Exploiting future Exascale systems with Partitioned Global Address Spaces

Bruno Manuel Mendes Amorim
bruno.amorim@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

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

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 [5, 3]. Additionally, shared memory cache-coherent hardware is expensive and hard to design [16]. 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 [15, 17]. But in reality, the paradigm requires more research and development before their promise of state-of-the-art message-passing performance can be delivered [11]. 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[1] (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

[1] `http://www.gaspi.de`

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 [1], using GPI-2[2], 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 [13, 21] to thermonuclear fusion [1]. 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 [4]. 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.

2. Parallel Programming

In this chapter, we will cover some parallel programming paradigms in use today, along with some implementations of each programming model. In conclusion, we give 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 [16]. 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 [3]. The use of caches might also introduce false sharing [22].

A good example of a modern implementation of the shared memory programming model can be found in OpenMP [18]. OpenMP[3] is an application programming interface (API) that supports multi-platform shared-memory parallel programming. OpenMP's compiler directives and callable runtime library routines extend well-established programming languages like C, C++, and Fortran [10].

[2] `http://www.gpi-site.com/`

[3] `http://www.openmp.org`

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 [3, 16]. Communication between workers is explicit and very frequently handled by a message passing interface, a well-established paradigm for parallel programming [5, 16]. The number of processors on modern massive parallel systems can reach the hundreds of thousands [3].

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 [16]. Thanks to simpler hardware [16], higher efficiency [5] and scalability [3] (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.

The most widely used implementation of this paradigm is the MPI (Message Passing Interface) standard [5]. MPI offers a rich collection of point-to-point communication procedures and collective operations for data movement, global computation, and synchronization.

Currently, MPI is extensively used in clusters and other distributed memory systems due to its rich functionality [3] and is considered the defacto standard for developing high-performance computing applications on distributed memory architectures [12]. It provides language bindings for C, C++, and Fortran [10]. Some well-known MPI implementations are MPICH [5] and OpenMPI[4].

However, 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 [20] 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.

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.

On the other hand, the technology we will be using is called GASPI[5]. GASPI (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 driven communication [6]. RDMA (Remote Direct Memory Access) [9, 14] 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.

2.3. Distributed Shared Memory

The final class of parallel systems we will outline is called distributed shared memory (DSM) systems. In these systems, the address space is, just like in distributed memory architectures, composed of the

[4] `https://www.open-mpi.org`

[5] `http://www.gaspi.de`

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.

Lacking affordable dedicated hardware for a distributed shared address space on a large scale [11], 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 [17].

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 [7]. Speedup can be experimentally obtained by the ratio between the execution time of the serial and parallel versions.

$$Speedup(p) = \frac{Sequential\ Execution\ Time}{Parallel\ Execution\ Time} \quad (1)$$

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):

$$Efficiency(p) = \frac{Speedup(p)}{p} \quad (2)$$

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 [1, 21], plasma simulation is a valuable and challenging research subject [24].

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 Particle-in-Cell (PIC) concept was formalized during the 1970s [24] and is well suited to study complex systems with a great degree of freedom and accuracy [4]. 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) [1]. In each iteration, these elements interact with each other to advance the simulation state. The way these elements interact can be summarized by Figure 1. 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.

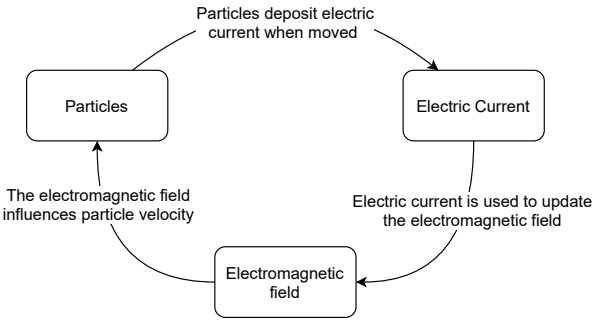

Figure 1: Sequence of computations in the PIC method.

3.1. ZPIC

ZPIC [1] is a fully relativistic PIC implementation focused on educational plasma physics simulations. ZPIC is not implemented to benefit from parallel execution and is meant to be run on common laptops and personal computers. This was a deliberate design decision with the intention of reducing code complexity.

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.

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 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 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.

Weibel instability [8, 19] 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.

Laser wakefield acceleration [21, 13] 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.

4. Implementation

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 implemented a procedure to divide the simulation space and assign the resulting regions to the participating processes.

4.1. 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.

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, we implemented 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.

4.2. 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.

In our distributed implementation, each process has a dedicated segment for each of its eight neighbors. This segment is used to both send and receive data and is shared by all species. Figure 3 shows an example of this segment organization in action. To relay the number of particles sent on each message we added an additional particle to the beginning of each data transmission, this particle will be known as the *fake particle*. In Figure 3 we demonstrate how the particle segments would be used to transmit two different species between neighbors.

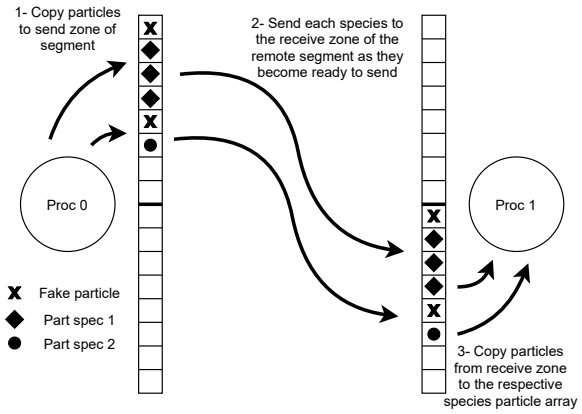

Figure 3: Particle communication example.

4.3. 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. Figure 4 shows 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.

The sending procedure is as follows: First, identify the cells to send. 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.

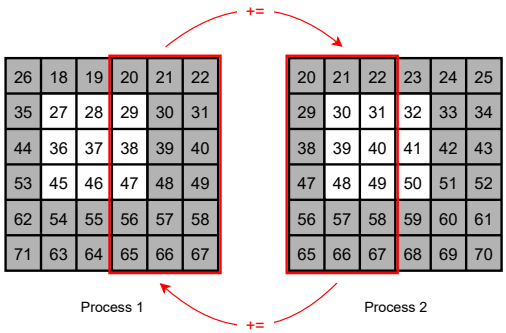

Figure 4: Electric current communication between two processes.

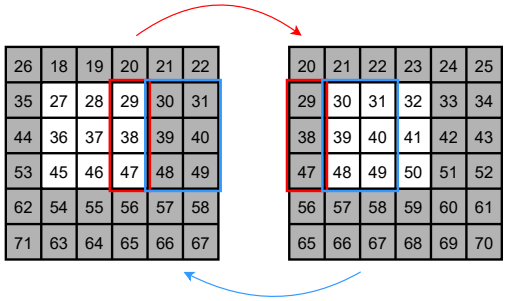

Figure 5: Electromagnetic field ghost cell update.

The procedure on the receiving side is just as simple. Wait for the GASPI notification from the neighbor we are waiting for. Then, 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.

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 5. As before, cells are marked with the number of the cell they represent. Also, note that ghost cells are overwritten with the value present on their respective real cell.

To implement this, we resorted to the same segment structure as explained in Section 4.3. 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.

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 **??**. 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.

5. Experimental Evaluation

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 [2] 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.

5.1. Performance evaluation

Figure 6 shows the speedups obtained by both our GASPI implementation and the reference MPI implementation 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.

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. Figure 7 shoes the speedups obtained by the GASPI and MPI implementations.

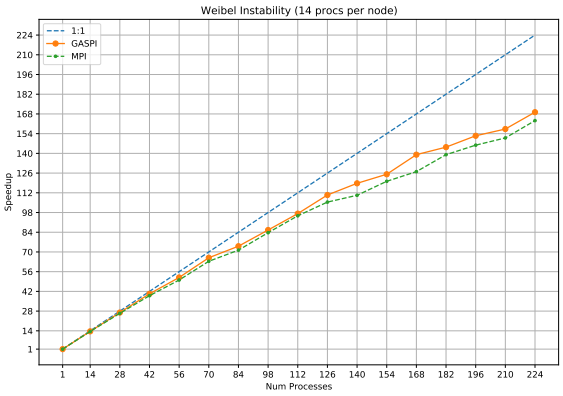

Figure 6: GASPI/MPI speedups obtained on a Weibel instability simulation.

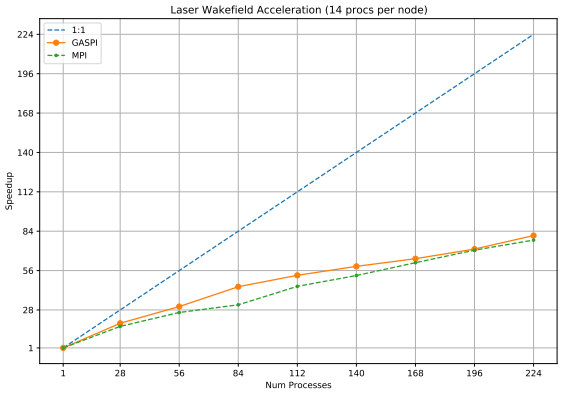

Figure 7: GASPI/MPI speedups obtained on a laser wakefield acceleration simulation.

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.

Armed with this knowledge, we modified our partitioning algorithm to allow the user to select a row-wise decomposition. Figure 8 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 our implementation with 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.

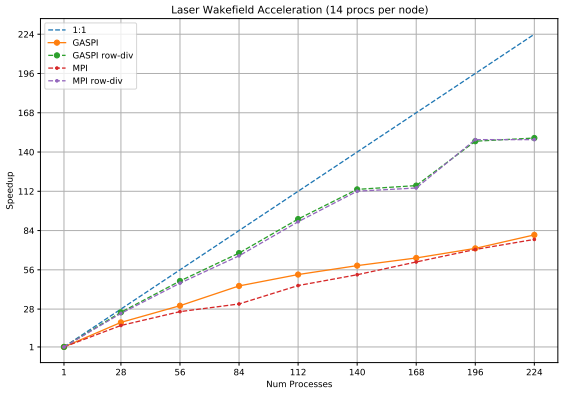

Figure 8: GASPI/MPI speedups obtained on a laser wakefield acceleration simulation using a row-wise decomposition.

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, 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 these 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.

References

[1] R. Calado. Computational toolkit for plasma physics education. Master's thesis, Instituto Superior Técnico (IST), 2018.

[2] P. Ceyrat. Applying modern hpc programming platforms to plasma simulations. Master's thesis, Instituto Superior Técnico (IST), 2019.

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

[4] 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*, pages 342–351. Springer, 2002.

[5] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[6] D. Grünewald and C. Simmendinger. The gaspi api specification and its implementation gpi 2.0. In *7th International Conference on PGAS Programming Models*, page 243, 2013.

[7] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[8] 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. *Nature Physics*, 11(2):173–176, 2015.

[9] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

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

[11] 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*, pages 3–14. ACM, 2015.

[12] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[13] 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. *Physical Review Special Topics-Accelerators and Beams*, 10(6):061301, 2007.

[14] 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)*, pages 103–114, 2013.

[15] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

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

[17] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[18] M. Sato. Openmp: parallel programming api for shared memory multiprocessors and on-chip multiprocessors. In *15th International Symposium on System Synthesis, 2002.*, pages 109–111. IEEE, 2002.

[19] R. Schlickeiser and P. K. Shukla. Cosmological magnetic field generation by the weibel instability. *The Astrophysical Journal Letters*, 599(2):L57, 2003.

[20] L. Smith and M. Bull. Development of mixed mode mpi/openmp applications. *Scientific Programming*, 9(2-3):83–98, 2001.

[21] T. Tajima and J. M. Dawson. Laser electron accelerator. *Physical Review Letters*, 43(4):267, 1979.

[22] J. Torrellas, H. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[23] D. Tskhakaya, K. Matyash, R. Schneider, and F. Taccogna. The particle-in-cell method. *Contributions to Plasma Physics*, 47(8-9):563–594, 2007.

[24] J. P. Verboncoeur. Particle simulation of plasmas: review and advances. *Plasma Physics and Controlled Fusion*, 47(5A):A231, 2005.