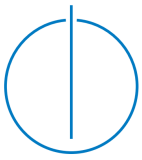
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Optimization of Molecular Dynamics Simulations Using One-Sided MPI-Directives

Andreas Schmelz



DEPARTMENT OF INFORMATICS

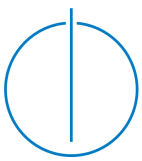
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Optimization of Molecular Dynamics Simulations Using One-Sided MPI-Directives

Optimierung von Molekulardynamik Simulationen Mittels Einseitiger MPI-Direktiven

Author:	Andreas Schmelz
Supervisor:	Prof. Dr. H.-J. Bungartz
Advisor:	Steffen Seckler
Submission Date:	15.06.2017



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.06.2017 Andreas Schmelz

Abstract

Molecular Dynamics (MD) Simulations are an interesting and important field of research and application. With molecular dynamics simulations we can model, observe and predict the behavior of molecules. Compared to a laboratory research we can get results faster, cheaper and with a wider range of variety. Limitations and regulations do not apply to simulations as they do to real-world experiments. In order to generate accurate results we need to create a simulation that is as close as possible to the real world. For this, models help us to describe the inter-molecular behavior with mathematical formulas of physical properties. When we want to simulate these models we need to discretize the time and the length to represent the particles in the simulation in correlation to the real world. A simulation, in order to be accurate, requires a small time step. To simulate a meaningful amount of matter, a huge number of particles needs to be computed. This creates a large computation effort. To scale the problem and compute it across multiple machines like a cluster or a supercomputer we need to parallelize the execution. Each instance of the program will compute and solve a subset of the entire problem. Their local results can be merged into a global representation. The exchange between computation nodes on a distributed memory system is often done with the MPI library.

MD simulations are a general purpose research tool that can model a wide variety of substances and chemicals. This differentiation is done by modifying a set of configuration parameters and by exchanging models. The MD simulation pattern is a widely used stencil computation halo exchange pattern. Knowledge gained by the MD simulations can thus be transformed to other applications using this model. This makes MD simulations an excellent field of application and research for large-scale parallel, distributed memory programming.

The Message Passing Interface (MPI) is an established, widely adopted approach for parallelization on scientific and commercial programs. In the last two major releases (2.0 and 3.0) MPI introduced and revised one-sided communication. This is a new programming model of how communication and synchronization is performed by the user's application. This new model promises an easy semantic, leveraging hardware features and delivering high performance data exchange. As the name 'one-sided' suggests it only involves one participant to issue a communication. This should allow a passive progression of communication during computing, thus a higher communication

to computation overlap.

In this thesis we want to look on one-sided communication, explain its core functionality, present and compare micro benchmarks to point to point (PTP) communication. Finally we want to optimize our molecular dynamics simulation for parallel performance. We will introduce one-sided communication and benchmark the impact on scalability.

Contents

1 Introduction

Molecular Dynamics simulations are a useful and interesting field of research in multiple components. First an MD simulation can be used in multiple fields such as biology, physics or chemistry. A particle simulation represents many small entities (the particles), and presents how they interact with each other. A high number of unique, individual particles can assemble a macrodynamic behavior, which is simply described by the interaction between a single pair of particles. By exchanging the interaction model we can simulate a wide variety of problems. Molecular dynamics simulations are used to simulate gases such as Argon, Ethylenoxide and Benzene [Eck12] but also in heterogeneous catalysis [Mat16], simulating frictional forces at nanoscale [Ben16] or the dynamics of transmembrane domains [Nor16] and ligand-protein binding free energies [Pet16].

Secondly it is a field of research for computer scientists to experiment with algorithms and methods to implement a large parallel program. Configurations like the particles' representation size or the time span allows us to arbitrarily scale the overall necessary resources for memory, arithmetic computation and data transfer. The accuracy of a simulation is determined by the used time step and by the magnitude that a particle is represented by. We can define a particle to represent a drop of liquid, a single molecule, an atom, or even simulate quarks, and electrons. The smaller we choose our scale, the more accurate are our results, but we also need to simulate more particles to simulate an equal sized amount of matter.

The architectural structure of a simulation is rather simple and interchangeable. This allows us to compare multiple different approaches about how a sub-problem of the simulation should be implemented in a specific case.

1.1 Molecular Dynamics

Our molecular dynamics simulation will consist of a fixed domain with a static amount of particles. The particles within the domain are independent from each other and each has its own properties such as position, velocities and forces. They can interact with any other particle within the simulation. Each particles position and properties are updated in a loop with a small, discrete time step. It is a N-body problem. As any

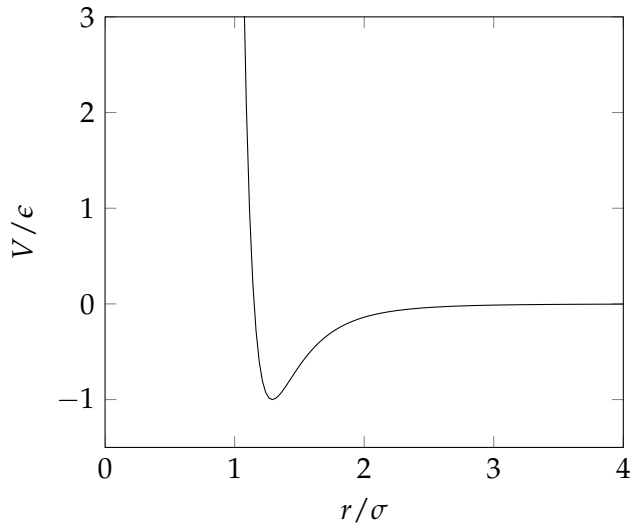


Figure 1.1: Lennard-Jones 6,12 potential

particle might exchange forces with any other particle the overall time complexity is $O(n^2)$ where n is the number of particles.

1.2 Lennard-Jones Potential

The Lennard-Jones potential is a model to approximately describe multiple complex physical principals in mathematical terms. It describes a potential that only depends on the distance of particles. We can derive the force between two particles from this potential.

$$V = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{1.1}$$

Equation 1.1 shows the Lennard-Jones Potential, where ϵ and σ are matter specific values. These can be adjusted to represent different materials in the real world. r is the distance.

In Figure 1.1 we see the plot of the 6,12 Lennard-Jones potential. For distances smaller than ϵ the repulsive part dominates the equation which converges towards infinity when approaching 0 from the right. These short range forces represent the Pauli repulsion. The attraction, long range forces represent van der Waals forces that converge to zero for growing distances. The Lennard-Jones potential is zero at $r = \sigma$, by defining σ, we can also define the distance at where the force between particles is

zero. At $2^{1/6}\sigma$ The Lennard-Jones potential has its minimum. This is the distance with the strongest attraction between particles.

While the calculation of position and velocity only requires to iterate over each particle once, we have to iterate over all pairs of particles to calculate th forces. This has the time complexity of $O(n^2)$, with n being the number of particles. with Newton's third law applied we can omit to iterate over half the pairs as $F_ij = -Fji$ holds. Every force has a counterforce of inverse direction.

1.3 Time Integration

Many models describe particle interactions with a system of partial differentials. These partial differentials with an initial configuration of particles gives that is called initial value or Cauchy problem. While we cannot directly solve the differentials, we can approximate the solution by time integration. This means we will run the simulation with a discrete, fixed time step. A time integration method describes in which way we will update the properties of a particle over a time step. This unavoidably introduces an error which increases with the time step. Different time integration methods try to minimize the error by updating values at different times within the interval of a time step, yet this comes at the cost that values have to be update multiple times within a time step. We will use a time integration method that is a good trade-off between computation time, complexity and accuracy [MZ07]. The velocity Verlet algorithm has been shown to be suitable for MD simulations. Its numerical rounding errors are lower than comparable algorithms [GKZ07]. It updates velocity and position at the same time step. A similar stepping algorithm, the leapfrog method splits up these two and updates velocity and position with an offset of a half time step.

$$\vec{x}_{n+1} = \vec{x}_n + dt \cdot \vec{v}_n + dt^2 \frac{\vec{F}_n}{2 \cdot m} \tag{1.2}$$

Equation 1.2 shows the calculation of the position. The new position is based on the old position, the velocity known from the previous time step and an Euler integrated velocity.

$$\vec{v}_{n+1} = \vec{v}_n + dt \cdot \frac{\vec{F}_n + \vec{F}_{n+1}}{2 \cdot m} \tag{1.3}$$

Equation 1.3 shows the calculation of the velocity. It is calculated according to Newton's second law with the average force of the current and the previous force. This means we will also have to store the value of the previous force for this update.

$$\vec{F}_i = \sum_{j=1, i\neq j}^{\#particles} \frac{24 \cdot \epsilon}{\left\|\vec{x}_i - \vec{x}_j\right\|^2} \cdot \left(\left(\frac{\sigma}{\left\|\vec{x}_i - \vec{x}_j\right\|^2} \right)^6 - 2 \cdot \left(\frac{\sigma}{\left\|\vec{x}_i - \vec{x}_j\right\|^2} \right)^{12} \right) \cdot \left(\vec{x}_j - \vec{x}_i\right) \quad (1.4)$$

Equation 1.4 shows the calculation of the force of one particle

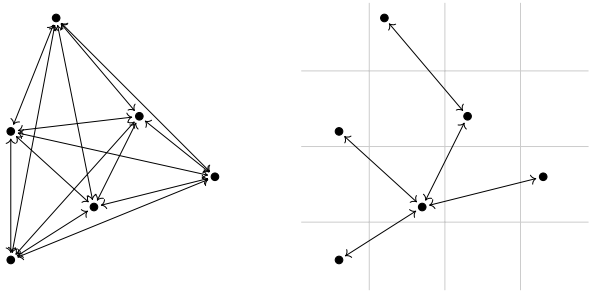
1.4 Domain Decomposition

The time complexity of $O(n^2)$ would not allow us to scale this simulation to a very high number of particles. The Lennard-Jones potential converges towards zero for longer distances. We could discard long range interactions as they are always dominated by shorter range forces. This means we cut off interactions between far distant particles. This does introduce an error. The error is smaller the larger this cut-off radius is. By setting a limit at which particles do not interact with each other, we only have to calculate forces of particle pairs that are at least as close as the defined cut-off radius. This approach still has the complexity of $O(n^2)$ as finding pairs of particles that are close by still requires us to iterate over all particles. Therefore we want to use a data structure that allows us to cluster and access particles that are nearby.

To do so, we define a regular lattice over the global domain shown in Figure 1.2a. This will slice the domain into smaller sub domains that contain particles with close positions that we will call cells. Each particle will be assigned to exactly one cell. With this approach we can access particles that are within certain, defined, known bounds in constant time. To now calculate the forces, we only have to iterate over the cells' particles and the cells neighbors particles that are within reach of the defined cut-off radius. When we set the cells length to be as long as the cut-off radius, the to iterated neighbors are only the adjacent neighboring cells (Figure 1.2b).

As we are modeling particles that have strong repulsive forces for near particles and weak attractive forces for longer range forces (compare the Lennard-Jones potential in Figure 1.1), the particles density is limited. A cell with its fixed volume will contain a limited amount of particles and thus be a constant in Landau notation. The overall time complexity drops to $O(n)$.

To ensure every particle is in its right cell, we need to update its assigned cell along with the position update. With the domain decomposition given, it is no longer required that the global domain has to be available for a simulation step update. A particle can be updated using only the neighboring cells particles. This allows us to spread the global domain with its particles across multiple processes that do not necessarily have to share memory. Those processes can solve the force, velocity and position update independently from each other. After an iteration step they need to exchange their



(a) Unstructured particles, no cutoff, $O(n^2)$ complexity

(b) Structured particles in lattice, $O(n)$ complexity

Figure 1.2: Introduction of Grid structure and cutoff radius decreases time complexity

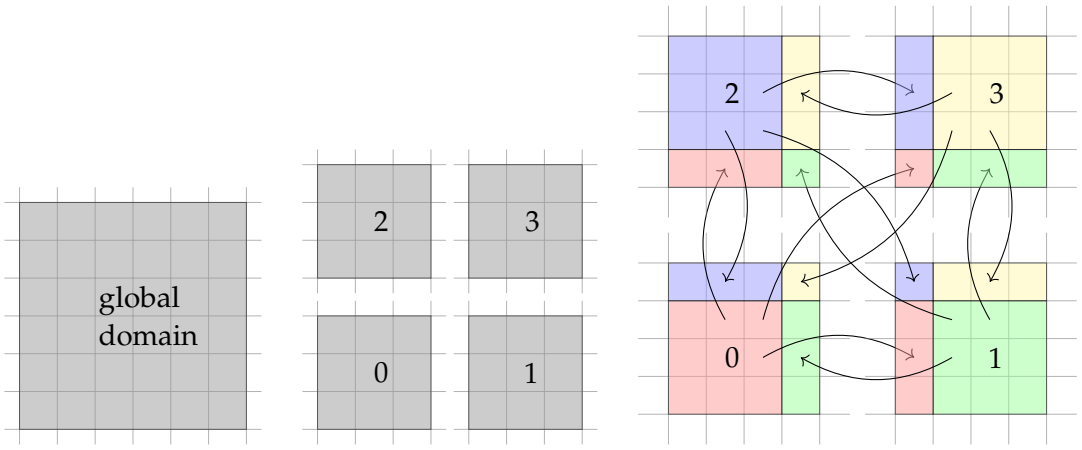
results with each other, as for the next step they again need to know about the updated position of their adjacent neighboring cells. A position update in one inner cell of a domain will have impact on the force calculation of another domain in case they are neighbors. Because of the cutoff radius a dependency of particles of cells is only given for adjacent cells. The process does not need to know all particles of its neighbors cells. Its only necessary to exchange the surface cells of the domain as shown in Figure 1.3c. These cells are a copy of the foreign surface cell layer called halo cells. These halo cells would only receive a part of the forces of the domain where it is stored, so they would drive apart from its original particle. This requires us to update the halo cells in every iteration step to keep them in sync (Figure 1.3c).

1.5 Simulation Algorithm

The molecular dynamics simulation that we will discuss and work on during this thesis is structured as follows.

1. We have a static number of particles, fixed time step.
2. Each particle has a position, a velocity and a force.
3. Particles interact with each other according to the Lennard-Jones potential.
4. Particle properties are updated with the presented formulas in Equation 1.2 (position), Equation 1.3 (velocities) and Equation 1.4 (forces).

Boundaries



(a) Entire global domain

(b) Splitting the domain and assigning it to multiple processes

(c) Adding halo boundaries and communication from inner cells to boundary

Figure 1.3: Splitting the domain

```
1 for (int i = 0; i < numIter; i++) {
2   domain->calculateX(in.delta_t);
3   domain->calculateF();
4   domain->calculateV(in.delta_t);
5 }
```

Listing 1.1: Simplified simulation main loop

The way the boundaries are treated can be used to model different borders. Possibilities are reflective, in- or outlets. We will use periodic boundaries. This will create a closed environment where particles cannot collide with anything but themselves and without external forces. The periodic connections for adjacent connections is shown in Figure 1.4. The global density remains the same. This should model a gas in a vacuum in zero gravity.

This also has an impact on needed computation and communication. Every particle also needs to be mirrored at the boundaries similar to how halo particles are exchanged. For the halo- and moving particles exchange we also need to adjust the position by the offset of the domain length at the mirrored axis.

In Figure 1.4 we can see the periodic connections of a sample domain with 8 cells. This is a three dimensional cube with each 6 neighbors (D3Q6), where only the direct

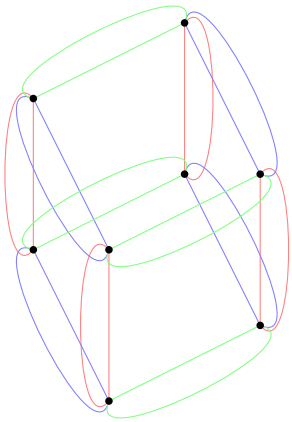


Figure 1.4: 3D Torus

adjacent surfaces are connected. We will use a D3Q26, where also all diagonals are connected. An domain pointing to its its 26 neighbors pointing is visualized in Figure 1.5.

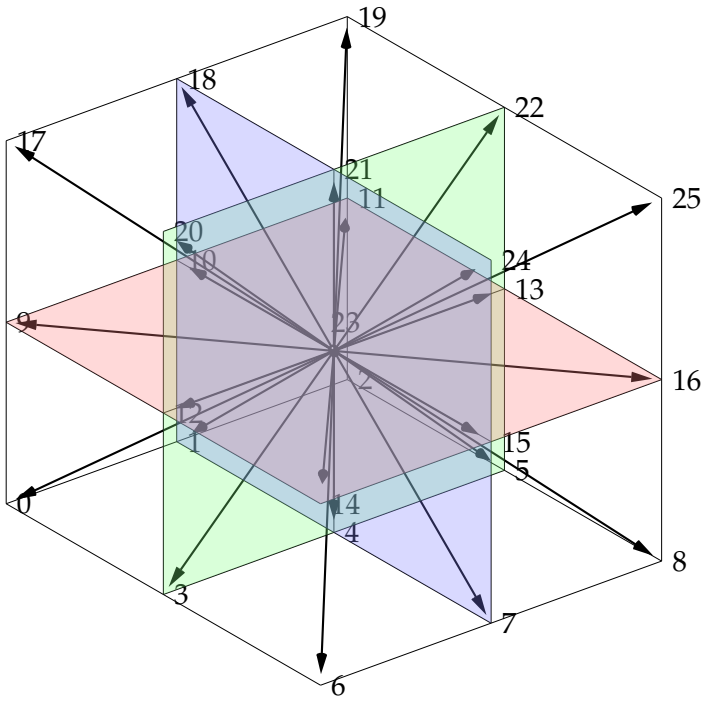


Figure 1.5: D3Q26

2 Message Passing Interface

MPI is the de-facto standard of writing parallel programs for distributed memory systems. The MPI standard itself is an interface definition that describes its semantics, but not its underlying implementation how each of the calls to MPI have to work. As a result of this, there are several MPI implementations that are independent of the software, operating system, programming language and hardware that is used to exchange data between participating processes. Open source implementations are MPICH and OpenMPI. MPI implementations are also provided by Hardware vendors such as Intel, IBM, HP and Mellanox. Those are tuned to utilize their underlying hardware. Even supercomputers get shipped with an optimized version of MPI to exploit the used hardware and network topology. For SuperMuc IBM developed an MPI implementation based on MPICH [1]. MPI is designed to be portable, hence a program can run on different hardware or MPI implementations. It is also possible to run a single MPI program on heterogeneous architectures without specific adjustments. MPI offers direct rank to rank and collective communication primitives. Due to the problem structure its communication pattern of our simulation, we will only use non-collective communication.

2.1 Two-Sided Communication

The traditional approach of communication is a two-sided model. Two processes directly exchange messages in a point to point (PTP) network. Every process within a communication group can send to and receive from every other, but perform only one of these actions at a time. The `MPI_Send` / `MPI_Recv` functions are available in blocking and a non-blocking (asynchronous, buffered) variant. The sending and receiving process both must issue their intents to exchange a message with the respectively other.

This means that for the communication step it has to be known upfront which processes will send to which processes at which time. This communication order has to be deterministic. If this is not given, an additional global communication step has to be performed before the actual exchange, in order to determine communication partners.

[1] https://www.lrz.de/services/software/parallel/mpi/ibmmpi/

This signalizing of the program's execution order will reduce parallel performance and complicate computation / communication overlapping [GHT15].

For static scenarios this programming model might be sufficient. On the other side dynamic applications, problems with unpredictable, irregular communication patterns are harder to implement with two-sided communication. Both target and origin needs to know that a message exchange is going to happen and both trigger communication calls .

2.2 One-Sided Communication

In the one-sided communication model only one process needs to become active in order to perform a data exchange. Additional the approach decouples communication and synchronization. The programmer has to take care of these separately. This makes one-sided communication programming more complex in the first step, but also allows new constructs for concurrent communication and overlapping computation / communication. Processes can receive (or passively send via a remote get request) without interruption. The MPI specification does not require remote direct memory access (RDMA) hardware for one-sided communication, yet it will highly increase performance if available and remote memory access (RMA) operations of MPI will be executed natively in hardware. One-sided communication can be emulated by point to point message passing. This means, that the performance depends on the internal implementation of the MPI library. Implementations might increase over their releases, because of incremental optimization [GHT15].

2.2.1 Introduction of MPI One-Sided Communication

MPI one-sided communication became part of the MPI specification in its second major version. Its early designs were criticized and generally rejected. Critique contained that its specification includes too many braking changes, incompatibility with developed tools, its API being too restrictive without filling an actual need [BD04]. Also its dynamic process management was incompatible with MPI's widely used batch processing systems on clusters. The official approved standard by the MPI forum was version 2.1 in 2008. Adoption of the specification to implementation took still very long. For many application 10 MPI methods are usually enough to implement their core message passing communication scheme. It has been criticized, that many specialized functions are not necessary for the majority of users. Because they are seldom optimized in the way the users application could use them, they perform poorly than using the basic functions. It took a long until one-sided MPI was implemented because multiple implementation steps had to be finished in sequence. First the new version has to

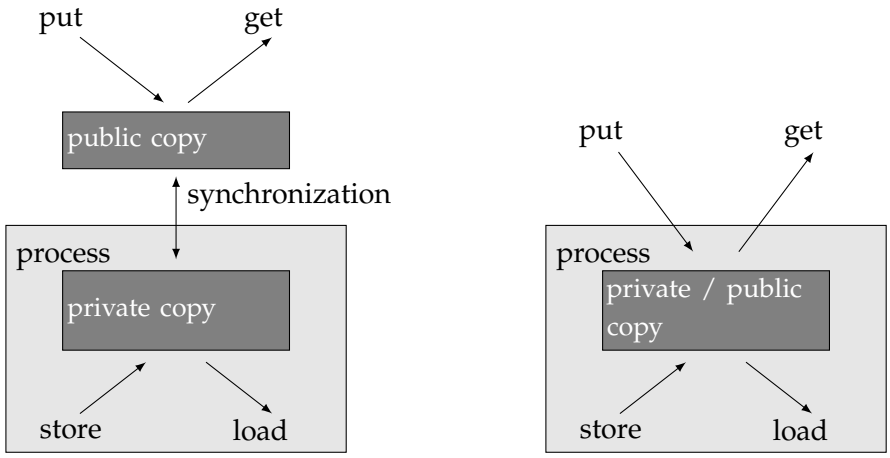
be implemented, then it can be adjusted for specialized hardware. Secondly large clusters have then to install and support this new version. Third, users have to learn a new semantic and applications have to be written using the new MPI specified functions. All of these steps slowed down adoption of one-sided communication. With version 3 of the MPI specification one-sided communication was revised. It introduced more specialized one-sided functions that should leverage RMA hardware to optimize performance. It also introduced non blocking collective operations [For15].

2.2.2 Window Creation

MPI communication structures its programming model into groups of processes in which they can communicate with each other directly. A process can be part of multiple groups. RMA support does not imply these. Other processes are not allowed to access and modify other processes memory by default. To allow RMA, each process must collectively expose a local memory region, a so called window. The program remains in control over its private memory, but additionally can declare memory as public. Communication can only happen within the same group. RMA operations are only allowed to exposed windows and those also only during certain time frames, called access epochs [GBH13].

The windows can be organized in one of two different memory models. Either a separated or a unified model. Those models think about declared windows as memory regions that can either be a public or private copy. A process usually has exclusive read and write access to its memory. In shared memory programming a process is forked into multiple threads that themselves have to organize the memory accesses in some way by using locks or atomic operations. In one-sided communication we might have support of RDMA, but we cannot rely on it. The public copy of a window is the copy that is accessible to other processes. Remote `MPI_PUT` and `MPI_GET` operation access this memory region. Local write and store operations are executed on the private memory. If public and private windows are logically identical then this corresponds to the unified memory model. Read and write updates to the unified memory are coherent.

One-sided communication can be seen in the same way as shared memory programming, where the exposed windows's memory is the shared memory. A modification of one process modification to a memory in an exposed memory should be visible to all other processes. This modification should be reflected in the same way as cache coherency protocols do in shared memory programming. A window can also be seen as an abstraction layer of memory. All windows together represent a global address space like memory region. We are only allowed to communicate to this layer with RMA calls.



(a) Separated model

(b) Splitting the domain and assigning them to multiple processes

Figure 2.1: Separated and unified memory model [GHT15]

The separated model should be used for compatibility reasons on systems that do not support RDMA. This model might be slower and also requires additional synchronization from the user program. As this decision is transparent to the user's program we cannot rely on the more advanced unified model.

Separated Separated means shared global memory is independent of local private memory, therefore MPI makes no assumptions about coherency supporting hardware. The users program is responsible to ensure coherency and synchronization in software. This relaxed model gives a lot of freedom to the MPI implementation, as it does not require specialized hardware to support MPI operations. This is done in order to be highly portable. Remote updates to a public exposed window may not be reflected directly on a private copy and the programmer has to synchronize these windows himself.

Unified In the unified memory model, MPI ensures coherency. Exposed windows can be seen as part of a cache hierarchy. The MPI specification explicitly encourages implementations to make use of underlying hardware that ensure ordering and coherence. Updates to a window is coherently reflected in both private and public windows, which is in case of underlying supported hardware. The unified, coherent model does not ensure consistency. If multiple processes write to the same remote memory region during an exposure epoch, order is not specified

by MPI. Even calls during one epoch are not ensured to be executed in the order they are issued. Without further synchronization or locks this could lead to inconsistent states of the windows.

The separated model is available in MPI since version 2.0. Version 3.0 introduced the unified memory model. Programming in a separated fashion will be upwards compatible. The unified model may be more convenient, but requires an MPI 3.0 implementation and hardware compatibility [For15].

The MPI 3.0 standard allows four different ways to expose memory windows. The initialization functions to create a window are collective calls. This means all of the processes within a group must call this function. Sizes of windows can vary across different processes. If a process does not have to expose memory, it still has to make the call, but can specify the size to be zero. The legacy MPI 2.0 `MPI_Win_create` function creates a window object on top of already allocated memory. Especially for a high number of processes this results in large translation buffers. It is recommended not to use this, but one of the three MPI 3.0 window creation functions:

`MPI_Win_allocate` allocates memory itself and tries to align the memory position across all processes.

`MPI_Win_create_dynamic` creates a window handle without allocated memory. Processes can attach memory to this window dynamically.

`MPI_Win_allocate_shared` creates a window object similar to `MPI_Win_create` but provides additional functionality for windows on local shared memory.

2.2.3 Communication

MPI offers three different communication operations. Basic operations like `MPI_Put` and `MPI_Get` that only involve one active process. The second type are more specialized accumulates, including atomic operations. All MPI one-sided communication methods are non-blocking. Thus their calls return immediately, the actual data transfer could be initiated but is not enforced by the specification. `MPI_Put` copies a memory buffer from an origin to a target window. `MPI_Get` specifies a remote memory location in a window from where to fetch the data.

```
1 MPI_Init(NULL, NULL);
2
3 int world_rank;
4 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
5
6 if (world_rank == 0) {
7     MPI_Send("Hello␣MPI\0", 10, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
```

```
 8 } else {
 9     char* buff = (char*) malloc(10);
10     MPI_Recv(buff, 10, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
11     cout << "received␣"<< buff;
12 }
13 MPI_Finalize();
```

Listing 2.1: MPI two-sided communication "Hello MPI" program

Listing 2.1 shows a minimal two-sided MPI "Hello MPI" example. It initializes MPI and sends the string "Hello MPI" from process 0 to process 1. Then prints out the received string. We do not need any synchronization as with the matching of `MPI_Send` and `MPI_Recv` we get implicit synchronization. It shows the strict matching of send and receive operations

```
 1 MPI_Init(NULL, NULL);
 2
 3 int world_rank;
 4 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 5
 6 char* buff;
 7 MPI_Win win;
 8
 9 MPI_Win_allocate(10, MPI_CHAR, MPI_INFO_NULL, MPI_COMM_WORLD, &buff, &
       win);
10
11 MPI_Win_fence(0, win);
12     if (world_rank == 0)
13         MPI_Put("Hello␣MPI\0", 10, MPI_CHAR, 1, 0, 10, MPI_CHAR, win);
14 MPI_Win_fence(0, win);
15
16 if (world_rank == 1)
17     cout << "received␣" << buff;
18 MPI_Finalize();
```

Listing 2.2: MPI one-sided communication "Hello MPI" program

The program in Listing 2.2 does the same as in Listing 2.1, but uses the one-sided communication model. `MPI_Win_fence` being a collective synchronization call has to be called by all processes, even process 1, which does not actively participate in the communication. The buffer in process one is undefined, as we never initialized it. We

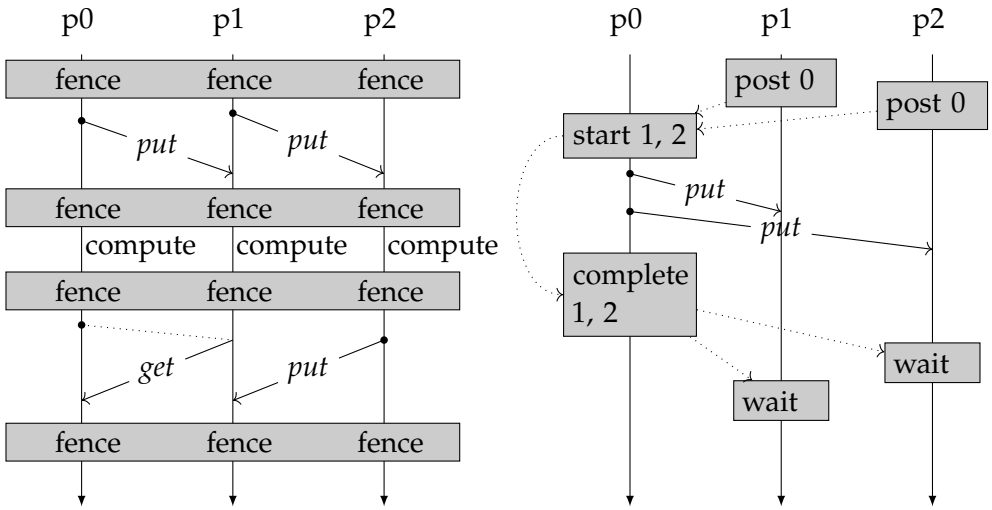
could also have set its size to zero bytes by using `10 * world_rank` as size definition in line 9.

2.2.4 Synchronization

MPI one-sided communication introduces the concept of access and exposure epochs to windows. All MPI operations to a remote window must happen during such an epoch and are not allowed outside of these epochs. To start and end an epoch the MPI program needs to call synchronization functions that indicate start and end of an epoch. These calls block until all necessary resources are available. During an epoch it is possible to arbitrarily access memory. There is no guarantee of ordering, atomicity or progress of communication operations given until the end of the epoch. In MPI we distinguish between two different principles of window synchronization: active and passive.

Active Target Synchronization Similar to MPI two sided communication both target and origin have to become active and collectively start an epoch to a window. One way to initiate an epoch is by using the fence synchronization, visualized in Figure 2.2a. All processes of a communicator that hold a window must start an epoch by calling `MPI_Win_fence`. It synchronizes similar to `MPI_Barrier`. Only once all processes have issued that call, RMA communication can start. After the end of an epoch it is guaranteed that all RMA accesses are completed and we can safely locally access window memory without intercepting ongoing remote memory access. The `MPI_Win_fence` approach does not define which processes will write or receive. Any process is free to perform any one-sided communication during the epoch. A sample program using the fence synchronization can be seen in Listing 2.3. This program will send a double value from rank 0 to rank 1.

Alternative to this approach is the post, start complete and wait (PSCW) mode. In this mode the synchronization defines which process will act as the origin and which as a target. The order of the calls is ensured to be PSCW. The target process has to call `MPI_Win_start` to expose its window, the origin process will call `MPI_Win_post` to imply an intend to perform operations on the remote window. After these synchronization points were bothreached, the origin process can perform its one-sided communication on this window. It will imply it is finished with a call to `MPI_Win_complete`. The target will call `MPI_Win_wait`. Similar to PTP communication those two calls always have to be called in pairs otherwise the program might deadlock. The last `MPI_Win_complete` and `MPI_Win_wait` calls block until the operation has been finished remotely. This is visualized in Figure 2.2b. The dotted arrows imply the ordering of the operations. PSCW



(a) Active target, fence synchronization (b) Active target, PSCW synchronization

Figure 2.2: Active target synchronization

synchronization is useful if only a subset of processes want to participate in the communication. A code sample using the PSCW mode can be seen in Listing 2.4.

Active target synchronization is best suited for regular access patterns like stencil boundary exchange. But it also degrades parallel performance, as all processes have to synchronize. On entering an epoch all processes have to wait for the slowest process to finish its local computation and when leaving all again wait to finish their respective RMA operations.

```
1 MPI_Win_fence(0, win);
2 if (world_rank == 0)
3   MPI_Put(&sendVal, 1, MPI_DOUBLE, targetPid, world_rank, 1,
        MPI_DOUBLE, win);
4 MPI_Win_fence(0, win);
```

Listing 2.3: Active target fence synchronization

```
1 if (world_rank == 0) {
2   MPI_Win_start(comm_group, 0, win); // access
3   MPI_Put(&sendVal, 1, MPI_DOUBLE, targetPid, world_rank, 1,
        MPI_DOUBLE, win);
4   MPI_Win_complete(win);
5 } else {
```

```
6    MPI_Win_post(comm_group, 0, win); // exposure
7    MPI_Win_wait(win);
8  }
```

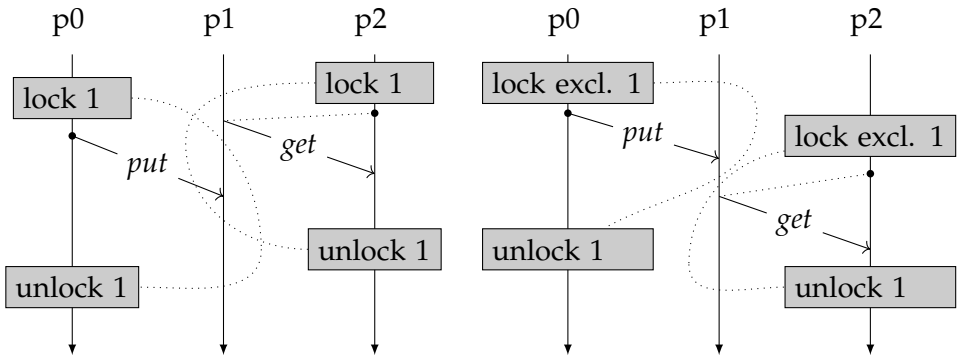
Listing 2.4: Active target PSCW synchronization

Passive Target Synchronization In passive target synchronization only the origin becomes active and accesses the target passively. It has to open an exposure epoch to a target. It does so by calling `MPI_Win_lock` / `MPI_Win_unlock` with passing the target window as the argument. A sample program is shown in Listing 2.5. The target process will not be notified about the foreign access, nor does it have to acknowledge the access. The lock can either be declared shared or exclusive. An exclusive epoch is only provided to a single process. In an exclusive epoch RMAs are protected against interfering with other remote accesses. The accesses are executed transaction, atomic, uninterrupted, yet in undefined order. In Figure 2.3a we can see a passive target synchronized put and get operation from processes 0 and 2. Because MPI does not ensure any ordering the time when those operations can happen does overlap between both processes (indicated by the dotted areas). Those operations can happen in arbitrary order. In case the put is a larger modification, the get operation on this window could see a partial modified state.

In case those overlapping operations can not be avoided by the program, an exclusive lock needs to be acquired. This is shown in Figure 2.3b. While the dotted areas do not overlap anymore, the order is still undefined. We can be sure a operation either has not started yet or is entirely completed. With a shared lock multiple processes can access the same window at the same time. `MPI_Win_lock` does not actually acquire a lock in a shared memory way and does not block. It starts an epoch, buffers the accesses and executes them whenever the lock can be provided. The only synchronization point in passive target synchronization is the `MP_Win_unlock`. At this point it is ensured that the RMA calls have finished, remotely and locally.

```
1  if (world_rank == 0) {
2    MPI_Win_lock(MPI_LOCK_SHARED, 1, 0, win);
3    MPI_Put(&sendVal, 1, MPI_DOUBLE, targetPid, world_rank, 1,
         MPI_DOUBLE, win);
4    MPI_Win_flush_all(win); // local and remote completion
5  }
```

Listing 2.5: Passive target mode, with local completion



(a) Passive target, shared locks (b) Passive target, exclusive locks

Figure 2.3: Passive target synchronization

MPI cannot guarantee that two remote processes write to the same window during the same synchronization epoch. In MPI 2.0 such overlapping accesses were declared as erroneous behavior. This meant it was up to the MPI implementation to handle this case. It could not only be the case that the program continues with undefined memory, but it could even lead to a termination of the program. This was changed in MPI 3.0 to the more strict "undefined" outcome. The programmer must ensure, that local and remote read / writes do not overlap unsynchronized.

Therefore the programmer needs to add additional synchronization during an epoch.

Synchronization can also be done by MPI two-sided communication calls like zero bytes messages with `MPI_Send` or a global `MPI_barrier`. This is possible and will achieve the desired effect, it might have a slowdown over more specialized functions.

MPI 3.0 introduced one-sided synchronization functions for this case [For15].

`MPI_Win_sync` synchronizes local and remote memory of a window of a process. `MPI_Win_flush` completes the local outstanding communication of a window to a process. Both of these functions are also available as an "all" variation, which execute the synchronization of the window to all processes. Note, that a similar effect could be achieved by closing and opening an epoch as the more specialized functions should perform better.

2.2.5 RDMA Mapping

MPI one-sided communication was designed directly to map to remote direct memory access (RDMA) networks. One-sided communication should profit from RDMA's low latency and high bandwidth. The MPI one-sided communications interface specification

is meant to be fully supported by architectures without RDMA support (gracefully degradation / progressive enhancement). RDMA can establish a socket connection without the network stack of the operation system, this is called operation system bypass.

2.3 Reasons for MPI One-Sided Communication

One sided communication can speed up our molecular dynamics simulations due to several reasons:

- RDMA mapping (low latency, high bandwidth, shallow abstraction / hardware near)
- Operation system bypass
- Simplified programming model (no tag-matching)
- No dependency on remote process to allow process skew
- Less intermediate copies

3 Implementation

For the implementation of the MD simulation we started with a given MD simulation program provided by Julian Spahl [Spa16]. He researched and implemented multiple different halo region schemes as his bachelor's thesis. The focus of our work is communication, therefore we removed alternative halo regions and kept one, the full shell approach. Additionally, we simplified the application to its purest core functionality. From there on we optimized hot code regions and applied general C coding optimizations [Fog17]. This minimized the overall execution time and set a basis for scaling the simulation to many cores.

In this chapter we will present the implementation adjustments we did in order to minimize execution time, memory consumption, maximized parallel efficiency and introduced one-sided communication.

3.1 General Performance Optimizations

Optimizing the single core performance is important. In ensures we are starting with an efficient basis that does not value artificial parallel efficiency over total execution time.

While the optimizations are not directly targeting one-sided communication, it ensures that the ratio of computation to communication strengthens the communication part by decreasing the time spent computing. This way later alternations to communication methods have a higher impact.

3.1.1 Vectorization

The data structure of the particles in the simulation is of fundamental importance. It defines the way we iterate and which blocks of memory have to be fetched from main memory into caches. The initial solution was an array of structs (AOS) data structure. This implementation packs every particle in its own struct, as visualized in Figure 3.1, a sample of the AOS definition in the code can be seen in Listing 3.1. While all the information of a single particle is in a contiguous memory, this is not required and leads to an undesired performance penalty, while iterating over only certain attributes

Figure 3.1: Array of structs (AOS) memory structure

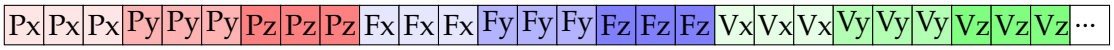


Figure 3.2: Structure of arrays (SOA) memory structure

of the particles. Therefore we switched from the array of structs (AOS) to a struct of arrays (SOA) data structure.

The SOA uses one large vector for every attribute component of all particles. Every vector is now split into three contiguous blocks of memory of the size of the number of particles within one cell. SOA uses the available cache more efficient as only the accessed attributes must be fetched from main memory. Attributes of different particles are locally stored close in memory. This is visualized in Figure 3.2 and sample code of the declaration of a cell can be seen in Listing 3.2. As an example, the update of the forces in the simulation accesses only the position and forces of the particles. The calculation of the velocity has to access the old velocity, current and old-force. The AOS approach had the downside that all attributes will be loaded into cache as they always will reside in the same cache line and therefore particles will be loaded entirely independent of whether the information was accessed or not.

```
1  class Cell {
2  public:
3      int type;
4      std::vector<Particle> particles;
5      ...
6  }
7  class Particle {
8  public:
9      utils::Vector<double, 3> pos, f, vel, old_f;
10     ...
11 }
```

Listing 3.1: AOS data structure of the cell before

```
1  class Cell {
2  public:
```

```
3     double *posX, *posY, *posZ;
4     double *fX, *fY, *fZ;
5     double *velX, *velY, *velZ;
6     double *old_fX, *old_fY, *old_fZ;
7     ...
8 }
```

Listing 3.2: SOA data structure of the cell after

Additionally to that, SOA was chosen as memory model in order to use vectorization. A vectorized code can execute the single instruction on multiple data (SIMD) at the same time. This requires that memory is accessed continuously and offsets are predictable.

Vectorization is usually used by the compiler itself when it detects code that allows for vectorization. It might not be necessary to alter the program, but compilers work with the best effort principle. Additional directives can give hints to the compiler to help it in oder to apply the most suitable transformation. In case the CPU supports the Streaming SIMD Extension two double precision floating operations can be executed simultaneously. Newer CPU architectures extend the vector registers size which allows to compute more data in in one operation to 256 bytes for the Advanced Vector Extensions (AVX) and 512 bytes for the AVX512.

Applying vectorization to an SOA structure may sound trivial, as the developer himself might not need to alter its code. Vectorization is a complex compiler optimization, thus compilers may or may not vectorize the code properly. Compilers can generate optimization reports in order to give the developer feedback about how the compiler optimized his code. Some compilers might report a successful optimized vectorization of a loop even if it did not do it properly. The compiler optimized code might be different from what the programmer expected the compiler to do. Compilers sometimes just use vector registers instead of the basic ones, creating a misleading report without a positive performance impact.

To get the vectorization successful we need to modify the code to make the optimizations easier and more obvious to the compiler. One of those optimizations is to align memory. Memory must be allocated at an address that is a multiple of 16 (SSE) or 32 (AVX) bytes. This is necessary in order to always fetch an entire cache line for one set of instructions without offsets. This aligning can be done by vendor specific memory allocation functions or manually by overallocation. If we use the overallocation approach, we shift the array pointer to the first offset that fulfills the requirement. Having aligned memory is not sufficient, we also need to tell the compiler before every critical region that our arrays are memory aligned by 16 or 32 bytes. This is done by method calls such as `__builtin_assume_aligned (arg, 16);` for the gnu c compiler or `__assume_aligned(arg, 16);` in case the Intel c compiler is used.

In some cases compilers might claim dependencies between variables inside the loop body that prevent the vectorization. In this case we need to ensure that there is indeed no dependency and tell the compiler. This assumed dependency can be overwritten by adding the preprocessor pragma `#pragma simd` in front of the loop.

After applying those modifications we were able to verify that our code was successfully vectorized by checking, that the compilers vectorization report does properly vectorize, by checking the assembly instruction contained packed vectorization op codes and by verifying that the execution time decreased.

3.1.2 Indices Table Lookup

In the simulation code we have different groups of cell types. The most important ones are innerCells, ownCells and haloCells as shown in Figure 3.3. Over all of these groups we have to iterate at some point in the simulation. These indices are fixed during the runtime. So we can create an array with the indices at the initialization phase of the program. When we iterate over the cells, we can iterate over those arrays and use them as a lookup table for the cells we need to iterate over. This is a convenient approach to implement as shown in listing 3.3. The code is simple and clean.

```
1 int cellId;
2 for (int i = 0; i < numOwnCells; i++) {
3   cellId = ownCells[i];
4   cells[cellId]->calculateVelocity();
5 }
```

Listing 3.3: example of a precalculated lookup table for the velocity calculation

But this introduces an indirection. Before we actually access the cell we need to know its cellId. We have to read and iterate over the lookup table in every single iteration step. It takes away precious cache space and does not allow the prediction of the access of the cellId. Even for the case that the called method fills the cache, the ownCells array will be evicted from the cache and has to be re-read from the main memory for every cell iterate over. This causes unnecessary memory usage, cache misses and non predictable branches [Fog17]. The iteration order is obscure by just looking on the calling code.

In many cases the iteration cell id scheme is continuous. During the iteration we only have to jump at the end of the axes by an offset. When we calculate those indices directly in the code where they are needed we can achieve higher performance due to less memory as we do not have to store the lookup table and less cache-misses. The index is already in cache and does not have to be fetched from main memory.

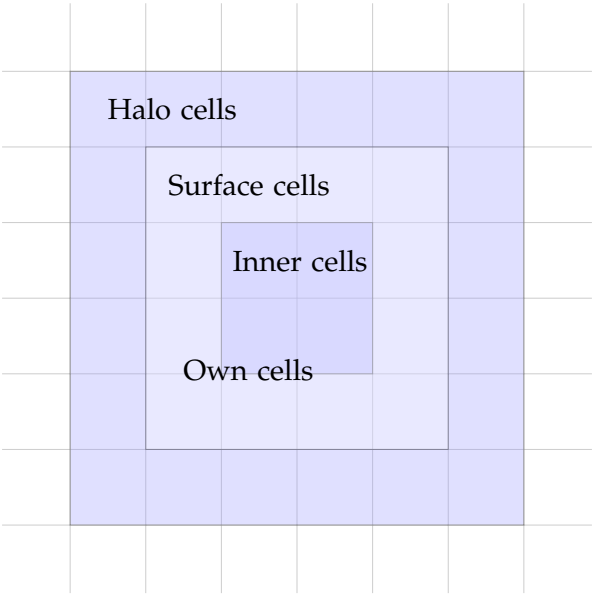


Figure 3.3: Layer naming of a domain

This comes at a cost of more code and more necessary computation. While computation is cheap the additional code is a penalty we are willing to pay for additional performance. The code to directly calculate the cellIds for the ownCells (entire domain without the halo boundary) is shown in Listing 3.4.

```
1  int cid = getInnerCellIdx(0, 0, 0);
2  for (int z = 0; z < domainSize[2]; z++) {
3    for (int y = 0; y < domainSize[1]; y++) {
4      for (int x = 0; x < domainSize[0]; x++) {
5        ...
6        cid++;
7      }
8      cid += factor_y - domainSize[0];
9    }
10   cid += factor_z - domainSize[1] * factor_y;
11 }
```

Listing 3.4: Example of a directly calculated cell ids of all ownCells

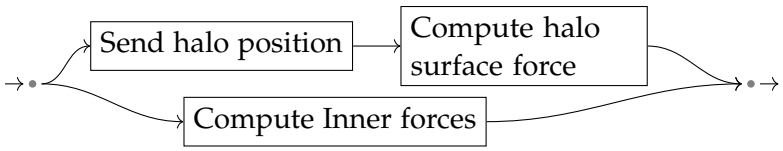


Figure 3.4: Concurrency graph of full-shell method

3.1.3 Boundary Force Exchange

For the halo exchange we can consider a variety of options. A process is responsible to compute the forces within its own cells. In order to do so it needs to accumulate all forces from within itself and its 26 adjacent neighbors. The naive approach is to send all neighboring halo cells particles to that process. This is called the full-shell method. This allow the process to compute all of the forces locally. A concurrency graph of this method can be seen in Figure 3.4. The advantage of this approach is that the forces can be computed in one step after the sending of the halo's particles positions has been finished. The inner cells can even be computed concurrently while data is being exchanged. Overall we only need one communication step. This comes at the cost of having to compute the forces of halo to own cells twice on different processes. With the computation of the forces being the most expensive processing operation we want to avoid this redundant calculation.

This redundant work increases with the number of processes that are assigned to the fixed problem. The ratio of halo cells to inner cells increases with every split, as the number of inner cells remain globally fixed, but we add halo regions between the surfaces of the split of the domains.

In the inner cells, we only have to iterate in one distinct direction over the lattice of cells. With Newtons third law applied every calculation of forces will apply its result to both participating particles. This means that we iterate over 13 of the 26 neighbors. We only have to ensure that the iteration scheme is distinct in such a way that we do not accidentally calculate the force on a pair of particles twice. For the iteration scheme we can go in any set of 13 directions as long as these do not also contain their inverse. This scheme is shown in a simplified 2d version in Figure 3.5.

We will use this iteration scheme and apply it to the halo region calculation. The concurrency graph of this approach can be seen in Figure 3.6. We define a set of 13 directions that do not contain their inverse directions. First, we send 13 directions of the current domains surface cells in the inverse direction. The receiving process can insert them as their halo region for these 13 directions. Each process will then calculate the forces starting from their own cells including the force calculation from the own cells to the received halo cells. The domain's neighbors will calculate the remaining

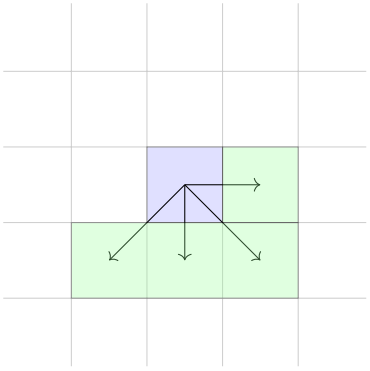


Figure 3.5: Iteration scheme in 2d

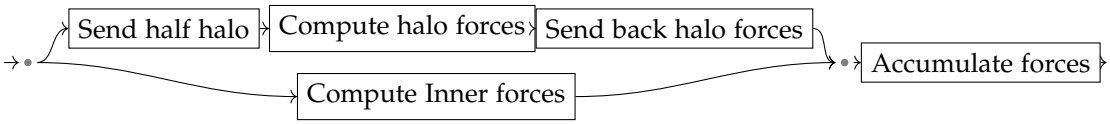
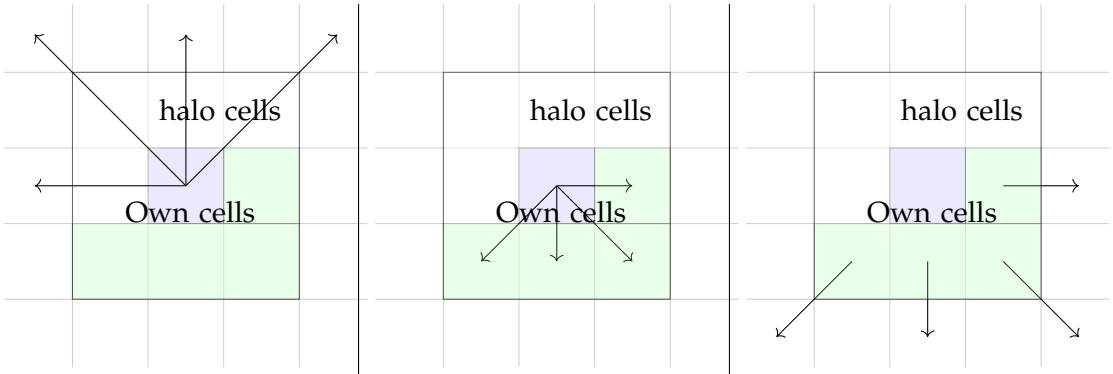


Figure 3.6: Concurrency graph of half-shell method

forces. Then the forces of the halo region can be send back to their origin and added up to the final force. This approach is called the half-shell method as only half the halo particles are exchanged. This splitting trades the redundant calculation with a second communication and synchronization step. In the first approach we send a vector for all particles in surface cells. In the second approach we send a vector of positions for only half the amount of particles of these surface cells, but we will also send one vector of forces back for every particle being transferred in the first step. So for every particle on the surface of a domain we have to send one vector of values. The overall data that is send over the network is in total the same for both variants.

Which of those variations should be favored depends on the input data and the number of executing processes. In case the synchronization is more expensive than the computation of the halo forces the full-shell method should be preferred over the half-shell approach. We try to avoid this case by not scaling larger than suitable. The second approach should also be favored in order to have a more energy efficient implementation. Redundant computation adds unnecessary CPU cycles that consume power, in contrast waiting for communication to finish allows the CPU to idle. Our goal is to achieve a highly parallel and efficient implementation. This can only be accomplished if we do not perform computations that increase with the number of used processes.



Send particles of the surface cells in inverse direction. These will be received and inserted as halo particles (the green marked cells).

Calculate from the own-surface cells in the 13 directions and calculate the forces including their opposing ones.

Send the halo forces back. The neighbors will receive the forces as part of the forces they should add to their originally send surface cells. These are locally accumulated.

Figure 3.7: Sequence of the force exchange

3.1.4 Computation to Communication Ratio

In our molecular dynamics simulation we have a fixed number of scenarios that should reflect the overall usage of our simulation. In those given scenarios, the ratio of computation versus communication time is strongly in favor of the computation part if executed on a few processes (1-16). The computation takes the majority of the overall execution time. The ratio is based on the size of the local domain in comparison to the size of the halo region. By decreasing the local domain size we can reduce the necessary computation. This is the case for a given fixed size problem that we distribute to more processes. As we do not compute forces with halo regions twice but rather send computed forces to its neighbors, we have a globally fixed problem to solve while scaling up.

This mean we have a fixed amount of computation while the communication increases with the number of processes. With this approach we can scale towards a communication bound problem. A communication bound problem is given if a local process spends less time on the computation than on the communication. This can be achieved by either providing an initial small problem domain or using sufficiently many processes that the local domain decreases enough. Our benchmarked scenarios

will get introduced in Chapter 5.

Increasing the number of processes for a given problem size has the disadvantage that due to the necessary synchronization between simulation steps the overall simulation is bound to the slowest process in every step. The globally slowest process has to finish before its neighbors can continue computing. The delay will propagate in the next synchronization steps and ultimately slow down all processes. Conclusively the entire time spent waiting will increase with an increasing number of processes. Therefore the important outline of the communication step is to provide a low synchronization latency, while bandwidth is less important.

The most extreme scaling scenario is that every process has only one own cell. It will have to send and receive from each 13 neighboring cells to share his cells as their halo cells, receive their own-cells and insert them as halo cells. In this case the process has to compute the forces for pairs of particles in its own cellc and forced between paris of own and halo calls. The computation to communication ratio is ideal $1 : 1$. In this case we reach peak performance, spending equal amount of time computing and communicating. Many of these properties are out of our control, but we can benchmark when a scenario reaches its peak performance or when it becomes impractical to add further processes.

3.2 MPI One-Sided Replacements

In this chapter we will discuss the changes introduced in order to transform the PTP communication in the MD simulation to a one-sided. In some cases multiple variations are viable. In this case we present both implementation alternatives, discuss their use case and will benchmark its impact on performance in either microbenchmarks Chapter 5 or results, Chapter 4.

3.2.1 Window Creation

In one-sided communication we need to allocate memory and expose it to remote processes. This is done by declaring memory blocks as windows. As stated in Section 2.2.2 this window can be declared in 3 different ways. With all these possible window creation methods the question remains when and how large a window should be. In our MD simulation, we can estimate the amount of data that has to be sent and received in every step. This estimate is based on the number of cells on a surface and the density of particles per cell. With this estimate we can allocate enough memory initially to make a reallocation of memory and the window exposure less likely.

A reallocation of memory is very expensive. Especially as the affected process cannot predict when this would be necessary. Only the origin, a foreign process that wants to

send, knows if a reallocation is necessary. Transferring the information whether the current window's size is large enough would require an additional communication step.

In case a process wants to write more data to the receiver than its current window size, it cannot notify the receiver processes, it should allocate more memory, because there is no more message exchange. The problem of the affected process not being able to tell if it has a too small window can also not be solved in case dynamic memory allocation is used. Additionally it would be necessary to inform all processes that there has been a reallocation of a window, because any process must be able to write to the newly allocated window. While in our MD simulation only adjacent neighbors will access the neighbors windows the windows are globally exposed to all processes.

This lead to our simplified solution that the window buffers are only initialized once and then kept static during the run time of the simulation. A simulation will terminate in case it might run into the case that it does not have enough memory to transfer a halo region. This could be extended by a special handling that the communication is split into multiple chunks and then send in multiple steps. We revoked this approach as all our test scenarios showed that such an in-equal distribution within our domain does not occur. It was sufficient to allocate a buffer of the size that was derived from the initialization parameters. The size of the buffer is double the size of the largest surface with the overall highest density of particles in one cell.

To expose memory for communication we can either can use one region for every send and receive direction (26 in total) or we will use one large buffer that is being shared for all directions. In the latter case we have to ensure not to overlap writes of multiple origins into the window. This means we must use parts of these buffers exclusively to one direction each.

One window per region gives us more control over the granularity and can lead to a cleaner code as we differentiate directions on the code level in the implementation. It could also be favored to have multiple buffers of different sizes as the number of cells on the surface varies.

The surface areas directly adjacent to the neighbors hold the most particles while the edges are just one single cell. The number of cells on the surface directly translate to the amount of data to be transferred. The different needed sizes for each direction could be implemented with one large window and multiple offsets that indicate the start address of the memory region in that particular direction.

We implemented both variants and measured their performance in section 5.5.

3.2.2 Synchronization

We want to implement a high performance, scaling molecular dynamics simulation that still shows a high parallel efficiency on a high number of CPUs. For a fixed problem set, increasing the number of CPUs, the local computation and memory usage decreases, while communication increases. The time used for communication will at some point surpass the computation time with an increasing number of processes. In the MD simulation, there are two points in time, where the processes need to be synchronized.

When recalculating the position and inserting them into their respective cell, particles can cross boundaries of a local domain and have to be transferred to a different cell. As this domain is then responsible for storing and updating the particle all particle information must be exchanged. A second time, when the force calculation requires to have the inner cells of their neighbor domains as their halo cells. This is necessary to calculate the forces at the surface cells as they are dependent on their neighbors cells particles. The force between two particles only bases on their distance vector. Thus only positions must be communicated to the halo region.

After both of these communication steps the local process must be able to detect it has received the entire information before it proceed with its calculation. As we are computing a periodic scenario this synchronization propagates through the entire global domain. This has the effect that the simulation can only be as fast as its slowest CPU.

The PTP implementation uses the `MPI_Irecv` and `MPI_Isend` directives, which are non blocking asynchronous calls. These are matched and resolved vice versa by MPI based on a tag and their rank. This matching is easy to implement for our simulation as neighbors are known and static. The communication is homogeneous. The synchronization happens on a `MPI_Waitall` call that blocks until MPI can ensure all 26 neighbor messages are received and the own issued send calls are finished remotely.

In MPI one-sided communication we have two synchronization modes available as stated in Section 2.2.4. Active and passive target synchronization. In active target synchronization both processes, origin and target, have to become active and therefore know about the epoch in which data exchange happens. In the passive synchronization model only the origin process know about the communication progress. The target neither knows about the progress nor that an exchange has happened at all. The program must use additional synchronization to ensure the processes are notified about data transfers. This separates synchronization and communication further than desired for our MD simulation.

In one-sided communication we write to and read from the windows. A process will perform local computation on those memory windows. This means that we cannot send memory to this memory region as long as the process is still busy computing

and accessing the memory. In Figure 3.6 we show a concurrency graph of the force calculation. Here we have to ensure that for every send operation we also successfully finish the corresponding receive operation from our neighbors, because otherwise we would risk corrupting the data integrity. Sending must not start before the foreign process has finished accessing the critical memory region. The local process must not continue computing with the next batch until the data exchange has been finished entirely.

3.2.3 No Intermediate Buffering

Another inherent advantage of SOA with RDMA support is that we are able to access transferred memory directly without intermediate buffers. No buffers have to be packed, neither internally by the MPI library nor the application needs to modify the buffers in order to use them. In contrast to immediate sending of point to point communication, the MPI implementation must not allocate and copy an internal buffer to allow immediate send and receive operations. Those are necessary to allow a local modification of the send memory address while the data is being transmitted. One-sided communication does not have to save a copy of the send buffers, as they do not guarantee to actually have processed the operation until one of the synchronization primitives is called.

In case RDMA is natively supported and used, a communication call will issue a request to a work queue. This queue provides work to the network interface communicator (NIC) which will perform the communication in the hardware. The NIC will send the state of the memory at the point in time when the data transfer actually happens. This does not necessarily have to be the same as at the time when the call is issued. Only on local completion the buffer can be safely reused. It is the applications responsibility to ensure correctness that the issued sending buffer is not modified during the sending progress.

The SOA data structure provides an additional benefit. During different communication steps we are only interested in certain parameters of the available information that is being stored for each individual particle. The SOA structure allows us to access those parameters independently from other parameters in contiguous memory. For the calculation of the halo forces we need the halos' particles position. The next step involves sending the calculated halo-forces back to its origin. Those all profit of the SOA structure as the data does not have to be repacked when sent. The memory address of these memory blocks is static during one simulation step. Therefore, in comparison to immediate point to point communication we use fewer intermediate buffers. Point to point communication requires at least one additional buffer on the senders side. Depending on the implementation of the MPI library there might be

an additional buffer on the receivers side that the users application does not see. The receiver can still use the issued and created buffer that he provided to an asynchronous `MPI_Irecv` in the same way was one-sided communication.

The disadvantage of the SOA structure is at sending the leaving particles from one cell to another. In this case we need to gather memory from many different places. For every vector we have to get every single component from a distinct address. This is bad as it triggers multiple random accesses that result in many cache misses. The data has to be gathered, send and scattered again. Here the AOS structure would be beneficial, because we need to access and move the particle as a single object. We must transform the data structure from SOA to AOS at moving the particles between cells.

In the beneficial case of SOA, when either sending only the position or the forces, we have to send three components of these vectors. As they are each in contiguous memory blocks we have to only pass the pointer to an `MPI_Put` call. The benefit of not having to repack the data is also helpful on the receivers side. Because the structure is identical for sending and receiving, we do not have to modify it on either side.

The positions and forces are vectors, where we store every component separately. This allows us to either trigger three `MPI_Put` calls or we can copy the three buffers into a larger local buffer via `memcpy` and then only issue a single call to `MPI_PUT`.

3.2.4 MPI Status Flags

The MPI specification offers to pass a status information to the synchronization routines. The flags can be used to optimize the way data is send and synchronized. Those flags represent a hint from the application that specific memory accesses were not executed before or after the synchronization. The MPI library then does not need to check certain edge cases in order to speed up the synchronization.

- MPI_MODE_NOCHECK The matching synchronization method was already called or is implicitly given by the structure of the program. No handshake will be performed to check for the corresponding matching synchronization call on the target process.
- MPI_MODE_NOSTORE The window was not modified locally
- MPI_MODE_NOPUT There will be no remote modification to this window (via. `MPI_Put` or accumulate functions)
- MPI_MODE_NOPRECEDE There were no RMA calls to this window preceding this synchronization call

- MPI_MODE_NOSUCCEED The synchronization call does not start an epoch of locally executed RMA calls.

If the program violates any of those assertions, the operation is assumed erroneous and it is up to the library to either abort the application or continue running with an eventual invalid result. This could lead to bugs that are poorly traceable in case the application does not abort or report the erroneous usage, but continues with incorrectly synchronized memory [For15].

4 Micro Benchmarks

It is difficult to predict performance with a new technology without prior experience or reports about it. Performance has many aspects, especially for a technology whose performance might vary drastically depending on factors such as used hardware, library implementation or version. Therefore we want to collect and present information from isolated micro benchmarks. These should give insight about where optimization or trade-offs can be made. This should help us use communication and synchronization calls in the most efficient way. We will create benchmarks that reflect key features of our simulation. For this we will not focus on the raw performance but try to adapt common benchmarks in a way that they reflect our simulation most closely.

We will have at the following metrics:

1. round trip time,
2. synchronization,
3. bandwidth,
4. asynchronous completion,
5. libraries, versions and runtime flags.

To test these benchmarks in isolation, it is easier to measure key aspects of the program. This way the measured numbers will not interfere with unintended side effects of the simulation. Microbenchmarks will allow us to create multiple variations of a similar approach without adjusting a wider scope of the simulation.

The benchamrs will be executed on Super SuperMuc phase 2. It uses 14 core Xeon processor E5-2697 v3. As default we use the IBM MPI implementation and the Intel C compiler.

4.1 Round Trip

The round trip benchmark should tell us how long it takes to send a signal from one process to a neighbor and to then receive a response.

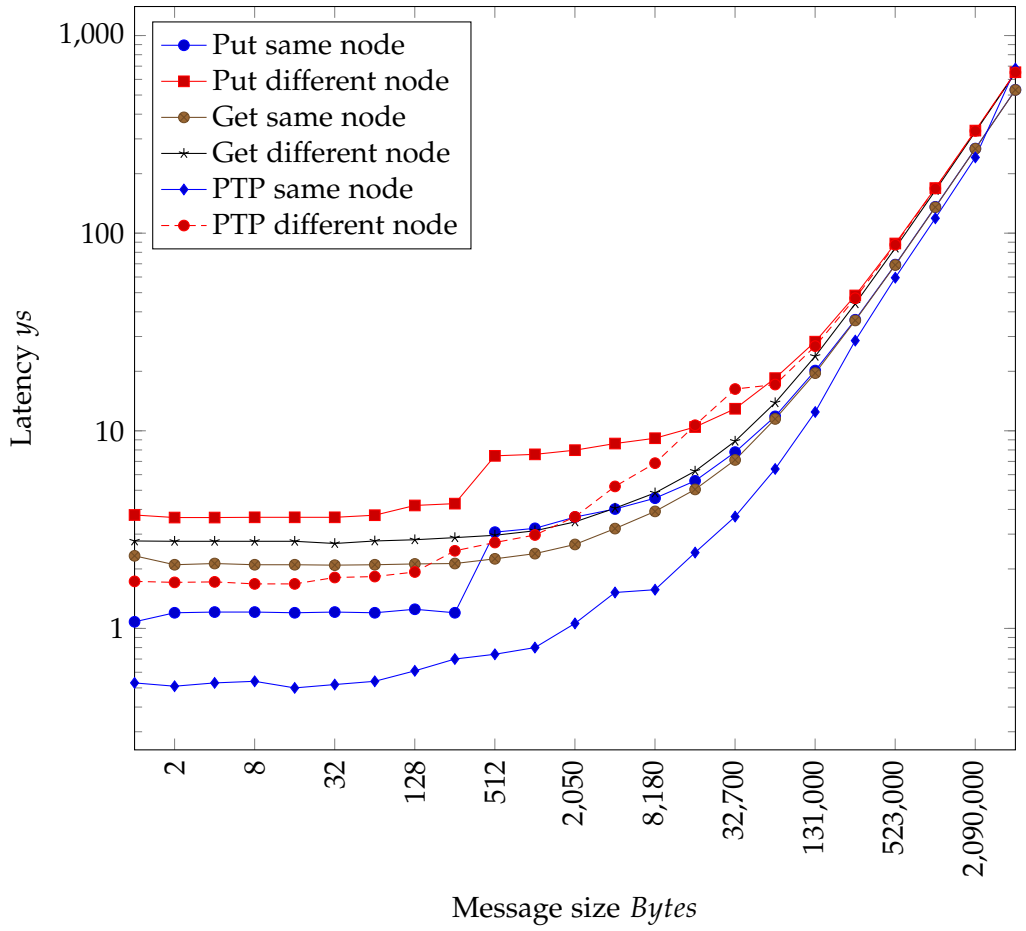


Figure 4.1: Latency micro benchmark with two processes

Figure 4.1 shows the measured time required to send N bytes between two nodes by using different method calls. We use `MPI_Get`, `MPI_Put` and `MPI_Send` and `MPI_Recv`. The two communicating ranks were executed on the same local node, as well as different nodes.

Overall PTP communication is faster than one-sided communication. A get operation is faster than a put operation. This is unexpected as a get operation has to perform two round trips until the data can be received. One-sided communication takes no time for zero byte messages. In this case, MPI does not actually transfer any messages. In case no data is being exchanged, one-sided communication, other than two-sided, does not inherently synchronize.

Communication within the same local node is unsurprisingly always faster than the

same operation externally. The time difference between local and remote accesses is higher for a put operation than a get operation. All methods show a jump in latency between sending 512 to 1024 bytes for remote one-sided calls. This might indicate a switch of the internal used protocol. It also indicates that unlike the one sided model the MPI calls are not truly independent from the receivers side.

4.2 Synchronization

We will look at synchronization separately as it is decoupled from communication in the one-sided communication model. Sending and receiving is decoupled from the actual point where these memory exchanges take place. The MPI specification does not declare when this exchange has to happen. The only point in time when communication has to be completed are synchronization calls. At this point memory has to be consistent according to the MPI semantic based on the triggered preceding calls. There are two possibilities to ensure synchronization in the one sided model: passive and active target synchronization: either passive target synchronization or active target synchronization as stated in Chapter 2.2.4.

We will compare the synchronization benchmarks with PTP communication, which implicitly already include synchronization. This will show us how well one-sided communication can perform when having to re-merge the communication and computation step again.

Passive Target Synchronization has the disadvantage that the target process does not know about incoming data. Yet it has to known when it has the necessary data received to perform the local computation step. MPI does not provide a native notification primitive. Therefore we will have to implement a polling mechanism of the process in order to tell if a process is ready to receive and finally if the target process has received the entire data buffer. It is necessary to verify these two steps because a process must not override memory of a remote while it may still access the window's data of an earlier communication step. If we would not verify these synchronization steps, we would risk to get erroneous memory accesses and erroneous results. Further to this a process must not start to read data which it is receiving while the sending operation has not terminated or not yet started. In conclusion, a communication step therefore requires two distinct identifiable synchronization points.

Active Target Synchronization In Active target mode all processes participate in the synchronization. All processes have to have finished their previous work before an RMA epoch can either start or finish. This splits the overall execution time of

the program in epochs of computation and communication. While the processes are still able to compute during the communication step, they are should not modify memory regions that are used for RMA operations. This is a more suitable approach, as it ensures a synchronization between all processes steps and thus gives a clear separation of computation and communication. The disadvantage of this approach is, that the `MPI_Win_fence` call introduces a global synchronization twice for every data exchange, as we need to ensure all processes have reached the same progress in the simulation step. This global synchronization is more than necessary. It would be sufficient to synchronize a process with its local neighbors.

Custom Synchronization Additionally to the given synchronization primitives, we wanted to implement a custom synchronization mechanism. This custom synchronization could be faster than the regular one, as we do not have to synchronize only the direct neighbors.

To ensure this in the passive target mode we write the current iteration step on a predefined memory offset. This data should then be used as a flag by the receiving process to determine that a foreign process has completed sending data. This approach relies on the eventual consistency model of MPI's one-sided communication. We cannot be sure this data modification is already present in the foreign process just at the same time, because the communication could still be ongoing. We only know that the data exchange might eventual be reflected on the foreign process and at latest on a explicit synchronization barrier. We hope that most of the time, due to overlapping computation the data has already been retrieved.

In such approaches we cannot simply wait until the data has been sent. Tests have shown that for this case it is not possible to re-poll the memory location where the flag has been set. The memory does not update the remote window with the local window's data. This can be explained with the relaxed memory model of the MPI specification described in Chapter 2.2.2. These split models should be manually synchronized though a a call to `MPI_win_sync`. Yet this does not resolve the issue with not updating data in the local windows memory.

In order to still enforce progress, we perform a remote get request to the sender and query the state about local and remote completion. This leads to the necessity to have an additional communication and synchronization step that needs to be executed from the receivers side. This synchronization only needs to happen on the receivers rank as it can perform a get request to the sender and query for the needed data. This approach works, but scales very poorly. The number of times a process hits is required to fetch the flag on its own unsurprisingly increase with the number of processes. Even then a fetch on the remote memory indicates that the memory is not successfully finish. On average we could measure that processes did two round trips per iteration step

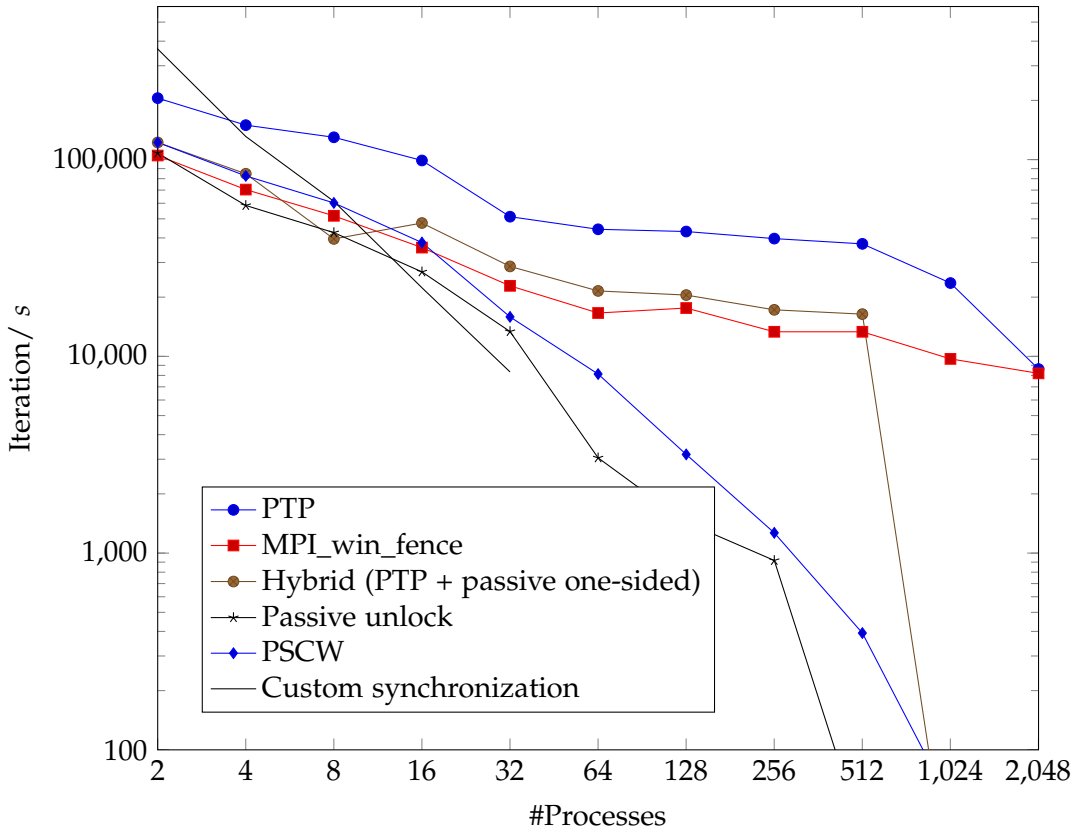


Figure 4.2: Synchronization comparison

rather than the optimum one. While we were able to replace MPI's expensive global synchronization mechanism with many independent local synchronizations its overall performed worse.

Figure 4.2 shows the iterations per second when using up to 2048 cores. The benchmark sends one double to 26 neighbors and then synchronizes. PTP synchronization with `MPI_Waitall` is the fastest. `MPI_Win_fence` scales similarly well, but is a factor slower, at 2048 process PTP and one-sided communication are almost up par. The hybrid approach of one-sided communication and PTP synchronization with `MPI_Barrier` is a little bit faster than `MPI_Win_fence` up to 512 processes, but then its performance drops by a significant amount which makes it unusable for any larger amounts of processes. The PSCW and passive `MPI_Win_lock` and unlock synchronization delivers similar performance as the other approaches for inner node synchronization. Here performance drops continuously with the number of processes. The custom synchronization approach starts very fast for very few processes. Its performance however,

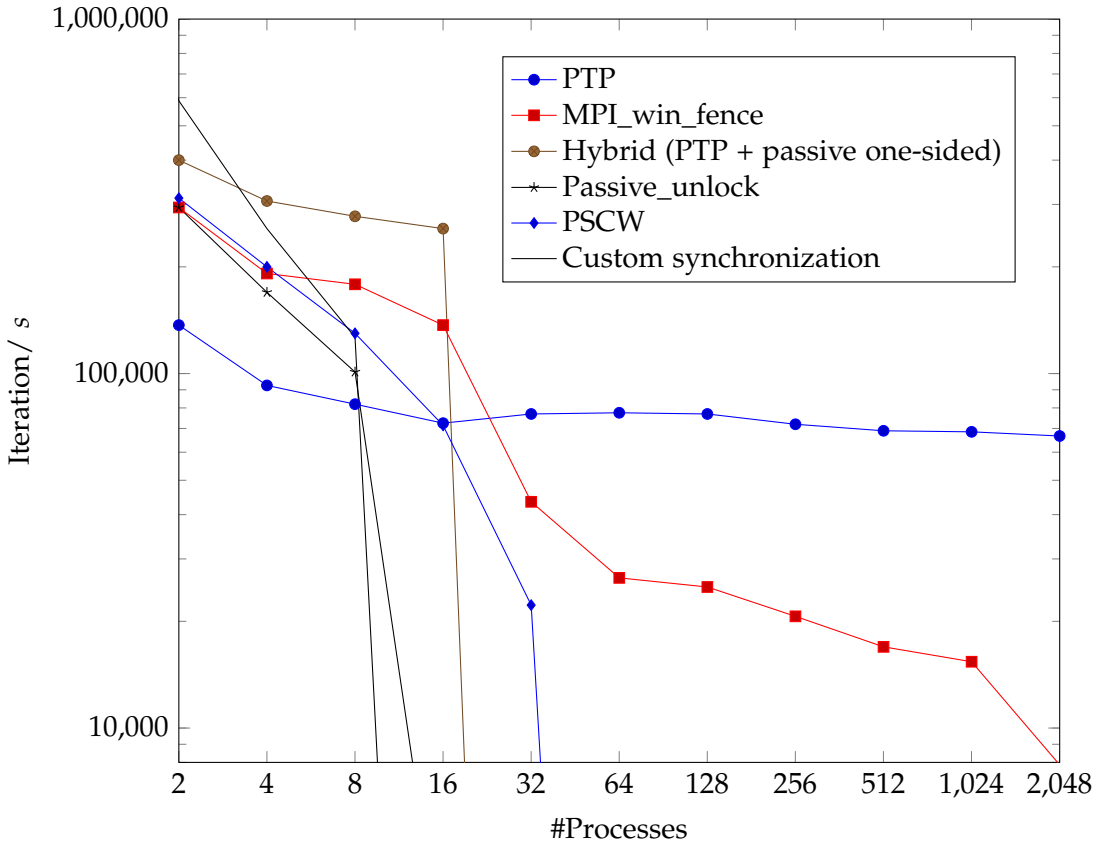


Figure 4.3: Synchronization comparison with environment flags MP_USE_BULK_XFER=no and MP_CSS_INTERRUPT=yes

degrades exponentially. It is possible to use it within two nodes, but for at least 64 processes progress stalls. This approach does not scale.

We ran the same benchmark again with the environment flags set to `MP_USE_BULK_XFER=no` and `MP_CSS_INTERRUPT=yes`. The results of this benchmark are shown in Figure 4.3. The performance for one-sided based synchronization rises and outperforms PTP synchronization within a single node. The performance degrades drastically for all one-sided approaches between multiple nodes when the flags are set. All, but the fence based approach becomes unusable at latest with 32 processes. `MPI_Win_fence` and PTP synchronization both profit from setting the environment flags. The PTP curve is much flatter with the flags set, at 2048 cores it performs almost 10 times the number of iterations.

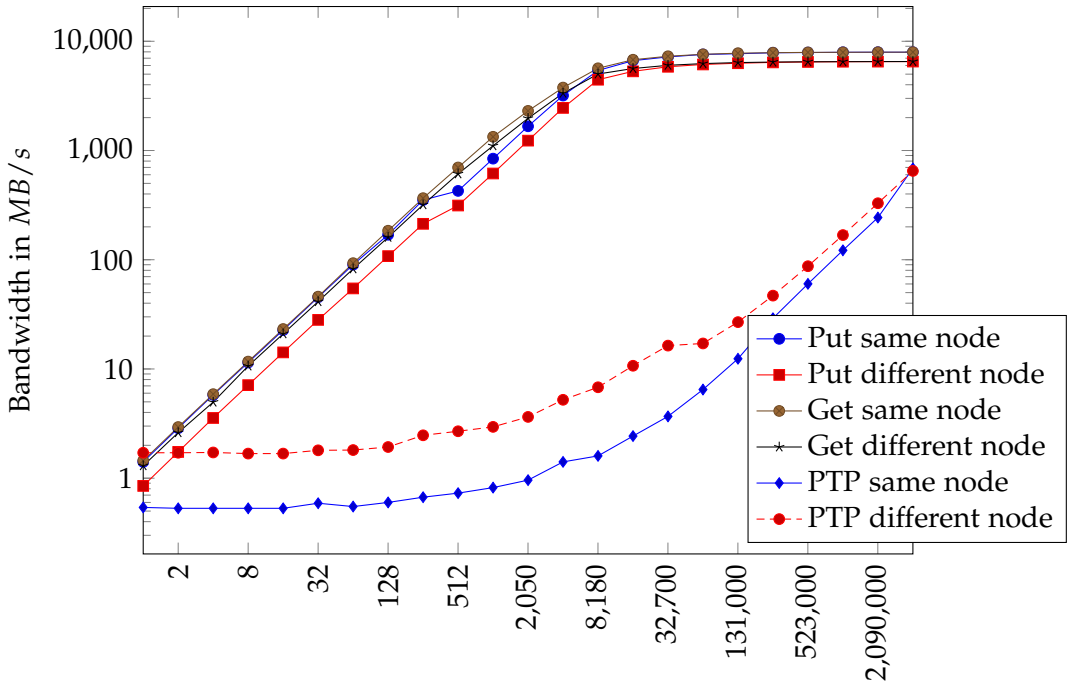


Figure 4.4: Bandwidth

4.3 Bandwidth

We want to compare the possible bandwidth between two nodes. For this we send a message from one process to the other. In the PTP variant `MPI_Isend` `MPI_Irecv` is used. For the one-sided implementation we use `MPI_Get` and `MPI_Put`. Here we are interested in only the data exchange so we use passive target synchronization.

In Figure 4.4 we see the measured bandwidth between two MPI processes executed on the same and different processors. One-sided communication is ahead of PTP communication. It does scale almost perfectly and approximates 8GB/s at message sizes larger than 8KB. One sided-communication does almost take not additional time for larger messages. Its latency is fixed but can send messages up to 4KB without additional slowdown. The get operation is a little faster than a put operation.

PTP communication shows a overall lower bandwidth. The bandwidth does only slowly increase with larger message sizes. Until message sizes of 2KB, the bandwidth does not exceed 1MB/s. After that the bandwidth does increase linearly with the message size, but does not reach the peak of one-sided communication.

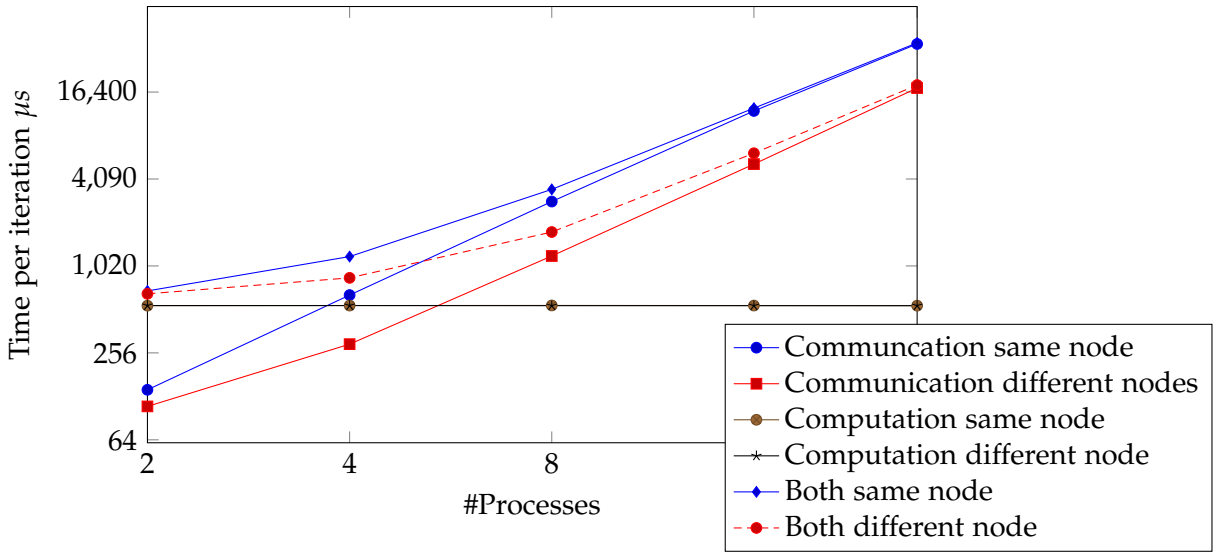


Figure 4.5: Computation - communication ratio

4.4 Computation Communication Ratio

In this benchmark we want to observe how much overlap between computation and communication can be achieve. MPI one-sided communication should ideally offload the communication and synchronization to specialized hardware and allow passive progression. In an ideal scenario, the network latency is shadowed by an ongoing local computation. The two operations take equally long and are executed parallel. The CPU can use all its cycles for the computation, while the network interconnect (NIC) handles the data exchange.

In order to measure the ratio computation and communication can be overlapped in one-sided communication we create a benchmark that measures this ratio. The benchmark consists of a loop that opens an MPI epoch, initiates a one-sided put, then computes and synchronizes the processes by closing the epoch with a call to `MPI_Win_fence` a second time. We will benchmark this program with some modifications to get the duration of the individual parts of the loop. Then we can compare the numbers to the total time it takes to perform both operations together. We will perform this benchmark on up to 28 cores. We will execute this on one node (2 processors with 14 cores each) and two nodes to see a difference between inner and inter node performance. We will send 100.000 doubles and perform 200 million additions per iteration step.

In Figure 4.5 we see the time it takes to finish the benchmark with soley communication or computation activated. We also report it takes time to execute these two together. The measured overlap is 0% as the time both operations need together is the sum of both individually. We could not achieve any overlap of communication and computation.

4.5 True Passive RMA Progress

One-sided communication promises to leverage RDMA passive progress. This should lead to asynchronous progress, which is key to overlap computation and communication phases. A process should be able to start, send and finish an one-sided operation even if the receiver is still busy. In Listing 4.1 we show a program of a sending and receiving process. The receiving process (rank 1) is heavily occupied by the loop in line 9. We assume rank 0 to finish before rank 1 because it should take overall longer to finish the computation on rank 1 than the sending operation from rank 0 to 1.

When testing this program we can observe, that the execution times are almost identical. See Table 4.1, both take around 240ms. This shows that the MPI library did not make use of asynchronous progression. Rank 1 blocks rank zero from sending. The Intel MPI library supports setting the runtime flag `MPICH_ASYNC_PROGRESS`. This flag advices the MPI library to use asynchronous progress. With this flag set the execution time on rank 0 drops to 11ms. Rank 1 remains at 242ms. This test shows that asynchronous progress is not enabled by the default configuration, but can be enforced.

```
1  if (world_rank == 0) {
2    MPI_Win_lock(MPI_LOCK_SHARED, 1, 0, win);
3    MPI_Put(arr, MAX_BUFFER , MPI_DOUBLE, 1, 0, MAX_BUFFER, MPI_DOUBLE,
         win);
4    MPI_Win_unlock(1, win);
5
6  } else {
7    for (int i = 0; i < MAX_BUFFER; i++) {
8      for (int j = 0; j < 1000; j++) {
9        arr[i] = arr[j] * arr[j];
10     }
11   }
12 }
```

Listing 4.1: RMA passive progress test

The flag advices MPI to perform all communication asynchronous. This is done by MPI implementation by launching one extra thread per rank that exclusively handles the communication. Therefore we have twice the number of running processes than physical cores when `MPICH_ASYNCH_PROGRESS` is enabled.

We started the benchmark again with 28 ranks. Rank 0 is still sending data to rank 1. Ranks 1 - 27 are all executing the dummy computation. The execution time for rank 0 again drops from 237ms to 11ms. The time for all remaining processes goes up from 244ms to 452ms. The time has almost doubled. Even for processes that did not perform any sort of communication. This shows that the spawned background processes takes a significant and measurable amount of CPU time.

A third variation of this benchmark is to set runtime variables that are recommended to use with one-sided communication. `MP_USE_BULK_XFER=no` and `MP_CSS_INTERRUPT=yes`. The first parameter disables buffering of multiple operations to one bulk. It therefore starts immediately when a communication call is issued. The second parameter allows MPI to interrupt the process in case an incoming message is received. With this configuration set we can achieve a true passive progression. Rank 0 needs 13ms showing it does not have to wait until the other processes finish their computation. The other ranks have no penalty in execution time. The data transfer is done without activating them.

2 processes			
execution:	default	ASYNC_PROGRESS 1	MP_USE_BULK_XFER=no, MP_CSS_INTERRUPT=yes
rank 0:	237ms	11ms	13ms
rank 1:	244ms	242ms	245ms
28 processes			
execution:	default	ASYNC_PROGRESS 1	MP_USE_BULK_XFER=no, MP_CSS_INTERRUPT=yes
rank 0:	237	11ms	5ms
rank 1 - 27:	238-239ms	452-258ms	245ms

Table 4.1: Execution time of true RMA test

This benchmark was taken on the SuperMuc Haswell nodes. Compiled and executed with Intel MPI, similar results were benchmarked with IBM MPI library, OMPI 1.10 and OMPI 2.0.

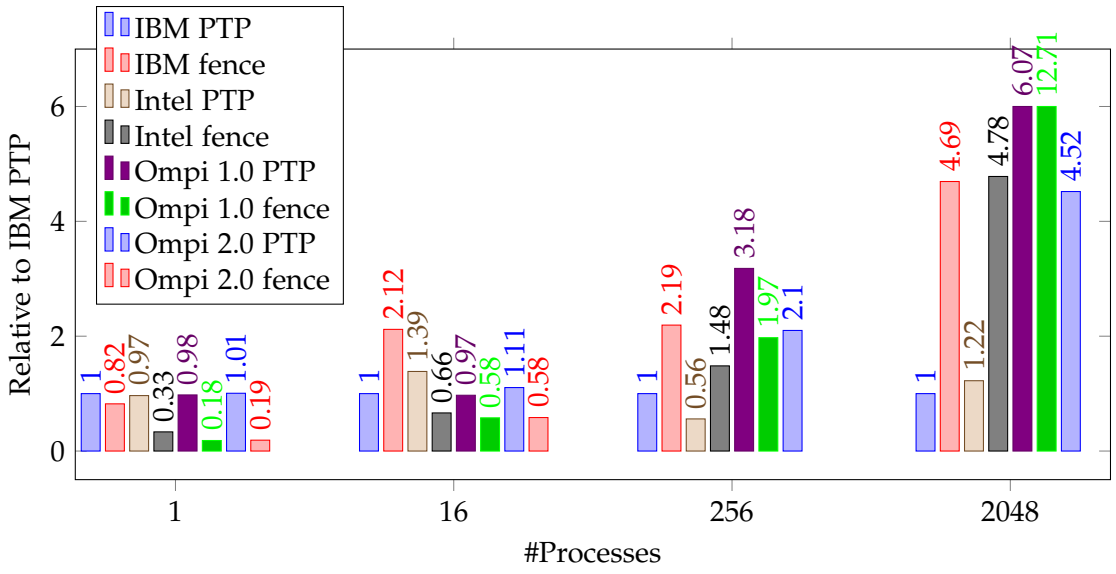


Figure 4.6: Execution time relative to IBM PTP, sending 1 double each

4.6 Different MPI Library Implementations

In this section we want to compare multiple MPI library implementations against each other. We will compare PTP vs fence synchronization. We will benchmark the libraries developed by IBM, Intel, OpenMPI in version 1.0 and 2.0. It is the same benchmark as in Section 4.2. The benchmark will consist of sending N bytes to the next 26 neighbors and then synchronizing, so that the processes are aware of a fully complete communication step. We will benchmark with different sizes and different numbers of processes. In order to visualize all of this data that we will produce with this benchmark, we will compare the performance to the execution time of IBM's PTP time.

In Figure 4.6 we see the benchmark executed for a small amount of data transferred, 1 double each. We can see that for 16 and 256 processes no implementation is dominantly fast. On 2048 processes most one-sided implementations are around 5 times slower than PTP. The Intel implementation is up par with IBM in PTP communication. Ompi fence synchronization does not scale to 256 processes or more. The benchmark terminates with an segmentation fault. In PTP Ompi is still 4 times slower than IBM. The Ompi 1.0 fence synchronization scales the worst, at 2048 processes it is 12 times slower than the IBM PTP variant.

In Figure 4.7 we see the benchmark executed with a message size of 1000 doubles. For inner node communication (< 28 cores) Intel and Ompi fence synchronization is faster

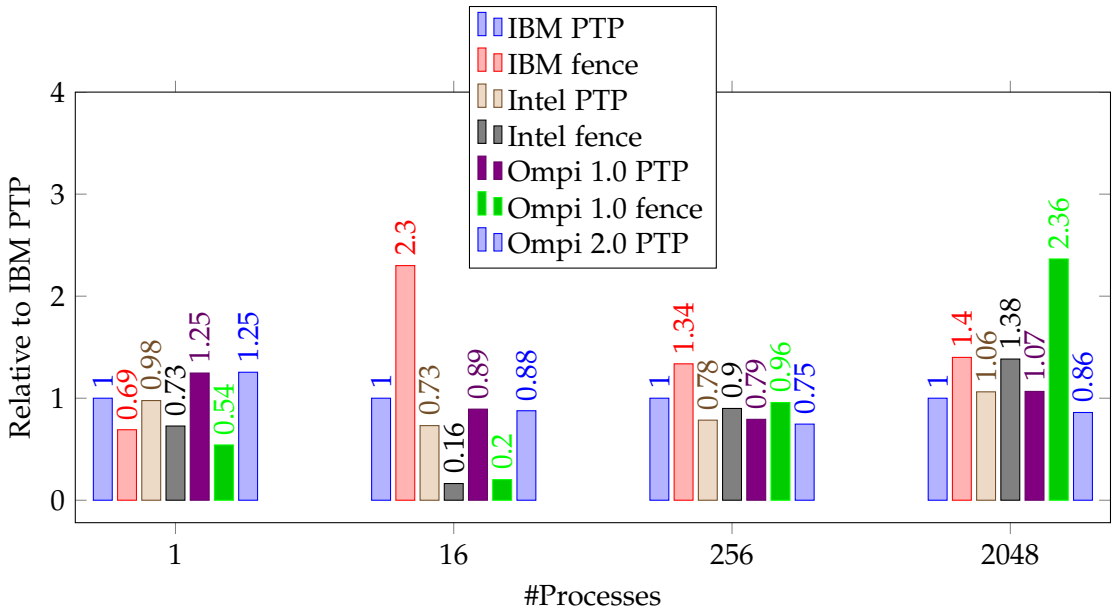


Figure 4.7: Execution time relative to IBM PTP, sending 1000 doubles each

than the IBM's PTP implementations. This is also the case for small messages. IBM shows to be relatively slow at fence synchronization compared to its PTP counterpart.

In Figure 4.8 we repeated the benchmark again with 1 million doubles.The differences are smaller than with smaller messages. IBM fence synchronization is comparatively slow at 16 cores. This diminishes again with more processes.

Overall the IBM implementation seems to be a competitive good implementation. Other implementations are faster in some specific cases, but cannot hold this lead with scaling message size or processes. Especially for smaller messages IBM's implementation leads in this benchmark.

When looking on one-sided communication only, IBM is is second place after Intels implementation, that is in most cases faster. IBM's fence synchronization is especially weak for inner node communication.

It is unfortunate that our tested Ompi version 2.0 did not work reliably. We verified the correctness of our program. Ompi 2.0 was not able to allocate a window size larger than 160MB. The program does stall if tried nevertheless. It also was not possible to start an Ompi 2.0 application with 256 processes and more processes. The application did terminate with an segmentation fault.

GASPI The Global Address Space Programming Interface (GASPI) is an specification

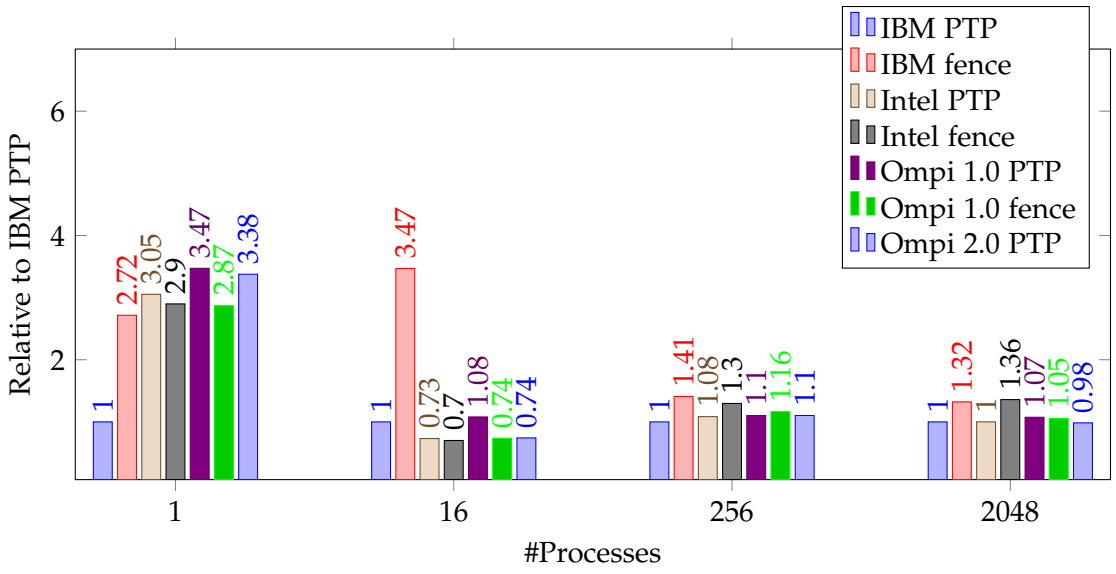


Figure 4.8: Execution time relative to IBM PTP, sending 1 million doubles each

for an library that allows global adress space programming. The specification is created, similar to MPI, in an open forum with with a diverse participating audience. One of the most prominent GASPI implementations is GPI2 [GS13] by the Frauenhofer institute. It focus on providing a high performance with asynchronous progress leveraging one-sided communication.

We wanted to compare how a GASPI implementation compares to MPI. For this we created again a benchmark in which we send a message to its neighbors. GASPI supports a notified write, that makes an additional synchronization in GASPI obsolete.

In Figure 4.9 we show the initialization and execution time of 500 thousand iteration steps in GPI-2. Additionally we display the results of the MPI execution. GASPI has an initialization method similar to `MPI_Init`, where the connection between the nodes is established and communication resources are allocated. We observed that this part takes a significant time of the overall execution time of the program. The blue graph does only consist of the time in the communication loop without the initialization. We can see from the graph that 50 thousand iterations almost take the same time as the initialization phase. We were not able to execute a GASPI program with more than 300 cores as at that scale the initialization already takes a minute. From the graph it seems that the initialization time scales exponentially with the number of processes, which makes the initialization a very expensive operations. Starting a GASPI application on more than 300 processes will stop at the initialization call.

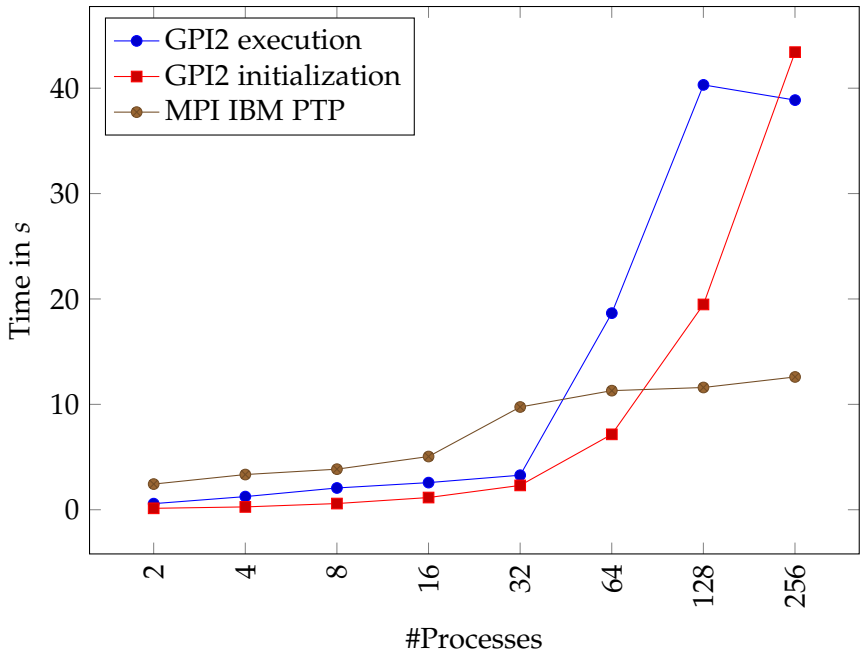


Figure 4.9: Execution of the GPI-2 benchmark

From this benchmark we see, that GASPI is faster than MPI for less than 64 processes. From 6 processes the initialization phase becomes very long and takes a huge amount of time to start.

Other tests (not in the graph) showed that GASPI performance also depends on the number of nodes an application is started on. The same number of processes distributed among more nodes resulted in a faster execution. GASPI's desired application is a hybrid approach with one GASPI communicator per processor and multiple local threads on one physical node. Still we were not able to start an application with more than 300 processes, independent of the number of nodes we used.

5 Results

In this chapter we want to present the results of the optimizations and transformations when applied to our molecular dynamics simulation. We will benchmark each scenario and compare them against each other. Other than in Chapter 4 we will benchmark the entire simulation. This will show the impact of performance in a real program usage. Even if micro benchmarks show a significant change for some variants, it could still be the case that in a complex program the impact might be different. Some performance properties could have dependencies on each other, while others might have a lower impact than in an isolated benchmark.

5.1 Testing Setup

We can provide an input file to our MD simulation that defines a configuration of what we want to simulate. For the tests we will look on a sparse and a dense scenario. The input values are as follows:

	sparse	dense
domain size	16x32x32	32x32x32
particles	10x20x40 = 8000	111x111x111 = 1.37 mio
iterations	1000	50
average particle density	0.49 particle/ cell	42 particles/ cell

Table 5.1: Testing inputs

Additionally other variables were identical in both scenarios $particle\_distance = 3.0$, $\sigma = 1.15$ $\epsilon = 2.0$ $r\_cutoff = 3.5$.

This benchmark is executed with and without the optimization applied. As our reference implementation we use our molecular dynamics code that was optimized by applying general c++ programming guidelines from [Fog17] and profiling as stated in Section 3.1. This implementation uses point to point communication and an AOS data structure.

We use the SuperMuc phase 2 which consists of 3072 nodes each with 2 Haswell Xeon Processor E5-2697 v3. Every node with two Haswell chips has 28 cores. Until

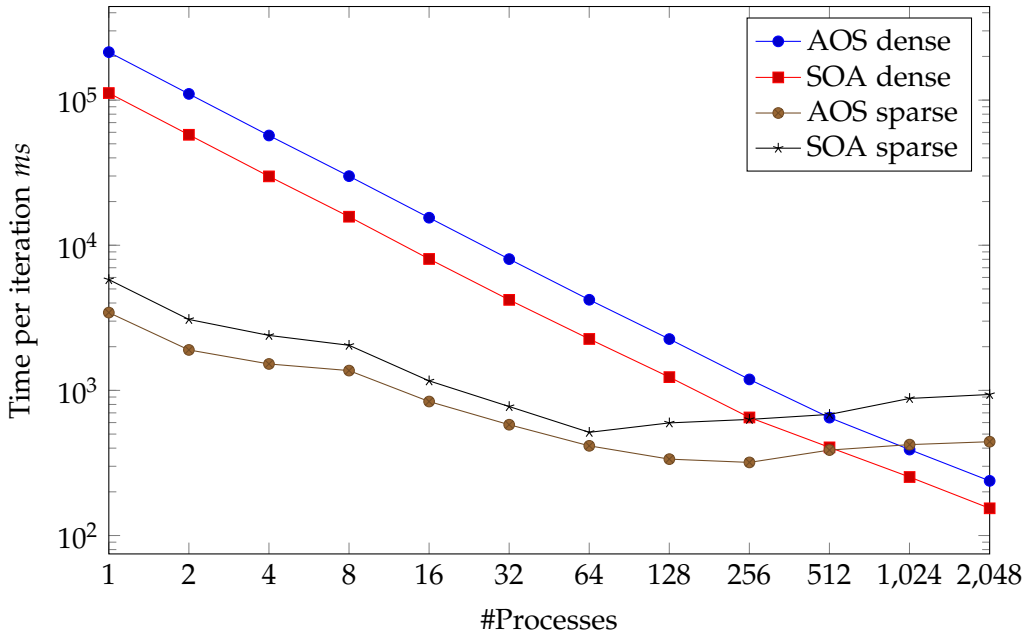


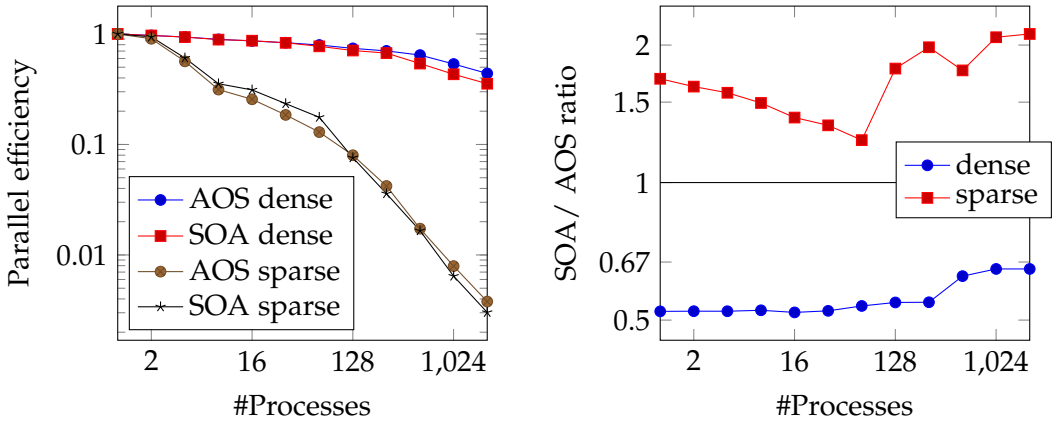
Figure 5.1: SOA vs AOS comparison for a dense and sparse scenario, time per iteration, PTP communication

the limit of 8, the program is executed on the same physical chip, until 16 on the same node. For tests that are executed on more than 28 cores the communication has to use the network, which in our case is Infiniband FDR14.

5.2 Vectorization

To benchmark the vectorization we compared the reference implementation with the SOA implementation, that applies vectorization in the position, velocity and force calculation. In the force calculation we can only efficiently apply vectorization to the forces within a cell, but not to the forces between the neighbors.

In the Figure 5.1 we can see the time it takes to finish the scenarios with different implementations and inputs. Figure 5.2 shows derived data such as the parallel efficiency of each run and the ratio of time between the SOA and AOS implementation. Figure 5.2a shows, that the dense scenarios scales very well, the parallel efficiency drops to 50% at 1024 cores used. The vectorized code is around twice as fast as the non-vectorized code. Because of the lower baseline for the non-vectorized code the reference implementation shows a better parallel efficiency with more cores while



(a) Parallel efficiency

(b) Ratio of SOA to AOS, faster SOA < 1 < faster AOS

Figure 5.2: Derived data from the SOA to AOS comparison in figure 5.1

overall taking longer.

The sparse scenario scales extremely poorly. There is no more speedup improvement once 32 cores are utilized. The vectorized code even takes longer overall to terminate with more cores. The non-vectorized code is faster. We can conclude vectorization introduces an overhead that is not compensated by faster vectorized execution for the sparse scenario.

5.3 Lookup Tables

We will compare a simulation run with a lookup table against a direct calculated cell index. Iterating over the own cells jumps at the end of the axis by the number of cells the halo adds on this axis surface. The cells' access pattern is the same for both implementations.

We measure the presented scenarios and plot the ratio of computed index vs lookup approach. The index calculation implementation has been set to be 100% to emphasis the ratio between these implementations.

We can see from Figure 5.3, that the lookup table for the sparse scenario is on average 3% slower than directly calculated id's until 64 cores. The lookup table for the sparse scenario shows a spike, where it is 50% slower than the directly calculated ids. We do not have an explanation for this sudden spike, but can assure repeated measurements produce very close results with a spike at 128 and 256 cores used. After this the curve

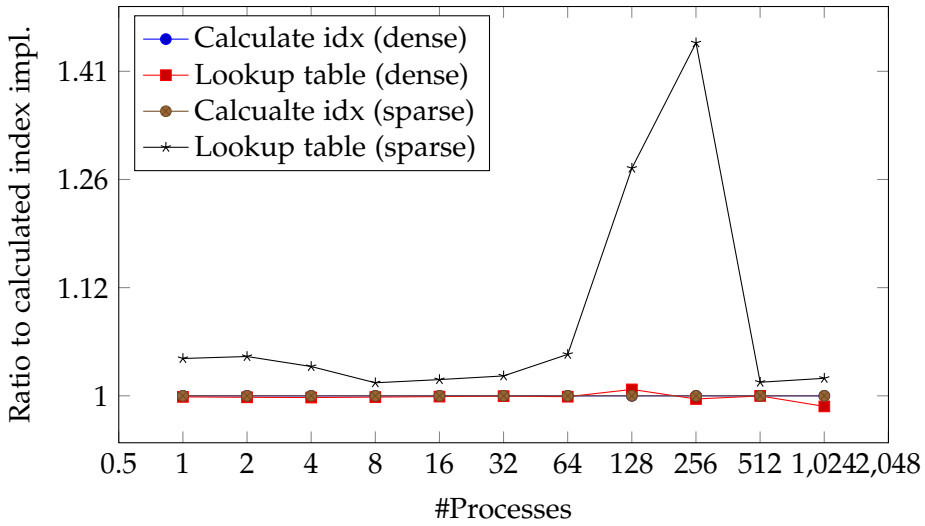


Figure 5.3: Time ratio direct index calculation to lookup table.

becomes flat again whilst being 3% slower than directly calculating ids. For the dense scenario both implementations are almost the same. On average the lookup table is 0.1% faster. This can be attributed to the fact that the dense scenario is computation bound therefore additional computations for the cell id can be impact performance noticeably. For the sparse scenario, memory lookups result in a cache miss. There lookups will stall the CPU, as the data is not available immediately in caches and has to wait for main memory. The direct calculation is up to 50% faster.

5.4 Boundary Force Exchange

As described in Chapter 3.1.3 we want to observe the effect of the full-shell vs the half-shell approach. This is a trade-off between multiple, smaller communication steps and redundant calculations. A process can either send all the halo to its neighbor cells and calculate all forces locally or send only half the halo cells, compute their forces and send the forces back to their owning processes, where all forces are accumulated. We did compare these two approaches, the time each scenario takes to finish and the ratio between both implementations.

In Figure 5.4 we can see the comparison of the full-shell / half-shell method for the dense scenario. The y axis shows the accumulated time spend of all processes. It shows the overhead of an implementation to compute a scenario. In order to get the actual execution time of the program, the plotted time must be divided by the number of

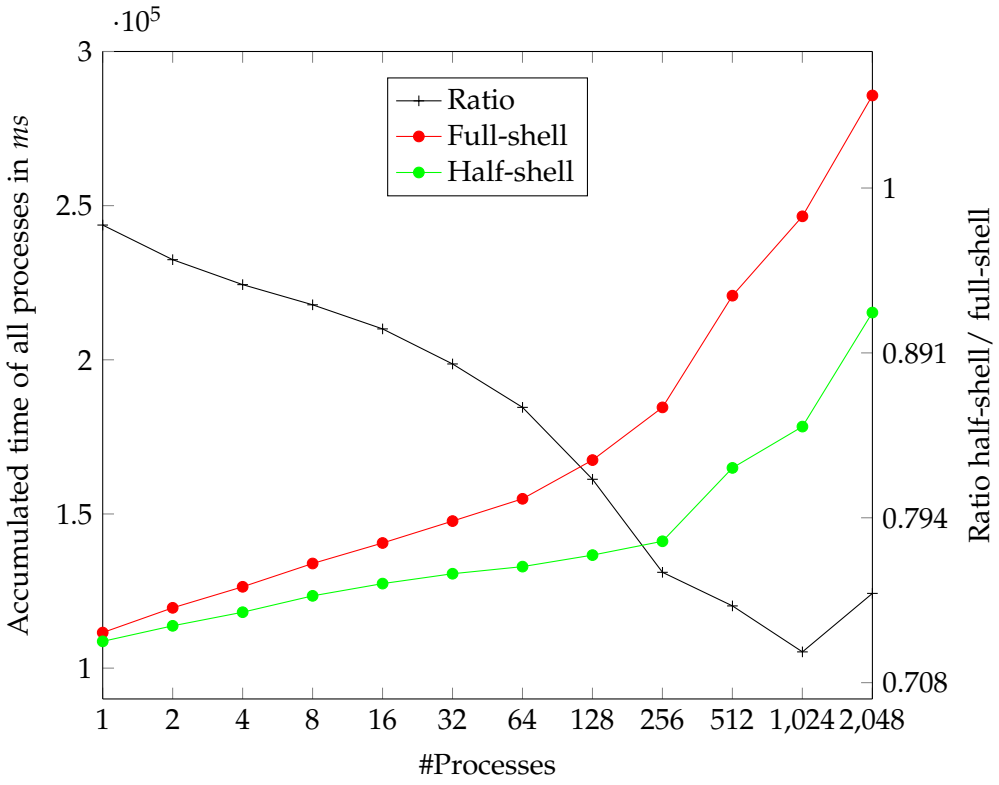


Figure 5.4: Full shell vs half shell comparison on dense scenario

cores used.

The dense approach scales good. With 2048 processes the full-shell approach computes 2.6 time the wok than with one core. As comparison the half-shell approach computes twice the work with 2048 cores than with one. This is equivalent to a 50% parallel efficiency. The ratio between half-shell and full shell decreases with the size of the local domain. This means the full-shell approach becomes less efficient with a rising number of cores. At one core the performance is almost the same, as the halo to own cell ratio is also low. The additional redundant computation of halo force computation is low.

In Figure 5.5 we see the full-shell half-shell comparison with the sparse scenario. The ratio of between the two implementations is in favor of the full-shell approach for more than 32 processes. This is the case, because the additional synchronization step in the half-shell approach is more expensive than the redundant force calculation of the full-shell approach. At less than 32 cores, the execution times for both implementation is very close with the half-shell method being 10% faster. It is not reasonable to scale

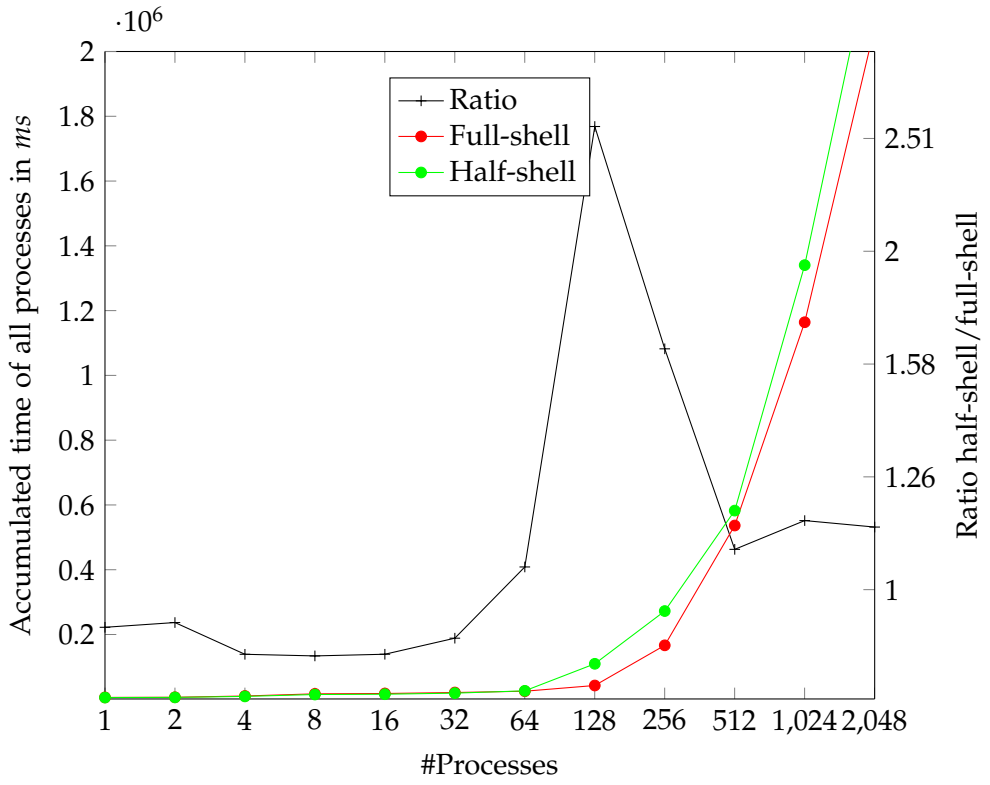


Figure 5.5: Full shell vs half shell comparison on sparse scenario

the sparse scenario to more than 64 cores, as it's parallel efficiency decreases too much.

In both scenarios the half-shell approach can increase the efficiency. The sparse scenario is too small to scale larger than 32 cores. Until this point the half-shell approach is also faster for the sparse scenario.

5.5 Window Creation

As discussed in Section 3.2.1 we have the options to issue multiple one-sided calls, for each vector component, or copy data into one buffer locally and only trigger one `MPI_Put` call once.

The split approach with multiple calls could be faster as it does not have to allocate and copy data, whilst the other approach only has to issue one MPI call in comparison to three.

We measured the simulation with both variants.

In Figure 5.6 we can see the time necessary to finish the scenarios. For the dense

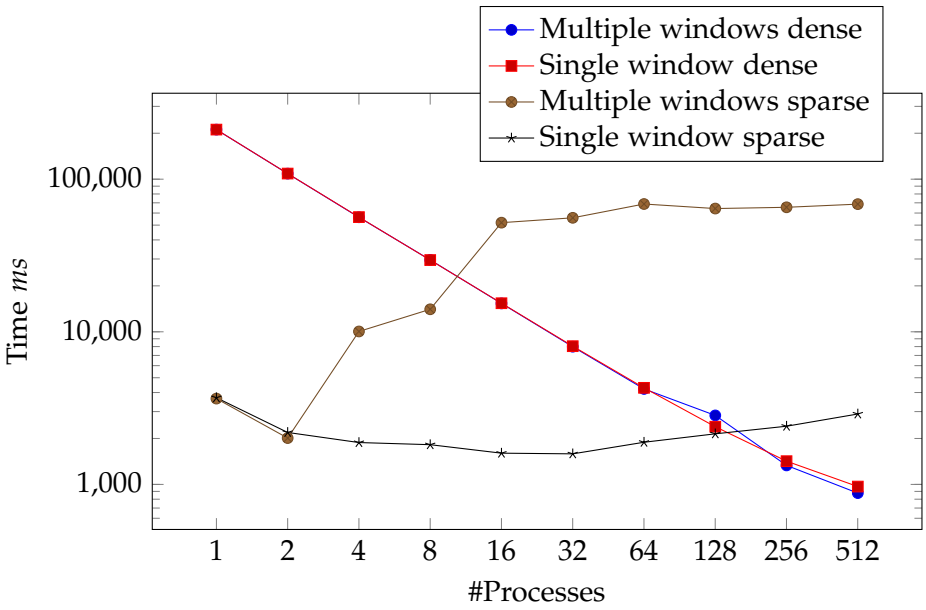


Figure 5.6: Comparison multiple vs single window

scenario, there is no difference between using multiple windows or one. In the sparse scenario, the single window implementation is significantly faster than using multiple windows. It is between 20 to 30 times faster than the multi window approach. The speed improvement does not scale with the number of processes. This shows that windows are synchronized individually. The multi window approach is as many times slower than it has windows. In our case this is 26.

This shows that MPI one-sided communication is not as efficient as it could be considering that the calls are issued almost at the same time and the MPI implementation could repack the blocks of data into a larger buffer. Our benchmark shows that repacking of smaller blocks of memory into one larger one is more efficient. It might require more overall memory usage, but is faster. The MPI library should repack on their own to use this effect. With this benchmark in mind we implemented local copying in our MD simulation. This trades memory usage over execution time. On the downside, we do additional local work. This implementation could become obsolete in case future MPI implementations optimize small memory transfers to be equally fast as one larger transfer. Yet the assumption how future implementations will perform is pure speculation and thus we prioritize to optimize for the current MPI version.

5.6 MPI Status Flags

In Chapter 3.2.4 we presented the MPI status flags. Those are flags that could help the MPI implementation to optimize for the cases where an assertion must not be validated as it is already provided by the user application. In this chapter we want to validate how effective the presented MPI flags are.

We created a benchmark of applications that did synchronize with `MPI_Win_fence`. On the opening epoch we set the `MPI_MODE_NOSUCCEED` flag, indicating no RMA calls were issued before the fence. On the closing fence we set the flags `MPI_MODE_NOSTORE` indicating no local modification to this window was done during the epoch and `MPI_MODE_NOPRECEDE` to tell MPI we will not precede with RMA to this window after the fence.

We did not consider to use `MPI_MODE_NOCHECK` as we want to have synchronization with one-sided communication and this flag is not supported by the fence synchronization call. We also did not use `MPI_MODE_NOPUT` because in our communication scheme as every process participates in the communication.

The benchmark was executed and compared against a variant without those flags set.

The results did not show any significant difference between using those flags or not using them. As the MPI specification says that those assertions might be used to increase performance, they might also be ignored [For15]. We assume that this is the case as not even a benchmark created to leverage these assertions could show an improvement. Future MPI implementations might use these flags, but the libraries from IBM, Intel OpenMPI version 1.0 and 2.0 did not use them.

6 Conclusion

MPI one-sided communication can be easily implemented. It just needs a few changes to a given PTP code in order to transfer it to use one-sided calls. This replacement can also be done gradually, as both models can be used along with each other. Yet its performance is generally worse than existing PTP implementations, especially for those that could mature over multiple iterations.

The programming model, which allows to directly write memory in another processes memory simplifies programming. Especially for use cases with unpredictable communication patterns this can be useful. One-sided communication must be supported by hardware and the implementation to be a performance improvement over PTP communication. This is hard to check and verify with MPI's platform independent, gracefully degradation approach a function will always work, but result in unpredictable performance.

MPI's lack of a notified put or get operation makes it very costly as every operation must be encapsulated withing synchronization barriers that involve the entire communication group. In many cases this leads to a global synchronization barrier that scales with the number of processes.

G.Santhanaraman et al. also came to the conclusion, that one-sided communication is impractical to use "mainly owing to the inefficiencies in current MPI implementations that internally rely on synchronization between processes even during one-sided communication.. "[San+09]

MPI's one-sided semantic might take some time to read and understand, especially considering that it splits communication and computation. This may be uncommon for MPI programmers that expect to pass messages between nodes, but for programmers that have experience with shared memory programming it should not be difficult to pick up the overall concept together with the synchronization primitives. Some MPI method names could be named differently to express its function further. The `MPI_Win_lock` calls do not guarantee any locking, `MPI_Win_fence` could have been named differently to indicate whether a fence is opening or closing the epochs.

MPI's interoperability and its platform independence has both its advantages and disadvantages. Having an application that can be implemented independently of used hardware, library or platform is very convenient. Yet it is not easy for the programmer to guess some of the performance metrics as different physical networks between nodes

can have a huge unpredictable performance impact. Therefore a different topology could change the way communication should be performed. MPIs opaque abstraction of the used underlying hardware makes it hard to write applications that adapt to available hardware. This makes it hard to guess latency, bandwidth and distance of a specific memory exchange from the applications during runtime. Overall we could show that MPI does not use local shared memory copies in a single node as efficiently as it could. Also asynchronous progress is not being used to the extend that we were hoping for in 2.2.5. A mapping of one-sided calls to RDMA primitives can not be observed. The lack of a notified write increases the necessary synchronization and its inherent costs significantly. The lack of a notification about a read makes it necessary to introduce global synchronization barriers that scale with the number of process in the communication world and thus makes it very expensive for a large scenario to synchronize every simulation step. This makes it inefficient to use on many processes and decreases parallel efficiency. MPI libraries still often use point to point communication which bounds one-sided communication to be at best as fast as point to point communication. Hardware features of Infiniband and RDMA are not used extensively enough in order to be noticeable. Infiniband and RDMA messaging support messaging queues that can tell about local and remote progress and can tell about incoming messages. This could be used to create a notified write.

MPI one-sided communication should most likely only be used on a problem scenario with unpredictable heterogeneous data exchange. There it can leverage its power of being able to write and read to any process in the communication group. For these heterogeneous scenarios point to point communication could be harder to implement the communication scheme has to be agreed on though message exchanges that precede the actual data exchange.

Thakur R. et al. have shown that one-sided synchronization can be sped up by 50% [RT05]. Which shows that synchronization has great potential to catch up with PTP synchronization. Future MPI implementations might optimize there. However the pace with which one-sided communication has been developed has been slow in the last years. MPI's core functionality is still passing messages in a convenient platform independent way with a point to point scheme. Other libraries that are not generic and platform and hardware independent can be optimized easier to a specific hardware. One specific libraries also do not have the baggage of having to support an API that can be supported by a wide variety of hardware.

6.1 Outlook and Alternative Implementations

Hybrid programming models that use shared memory programming on a local node level and an MPI interconnection between nodes reduces the number of ranks that need to participate in the communication. A hybrid model also allows to use a single core of a process to be exclusively used as a message broker. The threads within the processes can use shared memory programming to exchange data. This is faster than a communication that has to go through the MPI stack and potentially over a slower interconnection interface. This comes at a cost of an inhomogeneous programming code that needs to take care of different cases. This handling could quickly become very complex and often must be adjusted for different hardware. In the end this handling is abstracted away from the application by MPI. MPI promises to use he fastest link available. This generic approach supports developers, but comes at the cost that it is slower than a hand optimized hardware specific solution.

A hybrid solution is becoming a more reasonable approach with the ever rising number of cores on a single processor. MPI's abstraction comes at a cost that limits bandwidth, introduces latency and consumes memory over a threaded shared memory approach. Writing shared memory programs makes it unnecessary to communicate within the node, but only requires us to set locks on critical regions to avoid race conditions. Its lower abstraction should be more efficient than a library that internally could rely on shared memory data exchange at best. Shared memory programming does have its disadvantages. In non uniformed memory architectures main memory has different latencies to different cores on a chip, especially for motherboards with more than one socket. This could degrade performance if unnecessary long paths have to be taken over the memory bus. Another pitfall in shared memory programming is false sharing on cache coherent memory. False sharing is when multiple threads share a memory location, that is placed on the same cache line without intention. This leads to the effect that if one of the processes modifies data on their part of the cache line, the entire line has to be evicted and will have to be written back to main memory in order to ensure cache coherency with the other threads. Rabensteiner et al. also came to the conclusion that a hybrid approach can achieve a more efficient implementation than pure MPI implementations [RHJ09]. He also mentioned that a hybrid solution has more development effort, yet its potential gain in scalability is worth it.

With accelerator chips or general purpose graphics processing units programming the computation power of a single node becomes extremely powerful. More applications can be computed on one local node within reasonable time constrains and no longer require distributed programming. For the applications that still will not fit on a single local node, the overall architecture changes to become more heterogeneous. With more and more specialized chips and accelerators this increases even further. A

more heterogeneous architecture, either by using MPI plus threads or by some sort of accelerator cards, is in general more complex to implement. This requires more development effort which inherently is more expensive and error prone. This goes in the direction of a hardware aware implementation.

Another alternative viable approach in order to speed up the MD simulation is to implement the communication on a deeper abstraction level. MPI's convenience comes at a cost. It consumes CPU cycles and in our MD simulation introduces unnecessary synchronization. Going close to the operation system supported Infiniband library could gain us performance by getting rid of unnecessary overhead. The Infiniband ibverbs is a close abstraction of the RMA interface that we can use in Linux. It uses a queue for sending and receiving requests that can be used to observe progress of outgoing and incoming data. It uses a socket approach to connect to other processes. Because of it low abstraction we are responsible to handle a lot more cases ourselves. While ibverbs exposes us the raw performance of our machines it comes at a cost. Firstly we have to write more code, that in its nature is prone to errors. An initialization and simple data exchange can easily make up 100's of lines of code that can be implemented with 5 MPI calls. We also lose the platform independence of MPI. Our program would only be able to run on hardware with an Infiniband connection. The ibverbs approach of sockets also requires us to create a socket connection for every pair of processes that we want to communicate. There is no built in mechanism to declare a group of processes as a communicator the same way we are able to do in MPI. We have to build a communication topology on our own.

These alternative solutions show a potential of improving the MD simulation performance and scalability. Its heterogeneous approach requires us to focus and solve many facades of the implementation. Many components have to flawlessly work together. This might be a complex and challenging goal but with a huge potential.

Bibliography

[BD04] D. Bonachea and J. Duell. "Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations." In: *International Journal of High Performance Computing and Networking* 1.1-3 (2004), pp. 91–99. ISSN: 1740-0562. DOI: 10.1504/IJHPCN.2004.007569.

[Ben16] A. Benassi. "Phase transition based control of friction at the nanoscale." In: *High Performance Computing in Science and Engineering*. Ed. by S. Wagner, A. Bode, H. Satzger, and M. Brehm. Bayerische Adademie der Wissenschaften. Garching/Munich: Leibniz-Rechenzentrum, 2016, pp. 216–217. ISBN: 978-3-9816675-1-6.

[Eck12] W. Eckhardt. *Memory-Efficient Implementation of a Rigid-Body Molecular Dynamics Simulation*. Garching: Institut für Informatik, Technische Universität München, June 2012.

[Fog17] A. Fog. *Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms*. 2017.

[For15] M. P. I. Forum. *MPI: a Message-passing Interface Standard: Version 3.1*. High-Performance Computing Center, 2015.

[GBH13] R. Gerstenberger, M. Besta, and T. Hoefler. "Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013, 53:1–53:12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503286.

[GHT15] W. Gropp, T. Hoefler, and E. Thakur Rajeev Lusk. *Using advanced MPI modern features of the message-passing-interface*. Cambridge, MA: The MIT Press, 2015.

[GKZ07] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. 1st. Springer Publishing Company, Incorporated, 2007. ISBN: 3540680942, 9783540680949.

[GS13] D. Gr"unewald and C. Simmendinger. "The GASPI API specification and its implementation GPI 2.0." In: 243 (2013).

[Mat16] S. R. Matteo Maestri Sebastian Matera. "Coupling kMC and CFD in Heterogeneous Catalysis." In: *High Performance Computing in Science and Engineering*. Ed. by S. Wagner, A. Bode, H. Satzger, and M. Brehm. Bayerische Adademie der Wissenschaften. Garching/Munich: Leibniz-Rechenzentrum, 2016, pp. 182–183. ISBN: 978-3-9816675-1-6.

[MZ07] S. K. M. Griebel and G. Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. 1st ed. Springer, 2007.

[Nor16] M. G. Norman Blümel Alexander Götz. "Dynamics of Transmembrane Domains: Impact on Complex Membrane Processes." In: *High Performance Computing in Science and Engineering*. Ed. by S. Wagner, A. Bode, H. Satzger, and M. Brehm. Bayerische Adademie der Wissenschaften. Garching/Munich: Leibniz-Rechenzentrum, 2016, pp. 248–249. ISBN: 978-3-9816675-1-6.

[Pet16] A. B. Peter Coveney Shunzhou Wan. "Rapid and accurate calculation of ligand-protein binding free energies." In: *High Performance Computing in Science and Engineering*. Ed. by S. Wagner, A. Bode, H. Satzger, and M. Brehm. Bayerische Adademie der Wissenschaften. Garching/Munich: Leibniz-Rechenzentrum, 2016, pp. 270–271. ISBN: 978-3-9816675-1-6.

[RHJ09] R. Rabenseifner, G. Hager, and G. Jost. "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes." In: *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE. 2009, pp. 427–436.

[RT05] W. G. Rajeev Thakur and B. Toonen. "Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication." In: *LNCS* 3241 (2005). DOI: 10.1007/978-3-540-30218-6_15.

[San+09] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. K. Panda. "Natively Supporting True One-Sided Communication in MPI on Multi-core Systems with InfiniBand." In: *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. May 2009, pp. 380–387. DOI: 10.1109/CCGRID.2009.85.

[Spa16] J. Spahl. "Evaluation of Zonal Methods for Small Molecular Systems." Bachelor's thesis. Institut für Informatik, Technische Universität München, Nov. 2016.