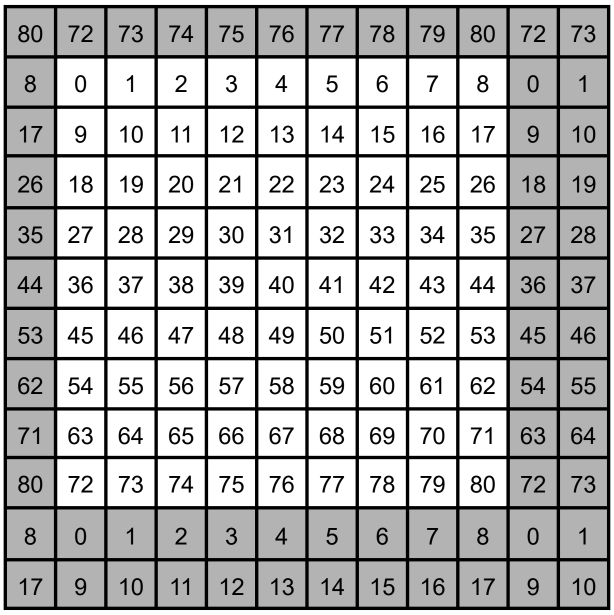
Moreover, ZPIC's included laser wakefield acceleration simulations employ an additional processing step, known as current smoothing. In practice, current smoothing is just another computation step taken each iteration to modify the electrical current value of each cell. Current smoothing can be applied on the x or/and y-axis, meaning that to update the electrical current value of each cell ZPIC will make use of the values of cells on the same row or column respectively. Current smoothing can be performed over multiple steps per simulation iteration. After each smoothing step, current ghost cells need to be updated with the new values.

A ZPIC simulation, like on many other PIC implementations, implements periodic boundaries. This means that the two-dimensional simulation space warps in a way that the cells on an edge of the simulation space border the cells on the opposite edge. Static window simulations use periodic boundaries on both the x and the y-axis, meaning the simulation space can be thought of as the surface of a torus. In contrast, moving window simulations only enforce periodic boundaries on the y-axis, so their simulation space is reminiscent of the surface of a horizontal cylinder. This means that, for example, if a particle leaves the simulation space of a static window simulation through the left border it will be reallocated to the rightmost cell on that row, while if the same was to happen on a moving window simulation the particle would be deleted because moving window simulations do not employ periodic boundaries on that axis.

This is also true for electric current and the electromagnetic field. For instance, if a particle that is on the bottom border is moved and deposits electrical current in a way that the current might leave the simulation space, it is deposited on a ghost cell that represents the correct cell on the top border.

Ghost cells are used on the edges of the simulation space to mimic real cells. They simplify interactions with cells by streamlining them. Going back to the previous example of the moving particle, if there were no ghost cells, every time a particle was moved the simulation would have to check if the cell where it is trying to deposit current exists. If not, the simulation would then have to identify the correct cell to deposit the electric current. This extra step would add additional branching and complexity to the particle mover, an already complex and expensive operation.

Figure 3.2 shows an example of a simple ZPIC simulation space (real simulations have many more cells). The gray cells on the figure are ghost cells. Note that they are marked with the number of the cell they represent. Ghost cells are, on each iteration, synchronized with the cell they represent, so that both have the same electric current and electromagnetic field values. Also, the figure shows an additional column/row of ghost cells on the right/bottom borders, they are required by the underlying logic of the ZPIC simulation.



80	72	73	74	75	76	77	78	79	80	72	73
8	0	1	2	3	4	5	6	7	8	0	1
17	9	10	11	12	13	14	15	16	17	9	10
26	18	19	20	21	22	23	24	25	26	18	19
35	27	28	29	30	31	32	33	34	35	27	28
44	36	37	38	39	40	41	42	43	44	36	37
53	45	46	47	48	49	50	51	52	53	45	46
62	54	55	56	57	58	59	60	61	62	54	55
71	63	64	65	66	67	68	69	70	71	63	64
80	72	73	74	75	76	77	78	79	80	72	73
8	0	1	2	3	4	5	6	7	8	0	1
17	9	10	11	12	13	14	15	16	17	9	10

Figure 3.2: ZPIC simulation space example.

3.2 Plasma Experiments

The Particle-in-cell method is very versatile and allows for many different scientific experiments to be simulated. Following is the description of two simulation categories which come bundled with the ZPIC code that we used in our experimental evaluation.

3.2.1 Weibel instability

Weibel instability [25, 26] simulations model the instability observed on homogeneous plasmas when energized with a certain amount of thermal energy. ZPIC models this simulation by using two different species, electrons and positrons, with opposite charge values. These two plasmas are energized with opposing momentums, forcing them to clash and generate instability patterns in the electromagnetic field. Figure 3.3 shows the magnetic field structure generated by a ZPIC Weibel instability simulation.

3.2.2 Laser wakefield acceleration

Laser wakefield acceleration [9, 10] uses powerful lasers to accelerate particles to near light-speed velocities over short distances. ZPIC portrays these experiments by modeling a laser pulse as an electromagnetic wave colliding with a low energy electron plasma. In order to capture the instant the laser pulse collides with the electron plasma, the simulation starts with no particles. They are later injected into the simulation space and are periodically shifted to the left, giving the illusion that the laser pulse is moving through space. Figure 3.4 shows the magnetic field structure generated by a ZPIC laser wakefield acceleration simulation.

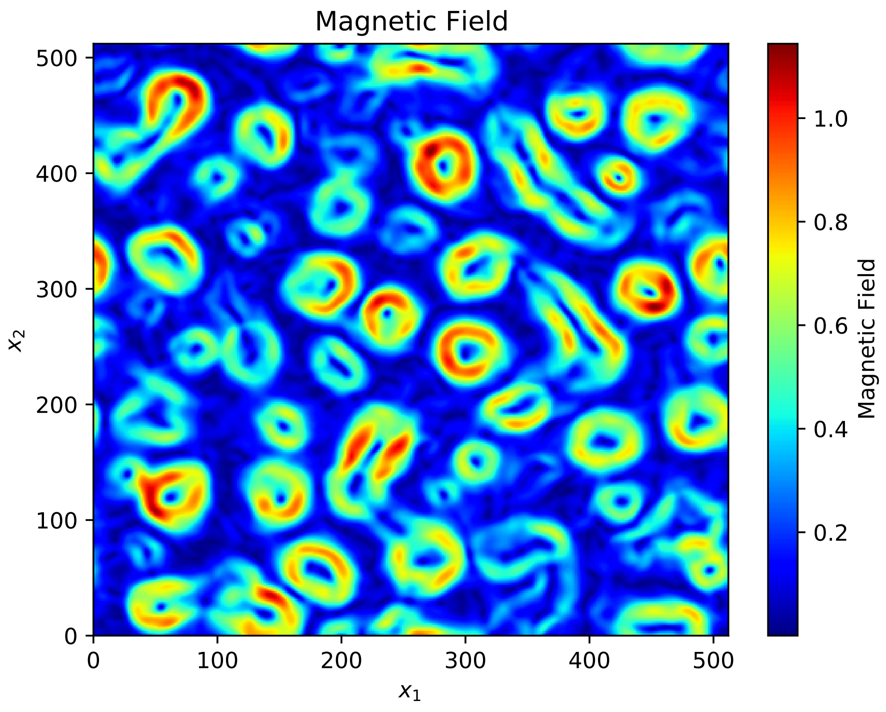


Figure 3.3: ZPIC Weibel instability magnetic field structure.

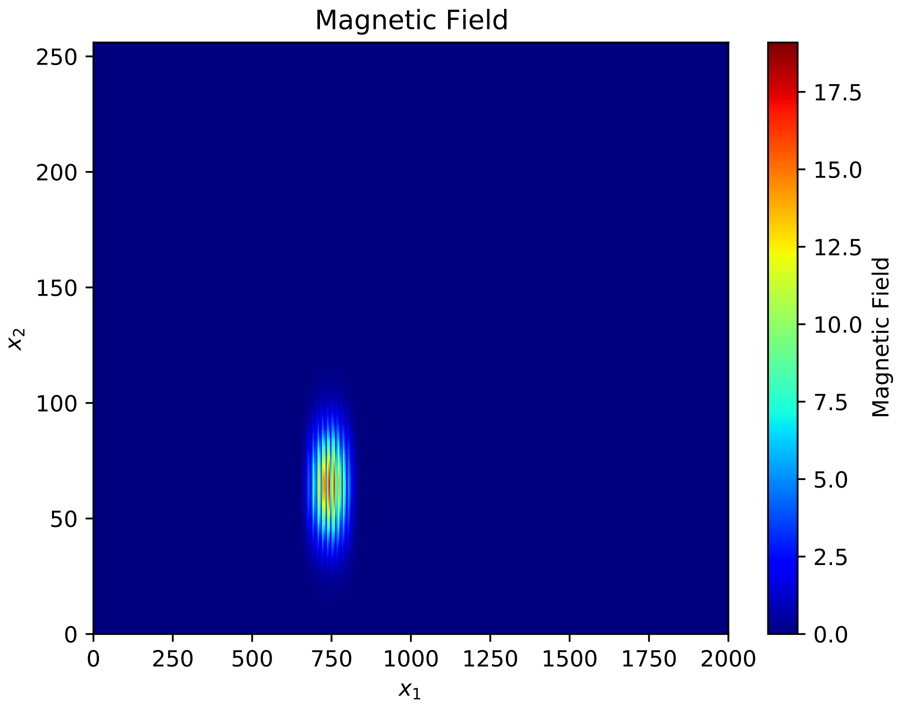


Figure 3.4: ZPIC laser wakefield acceleration magnetic field structure.

Chapter 4

Implementation

In this chapter, we will give an in depth description of our implementation and the decisions made during its development. First, a broad overview of the work needed to distribute the simulation. Then, a detailed description of our partitioning scheme, followed by our solution for the distributed particle generation, and a thorough description of the implemented communication procedures. Finally, we address our GASPI/OpenMP hybrid implementation.

4.1 Implementation overview

Now that we know the components that define the state of a ZPIC simulation and how they interact with each other we need to distribute the simulation between the available processes. The most common, and natural, decomposition for parallel Particle-in-Cell implementations is a spacial decomposition, where the cells that make up the simulation space are distributed and assigned to the available processes. Of the possible ways to divide a two-dimensional space, the usual approach to these kinds of simulations is to adopt a checkerboard decomposition, where the simulation space is partitioned into smaller rectangular sections. To achieve this spatial decomposition we need a procedure to adequately divide the simulation space and assign the resulting regions to the participating processes. While MPI offers routines to help the programmer with this effort, GASPI does not implement such features.

To guarantee that the distributed simulation has a similar end result as the original ZPIC simulation, it is crucial that both share the same initial state. Since ZPIC implements a custom pseudo-random number generator to instantiate particle values, a distributed solution with the same behaviour will need to be implemented so that each process starts with the correct state for the simulation zone assigned to it.

In addition to the partitioning problem, the distributed simulation will introduce inevitable data dependencies between processes, leading to costly synchronization between processes, which is incompatible with an efficient parallel implementation and should be minimized. These data dependencies can be deduced from Figure 3.1. At the start of each iteration, the simulation will advance the particles present in the simulation by moving them and, using electromagnetic field values, update their velocity. Moving particles deposit electric current to the cells they move through, this newly generated current needs to be transmitted to the neighboring processes. Moving particles can also leave the simulation space of a process, if this is the case they need to be reallocated to the correct process, requiring additional data transmissions. Lastly, the electric current generated by the moving particles will be used to update the electromagnetic field values present on each cell. These values, just like the previously computed electric current values, need to be broadcasted to neighboring processes.

4.2 Partitioning

Following the traditional decomposition method for Particle-in-Cell implementations, the simulation space was partitioned and organized into a checkerboard arrangement. In order to dynamically partition cells based on the number of processes, we can use the following block decomposition formulas (where id represents the block id, p is the number of desired partitions (blocks), and n is the number of elements we want to divide):

$$BlockLow(id,p,n) = \left\lfloor \frac{id \times n}{p} \right\rfloor \tag{4.1a}$$

$$BlockHigh(id,p,n) = BlockLow(id+1,p,n) - 1 \tag{4.1b}$$

$$BlockSize(id,p,n) = BlockHigh(id,p,n) - BlockLow(id,p,n) + 1 \tag{4.1c}$$

Figure 4.1 shows an example of an array divided using these formulas ($p = 3$, $n = 10$):

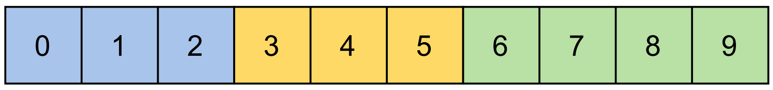


Figure 4.1: One dimensional partitioning.

However, these formulas are not immediately applicable to two-dimensional partitioning. Regardless, we can implement partitioning for n dimensions by having n different partition counts, one for each axis. MPI offers this functionality through the *MPI_Dims_create* function, GASPI however does not implement such operation. To this end, we need an algorithm to compute the number of blocks on each axis, based on the number of available processes. This was implemented by applying a prime factorization algorithm to the number of processes and then distributing the resulting factors through the desired number of dimensions.

The factorization algorithm is used to decompose the number of processes into smaller prime integers, called factors. Our implementation is as follows: First of all, we count the number of occurrences of the factor 2 in the number of processes, dividing the number of processes by 2 on every occurrence. Then, we count the number of occurrences of each odd factor, up to the square root of the initial number of processes. Again, dividing the remaining number of processes by every newly discovered factor. Finally, if the last factor is different from 1, we save that remaining factor.

Next, we need to assign the obtained factors to each axis. Starting with one block on each axis, we assign factors from highest to lowest to the axis with the lowest number of blocks. In the event that both dimensions have the same number of blocks, the algorithm will assign the next factor to the y-axis. After all factors are assigned, an additional check is made to make sure that the y-axis has more divisions than the x-axis, if not the number of divisions on both axes are swapped. This is done because in moving window simulations with custom particle density profiles the number of particles between columns is not balanced, some may have no particles at all for many iterations. By employing a checkerboard decomposition while prioritizing divisions on the y-axis, only on situations where this bias would not result in an unbalance in the number of blocks, we can slightly minimize the impact of this lack of balance in the initial particle arrangement. This bias does not negatively affect static window simulations because the number of divisions on each dimension is still as balanced as possible.

Let us consider a simple example of a small simulation space with 100 cells (on a 10x10 configuration) and walk-through how the partitioning algorithm would divide the cells between 8 processes:

To obtain the number of blocks per axis we first need to apply the factorization algorithm the number of processes. In this case, it will result in:

$$Factor(8) = [2, 2, 2]$$

We then distribute these factors among the x and y-axis, picking the dimension with the lowest number of blocks and multiplying it by the highest factor (the algorithm prioritizes divisions on the y-axis):

$$AssignFactors([2, 2, 2]) = [2, 4]$$

This means that, with 8 processes, the simulation space will have 2 partitions on the x-axis and 4 partitions on the y-axis. Using these values with the block decomposition Formulas 4.1 we get the arrangement shown in Figure 4.2.

Now that we have the partition structure, we need to assign a process to each block. To that end, the implementation makes use of process coordinates, represented by the pair ($column$, row). Process coordinates are computed as follows ($NumColumns$ represents the number of blocks on the x-axis):

$$ProcessColumn(Rank) = Rank \ \% \ NumColumns$$

$$ProcessRow(Rank) = \left\lfloor \frac{Rank}{NumColumns} \right\rfloor$$

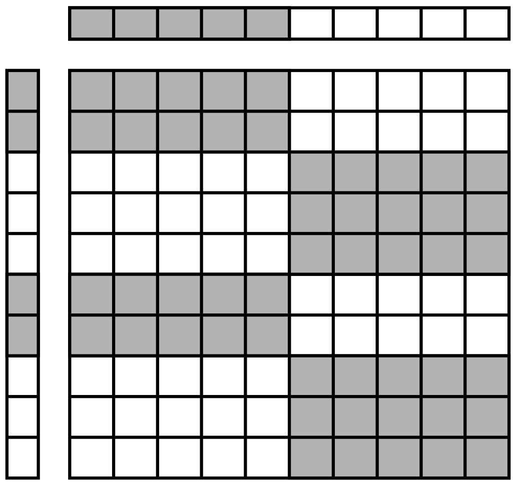


Figure 4.2: Two dimensional partitioning.

These are used by each process to compute the size and location of their assigned block. Processes also use their coordinates to identify neighboring processes, with whom they will send and receive data on the communication stages of each iteration. The following formula can be used to obtain the rank of a process through its coordinates:

$$ProcessRank(Column,\ Row) = Column\ +\ (Row\ \times\ NumColumns)$$

4.3 Particle Generation

Particles on ZPIC are grouped into species, they are used to represent diverse types of particles like, for example, electrons or positrons. Each can be configured with their own electrical charge value, initial velocity, density profile, and many other attributes. All cells start with the same number of particles of each species, the position of the species' particles inside each cell is also equal for every cell. It is defined by the density profile specified on the input file. Their initial velocity, however, can be specified on the input file or can even be random.

ZPIC implements a custom pseudo-random number generator to assign each particle a random speed value in each axis. As with any pseudo-random number generator, it still generates numbers in a deterministic fashion. This means that if starting state of the number generator is known, the sequence of generated numbers can be reproduced. Taking this into account, there are several ways to guarantee that the parallel version generates particles with the same velocity as the serial version.

We could implement a truly distributed approach, where each process would receive the state of the number generator, instantiate particles, and then send the resulting state of the number generator to the next process. However, ZPIC instantiates particles row-wise, from left to right. Implementing this approach would mean synchronizing between multiple processes, just to instantiate a single row worth of particles.

Alternatively, we could opt for a more centralized approach, where one process would generate all the particles required by the simulation, and then send them to the correct processes. This alternative requires less synchronization between processes than the previous option, but it does introduce a problem. Moving window simulations instantiate additional particles every time the window is moved. This would mean that, every time the window is moved, the processes on the right edge of the simulation space would need to wait for the arrival of the newly instantiated particles. Consequently, synchronization and communication between processes would increase, something we want to avoid.

In the interest of minimizing communication, we could implement a method reliant on redundant computation. The implementation is quite straightforward, before saving the velocity of a particle, generate the same amount of random numbers the serial version would need to generate before reaching that particle. That way, when we start saving the generated numbers, the pseudo-random generator is on the correct state for that particle. Since the serial implementation starts on the top row and generates particles from left to right, we can follow the same pattern, but only save the particles we need.

Using this method we are able to generate all the particles of the simulation, distributed through several processes, with no communication between processes. However, it does introduce a considerable amount of redundant computation. Some particles, especially the ones meant for the top rows of cells, will be generated by almost all processes participating in the simulation. Luckily, the pseudo-random number generator implemented on ZPIC is considerably fast, so this redundancy in computation does not become a problem. This is true even on the biggest simulations included with ZPIC, where the particle instantiation is still dwarfed by the rest of the simulation.

In the end, we opted for this last approach, as it required no additional communication, and the resulting redundancy in computation was not significant when compared to the rest of the simulation.

4.4 Communication

4.4.1 Particle Communication

A ZPIC iteration starts with particle processing. Each particle is moved and, if a particle leaves the simulation space, it is reallocated to the correct cell, respecting the imposed periodic boundaries. On a distributed implementation of the sequential particle mover, the only thing that would change is the procedure to take when a particle leaves the simulation space of the task that currently holds it. In a distributed implementation the particle would need to be reallocated to the correct process, implying communication between processes.

As explained before in Section 2.2.3, when sending data with GASPI, it is required that the data is sent from/to predefined memory segments. We could have each task save its particles on a GASPI segment at all times. Implying that each process would only have a single segment to hold all its particles for the duration of the simulation, with particles then being removed from and copied to this segment as the simulation progressed. This approach would allow us to transmit particles directly between the species data structures, achieving a no-copy data transmission. However, this solution introduces multiple problems.

If we send a particle immediately after it was determined that it should be moved, we will end up with massive amounts of tiny messages every time a species was processed, likely flooding the network. Furthermore, while developing the distributed implementation we were informed by the GPI-2 developers that, in the current implementation, it is more efficient to group non-contiguous data into larger messages[1].

Also, using this approach, the sending process would not know where in the remote segment it should write the data to, possibly leading to data overwriting. To avoid data overwriting, the receiving process would need to coordinate with the sending processes so as to establish where each one should write their data to, increasing synchronization across processes.

To limit the number of messages, and completely negate the data overwriting issue, each process could have a dedicated segment for particle communication with each of its eight neighbors. This segment could be used to both send and receive data, by having dedicated send and receive zones, and could be shared by all species. Figure 4.3 shows an example of this segment organization in action. Since GASPI segments can not be resized, they need to be instantiated with enough space to hold all the particles transiting between two neighboring processes on each iteration, but not too large as to not waste memory.

The segment size will need to be determined dynamically. It must account for multiple species, with different particle densities, and should scale with the number of cells at that process border. We came up with the following formula to compute the number of particles each segment should be able to send/receive on a single iteration (where b represents the border to one of the neighboring processes):

$$Particle\ Segment\ Size(b) = \sum_{spec=0}^{n\ spec} Particles\ per\ Cell(spec) \times Num\ Cells\ Border(b) \times K \tag{4.2}$$

This means that each particle segment will have enough room so that, on a single iteration, each species can send/receive its initial number of particles per cell, times a constant K, per cell on the border. The constant K can be obtained through trial and error, if segment size computed by Formula 4.2 is exceeded during a simulation, an error message is displayed informing the user and the simulation is stopped. A value of $K = 4$ seems to be adequate for most simulations. Each segment is then created with double the value returned from this equation, as they will have their own send and receive zones, to avoid overwriting. Since neighboring processes will have the same number of border cells between them, their particle segments will have the exact same size, simplifying memory offset calculation needed for a *gaspi_write* call.

[1]This information was passed to us during a GASPI workshop. We have no formal references for this statement.

Now that we have the particle segments instantiated and their sizes are known, we can start using them to transmit particles. Every time a particle is moved and leaves the simulation space of a process we determine which process it will be sent to, based on its cell coordinates. Particles hold the coordinates of the cell they are in. In the serial version of ZPIC, the cell coordinates of a particle can be represented by the pair $(column,\ row)$, where the top-left (non-ghost) cell has coordinates $(0,\ 0)$, similar to the process coordinates introduced before. In the distributed implementation, the particle coordinates are relative to the simulation space of the process they are currently in. Identifying which particles need to be reallocated, and to which neighbor, can be done through their coordinates. If either the column or row coordinate of a particle is less than zero or exceeds the number of columns/rows of the simulation space of the process it is currently in, it means that the particle is no longer in any of the cells assigned to that process and needs to be transferred. Additionally, when a particle is reallocated we always have to correct its cell coordinates to reflect the change of the origin of their coordinate space.

Particle coordinates can either be corrected by the sender or by the receiver. In our initial implementation, this burden fell to the sender. We experimented with switching this responsibility to the receiver. However, in our experimental evaluation[2], both alternatives yielded comparable execution times. Our final implementation leaves the particle coordinate correction burden to the receiving process.

The simulation logic guarantees that, in each iteration of a static window simulation, particles will move at most one cell on each axis. This makes it easy to compute the new coordinates of a newly received particle. For example, if a particle leaves the simulation space of a process through the left border, we can be sure that it will be reallocated to one of the cells on the rightmost column of the process to its left, meaning its column coordinate will be equal to the number of columns of the receiving process minus one. On the other hand, if the particle leaves the cells assigned to a process through the right edge of the simulation space, the receiving process can be sure that the received particle should be reallocated to one of the cells on the leftmost column, meaning its cell coordinate will be zero. We can extend this logic to particles arriving through other borders. However, in moving window simulations, it is possible that a particle, through its own motion, moves one cell to the left on a moving window iteration, causing it to then move an additional cell to the left, resulting in the particle moving a total of two cells on the x-axis. Now, there are two possible columns that the received particle might need to be reallocated to, not just the rightmost one like on static window simulations. To account for this possibility, instead of just overwriting the column coordinate of particles received from processes to its right, the receiving process will add the number of columns of its simulation space to the column coordinate of the received particle.

After a process has all the particles of a species ready to be send, a single *gaspi_write_notify* call is made to each neighbor, for that species. After the remote process receives the notification indicating that some particles have arrived at target particle segment, the receiving process needs to copy the newly received particles to the species particle array. To do that, the receiving process needs to know how many particles have arrived. On our initial implementation, the number of

[2] Experiments where performed on MareNostrum 4 on multiple Weibel instability [25, 26] and laser wakefield acceleration [9, 10] simulations.

transmitted particles was indicated by the notification value, a number that is included with every GASPI notification that can be used by the programmer to convey additional information about the notification. However, when evaluating the implementation on the MareNostrum 4 supercomputer we found that when installing GPI-2 there, notification values can only hold 1 byte of information, while on the version we used to develop and test our distributed implementation, notification values can be used to transmit 4 bytes. Meaning that, with this implementation of the particle communication, when running on MareNostrum 4 we can have at most $2^8 - 1 = 255$ particles per species sent/received between two neighbors each iteration, a very low number.

To overcome this limitation, we added an additional particle to the beginning of each particle transmission, this particle will be known as the *fake particle*. This particle is used to relay the number of particles sent. In Figure 4.3 we demonstrate how the particle segments would be used to transmit two different species between neighbors.

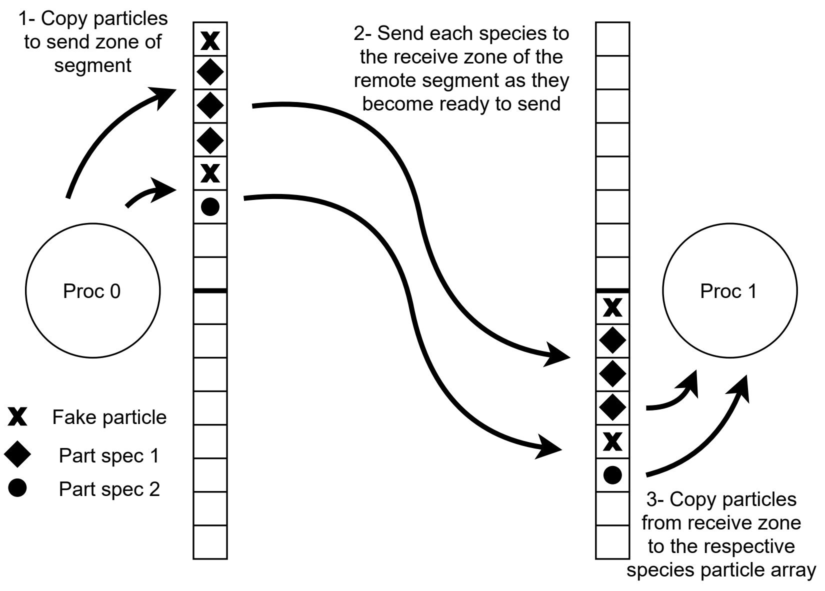

1- Copy particles to send zone of segment
2- Send each species to the receive zone of the remote segment as they become ready to send
Proc 0
Proc 1
Fake particle
Part spec 1
Part spec 2
3- Copy particles from receive zone to the respective species particle array

Figure 4.3: Particle communication example.

However, adding an extra particle to each transmission will result in some overhead. When compiling the code on the Intel Xeon Platinum 8160 CPU, present on the MareNostrum 4 supercomputer where we performed our experimental evaluation, the data structure that represents each particle holds 28 bytes of information. On the *fake particle* only 4 of the 28 bytes are useful, those 4 bytes being the integer representing the number of particles sent. This results in 24 bytes of overhead on each particle transmission. Depending on the simulation parameters and the number of tasks assigned to the execution, the number of particles per transmission is often in the hundreds[3]. In the end, the resulting overhead is negligible when compared to the rest of the

[3]As observed on a Weibel instability simulation with 128x128 cells, 2 species each with 256 particles per cell on 8 processes.

data transmission.

After a process receives a notification from each neighbor, signaling the arrival of all the particles of a species, the receiving process will compute the new total number of particles of that species. Then, if there is not enough room for the newly received particles, extend the species array. Finally, the process can copy the particles from each segment to the species particle array.

4.4.2 Electric Current Communication

After advancing each species in the simulation, ZPIC synchronizes the electric current values of each ghost cell with its respective real cell. To do that, ZPIC sets both cells with the sum of the electric current value of both cells. Keep in mind that each cell has three electric current values, one for each dimension. However, during this update procedure, those three values are all treated in the same way. For that reason, each cell will be regarded as only holding a single value.

To update the cells of each process on a distributed simulation we have to follow the same pattern, set the value of each real cell, and all ghost cells that represent that real cell, to the sum of all values. In the end, all cells that represent the same cell must have the same electric charge value. In Figure 4.4, we illustrate an example of what we need to achieve dynamically for each neighboring process pair. Note that cells are marked with the number of the cell they represent and that all the cells present on both processes are exchanged and updated.

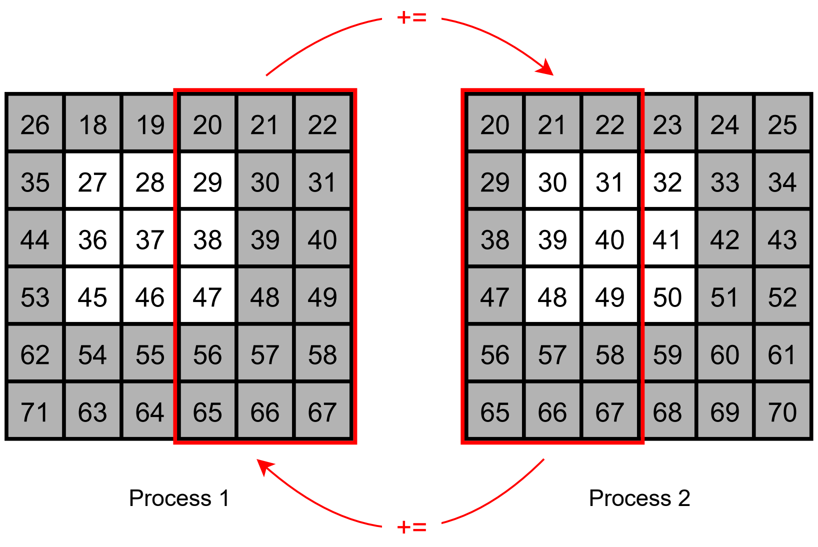


Figure 4.4: Electric current communication between two processes.

To accomplish this, at the start of the simulation, each process computes the cell coordinates ($column$, row) of the cell at the top-left corner of the zone that it will send to each neighbor, as well as the number of rows and columns. Keep in mind that cell coordinates use the top-left real cell of the local simulation space as origin. Therefore, in Figure 4.4 the origin of the coordinate space of process 1 is the cell 27, while process 2 uses cell 30 as the origin.

In the example shown in Figure 4.4 the process on the left needs to send to its right neighbor the area composed of 3 columns and 6 rows with the top-left cell 20, that has the coordinates $(2,-1)$. While the process on the right will send 3 columns and 6 rows worth of cells starting from the cell 20, with coordinates $(-1,-1)$.

After sending local current data, each process needs to add the remote current values to its own cells. So this data needs to be sent to an intermediate location, where it can be later accessed when the receiving process is ready to update its cells. Also, since we are trying to minimize the number of small messages, we would like to group data into a larger single message, so the sending process will also need a dedicated memory area to group all the data it needs to send to a neighbor before sending it. Therefore, we followed an approach similar to our particle communication implementation, where each process has a GASPI memory segment dedicated to each neighbor, where it can organize the data it needs to send, and also have a dedicated zone where that same neighbor can write their data to. Bearing in mind that we know the exact number of cells we will send to each neighbor, we can create these segments with the exact size we need and also know the offset of the receiving zone of the segment.

The sending procedure is as follows: First, identify the cells to send by making use of the (previously computed) top-left cell coordinates and row/column counts. Then, copy cell data from the electric current array to the GASPI segment row-wise. And finally, write that data to the remote memory segment with a single data transmission. The procedure on the receiving side is just as simple. Wait for the GASPI notification from the neighbor we are waiting for. Then, since the values we need to send to a neighbor are from the same cells as the ones we will receive from it, we use the top-left cell coordinates and row/column counts for that neighbor again to make out the cell where each received value should be added to. Keeping in mind that the cells were packed on to the data transmission row-wise, the receiving process can iterate the received data, adding each received value to the correct cell by also iterating through its own cells row-wise.

One possible drawback inherited by this approach is the blocking for each neighbor's data transmission. This could mean that a process could block while waiting for the data transmission of a slower neighbor instead of processing the received data of faster neighbors whose data transmissions have already arrived. We experimented[4] with an implementation where a process would, instead of blocking, just test for a notification. If that data transmission has not arrived yet, the process would test the notification of another neighbor. This would allow for a task to process remote writes as their data became available. However, this change did not only show any improvement to the execution times whatsoever, it also introduced a problem.

Up until now, when executing the our implementation with a fixed number of processes, the simulation result was deterministic. All operations involving floating-point numbers while performed by different processes at non-deterministic instants, the resulting data is always joined with other processes in the same order. In other words, the order in which processes wait for and integrate remote data of each of their neighbors into their local simulation state is always the same. Since

[4]Experiments where performed on MareNostrum 4 on multiple Weibel instability [25, 26] and laser wakefield acceleration [9, 10] simulations.

cell data is represented with floating-point numbers and, with this change, the orders of the operations involving these floating-point numbers are no longer guaranteed, so the simulation result was no longer deterministic. Considering that this change did not improve execution times, and also compromised the determinism of the simulation, we chose to revert this change and keep the original blocking approach.

It is important to note that, unlike electromagnetic field values, electric current is not persistent between ZPIC iterations. Every iteration, the electric current values present on each cell are set to zero before the particle processing stage is reached. This step is also performed by every process on our distributed implementation, right before the particle processing stage.

4.4.3 Electric Current Smoothing Communication

After updating the electric current values of its cells each process will apply smoothing to its electric current data, if the simulation is configured to do so. Current smoothing changes the current values of cells based on the values of nearby cells. After each smoothing step, ZPIC updates ghost cells with the newly computed values. The laser wakefield acceleration simulation included with ZPIC performs, in each iteration, five steps of current smoothing.

A distributed implementation must do the same, updating each ghost cell with the value of its respective real cell after each smoothing step. Figure 4.5 illustrates the communication procedure we need to implement. Note that cells are marked with the number of the cell they represent and that ghost cells are overwritten with the value present on their respective real cell.

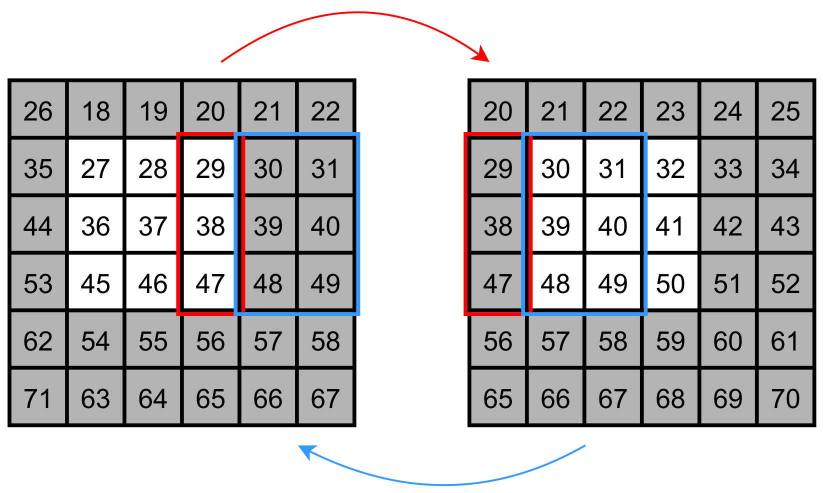


Figure 4.5: Electric current smoothing communication between two processes.

We resorted to the same approach used to update electric current, explained in section 4.4.2, using pre-computed top-left cell coordinates as well as row/column counts. However, this time the cells we are sending are not the same we will be receiving later, so each task also needs to compute where the received data should be copied to. The GASPI segment layout is also identical, each process has a dedicated segment for each neighbor, where it can prepare messages before sending them and also read received data.

When testing this implementation of the electric current ghost cell update we identified a problem. Sometimes processes would either block while waiting for the last current update notification while already having received that last notification, or the simulation result would be very different from the original ZPIC version. After investigating this issue we concluded that, since these messages were being sent back-to-back without any synchronization between the sender and the receiver, the sender would occasionally overwrite messages they had previously sent to a neighbor before it could process them. An example of this error provoking situation is shown in Figure 4.6. This erroneous situation can happen when two processes transmit ghost cell data to each other, but the message from process 1 is delayed. Then, if process 1 is fast enough to process and send the data regarding next smoothing step, it can overwrite the information present on the segment of process 2 before it can be used.

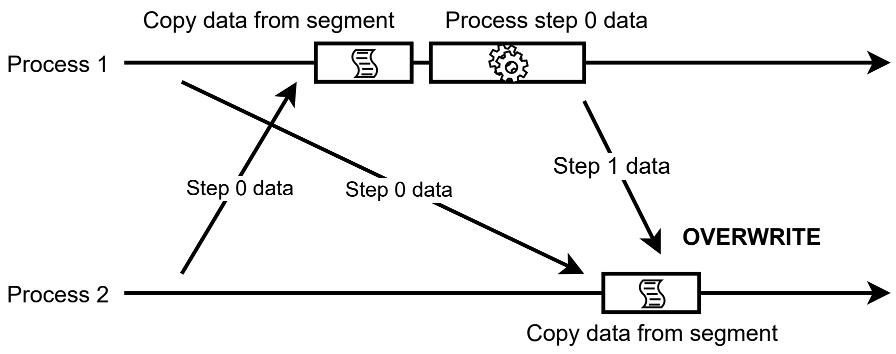


Figure 4.6: Electric current smoothing communication overwrite issue example.

To fix this issue, we could introduce synchronization points between each current smoothing step to make sure that each task would only send data when the receiving process was ready to receive it. However, that would conflict with the asynchronous focus of the implementation. Alternatively, by extending the area which the sending process can use to write data to we can prevent the data overwriting issue. After a careful analysis of the code and communication patterns, we concluded that each process can only receive up to two messages from a neighbor before the neighbor has to wait for a response. So, to fix this issue, we simply doubled the segment size. This way we can divide each segment to have two sending and two receiving zones, choosing which of the two to use based on the parity of the smoothing step number, completely avoiding the data overwrite issue.

4.4.4 Electromagnetic Field Communication

The last processing step of a ZPIC iteration is the update of the simulation's electromagnetic field. The electromagnetic field that take part in a ZPIC simulation are composed of electric and magnetic fields. Since both fields are always updated and used at the same time each iteration, they will be treated as a single entity for the rest of the document, as the procedure to update these fields in our distributed implementation is also the same for both.

To update the electromagnetic field, ZPIC overwrites each ghost cell with the value present on their respective real cell. The distributed implementation must do the same, each neighbor process pair must send the cells that its neighbor will need. Then, each process must wait for the remote cell data to then overwrite their ghost cells with. This procedure is exemplified in Figure 4.7. As before, cells are marked with the number of the cell they represent and ghost cells are overwritten with the value present on their respective real cell.

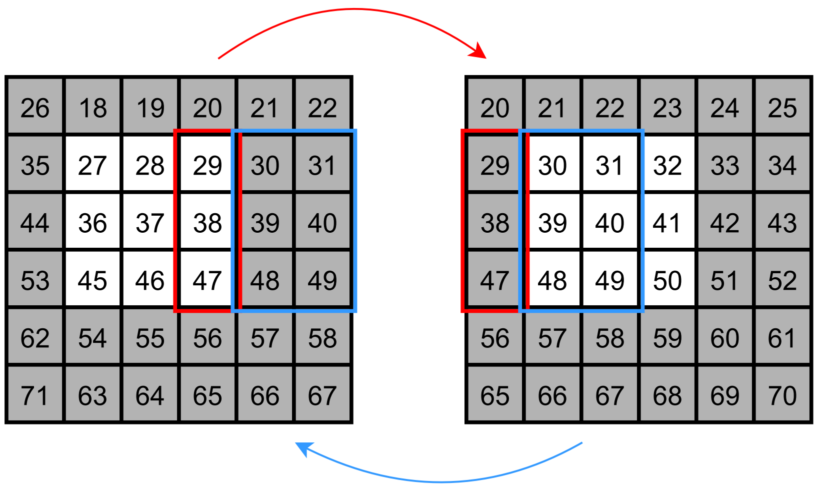


Figure 4.7: Electromagnetic field ghost cell update.

To implement this, we resorted to the same segment structure as explained in Section 4.4.2. Each process has a dedicated segment for each of its eight neighbors. Each segment has a dedicated sending and receiving zone, where it can prepare data before sending it, and also receive and hold remote data until the process is ready to overwrite its ghost cells with the newly received values. In order to identify which cells need to be sent, and which cells must be overwritten with the remote data, we implemented the same approach as the one used in Sections 4.4.2 and 4.4.3.

However, contrary to the electric current values, the electromagnetic field values are persistent between iterations and not zeroed after each iteration. This means that, to give the illusion of a moving window on moving window iterations, ZPIC has to shift all electromagnetic field values one cell to the left. Additionally, on moving window iterations, ZPIC sets all cells in the three rightmost columns, including ghost cells, to zero so they appear as new untouched cells.

On moving window iterations, our distributed implementation also has to shift cell data of all real cells one cell to the left, and if a process is on the right edge of the simulation space, zero its three rightmost columns (including ghost cells). However, the inter-process communication procedure is a bit more complicated. Figure 4.8 show an example of how a process would update its electromagnetic field ghost cell data on moving window iterations. Note that cells are marked with the number of the cell they represent. Also, note that this time, now ghost cells are not updated with the value of their respective real cell. Instead, ghost cells are set with the value of the cell to the right of the cell they represent. Additionally, since each process only has one column of ghost cells on the left edge of their simulation space, processes do not exchange electromagnetic field values with neighbors to their left on moving window iterations.

To achieve the moving window effect, our distributed implementation does as follows: first, send real cell data to each neighbor; then, if the window needs to be moved in this iteration, shift real cell data one cell to the left and if the process is on the right edge of the simulation space, zero the three rightmost columns (including ghost cells); finally, when the remote data is ready, overwrite local cells with the received values.

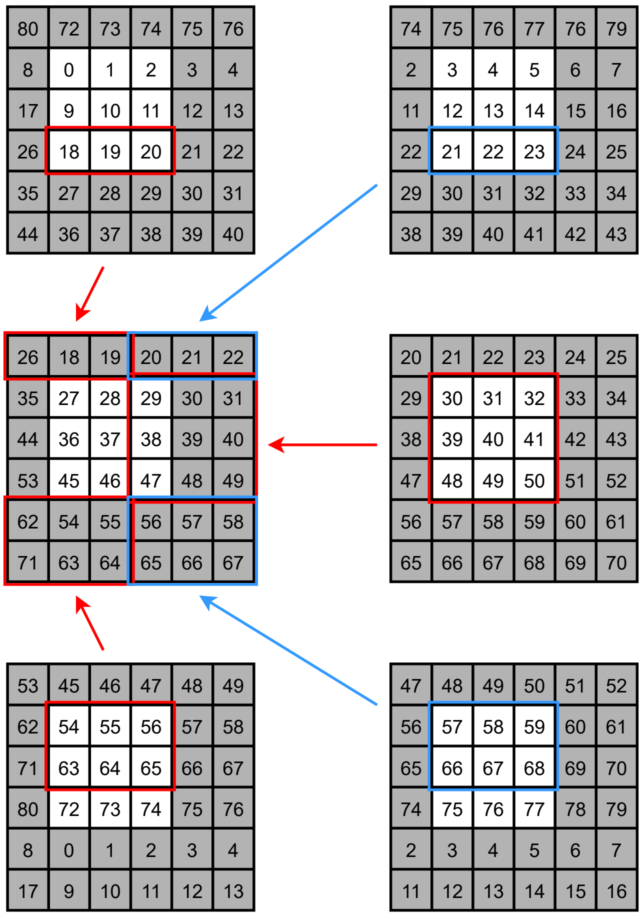


Figure 4.8: Moving window ghost cell electromagnetic field update.

4.4.5 Reporting

To be able to obtain the state of the simulation at any point during its execution, ZPIC implements a reporting functionality (ZPIC documentation also calls this feature simulation diagnostics). This reporting functionality can be used to obtain the current state of the various components that make up a ZPIC simulation, such as particle and cell values. Implementing this utility on our distributed implementation allows us to validate our implementation by comparing its result with the original ZPIC's result and measuring the magnitude of the differences between the two simulation states.

Since distributed reporting is not the focus of this thesis, we committed to a simple implementation of this feature. We also chose to only generate diagnostic information for electromagnetic field and electric current values of the simulation space, as this data would be enough to check the correctness of our version.

In our implementation, the process with rank zero has three segments, one for each of the simulation components that it will generate the diagnostic files for. These segments are used to emulate the entire (global) simulation space of the distributed simulation. Additionally, all processes have a reporting segment. When the time comes to generate a file with the current simulation state, each process will copy its cell data of all spatial quantities (electromagnetic field and electric current values) to this dedicated reporting segment row-wise. Then, each process will perform a *gaspi_write* call for each row of data in the reporting segment, and write that data directly to the relevant data segment for that spatial quantity, at the correct memory location for those cells in the global simulation space. After all writes are completed, the process with rank zero will have the state of all cells of the global simulation space. The three segments with the spatial quantities are then passed to the original ZPIC reporting routine that will pack the simulation state into a file that can be accessed later.

4.5 GASPI/OpenMP hybrid

In addition to our GASPI implementation, we also developed a hybrid GASPI OpenMP version. This version is meant to show how GASPI would perform on a more realistic use case of the standard, in combination with a shared memory API. This hybrid implementation still uses all the communication procedures outlined in Section 4.4. The additional threads are employed in the most demanding section of a ZPIC iteration, the particle mover.

To parallelize the particle mover, we implemented an approach that relies on the reduction of the electric current values that are generated by every moving particle. The default OpenMP reduction operation allocates and deallocates memory every time a thread enters and leaves a reduction construct. So, to minimize the number of dynamic memory operations, each thread will allocate its own private current buffer at the start of the simulation. Then, using OpenMP's user reduction feature, we defined a custom reduction operation so that each thread uses one of these private current buffers. This thread private memory region is used throughout the simulation and zeroed after each reduction. Each iteration, every thread will advance the particles assigned to it, using their private current buffer to save the resulting electrical current values. After a thread finishes its work, the resulting current values are added to the shared current buffer concurrently.

Outside of the particle mover current reduction, a significant difference between the hybrid and the GASPI-only implementation is the point in each iteration where leaving particles are moved from the particle array to the particle segments. In our GASPI implementation, after each particle is moved it is immediately determined if the particle needs to be reallocated into one of the neighboring processes and, if needed, the particle is promptly moved to one of such particle segments. In the hybrid implementation, we are using a *parallel for* construct to iterate through the particles. So removing particles from the particle array during the particle processing stage would involve changing the number of total particles left in the process while it is being used for the exit condition of the *parallel for* construct. Because of this, our hybrid implementation only identifies which particles should be reallocated at the end of each iteration right before the particle sending procedure is performed. To this end, each thread will iterate through its share of the particle buffer and decide which particles need to be transferred, and to which neighbor. After all particles are checked, the master thread (thread with id zero) will then move all outgoing particles to the correct particle segments. While the master thread moves outgoing particles of a species, other threads can identify the leaving particles of the next species.

Chapter 5

Experimental Evaluation

In this chapter, we start by describing the conditions for our experimental evaluation, followed by a brief examination of the correctness of our implementation. Then we analyze the performance of our GASPI implementation on two different ZPIC simulations, comparing the obtained performance with a reference MPI implementation. Finally, we examine the performance measured with our GASPI/OpenMP hybrid code.

5.1 Setup

All executions of our performance evaluation where performed on the MareNostrum 4 (MN4) supercomputer. MN4 is equipped with 3456 computing nodes, each fitted with two Intel Xeon Platinum 8160 CPUs, and 12×8GB 2667Mhz DIMMs for a total of 96 GB of RAM per node. Each of these CPUs has 24 cores running at 2.10GHz, totalling 48 cores per node, for a grand total of 165,888 CPU cores in the whole system.

To compile our code we used the Intel icc compiler version 17.0.0. We enabled the most powerful compiler optimizations available to the Intel Xeon Platinum 8160 CPU, including SIMD instructions. We used GPI-2 version 1.4.0 as the implementation of the GASPI standard.

The execution times used for the graphs are the average of three runs and were made with the reporting functionality disabled, as it would interfere with the execution times. We use the original ZPIC implementation for the sequential execution times and the MPI implementation in [27] as the basis for our performance comparisons. In all cases, the measured execution times ignore the setup phase of the simulation and only measure the time from the start of the first iteration up to the end of the simulation.

In our tests, out of the possible 48 processes per node, we could only manage to run 14 processes per node. This is due a compatibility issue with the current GPI-2 implementation in the MN4 hardware that limits the number of GASPI processes when using several MN4 nodes.

The performance tests we realized used simulation inputs based on the plasma experiments outlined in Section 3.2.

5.2 Implementation validation

To verify the correctness of our implementation, we used the reporting functionality offered by ZPIC and our distributed reporting implementation described in Section 4.4.5 to save the final state of the simulation grid. Then, using python scripts, we compare both end states and measure the magnitude of the differences between the two simulation results.

ZPIC uses single precision floating-point numbers to represent various quantities that define the state of the simulation. Since floating-point arithmetic is not associative, and the order of the operations involving these floating-point numbers is different from the original ZPIC implementation, it is very unlikely that both versions will ever yield the exact same result.

The MPI implementation [27] we are comparing our implementation to only supports distributed reporting of the magnetic field, so our analysis will be focused on that spatial quantity.

Figures 5.1 and 5.2 show the absolute error in one of the three magnetic field axes for a Weibel instability and a laser wakefield acceleration simulations respectively. The Weibel instability simulation has a grid size of 512 by 512 cells, two species each with 36 particles per cell each, and runs for a total of 500 iterations. While the laser wakefield acceleration has a grid size of 2000 by 256 cells, with the right half of the simulation space filled with 8 particles per cell, with more particles being injected as the window moves, and runs for 1450 iterations. Both simulations ran on 8 processes.

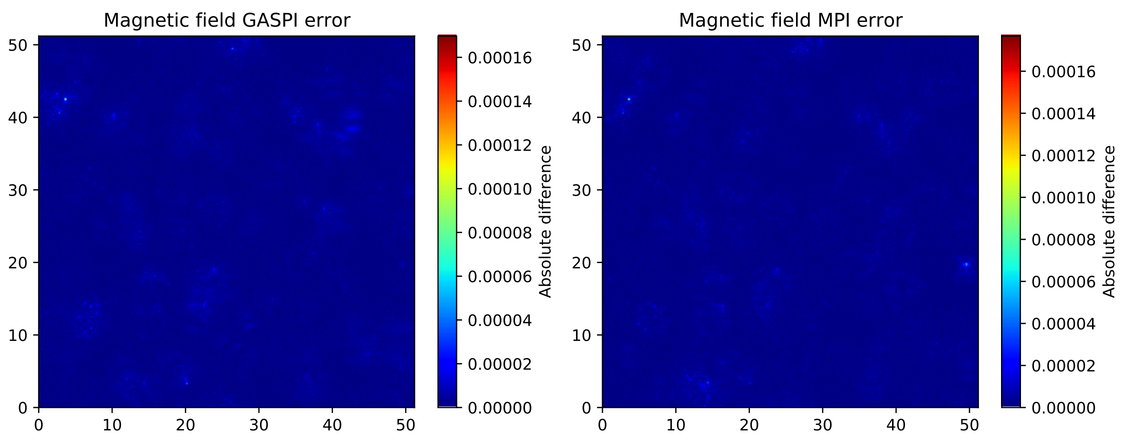


Figure 5.1: Weibel instability GASPI/MPI absolute error.

As we can see by the figures, the final result of both implementations differs slightly from the ZPIC result. We tested both implementations on a range of different simulations, on both MN4 and personal computers with different compilers, and no implementation proved superior to the other in this regard. The absolute error measured on both implementations is very comparable and on the same order of magnitude. In conclusion, the extent of the differences between the results of our GASPI implementation and the original ZPIC code is acceptable, validating the correctness of the implementation.

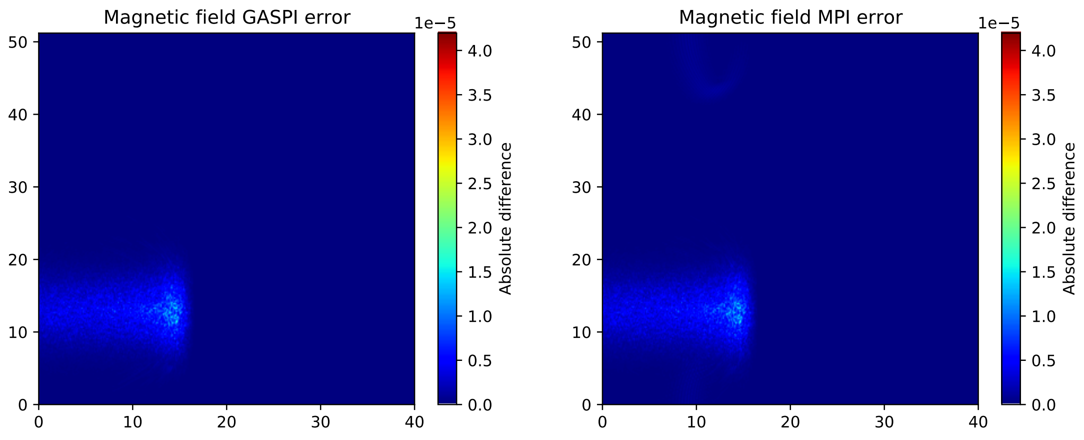
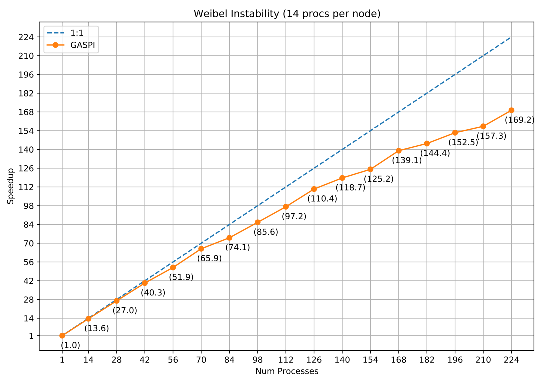


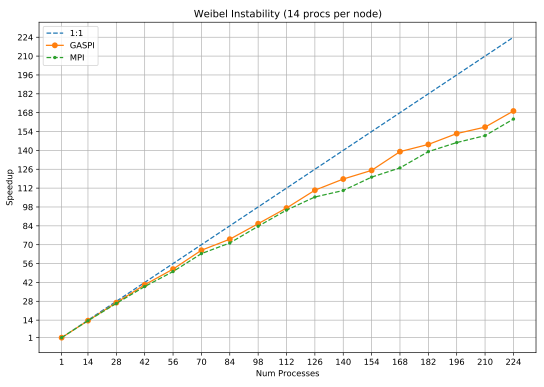
Figure 5.2: Laser wakefield acceleration GASPI/MPI absolute error.

5.3 Performance evaluation of the GASPI implementation

Figure 5.3a shows the speedups obtained in a 500 iteration Weibel instability simulation with grid size of 512 by 512 cells where each of the two species have 1024 particles per cell. For comparison, Figure 5.3b shows the speedups obtained by the reference MPI implementation in the same Weibel instability simulation.



(a) GASPI speedups.

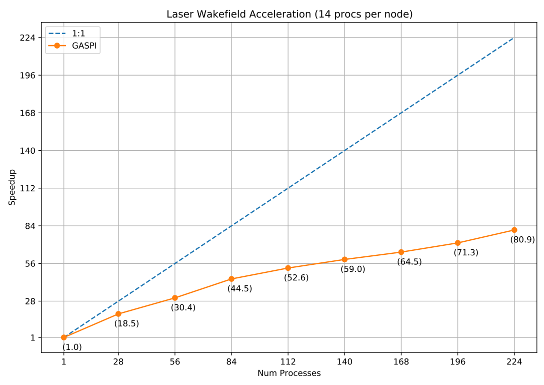


(b) GASPI/MPI speedups.

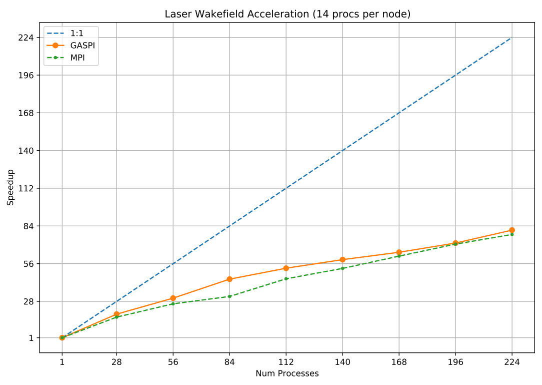
Figure 5.3: GASPI/MPI speedups obtained on a Weibel instability simulation.

As we can see by the graphs, our GASPI implementation behaves rather well. With 224 processes the GASPI implementation achieves a speedup of 169.2 compared to the 163.2 of the MPI version. These speedups translate to an efficiency of 75.5% for our GASPI implementation and 72.9% for the reference MPI version. In short, with 224 processes, the our implementation was 3.66% faster than the MPI code.

For the laser wakefield acceleration tests, we used a simulation with a grid size of 2000 by 512 cells. The simulation starts with zero particles, these are later injected to the right side of the simulation space with a density of 64 particles per cell. The simulations run for a total of 4000 iterations. Figures 5.4a and 5.4b show the speedups obtained by the GASPI and MPI versions respectively.



(a) GASPI speedups.



(b) GASPI/MPI speedups.

Figure 5.4: GASPI/MPI speedups obtained on a laser wakefield acceleration simulation.

This time the results are somewhat disappointing, neither of the implementations scale very well. With 224 processes our GASPI implementation only managed to achieve a speedup of 80.9, while the MPI version managed a speedup of 77.7. These result in the mediocre efficiencies of 36.1% and 34.7% respectively.

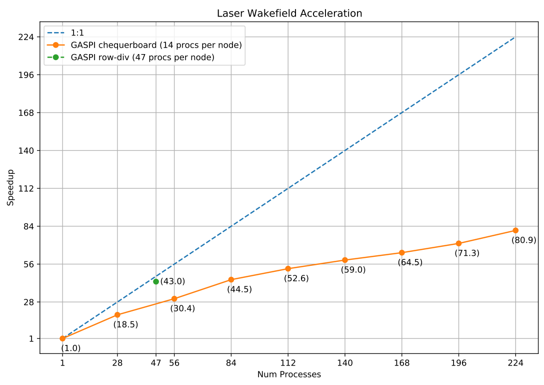
Upon investigating, we theorized that the possible cause for the mediocre scaling may be an unbalance of the workload between the participating processes. Since most columns will not hold any particles for many iterations, many processes, especially the ones in the left side of the simulation space, will have to wait for many iterations until particles reach them. This leads to a very significant amount of wasted CPU time for the better part of the simulation, until the simulation space is completely filled with particles.

To confirm our theory we tried running the same simulation with 47 processes (on a single node[1]). Since 47 is a prime number, our partitioning algorithm cannot decompose it into other integers. So, it will be forced to assign all 47 blocks to the y-axis, essentially achieving a row decomposition of the simulation space. Figure 5.5a shows the row decomposition test results[2] in comparison to our previous results for the chequerboard decomposition laser wakefield acceleration simulation. As we can see, the executions with 47 processes using a row-wise division of the simulation space manage to beat 56 processes organized in a chequerboard arrangement, and also almost achieves the same speedup as 84 processes in a chequerboard configuration.

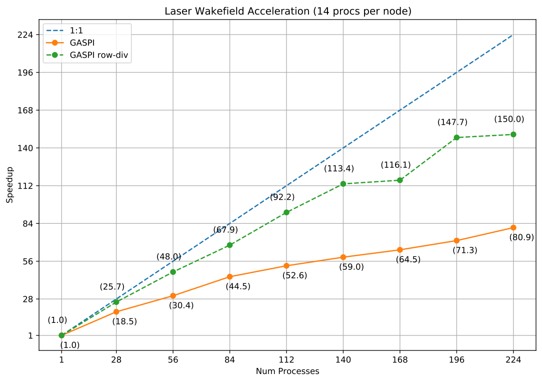
Armed with this knowledge, we modified our partitioning algorithm to allow the user to select a row-wise decomposition. If the user specifies that the simulation space should be partitioned

[1]Since we are running all processes on the same node, the 14 process per node limit does not apply.
[2]The green dot represents the speedup obtained with 47 processes with a row-wise decomposition.

using a row-wise decomposition, the algorithm, instead of trying to balance the number of blocks in both axes, will assign all divisions to the y-axis. Figure 5.5b illustrates the speedups measured in our GASPI implementation using the checkerboard and the row-wise decompositions in the same laser wakefield acceleration simulation. This time the results were much better. When running a total of 224 processes the row-wise decomposition achieves a speedup of 150, in contrast to the speedup of 80.9 measured with the checkerboard decomposition. These speedups translate to efficiencies of 70% and 36% respectively. In other words, this change in the simulation space partitioning scheme yielded a performance increase of 85.38% with 224 processes, a noticeable improvement.



(a) GASPI row-wise decomposition test (green dot).



(b) GASPI row-wise decomposition speedups.

Figure 5.5: GASPI speedups obtained on a laser wakefield acceleration simulation using a row-wise decomposition.

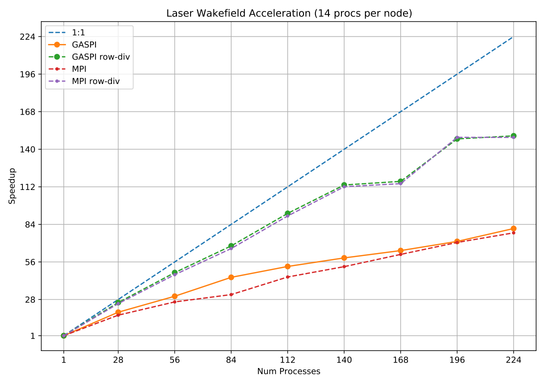
For comparison, we also implemented a row-wise decomposition in the reference MPI implementation. Figure 5.6a compares the obtained speedups by the two implementations using both decompositions on the same laser wakefield acceleration simulation. The improvement showed by our distributed implementation, by changing the decomposition to a row-wise decomposition, is mirrored by the reference MPI implementation. Using this decomposition the two implementations demonstrate very comparable performance. By employing a row-wise decomposition the reference MPI code achieved a speedup of 148.8 when running with 224 processes, very close to the speedup measured with our GASPI code of 150.

Additionally, we measured the performance of our GASPI code when running a Weibel instability simulation with a row-wise decomposition. As we can observe by Figure 5.6b, although the difference in performance is not very pronounced, a checkerboard decomposition is still preferable when the workload is more evenly distributed across the simulation space.

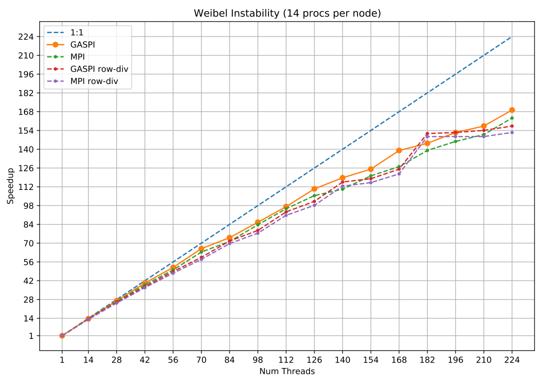
To understand why our implementation was not scaling any better, we analyzed the execution of our GASPI code with Extrae[3] and Paraver[4]. Both tools were developed by the Barcelona Supercomputing Center to aid programmers in profiling their multi-process and multi-threaded code. While Extrae captures instrumentation information during the program execution, Paraver offers

[3]https://tools.bsc.es/extrae
[4]https://tools.bsc.es/paraver

a user interface to browse and analyze the instrumentation data.



(a) Laser wakefield acceleration GASPI/MPI speedups.



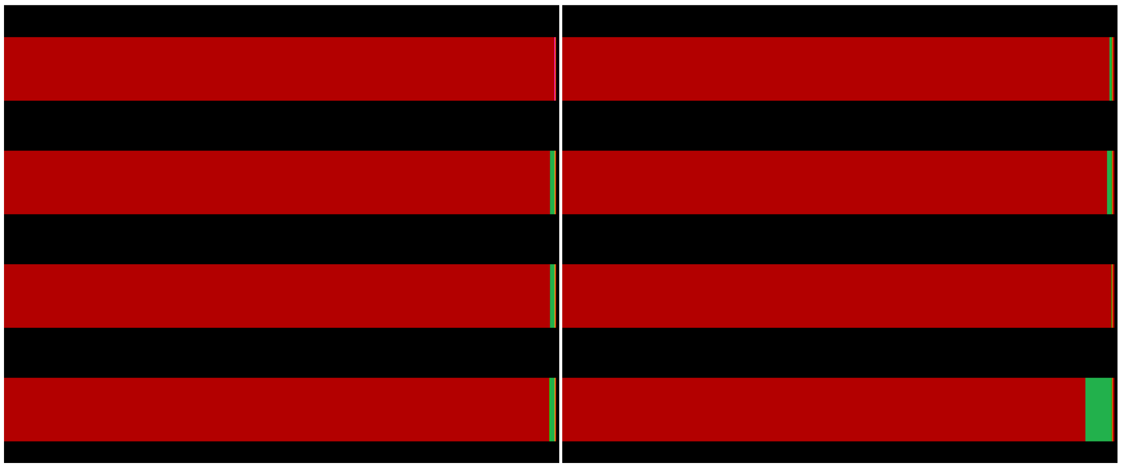
(b) Weibel instability GASPI/MPI speedups.

Figure 5.6: Speedups obtained with a row-wise/chequerboard decomposition.

After analyzing the instrumentation data, we concluded that the primary performance bottleneck in our GASPI implementation is workload distribution. In the early iterations of a Weibel instability simulation, the number of particles across all processes is fairly balanced. As the simulation progresses, and particles start to move between processes, an imbalance starts to emerge. After a few hundred iterations this imbalance evolves to a noticeable degree. At this point, some processes have to wait for a significant amount of time for remote data, because their neighbors are now taking longer to process all of their particles. Resulting in wasted CPU time. As the number of processes increases, the average number of particles per process decreases, but the imbalance in the number of particles per process is not affected to the same degree.

Figures 5.7 show the output of the Paraver tool when analyzing two iterations of a small Weibel instability simulation on four processes. While this execution is only running on four processes, the performance patterns reflect the behavior of the code on larger executions. The left figure shows the CPU time distribution in an early iteration, while the right figure represents an iteration closer to the end of the simulation. The red area that dominates all processes on both figures symbolizes the time spent by each of the processes on the particle mover, while the smaller green zone represents the time a process is waiting for remote electric current data. As we can observe, all four processes start the simulation with a very balanced workload. Neither of the processes has to wait very long for remote data. However, we can see by the figure on the right that as the simulation progresses the workload becomes unbalanced. This leads to wasted CPU time while processes wait for their neighbors to finish processing their workload and send the necessary data.

This issue is also present in laser wakefield acceleration simulations, on a much bigger scale. Some processes can spend many iterations with no particles, waiting for their busy neighbors. This can be addressed by using a row-wise decomposition instead of a chequerboard decomposition. However, the area where the laser pulse collides with particles sees much more activity than the rest of the simulation space, creating an imbalance of the workload.



(a) Early iteration. (b) Later iteration.

Figure 5.7: Paraver visualization of a Weibel instability simulation on GASPI code.

5.4 *Performance evaluation of the GASPI/OpenMP hybrid implementation*

When testing our GASPI/OpenMP hybrid implementation we could have utilized all 48 CPU cores available in each node, but to maintain the same testing conditions as our GASPI evaluation we only used 14 cores per node.

Figure 5.8 shows the performance comparison between the GASPI and hybrid versions in the same Weibel instability simulation used in Section 5.3. The orange line shows the speedups obtained by our GASPI implementation, running with 14 processes per node. The green and red lines show the measured speedups by our hybrid implementation, running with 1 process per node, 14 threads per process, and 2 processes per node, 7 threads per process respectively.

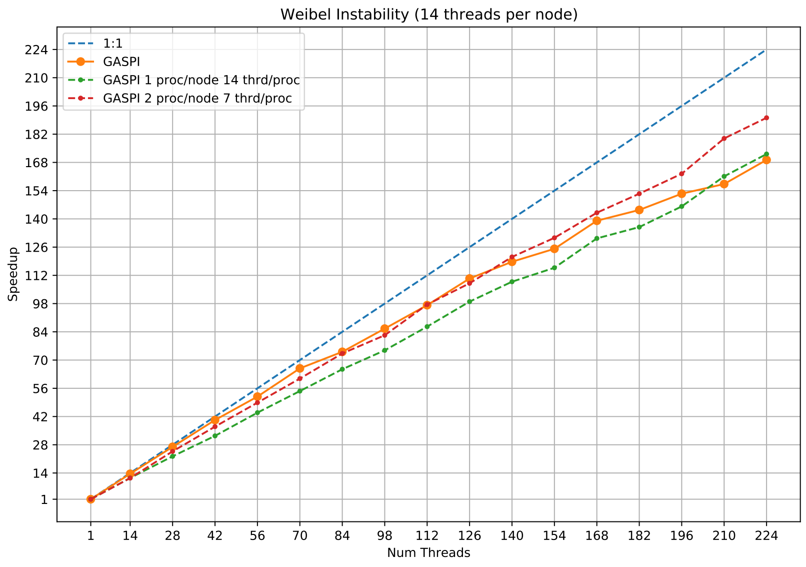


Figure 5.8: Weibel instability GASPI/OpenMP hybrid speedups.

The graph shows the hybrid version falling behind the GASPI implementation in the executions with a small number of threads. However, the hybrid implementation manages to outperform the GASPI version in executions with more processing cores. When using a total of 224 CPU cores the GASPI implementation achieves a speedup of 169.2, translating to an efficiency of 75.5%, while the hybrid version measures a speedup of 172.2 (1.72% faster than the pure GASPI code) with 1 process per node, 14 threads per process, and 190 (12.34% faster) with 2 processes per node, 7 threads per process. Achieving efficiencies of 76.9% and 84.9% respectively.

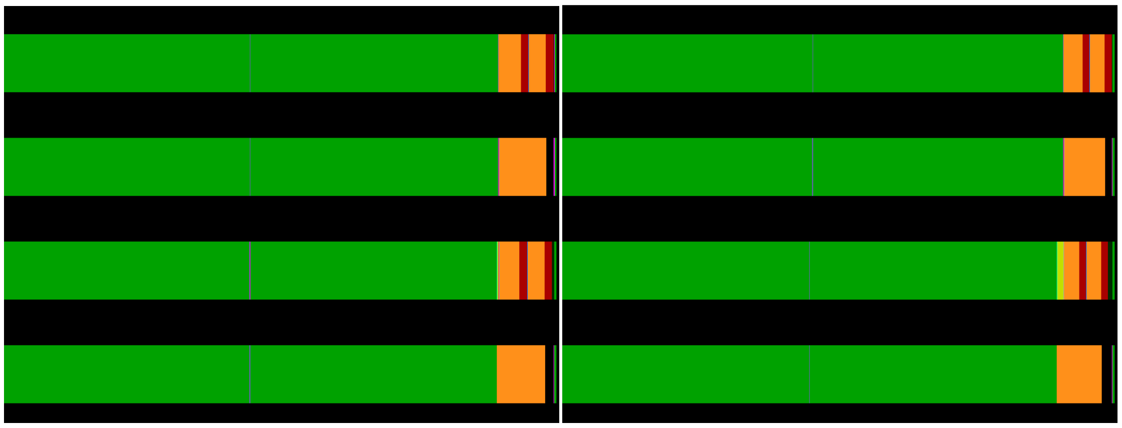

(a) Early iteration. (b) Later iteration.

Figure 5.9: Paraver visualization of a Weibel instability simulation on GASPI/OpenMP hybrid code.

To understand this performance behavior, we profiled the hybrid implementation. Figures 5.9 show the Paraver representation of the CPU time utilization of our hybrid code on a Weibel instability simulation with 2 processes, each with 2 threads. The first two colored rectangles represent the two threads of one of the processes, while the last two represent the other process. The bright green area represents the time spent on the particle mover. Here we see a situation similar to Figure 5.7, where the workload becomes more imbalanced as the simulation progresses. As expected, the serial portion of the hybrid code where the outgoing particles are reallocated to the particle, represented by the red areas, diminishes the benefit of the extra threads. Note the empty area on the second and fourth threads, this represents the idle time while extra threads wait for the main thread of each process while it moves outgoing particles to the particle segments. The orange zone represents the time spent iterating through the particles, to determine which need to be reallocated. Also, in the right figure, we can see the second process waiting for remote data, represented by the yellow and the darker green zones.

However, since the additional threads can dynamically divide the workload of a process between them, they mitigate the difference in the workload across processes. Also, as the number of processes increases, the average number of particles per process is lower, reducing the amount of time needed to copy the outgoing particles to the particle segments, where the additional threads are not useful. The under-decomposition of the workload of each process and the gradual reduction of the serial fraction of the in-process code allow the hybrid version to eventually outperform the pure GASPI implementation.

Chapter 6

Conclusions

The current rate of innovation of computer hardware suggests that near future supercomputers will have the processing power and communication infrastructure scalability necessary to reach exascale levels of performance (speeds in the order of 10^{18} floating-point operations per second). However, the industry-standard bulk-synchronous two-sided communication models in use today offer no guarantee that they will be able to power the next generation of supercomputers.

Looking for an alternative, we developed a distributed implementation of a well-known plasma simulation tool using GASPI, a novel communication API standard that focuses on asynchronous communication and execution. Then, we conducted a thorough performance evaluation of our GASPI implementation on the MareNostrum 4 supercomputer, comparing it to an optimized implementation of the same tool powered by MPI.

As we discussed in Section 5.3, GASPI performs rather well. Our GASPI based distributed implementation managed to remain competitive with the MPI implementation, often outperforming the current state of the art in our tests. However, usability falls short when comparing to the industry standard. Compared to MPI, installing GPI-2 and running GASPI applications requires some effort, even on a personal computer. Compared to synchronous two-sided communication models, GASPI requires more preparation from the programmer. For example, computing offsets of local and remote memory locations, and the asynchronous nature of the communication routines can easily lead to erroneous situations. The lack of debugging tools for GASPI code is a major drawback, requiring awkward workarounds.

Despite its current drawbacks, GASPI is a significant advancement for the high-performance computing industry. GASPI and GPI-2 are still in active development, and will only improve with time. We are confident that GASPI offers the performance and scalability required to power next-generation exascale supercomputers.

For future work, we propose exploring the potential performance improvements offered by the zero-copy mechanism present in GASPI. Our implementation did not leverage such a feature, and since this is one of the primary mechanisms implemented by GASPI, we think a study should be performed to assess the potential performance gain possible by properly leveraging this feature.

Bibliography

[1] H. El-Rewini and M. Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. USA: Wiley-Interscience, 2005. ISBN: 0471467405.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard." In: *Parallel computing* 22.6 (1996), pp. 789–828.

[3] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269.

[4] J. Protic, M. Tomasevic, and V. Milutinovic. "Distributed shared memory: Concepts and systems." In: *IEEE Parallel & Distributed Technology: Systems & Applications* 4.2 (1996), pp. 63–71.

[5] B. Nitzberg and V. Lo. "Distributed shared memory: A survey of issues and algorithms." In: *Computer* 24.8 (1991), pp. 52–60.

[6] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas. "Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory." In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM. 2015, pp. 3–14.

[7] *GASPI Forum*. URL: http://www.gaspi.de (visited on 11/20/2019).

[8] R. Calado. "Computational Toolkit for Plasma Physics Education." Master's thesis. Instituto Superior Técnico (IST), 2018.

[9] T. Tajima and J. M. Dawson. "Laser electron accelerator." In: *Physical Review Letters* 43.4 (1979), p. 267.

[10] W. Lu, M Tzoufras, C Joshi, F. Tsung, W. Mori, J Vieira, R. Fonseca, and L. Silva. "Generating multi-GeV electron bunches using single stage laser wakefield acceleration in a 3D nonlinear regime." In: *Physical Review Special Topics-Accelerators and Beams* 10.6 (2007), p. 061301.

[11] R. A. Fonseca, L. O. Silva, F. S. Tsung, V. K. Decyk, W. Lu, C. Ren, W. B. Mori, S Deng, S Lee, T Katsouleas, et al. "OSIRIS: A three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators." In: *International Conference on Computational Science*. Springer. 2002, pp. 342–351.

[12] J. Torrellas, H. Lam, and J. L. Hennessy. "False sharing and spatial locality in multiprocessor caches." In: *IEEE Transactions on Computers* 43.6 (1994), pp. 651–663.

[13] H. Kasim, V. March, R. Zhang, and S. See. "Survey on parallel programming model." In: *IFIP International Conference on Network and Parallel Computing*. Springer. 2008, pp. 266–275.

[14] M. Sato. "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors." In: *15th International Symposium on System Synthesis, 2002.* IEEE. 2002, pp. 109–111.

[15] *OpenMP*. URL: http://www.openmp.org (visited on 10/01/2019).

[16] J. Liu, J. Wu, and D. K. Panda. "High performance RDMA-based MPI implementation over InfiniBand." In: *International Journal of Parallel Programming* 32.3 (2004), pp. 167–198.

[17] *OpenMPI*. URL: https://www.open-mpi.org (visited on 11/01/2019).

[18] L. Smith and M. Bull. "Development of mixed mode MPI/OpenMP applications." In: *Scientific Programming* 9.2-3 (2001), pp. 83–98.

[19] D. Grünewald and C. Simmendinger. "The GASPI API specification and its implementation GPI 2.0." In: *7th International Conference on PGAS Programming Models*. 2013, p. 243.

[20] A. Kalia, M. Kaminsky, and D. G. Andersen. "Using RDMA efficiently for key-value services." In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 295–306.

[21] C. Mitchell, Y. Geng, and J. Li. "Using One-Sided {RDMA} Reads to Build a Fast, CPU-Efficient Key-Value Store." In: *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 2013, pp. 103–114.

[22] M. D. Hill and M. R. Marty. "Amdahl's law in the multicore era." In: *Computer* 41.7 (2008), pp. 33–38.

[23] J. P. Verboncoeur. "Particle simulation of plasmas: review and advances." In: *Plasma Physics and Controlled Fusion* 47.5A (2005), A231.

[24] D Tskhakaya, K Matyash, R Schneider, and F Taccogna. "The Particle-In-Cell Method." In: *Contributions to Plasma Physics* 47.8-9 (2007), pp. 563–594.

[25] C. Huntington, F Fiuza, J. Ross, A. Zylstra, R. Drake, D. Froula, G Gregori, N. Kugland, C. Kuranz, M. Levy, et al. "Observation of magnetic field generation via the Weibel instability in interpenetrating plasma flows." In: *Nature Physics* 11.2 (2015), pp. 173–176.

[26] R. Schlickeiser and P. K. Shukla. "Cosmological magnetic field generation by the Weibel instability." In: *The Astrophysical Journal Letters* 599.2 (2003), p. L57.

[27] P. Ceyrat. "Applying Modern HPC Programming Platforms to Plasma Simulations." Master's thesis. Instituto Superior Técnico (IST), 2019.