Comparison of PGAS Languages on a Linked Cell Algorithm

Martin Bauer, Christian Kuschel, Daniel Ritter, Klaus Sembritzki

Lehrstuhl für Systemsimulation
Friedrich-Alexander-Universität Erlangen-Nürnberg
Cauerstr. 11, 91058 Erlangen
{martin.bauer, christian.kuschel, daniel.ritter, klaus.sembritzki}@fau.de

Abstract: The intention of partitioned global address space (PGAS) languages is to decrease developing time of parallel programs by abstracting the view on the memory and communication. Despite the abstraction a decent speed-up is promised. In this paper the performance and implementation time of Co-Array Fortran (CAF), Unified Parallel C (UPC) and Cascade High Productivity Language (Chapel) are compared by means of a linked cell algorithm. An MPI parallel reference implementation in C is ported to CAF, Chapel and UPC, respectively, and is optimized with respect to the available features of the corresponding language. Our tests show parallel programs are developed faster with the above mentioned PGAS languages as compared to MPI. We experienced a performance penalty for the PGAS versions that can be reduced at the expense of a similar programming effort as for MPI. Programmers should be aware that the utilization of PGAS languages may lead to higher administrative effort for compiling and executing programs on different super-computers.

1 Introduction

A common trend in state-of-the-art computing is the parallelization of algorithms. A de facto standard in parallel scientific software is the Message Passing Interface (MPI) library. More recently, partitioned global address space (PGAS) languages have been developed to support and implement parallel computations, such as e.g. Co-Array Fortran (CAF), Unified Parallel C (UPC) and Cascade High Productivity Language (Chapel). MPI works on a distributed memory model in which each processor has a memory of its own and sends/receives messages to exchange data with other processors.

A different approach is conveyed by PGAS languages: Each processor is assigned a part of the global memory space which can be freely accessed by the respective processor. When accessing other memory performance penalties have to be expected due to either non-uniform memory access (NUMA) or network communication effects. The languages provide synchronization mechanisms to avoid memory inconsistencies. In contrast to distributed memory models, the PGAS model allows each processor to access the entire memory directly without manually invoking transactions.

In this paper we analyze the three above-mentioned PGAS languages—CAF, Chapel and

UPC—by means of implementing a linked cell algorithm. The analysis focuses on two aspects: a. the ease of use of the language and b. the performance of the parallel implementation. Since the PGAS languages promise a good trade-off between parallel performance and ease of use (compared to MPI), both aspects are crucial to our tests. Therefore the analysis focuses on this trade-off. Similar evaluations have already been performed in [BSC+12] and [TTT+09]. The choice for the implementation of a linked cell algorithm is that it is not trivially parallelizable as e.g. solving a partial differential equation as in [BSC+12], because the moving particles may impose computational load imbalances.

The tests show that, using CAF, Chapel and UPC, the coding effort for the linked cell algorithm is less than for the MPI version. However, the performance of the PGAS implementation is worse than that of MPI. For UPC, the performance gap can be closed, if it is used in an MPI-like fashion.

Initially, an MPI parallel reference implementation in C of the linked cell algorithm is ported to CAF, Chapel, UPC. The details of the reference implementation and the linked cell algorithm are presented in section 2, the ported implementations are elaborated in section 3. The performance results are evaluated in section 4.

2 Reference Implementation

Most parts of the program were implemented in C according to [GDN98]: a Velocity-Störmer-Verlet time stepping and a linked cell data structure was used (cf. [Ver67]). The algorithm can be paraphrased by the pseudo code in algorithm 1.

Algorithm 1 Linked Cell Algorithm per Time Step

```
for all Particles do
  update particle forces
end for
for all Particles do
  update particle positions
end for
```

There are several design decisions to make when implementing a linked cell algorithm. To begin with, an appropriate data structure for storing particles has to be chosen. Therefore the algorithm is analyzed: at first, the inter-molecular forces are computed for every particle. Given this force, the velocity and the position of the particle can be determined by explicit integration. This approach requires to store the total force $\mathbf{F}_k$ acting on the k-th particle, as well as its current position $\mathbf{p}_k$ and current velocity $\mathbf{v}_k$. These quantities can be subsumed in a C struct (see Listing 1).

In order to update the force $\mathbf{F}_k$, the following computation scheme is used:

$$\mathbf{F}_k = \sum_{i \in \mathcal{I}(k), i \neq k} \mathbf{F}_{i,k}. \tag{1}$$

```
struct Particle {
    unsigned short type_id;
    double force [3];
    double position [3];
    double velocity [3];
};
```

Listing 1: Particle data structure.

$\mathcal{I}(k)$ denotes the index set of particles which are in the same cell as the k-th particle or in one of the adjacent cells (26 in 3D). The force $\mathbf{F}_{i,k}$ between particle P_k and P_i ($i \in \mathcal{I}(k) \wedge i \neq k$) is computed by the Lennard-Jones Potential:

$$\left|\frac{\mathbf{F}(d)}{d}\right| = \frac{24\varepsilon}{d^2} \cdot \left(\frac{\sigma}{d}\right)^6 \cdot \left(1 - 2 \cdot \left(\frac{\sigma}{d}\right)^6\right) \tag{2}$$

d denotes the distance between the two particles P_i and P_k. ε and σ are defined as

$$\varepsilon := \sqrt{\varepsilon_k \cdot \varepsilon_i} \qquad \sigma := \frac{\sigma_k + \sigma_i}{2}. \tag{3}$$

(2) can be rewritten in a form that can be evaluated faster:

$$\left|\frac{\mathbf{F}(d)}{d}\right| = \frac{A(P_k, P_i)}{d^8} - \frac{B(P_k, P_i)}{d^{14}} \tag{4}$$

The terms A and B, depending on mass m_k, ε_k, σ_k and m_i, ε_i, σ_i, are computed by

$$A = 24\varepsilon\sigma^6 \text{ and } B = 48\varepsilon\sigma^{12}. \tag{5}$$

Consequently, each particle P_k needs the parameters ε_k, σ_k and m_k in addition to $\mathbf{F}_k$, $\mathbf{p}_k$ and $\mathbf{v}_k$. In typical use cases, on the other hand, there are thousands of particles, but only few different types of particles. Instead of storing the constants ε_k, σ_k and m_k in each particle data structure, a look-up table for A and B can be precomputed for each pair of particle types. In that way, only a type id has to be stored in the particle data structure. This type id refers to a row or column in the look-up table[1]. In order to compute the potential between two particles, the respective type ids specify the entry in the look-up table which is used. With this approach the efficiency of the algorithm is improved and the size of the particle data structure for particles is kept small.

The particle data structure is directly stored in the cells as shown in Fig. 1. In our implementation each cell holds an array of particles (instead of a linked list as in [GDN98], speeding up iterations over particles). When a particle leaves a cell, this particle is swapped with the last valid entry and the number of valid entries is decremented. Memory is not freed when particles leave the cell to reduce the number of the expensive reallocation and copying steps. If more memory is required, the newly allocated space has twice of its previous size and the previous content is copied. At the start of a simulation, the number of

[1] Since $\mathbf{F}_{i,k} = \mathbf{F}_{k,i}$, the look-up table is symmetric.

particles per cell is estimated and memory of particle arrays is pre-allocated accordingly. So in a typical run the number of reallocation operations is very low.

The inner loops of the linked cell algorithm can be parallelized by simply distributing cells to different processors. PGAS languages are suited for this task, since the cells are distributed among processors and the particle data structure need only locally be available.

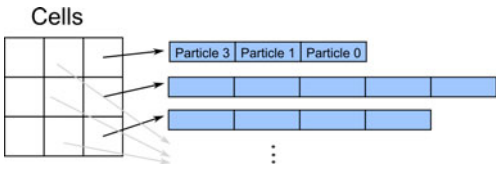


Figure 1: Cell-based data layout.

3 Parallel Implementation

In this section we analyze the data dependencies, parallelization strategies for the linked cell algorithm, and address language specific issues. The cells are split into slices and are distributed among different processors, which in turn assign the cells to different threads (cf. [SBG96]). Each thread operates on cells which are stored in its local memory without imposing race conditions. At first, the force update loop is parallelized. In this step the velocities and positions, and thus the distances between the particles, are fixed. Consequently, the force acting on each particle can be computed independently, without inter-thread communication. In contrast, the position updates are not independent. When updating the position of a particle, it may cross the boundary of a cell. Then the particle is copied to the neighboring cell, which may be owned by a different thread. In this case a locking mechanism is needed to synchronize the communication between the two threads.

3.1 UPC

The preliminary goal was to implement a working parallel version as fast as possible. In this attempt, we chose to share the global array of cells among all threads. Sharing of the domain is necessary, since otherwise a thread has no access to the cells of the neighboring thread which is required when particles move across cell boundaries. We decompose the domain and assign each thread a subset of the domain. For simplicity we chose a 1D domain decomposition by distributing the z-coordinate uniformly across the threads. This is simple to realize for the force update loop, since only the iteration bounds of the outermost loop have to be adapted according to the current thread number.

As discussed above, the iterations of the position update loop are not independent and therefore the parallelization requires additional effort. Consider an updating thread in slice $n-1$ and another one in slice $n+1$: Assume a particle moves from slice $n-1$ to n and

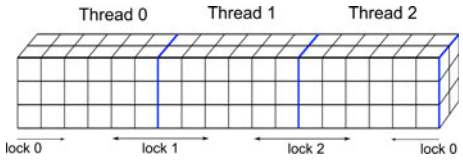


Figure 2: Locking scheme (periodic boundary).

simultaneously another particle moves from slice $n+1$ to n. Both threads write to slice n at the same time. This leads to a race condition, if the cell data structure is not thread-safe. One option to prevent race conditions is to implement a locking mechanism for reading from and writing to the cell data structur. Since we want to parallelize the original code with as less programming effort as possible, we chose another simpler approach: To avoid the race condition described above, we impose the following restriction: While thread i processes its first two slices, process $i-1$ must not update its last slice. This condition is enforced using UPC's locking mechanism. Each lock protects four slices, as depicted in Fig. 2, and enforces that only one thread is in the critical region.

Few changes to the original program were required to parallelize the more complex position update loop.Due to our design, the implementation was kept easy, but there is still plenty of room for optimization. The UPC version was further improved with respect to parallel performance with respect to the suggestions of [CSC+07], section 8.3, p. 68:

> "1. Use local pointers instead of shared pointers for local shared data through casting and assignments
>
> 2. Use block copy instead of copying elements one by one with a loop
>
> 3. Overlap remote accesses with local processing using split-phase barriers"

Whereas the first item is a mere technicality, the second requires a new communication scheme. The threads have to be decoupled as much as possible and the data, which need to be exchanged, has to be assembled in a continuous block in memory. This entire block is then copied to the neighboring thread. Considering the third item, computations can also be performed while copying data. The UPC implementation and the network type on which the program was tested (see section 4) did not support overlapped communication and computation. Therefore it was not implemented. The entire communication had finished during the UPC notify call, before the computation started. However, on a different network type or UPC compiler, overlapping may yield performance improvements.

Focusing on the message passing (item two of the list of suggestions), each part of the domain is assigned to a different thread. These subdomains are linked by ghost layers (gray cells in Fig. 3) which are just a copy of the next thread's outermost layers. So the g layer on $T1$ is merely a copy of the $i1$ layer on $T2$. We stick to the 1D domain decomposition, since only one synchronization point per time step is required. Relevant information for adjacent threads is collected and packed in a message buffer. The content of this buffer is sent using UPC's block copy mechanism. The first element of the Particle Message stores the number of particles, the next how much memory is reserved for this

message, followed by the particle data structures together with the cell index in which the particle is located. The only required UPC language construct for this approach is the `upc_memput` function and one shared particle message buffer.

The information which has to be exchanged in the communication step consists of two parts. First, the particles on $T1$ which moved into ghost cells g have to be sent to the neighboring thread $T2$ which stores them in slice $i1$. Second, slice $i1$ on $T1$ is copied to the ghost layer g on $T2$. This can be implemented using two messages and two synchronization points per time step. However, the same result can be achieved by sending only one message: First, the outgoing messages are stored in a buffer. The first part of the message consists of all particles in $i1$ on $T1$. The second part of the message contains all particles which moved from $i1$ to the ghost layer g on $T1$. The unpacking of the message works as follows: first, the content of the ghost layer on $T2$ is deleted and the particles of the first message part are copied into it. Then the particles of the second part of the message are copied to $i1$ on $T2$. In that way, the communication overhead when running the linked cell algorithm on many processors is kept small.

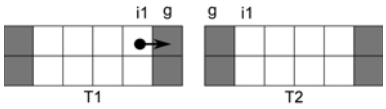


Figure 3: Domain decomposition.

3.2 MPI in C

In the optimized UPC version described above, basically a message passing scheme has been implemented. A version was written in C which uses MPI instead of UPC's `memcpy` mechanism. For all messages, the synchronous version of the MPI functions was used to stay comparable to the UPC implementation.

Concerning programming effort, there is almost no difference between the optimized UPC and the MPI version. In both versions the programmer has to specify manually which data is exchanged between threads/processes and how to pack and unpack the messages.

3.3 CAF

Similar to MPI, in CAF all instances (called images in CAF) work on their private part of the data. They run asynchronously and synchronization is specified explicitly in the source code. Data is exchanged using co-arrays, which are variables that have so-called co-dimensions in which each element is stored by exactly one image. An image can access another image's data by explicitly specifying the co-dimension of the variable. This makes it possible for the compiler to efficiently store and access co-arrays in consecutive memory on shared memory architectures and, on distributed memory architectures, to use

a protocol like MPI or the SHMEM library to perform communication. More information about the features of CAF can be found in [ISO10b] and [ISO10a].

Parallel execution of the linked cell algorithm requires each process P to receive information from its neighbors: The position of all particles in the cells adjacent P's domain and position, velocity and net force of each particle that enters P's domain.

The data exchange can be implemented in two ways: Either the cell data structures are declared as co-arrays and each image directly accesses its neighbors' data structures, or the data is packed by each image and transferred to the other images, like in a typical MPI implementation. The latter approach was chosen as it also allows to run the code efficiently, if small messages can not be transferred efficiently. With this choice the implementational effort was approximately as high as the initial MPI implementation.

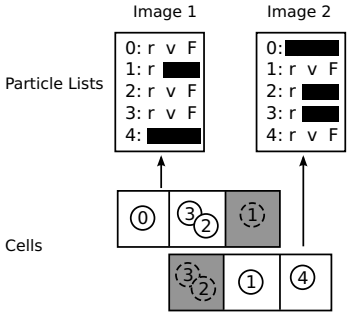


Figure 4: CAF memory layout.

The memory layout is depicted in Fig. 4. Each image allocates memory for all particles, even for those that it does currently not contain. In Fig. 4, image 1 contains the particles 0, 2 and 3. It keeps position, velocity and force for those particles up-to-date. Only the position is stored for particle 1, its velocity and force are irrelevant for image 1. After each time step, image 1 sends the positions of particles 2 and 3 to image 2, which requires a message of size $2 \cdot (4 + 3 \cdot 8)$ bytes: An index of type integer and the three components of the position vectors. If particle 2 leaves the domain of image 1, then also the velocity and the force will have to be sent to image 2.

```
sync images(neighbor_indices) ! sync with all neighbors
do i=1,26
  ! "push" communication with neighbor in direction i
  buffer[i] = sendbuffer(i)
end do
sync images(neighbor_indices)
```

Listing 2: Push communication.

"Push" communication (see Listing 2) is used for the buffer exchange. So each image performs remote write operations on neighboring images. Two synchronization statements

surround the actual buffer exchange. The first synchronization ensures that the receiving image's receive buffer can be written to, the second statement ensures that the communication has finished before the receive buffers are read.

3.4 Chapel

Chapel has been recently developed by Cray Inc. and is designed to provide parallelization of programs from a high-level programming language perspective. Unlike CAF or UPC, which extend existing languages by new concepts, Chapel is a whole new language. It provides syntax to mark sections to be executed in parallel. The entailed inter-thread communication is automatically generated by the compiler without the influence of the programmer (cf. [Inc10]). Therefore the first parallel linked cell implementation in Chapel merely consisted of porting the initial C implementation to Chapel without heeding parallel aspects. As a result of Chapel's design, the ported code looked better arranged and could be executed in parallel right away. In particular, vector arithmetics can easily be implemented and do not require loops as in C. Consequently, the implementational effort is low as compared to the other languages; especially since a parallel program is obtained for free.

Although a parallel program is easily created, it can still be made more efficient by explicitly labeling sections to be executed in parallel (e.g. forall loops and reduction functions). In this way, the domain decomposition is carried out like discussed above. In reference to algorithm 1 this corresponds to parallelizing both loops running over the cells. This is the most the Chapel code can be optimized with respect to parallelization. By its design, Chapel does not allow the user to explicitly define the domain decomposition and communication.

3.5 Programming Effort

The parallel C++ implmentation took only a moderate amount of time to program approx. 1,400 lines of code. The 'message passing' CAF and UPC implementation took more effort than the reference implementation, because we had to get used to the languages. In total UPC implementation has approx. 4,000 lines and CAF approx. 2,400. As described in section 3.1 the 'shared memory' implementation in UPC reduces the lines of code to 2,200, which reduces the effort to program. Chapel was implemented with approx. 800 lines with fairly little effort. Because of Chapel's C-like syntax the linked cell algorithm was developed fast.

4 Test Results

The setup for the evaluation was a Rayleigh-Taylor Instability (see Fig. 5) with two types of fluids. One fluid, having a higher specific mass than the other one, is lying upon the

other fluid. Because of external forces (e.g. gravity) and the different specific mass the two layers mix: The higher specific mass fluids displace the lower specific mass fluids and create characteristic mushroom shapes (cf. [GDN98]).

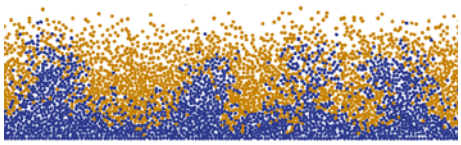


Figure 5: Snapshot of a Raleigh-Taylor Instability.

In terms of the linked cell algorithm, the two different fluids are represented by two types of particles with different mass. In the tests the heavier particles are twice as heavy than the lighter ones. In total 754,992 particles were distributed among 138,240 cells, which yields approximately 5.5 particles per cell. The cells were arranged in a cuboid of the dimensions $240 \times 24 \times 24$. The Lennard-Jones Potential used to compute the inter-molecular forces was parameterized with $\varepsilon = 1.0$ and $\sigma = 1.2$. The tests were run on a single Fat Node of the SuperMUC petascale system in Munich. This system is composed of 18 Thin Node islands and one Fat Node island. The Fat Node contains four Intel Xeon E7-4870 "Westmere-EX" processors with ten cores each, resulting in 40 physical cores per node. All of these 40 cores share 256 GB of main memory, so that each core has approximately 6 GB available. All ten nodes of a processor share the L3 Cache and make up one NUMA domain.

Since the investigated PGAS languages are not as common as the standard parallelization libraries OpenMP and MPI, it can be very time consuming to compile and execute a UPC, CAF or Chapel program on a cluster, due to compiler issues or lacking support of the queuing system to execute the parallel program.

To compile the two UPC versions of the molecular dynamics simulation, the Berkeley UPC compiler version 2.14.0 was used. This is a code-to-code compiler, i.e. it first translates the program to C code, then another C compiler is used to create the final binary. On the SuperMUC, the Berkeley UPC compiler is configured to use the GNU C compiler for this task. To be comparable, we also compiled the MPI parallel reference version with the GNU C compiler, instead of the recommended Intel compiler, which is known to produce slightly faster code on this machine.

UPC programs can be run with different networks. In our testcase all threads run on a single node and therefore share the memory, the "Symmetric multiprocessor" network type was used. On the SuperMUC, UPC communication can also be done using the "ibv" (InfiniBand) network type, which makes inter-node communication possible. However, this was not tested in our experiments.

The CAF versions were created using the built-in co-arrays of the Intel Fortran Compiler version 12.1.6 which used Intel MPI version 4.1 for the actual communication. The optimization level was set to "-O3" and the flag "-coarray=shared" instructed the compiler to create a shared memory version.

Preliminary tests showed the linked cell algorithm implemented with Chapel 1.4 cannot keep up with the other implementations in terms of efficiency. On the one hand, Chapel is a

new language and not all promised features have been implemented yet and others require optimization. It can be expected this will change over time as Chapel is further developed. On the other hand, Chapel is optimized to be run on Cray architectures. The SuperMUC, however, uses generic, less efficient libraries to carry out communication. This imposes overhead rather than allowing efficient execution. For our tests we only had access to the SuperMUC without Cray architecture. Therefore we did not evaluate the parallel efficiency of Chapel, as no appropriate hardware was available. However, the serial implementation of Chapel performed reasonably well. Although being slower than the serial C, CAF and UPC counterpart, the Chapel program needed the least implementational effort.

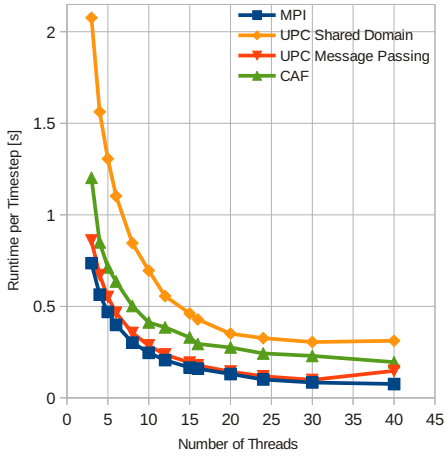
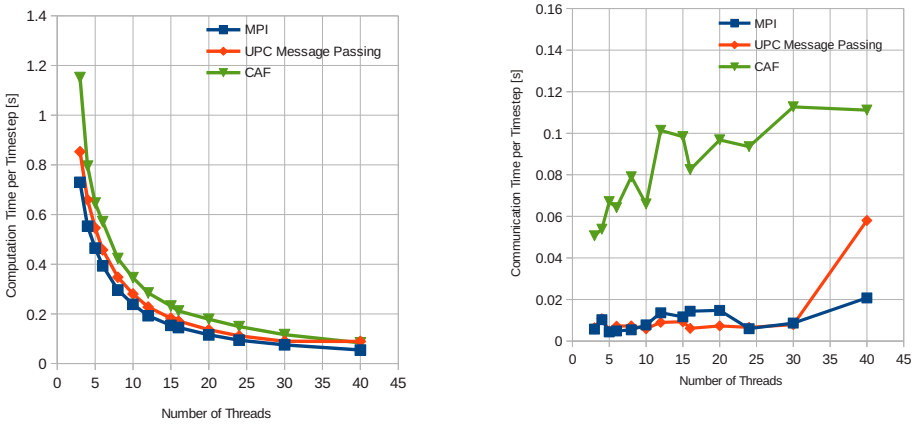


Figure 6: Scaling behavior of total runtime.

The following results are timing measurements from a strong scaling experiment, where slices of the x-coordinate where distributed on different threads. The total number of threads for each run, was chosen such that every thread has the same amount of cells. The runtime results of all implementations are shown in Fig. 6. As expected the MPI implementation in C performs best, since a vendor optimized MPI version from IBM was available on the cluster. The UPC Message Passing variant performs very similar to the MPI version. This is not surprising, since it implements message passing, but using UPC functions insteads of MPI. The UPC compiler is able to map the used `upc_memput` command efficiently to the underlying hardware. CAF generally performed worse than UPC, which is due to a poorer single core performance and due to a lower communication bandwidth. The simple shared domain UPC implementation is about a factor of 2 to 3 slower than the message passing versions. Nevertheless it shows a similar scaling behavior. So with UPC a parallel version can be developed very fast, if one is willing to pay the above mentioned performance penalty.

In Fig. 7a the computation time for each implementation is shown, i.e. the time spent in the force and position update loops, not considering time for communication. In the shared domain UPC implementation this separate measurement is not possible, since there is no clear separation between local and remote memory accesses in the code. Considering

only the pure computation time, the small runtime difference between the MPI and UPC Message Passing variant is due to computation, not communication. Since both implementations differ only in the communication part, this has to be a compiler effect. The two phase UPC compilation process seems to generate slightly less optimized code, compared to directly compiling with the GNU C compiler. The CAF implementation shows a little poorer single core performance than UPC and MPI. We think this effect can be overcome by investing more time on the single core optimization of the CAF version.



(a) Computation time.

(b) Communication time.

Figure 7: Scaling behavior.

Fig. 7b shows the communication times for the different implementations. For the MPI and UPC versions, the communication times stay very small for up to 30 threads compared to the computation times (they are about a factor ten lower). When using the full node there seems to be a problem with the UPC communication method, since the communication time increases drastically, which is not the case for the similar MPI version. The poor communication performance of CAF is explained by [SHK+12], which shows that the CAF implementation of the Intel Fortran Compiler performs element-wise communication even when contiguous Fortran constructs are used in the source code.

5 Conclusion

With UPC, the existing reference implementation was parallelized very fast, compared to MPI. Considering the optimization hints from [CSC+07], a more efficient version is obtained, however the required implementation effort is comparable to MPI.

The co-array concept could successfully be applied to the linked cell algorithm. Like with UPC, the chosen communication strategy would not have required more effort in MPI and would have given a better performance and portability. CAF may be better suited

for stencil codes and other codes that have more regular communication patterns. The poor communication performance we experienced with Intel CAF is due to element-wise communication that the Intel Compiler performs.

Chapel has proven to be a programming language which is easy to use. It turned out, however, that the environment which was used in our tests is inappropriate for Chapel. Therefore no results comparable with CAF, MPI and UPC could be achieved.

As for all three PGAS languages, it is necessary to check whether suitable hardware and libraries are available. In contrast, the de facto standards MPI and OpenMP are widely used and can be assumed to run efficiently on virtually every machine. In our tests, using the PGAS paradigm in CAF and UPC results in worse performance than MPI. The parallel efficiency in UPC highly profited from switching to an MPI-like communication scheme. In this case, however, the effort to arrive at an efficient parallel program is similar to programming in MPI.

References

[BSC+12] H. Burkhart, M. Sathe, M. Christen, O. Schenk, and M. Rietmann. Run, Stencil, Run! HPC Productivity Studies in the Classroom. In *PGAS12*, In Press, 2012.

[CSC+07] S. Chauvin, P. Saha, F. Cantonnet, S. Annareddy, and T. El-Ghazawi. UPC Manual. http://upc.gwu.edu/downloads/Manual-1.2.pdf, 2007.

[GDN98] M. Griebel, T. Dornsheifer, and T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics: A Practical Introduction*. Monographs on Mathematical Modeling and Computation. Society for Industrial and Applied Mathematics, 1998.

[Inc10] Cray Inc. Chapel Language Specification 0.795. http://chapel.cray.com/spec/spec-0.795.pdf, 2010.

[ISO10a] ISO. Coarrays in the next Fortran Standard, ISO/IEC JTC1/SC22/WG5 N1824. ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf, 2010.

[ISO10b] ISO. Fortran 2008 Language Draft, ISO/IEC JTC1/SC22/WG5 N1826. ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1826.pdf, 2010.

[SBG96] B.F. Smith, P.E. Bjorstad, and W.D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[SHK+12] K. Sembritzki, G. Hager, B. Krammer, J. Treibig, and G. Wellein. Evaluation of the Coarray Fortran Programming Model on the Example of a Lattice Boltzmann Code. In *PGAS12*, In Press, 2012.

[TTT+09] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan. Evaluation of UPC programmability using classroom studies. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, PGAS '09*, page 10:1–10:7, 2009.

[Ver67] L. Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review Online Archive (Prola)*, 159(1):98–103, July 1967.