A performance-oriented comparative study of the Chapel high-productivity language to conventional programming environments

Guillaume Helbecque, Jan Gmys, Tiago Carneiro, Nouredine Melab, Pascal Bouvry

A performance-oriented comparative study of the Chapel high-productivity language to conventional programming environments

Guillaume Helbecque
guillaume.helbecque@univ-lille.fr
Université de Lille
CNRS/CRIStAL UMR 9189
Inria Lille-Nord Europe
France

Jan Gmys
Université de Lille
CNRS/CRIStAL UMR 9189
Inria Lille-Nord Europe
France

Tiago Carneiro
University of Luxembourg
FSTM
Luxembourg

Nouredine Melab
Université de Lille
CNRS/CRIStAL UMR 9189
Inria Lille-Nord Europe
France

Pascal Bouvry
University of Luxembourg
DCS-FSTM/SnT
Luxembourg

ABSTRACT

The increase in complexity, diversity and scale of high performance computing environments, as well as the increasing sophistication of parallel applications and algorithms call for productivity-aware programming languages for high-performance computing. Among them, the Chapel programming language stands out as one of the more successful approaches based on the Partitioned Global Address Space programming model. Although Chapel is designed for productive parallel computing at scale, the question of its competitiveness with well-established conventional parallel programming environments arises. To this end, this work compares the performance of Chapel-based fractal generation on shared- and distributed-memory platforms with corresponding OpenMP and MPI+X implementations. The parallel computation of the Mandelbrot set is chosen as a test-case for its high degree of parallelism and its irregular workload. Experiments are performed on a cluster composed of 192 cores using the French national testbed Grid'5000. Chapel as well as its default tasking layer demonstrate high performance in shared-memory context, while Chapel competes with hybrid MPI+OpenMP in distributed-memory environment.

KEYWORDS

Chapel, MPI, Multi-core, OpenMP, Parallel computing, Productivity-awareness

1 INTRODUCTION

Nowadays, we observe a dramatic increase in complexity, diversity and scale of High Performance Computing (HPC) environments. According to the TOP500 bi-annual ranking of the most powerful systems [14], modern supercomputers are increasingly large (millions of cores) and heterogeneous (CPU-GPU). On the other hand, HPC applications and algorithms also tend to be more and more sophisticated. Actually, in order to benefit from the parallelism provided at different levels of the hardware, hybrid hierarchical parallelism is recommended.

This is a reason why there is an increased research interest in software development productivity in HPC [5, 8]. Different programming models/languages, runtimes and libraries need to be used together to efficiently exploit all levels of parallelism. As a consequence, to deal with such complexity, efforts towards productivity are crucial for harnessing the processing power of modern supercomputers.

To this end, the DARPA High Productivity Computing System (HPCS) program represents an effort for creating high-productivity languages for the next generation of supercomputers [10]. Among these languages, Chapel stands out, as it is competitive to both C+OpenMP and MPI+X in terms of performance and scalability [3].

The recent arrival of this language raises the question of its competitiveness with well-established conventional parallel programming environments. For this purpose, this paper provides a comparison point between Chapel and the widely adopted OpenMP and MPI+X parallel programming libraries. The focus is put on the parallel computation of the Mandelbrot set. This embarrassingly parallel application allows massive parallelism, while facing irregular workload.

In this paper, we illustrate the main parallel features of Chapel, MPI and OpenMP using the Mandelbrot test case. Six implementations are proposed using Chapel, C+OpenMP, C+MPI with different communication models (two-sided blocking and non-blocking, one-sided), and a hybrid one, combining C+MPI and OpenMP. Chapel as well as its default tasking layer demonstrate high performance in shared-memory context. Moreover, in distributed-memory environment, it presents similar performance to hybrid MPI+OpenMP, even though we did not explore other features, such as the distributed iterators.

The remainder of the paper is structured as follows. Section 2 presents related works. Section 3 defines the Mandelbrot set computation. Section 4 provides an overview of each parallel programming environment and describes the corresponding implementations. Experimental results are reported in Section 5. Finally, we draw the conclusions in Section 6 and outline some future works.

2 RELATED WORKS

In various domains, empirical studies compare different parallel programming environments for a typical problem in the respective field of research. In [13], Parenteau et al. develop Chapel-based alternatives for two CFD applications. These were compared to conventional MPI-based algorithms, and the competitiveness of Chapel is highlighted. In [6], Gmys et al. are dealing with three productivity-aware languages (Chapel, Julia and Python) to solve the 3D Quadratic Assignment Problem parallel metaheuristic on a multi-core shared-memory computer. The comparison is done in terms of performance, scalability and productivity. Moreover, related works in the context of irregular applications (like Branch-and-Bound tree-search algorithms) show that it is possible to achieve parallel efficiency and performance, but also high productivity, by using the Chapel language [2].

The parallel Mandelbrot set computation is a well-studied problem. Some authors use this test-case to describe how the fundamentals of task parallel programming are dealt with different efficient parallel environments, like Chapel, OpenMP, X10, OpenCL, *etc* [9]. However, the latter focus on illustrating the fundamentals of task parallel programming without performing a performance analysis. Another study conducts performance evaluation on shared-memory architecture using MPI and OpenMP [7], but doesn't include PGAS-based environments.

Our paper aims at providing a useful data point using shared- and distributed-memory multi-core systems for supercomputer programmers. The well-known parallel Mandelbrot set computation is chosen because it is a complete application (not a microbenchmark kernel) that allows easy decomposition into a large number of irregular subproblems, while retaining a relative simplicity. The latter allows us to illustrate the programming effort in each environment by short but complete code snippets and thereby provide to the reader a sense of "productivity", which is a hard-to-evaluate and necessarily somewhat subjective measure.

3 MANDELBROT SET COMPUTATION

In this paper, we consider the parallel computation of the Mandelbrot set as a test-case. It is defined as the set of complex numbers $c = a + ib \in \mathbb{C}$ such that the sequence $(z_n)_{n\in\mathbb{N}} \subset \mathbb{C}$ defined by

$$z_0 = 0, \qquad z_{n+1} = z_n^2 + c, \tag{1}$$

remains bounded in $\mathbb{C}$ (see Figure 1). In practice, we can prove that a point c of the complex plan belongs to the Mandelbrot set if and only if $|z_n| \leq 2,\ \forall n \in \mathbb{N}$. This observation directly yields a simple "escape time" algorithm shown in Algorithm 1. Considering a bounded domain Ω (hereafter called *image*) uniformly cut into pixels. For each pixel $(a, b) \in \Omega$ the algorithm determines the smallest integer n required for observing divergence of $(z_n)_{n\in\mathbb{N}}$. Then the luminous intensity I of pixel (a, b) is defined as

$$I(a, b) = \frac{1}{N} \min_{n=0,\ldots,N} \{n : |z_n| > 2\}, \tag{2}$$

where $N \in \mathbb{N}$ is an arbitrary maximum number of allowed iterations.

This well-known parallel application allows to illustrate the main parallel features and to investigate the parallel efficiency of the programming environments considered in this paper. As we can see in Algorithm 1, the nested `for`-loops can be easily and massively parallelized due to the independence of each pixel. The granularity of the application (amount of work performed per pixel/line of the image) can be controlled by adjusting the maximum number of iterations N. However, one has to deal with the irregular workload resulting from the drastically different amount of work performed per pixel (see Figure 2).

The parallelization of Algorithm 1 is thus based on a domain decomposition. In order to preserve the relative simplicity of the implementation, the decomposition is assumed only along the lines, *i.e.* only the first `for`-loop in Algorithm 1 (line 13) is parallelized. Moreover, in order to achieve some workload regularization, lines are mapped to processing elements in round-robin fashion. For comparison purposes, we require that the computation is deterministic. It means that the domain decomposition is done statically and we know in advance which thread computes which lines.

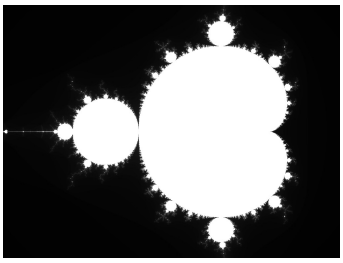

Figure 1: Monochrome Mandelbrot set.

Algorithm 1: Pseudo implementation of the Mandelbrot set computation

```
1  function Compute_pixel(a, b) :
2      x = y = 0;
3      n = 0;
4      while x^2 + y^2 < 4 and n < N do
5          t = x;
6          x = x^2 - y^2 + a;
7          y = 2ty + b;
8          n = n + 1;
9      end
10     I(a, b) = n/N;
11 end

12 function Compute_image() :
13     for a = 0 to nb_lines do
14         for b = 0 to nb_columns do
15             Compute_pixel(a, b);
16         end
17     end
18 end
```

4 PARALLEL PROGRAMMING ENVIRONMENTS

This section presents the studied parallel programming environments and highlights their main features. Six parallel implementations of the Mandelbrot set computation are described using Chapel, OpenMP, MPI with different communication models (two-sided blocking and non-blocking, one-sided), and a hybrid one, combining MPI and OpenMP.

4.1 OpenMP

The OpenMP API (Open Multi-Processing Application Programming Interface) [12] is a collection of compiler directives, library routines and environment variables for shared-memory parallelism in C, C++ and Fortran programs. It is portable across different architectures and supported by numerous compilers. It relies on the Single Program Multiple Data (SPMD) execution model, where a set of tasks share a common address space and are executed in an asynchronous way.

OpenMP directives are added to the code using the `#pragma` preprocessor mechanism. The most common way to introduce parallelism is the `omp parallel` directive which creates a parallel region where concurrent threads execute the same code. The `omp for` work sharing directive allows to distribute iterations of a `for`-loop among threads. Several clauses can be employed to modify the behavior of the work sharing construct, for instance whether loop iterations are distributed using a static scheduling (Master-Worker model) or a dynamic one (Work-Pool model). Round-robin static distribution of lines onto threads is achieved by the `schedule(static,1)` clause, where the parameter 1 represents the chunk size. Another important point is that OpenMP relies on a *fork-join* model, where a set of threads is created entering a parallel region (fork), and is destroyed at the end (join). This model thus implies implicit synchronization mechanisms. A more complete documentation of OpenMP is available in [12].

The OpenMP parallel Mandelbrot generation is described in Algorithm 2. One can see that parallelization is achieved through a minimal and incremental modification of the sequential code (addition of line 2). The composite `omp parallel for` directive is added in order to distribute the iterations of the first `for`-loop (line 3) onto threads.

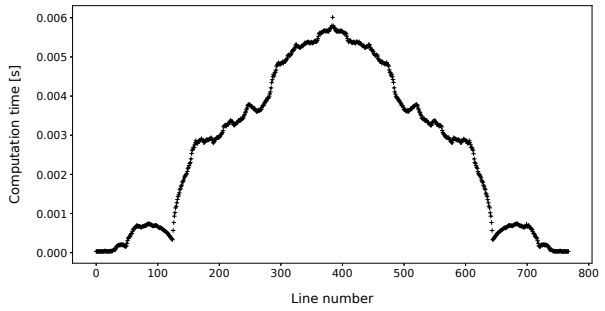

Figure 2: Computation time per line of pixels for an image of size 1024**x**768 **and** $N = 1000$**, on AMD EPYC 7301 CPU @2.20GHz.**

Algorithm 2: Pseudo OpenMP parallel implementation of the Mandelbrot set computation

```
1 function Compute_image_omp() :
2     #pragma omp parallel for schedule(static, 1);
3     for a = 0 to nb_lines do
4         for b = 0 to nb_columns do
5             Compute_pixel(a, b);
6         end
7     end
8 end
```

4.2 Chapel

Chapel (Cascade High Productivity Language) [4] is an open-source high-productivity and high-performance programming language that follows the Partitioned Global Address Space (PGAS) programming model. Chapel allows shared- and distributed-memory executions, and runs on top of the GASNet one-sided communication and active message library. Furthermore, as Chapel belongs to the PGAS class of languages, the application has a global memory addressing space, and each segment of this space is assigned to a different locale [1]. In turn, a locale refers to a unit of the machine resources that can store variables and run Chapel tasks, which is similar to an MPI process. It is also worth to say that the language supports object-oriented design and C or Fortran interoperability features.

In Chapel, parallelism is expressed in terms of tasks, which can be run on one or several locale(s). The program is started with a single task, and parallelism is *added* through data or task-parallel features, such as the `coforall` statement, which is a parallel version of the `for`-loop, introducing task parallelism by creating a distinct task per loop iteration. This feature is suitable to create concurrent tasks, especially when the loop iterations are independent, like in the Mandelbrot set computation. Moreover, it is possible to explicitly control locality via the `on` clause that allows to migrate a task on the specified locale. The number of locales is passed to the implementation using the command line parameter `-nl` L, where L is the number of locales on which the application is executed. The detailed Chapel documentation is available in [4].

The implementation of the Mandelbrot written in Chapel is depicted in Algorithm 3. First of all, a `coforall` statement is used to create as much tasks as locales (line 2). Then, each allocated task is migrated on its associated locale (line 3). Following this, another `coforall` statement is used to create as much tasks as threads per locale (line 4). Finally, each task thus created computes sequentially its lines of pixels. Unlike the OpenMP implementation, the distribution of lines onto threads is done manually, by computing the corresponding indices (line 5).

4.3 MPI

MPI (Message-Passing Interface) [11] is a library interface specification for message-passing on shared- and distributed-memory multi-core computers. MPI is widely used in academic and industrial

Algorithm 3: Pseudo Chapel parallel implementation of the Mandelbrot set computation

```
1  function Compute_image_chpl() :
2      coforall loc = 0 to nb_locales do
3          on loc do
4              coforall rank = 0 to nb_tasks do
5                  for a = loc.id + rank * nb_locales to
                     nb_columns by nb_tasks * nb_locales do
6                      for b = 0 to nb_columns do
7                          Compute_pixel(a, b);
8                      end
9                  end
10             end
11         end
12     end
13 end
```

areas for its portability, standardization and its high performance. MPI defines a set of subroutines, usable in C or Fortran. Typically, a SPMD model is used, where a set of MPI processes, having their own exclusive address space, execute the same program.

Executing an MPI program in parallel consists in launching multiple copies of the same program on a set of specified hosts. The MPI environment can be initialized using the `MPI_Init` routine, enabling interprocess communications. `MPI_Finalize` shuts down the environment and cleans up all MPI-related state. Other process management operations like `MPI_Comm_size` and `MPI_Comm_rank` allow to query the number of MPI processes in a given MPI communicator and the rank of the calling MPI process, respectively.

Initially, the MPI standard (MPI-1) specifies a point-to-point communication model through send and receive operations. Since this latter implies both sender and receiver sides, we usually refer to this model as *two-sided*. Later, MPI-2 introduces new functionalities, like one-sided communications, also known as Remote Memory Access (RMA). MPI one-sided communications are limited to accessing only a specifically declared memory area on the target, called a *window*. Unlike with two-sided operations, the target process doesn't perform any action. The detailed MPI documentation is available in [11].

In the remainder of this section, four MPI implementations are described using different communication models: two-sided with blocking and non-blocking operations, one-sided, and hybrid MPI+OpenMP. Each MPI process will perform the computation of its lines, while only the master process stores the image. This implies communication of data between processes.

4.3.1 Two-sided communications. The two main routines for MPI two-sided communications are `MPI_Send` and `MPI_Recv`. The send/receive buffers, sender/receiver ranks and data types must be explicitly specified by the programmer. Moreover, these are blocking communications involving implicit synchronizations.

The MPI parallel implementation with two-sided blocking communications is described in Algorithm 4. First, the MPI environment is initialized, and each process reads its rank and the number of MPI processes (lines 1-3), which are used to explicitly map processes

Algorithm 4: Pseudo MPI parallel implementation of the Mandelbrot set computation with two-sided blocking communications

```
1  MPI_Init();
2  MPI_Comm_size(commsize);
3  MPI_Comm_rank(rank);
4  for l = rank to nb_lines by commsize do
5      Compute_line(l);
6      if rank ≠ 0 then
7          MPI_Send(/*args*/);
8      else
9          MPI_Recv(/*args*/);
10     end
11 end
12 MPI_Finalize();
```

onto work items (lines of the image). Each MPI process sequentially computes the lines it is assigned to (lines 4-5). Completed lines are send to the master process (rank 0) introducing synchronization points between the latter and all non-zero ranks. When the work is finished, the MPI environment is shut down and the master process writes the image to an output file (not included in time measurings).

The previous MPI algorithm represents the most basic MPI parallel implementation of the Mandelbrot set computation. However, implicit synchronization could induce communication overheads that become significant for fine-grained computations. This is why MPI also includes non-blocking operations. The most basic ones are `MPI_Isend` and `MPI_Irecv`. The use of non-blocking communications leave the user responsible for explicit synchronization, for instance by using the `MPI_Waitall` routine. A variant of Algorithm 4 using non-blocking communications is shown in Algorithm 5.

Algorithm 5: Pseudo MPI parallel implementation of the Mandelbrot set computation with two-sided non-blocking communications

```
1  MPI_Init();
2  MPI_Comm_size(commsize);
3  MPI_Comm_rank(rank);
4  for l = rank to nb_lines by commsize do
5      Compute_line(l);
6      if rank ≠ 0 then
7          MPI_Isend(/*args*/);
8      else
9          MPI_Irecv(/*args*/);
10     end
11     MPI_Waitall();
12 end
13 MPI_Finalize();
```

4.3.2 One-sided communications. When dealing with one-sided communications, the main calls are `MPI_Put` to send data to the window of another process, `MPI_Get` to fetch data from the window of another process, and `MPI_Accumulate` to update data by

combining the existing data and the data sent. Moreover, like in two-sided non-blocking operations, explicit synchronizations are required. It can be achieved via the `MPI_Fence` routine.

Algorithm 6: Pseudo MPI parallel implementation of the Mandelbrot set computation with one-sided communications

```
1  MPI_Init();
2  MPI_Comm_size(commsize);
3  MPI_Comm_rank(rank);
4  MPI_Win_create(win);
5  MPI_Fence();
6  for l = rank to nb_lines by commsize do
7  |   Compute_line(l);
8  |   if rank ≠ 0 then
9  |   |   MPI_Put(/*args*/);
10 |   end
11 |   MPI_Fence();
12 end
13 MPI_Win_free(win);
14 MPI_Finalize();
```

Algorithm 6 shows a variant of the previous MPI-based algorithms using one-sided communications. Similarly, the implementation begins with the initialization steps, followed by the window creation procedure `MPI_Win_create` (line 4). Synchronization on the window (line 5), effectively acting as a global barrier, is required to ensure that the window is completely accessible by every process before performing any operations. The mapping of processes onto work items is identical to the previous MPI-based algorithms. The difference is that the master process is no more involved in the communications. At the end, the windows is freed with `MPI_Win_free` (line 13) and the MPI environment is closed.

4.3.3 Hybrid MPI+OpenMP. In this last section, we focus on a hybrid MPI+OpenMP program to combine the distributed features of MPI with the multi-threaded execution model of OpenMP. The MPI environement is initialized using the `MPI_Init_thread` subroutine with the `MPI_THREAD_FUNNELED` multi-threading support, indicating that MPI calls will only be issued from the master thread of each process.

The hybrid approach is illustrated in Algorithm 7. At most one MPI process is allowed per computer node. This can be done using the `--map-by ppr:1:node` mapping policy. Locally to each node, the computation of lines is done in parallel using OpenMP, like in Algorithm 2 (line 4). Then, MPI is used for the inter-processes communications (lines 5-9). As lines computed per process are send to rank 0 in a single batch, we choose to deal communications with the basic two-sided blocking communication model.

5 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the implementations introduced in Section 4. Section 5.1 introduces the experimental testbed, and Section 5.2 defines the experimental protocol. The performance results in shared- and distributed-memory environments are presented and analyzed in Section 5.3 and 5.4, respectively.

Algorithm 7: Pseudo MPI+OpenMP parallel implementation of the Mandelbrot set computation

```
1  MPI_Init_thread(MPI_THREAD_FUNNELED);
2  MPI_Comm_size(commsize);
3  MPI_Comm_rank(rank);
4  Compute_image_omp(rank,commsize);
5  if rank ≠ 0 then
6  |   MPI_Send(/*args*/);
7  else
8  |   MPI_Recv(/*args*/);
9  end
10 MPI_Finalize();
```

5.1 Experimental testbed

The experiments are carried out using the French national Grid'5000 experimental testbed. All computer nodes operate under Debian GNU/Linux 11 (bullseye), 64 bits and are equipped with *two* AMD EPYC 7301 CPUs @2.20GHz (a total of 32 cores/64 threads per node) and 192 *GB* RAM. All computer nodes are interconnected through a 25 Gbps Ethernet network Intel Ethernet Controller XXV710. The number of computer nodes in the experiments ranges from 1 to 6. Thus, up to 192 processing cores are used in the experiments. Concerning the Chapel implementation, each computer node hosts one Chapel locale. Table 1 presents the tools/libraries and optimization flags used for compiling the programs.

Table 1: Summary of the tools/libraries and optimization flags used for compilation and execution.

Tools/libraries	Version
C compiler *gcc*	10.2.1
Open MPI	4.1.0
OpenMP	4.5
Chapel	1.25.0
C optimization flag	`-O2`
Chapel optimization flag	`--fast`

Chapel's multi-locale code runs on top of GASNet, and the runtime should be built taking account the characteristics of the system on which the multi-locale code is supposed to run. One can see in Table 2 a summary of the runtime configurations for multi-locale execution. The UDP GASNet implementation is the one used for communication (`CHPL_COMM_SUBSTRATE`) along with SSH, which is responsible for getting the executables running on different locales (`GASNET_PSM_SPAWNER`).

5.2 Experimental protocol

For each implementation, the computation time is measured while varying the granularity, controlled by the maximum number of iterations N, that takes its value in $\{100, 1000, 10000, 100000\}$. The image size is fixed to 1024x768 and 5120x3840 for shared- and

Table 2: Summary of the Chapel environment configuration for multi-locale execution.

Variable	Value
CHPL_RT_NUM_THREADS_PER_LOCALE	64
CHPL_TARGET_CPU	*native*
CHPL_HOST_PLATFORM	*linux*64
CHPL_LLVM	*none*
CHPL_COMM	*gasnet*
CHPL_COMM_SUBSTRATE	*udp*
GASNET_PSM_SPAWNER	*ssh*

distributed-memory experiments, respectively. For each experiment, the relative speed-up S, defined by

$$S(p) = \frac{t(1)}{t(p)}, \tag{3}$$

is then measured, where $t(p)$ corresponds to the execution time using p processing units, and $t(1)$ is the corresponding sequential time. The `clock_gettime` timer C function is used, since it is valid for all implementations (using the C interoperability in Chapel). Experiments are performed multiple times, and the average is considered.

5.3 Results on shared-memory system

Figure 3 depicts the relative speed-up measured while varying the number of processing units from 1 to 64. Different maximum number of iterations N are considered in order to see the effect of the granularity. Hyperthreading is allowed for the Chapel and OpenMP implementations, but the overloading of MPI processing elements is disabled (with the `-nooversubscribe` flag).

We can see that the MPI-based implementations suffer from high parallel overheads. Indeed, for low granularity, $N = 100$, the workload is not sufficiently large to counterbalance the MPI communication overheads. This is why speed-up remains bounded by 10. However, when the granularity increases, the MPI implementations are able to take advantage of parallelism to reach a high speed-up, very close to the ideal one ($N = 100000$). In addition, we note that the implementation based on two-sided non-blocking communications performs better than its blocking counterpart. In these experiments, no difference appears between the one-sided and the two-sided non-blocking MPI implementations. Concerning the speed-up of the OpenMP implementation, we observe the same behavior as in MPI, *i.e.*, the application scales poorly for very fine granularity. This is explained by the fact that the *fork-join* model of OpenMP is costly, and need to be counterbalance by a sufficiently large workload. When N increases, the OpenMP speed-up is getting closer to the ideal one. We also note that hyperthreading allows OpenMP to obtain good performance, although the speed-up for 64 threads is quite far from ideal. Finally, interestingly, we can see that the Chapel implementation doesn't suffer much from parallel overheads. Even with a low granularity, we note that Chapel can easily obtain a speed-up that reached 50 using hyperthreading, and scales well with the ideal speed-up when N is growing.

In order to investigate the high performance of Chapel compared to the other implementations, we suggest to measure the computation time when fixing $N = 1$. In accordance with Algorithm 1, it represents the minimum valid value of N for which the workload for each pixel is negligible, almost zero, and thus the computation time reflects the parallel overheads. Experiments are performed fifty times, and the average is considered.

These results are depicted in Figure 4. Firstly, concerning the MPI-based implementations, we can see that for two-sided communications, the computation time is slightly increasing with the number of processing cores, and we note that the removal of implicit synchronizations through non-blocking operations allows a minor gain. Moreover, we observe that the MPI one-sided approach is more expensive than others, due to the fact that this model requires the creation of a window, in addition to communication and synchronization mechanisms. Secondly, we observe also that the OpenMP computation time is growing with N. We note that OpenMP is faster than all MPI-based implementation, except when using hyperthreading. Finally, we observe that Chapel is consistently faster than all others. Interestingly, the computation time is decaying with the number of processing units. This behavior can partly explain the very good Chapel performance in the shared-memory multi-core experiments of Figure 3, whatever the granularity. The Chapel *qthreads* default tasking layer, that provides a lightweight implementation of Chapel tasking as well as an optimized implementation of synchronization variables, is thus very appropriate, and allows high performance [15, 16].

5.4 Results on distributed-memory system

In this section, the parallel speed-up is observed while varying the number of processing units from 1 to 192. MPI processes overloading and hyperthreading are not allowed. The hybrid MPI+OpenMP implementation uses 1, 2, 3, 4, and 6 MPI process(es) with 32 threads each. Communications are performed by process 0. The results are presented in Figure 5.

Firstly, when considering low granularity, $N = 100$, all implementations suffer from high parallel overheads. Each speed-up associated to MPI-based implementation remains bounded by 20. This is also true for the hybrid MPI+OpenMP program. Actually, the OpenMP parallelism cannot counterbalance the MPI overheads generated by the distribution of work onto computer nodes. Regarding Chapel, we observe an almost ideal speed-up for 32 processing units, since it implies only one locale. Indeed, we have seen in the previous section that Chapel is particularly efficient in single locale execution. However, when considering multi-locale executions (more than 32 cores), the performance dramatically drops. When N is growing, each implementation scales well, since the parallel overheads are counterbalance by the workload. The distributed-memory experiments allow to better see the differences between the MPI-based implementations. The two-sided blocking communication model is outperformed by its counterparts, whatever the granularity. Moreover, we note for this application that the one-sided communication model is not as good as the two-sided non-blocking one. Actually, it can be explain by the fact that tasks are independent, and thus few inter-process communications occur. The hybrid MPI+OpenMP program is as good or better than all others, whatever the value of N. In addition, we observe that for high granularity ($N = 100000$ for example), Chapel is able to compete with the hybrid program.

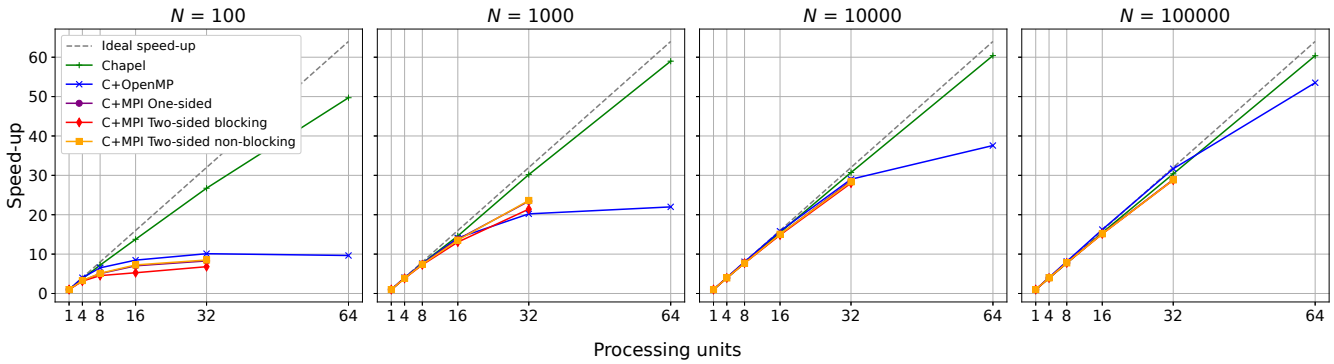

Figure 3: Speed-up achieved by all five shared-memory implementations. The number of processing units varies from 1 to 64 (hyperthreading enabled for Chapel and OpenMP). Results are for different maximum number of iterations N.

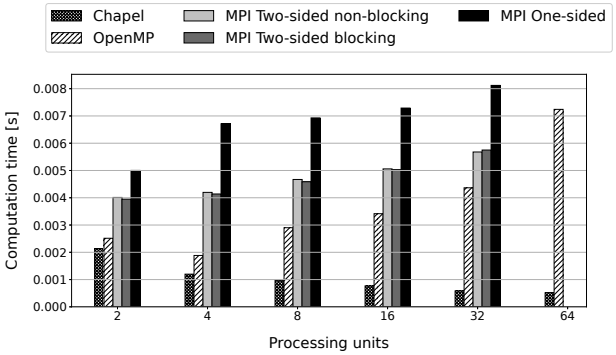

Figure 4: Computational overhead measured by all five shared-memory implementations when considering $N = 1$. The number of processing units varies from 2 to 64 (hyperthreading enabled for Chapel and OpenMP)

As we did in the previous section, we investigate the communication overheads in the distributed-memory experiments by fixing $N = 1$. Again, experiments are performed fifty times, and the average is considered.

The results are depicted in Figure 6. First of all, we can see that for two-sided MPI, the communication overheads for both blocking and non-blocking communication models are approximately the same. Moreover, we note a light increase when the number of processing units growths. For one-sided MPI, this last observation is also true. The higher the number of processing units, the higher the communication overheads. However, compared to two-sided MPI, the latter are 2 times higher for 96 and 128 processing units, and almost 6 times higher for 192. Like in the shared-memory experiments, it can be attributed to the window management, that requires creation, communication and synchronization mechanisms. Concerning the hybrid MPI+OpenMP approach, we can see that the overheads remain constant, whatever the number of processing units. This can be justified by the fact that only one MPI process is allowed per computer nodes. Finally, we note that the communication overheads of Chapel in our distributed-memory experiments are high. Indeed, from 3 computers nodes (96 processing units), overheads are 10 to 21 times higher than the ones observe using the hybrid model. Chapel may suffer from the fact that our cluster is not equipped with high-performance network between nodes. This last observation can explain the poor performance of Chapel in distributed-memory when N is low (see Figure 5).

6 CONCLUSION

In this paper, we have compared the Chapel high-productivity programming language to the well-established conventional parallel programming libraries OpenMP and MPI+X, in terms of performance. Shared- and distributed-memory multi-core experiments were conducted on a cluster composed of 192 processing cores, using the French national testbed Grid'5000.

In this work, the embarrassingly parallel computation of the Mandelbrot set was chosen as a test-case for its high degree of parallelism and its irregular workload. This study represents a good comparison element between Chapel and its OpenMP and MPI counterparts. However, this study should be extended to more complicated problems, involving dependent tasks, and thus more communication between processing units.

Chapel outperforms its counterparts in shared-memory context. This may be explain by its *qthreads* default tasking layer that provides a lightweight implementation of Chapel tasking as well as an optimized implementation of synchronization variables. Moreover, in distributed-memory environment, Chapel competes with hybrid MPI+OpenMP, even though its *qthreads* tasking layer seems to suffer from the lack of high-performance network of the cluster.

Finally, we plan to investigate the use of Chapel and other high-productivity languages for more complex applications, in particular irregular ones (such as tree-search algorithms). Another important aspect is GPU programming support. Although there exist two modules that facilitate GPU programming in Chapel (`GPUIterator` and `GPUAPI`), they are not yet mature.

ACKNOWLEDGEMENT

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted

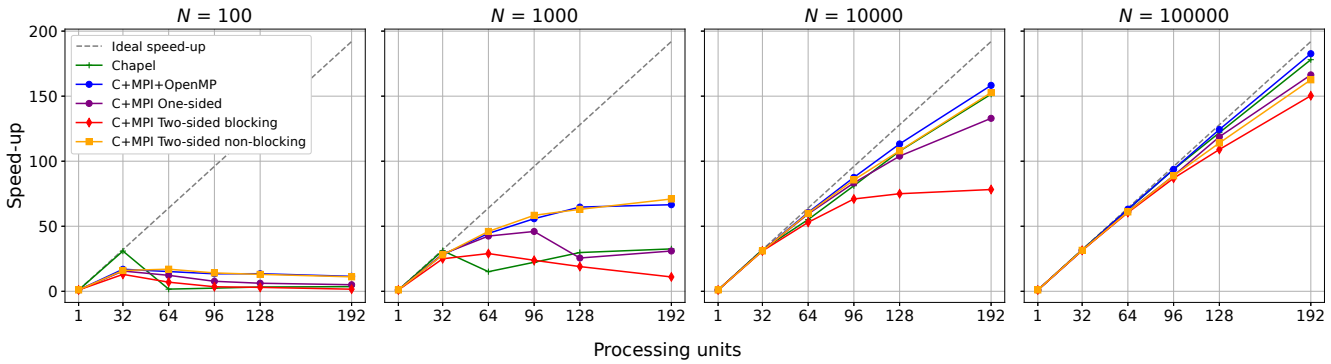

Figure 5: Speed-up achieved by all five distributed-memory implementations. The number of processing units varies from 1 to 192 using 6 computer nodes (hyperthreading disabled for Chapel and OpenMP). Results are for different maximum number of iterations N.

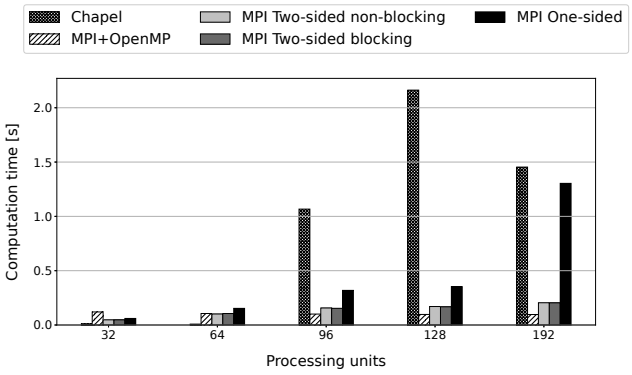

Figure 6: Computational overhead measured by all five distributed-memory implementations when considering $N = 1$. The number of processing units varies from 32 to 192 using 6 computer nodes (hyperthreading disabled for Chapel and OpenMP).

by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] George Almasi. 2011. PGAS (partitioned global address space) languages. In *Encyclopedia of Parallel Computing*. Springer, 1539–1545.

[2] T. Carneiro, J. Gmys, N. Melab, and D. Tuyttens. 2020. Towards ultra-scale Branch-and-Bound using a high-productivity language. *Future Gener. Comput. Syst.* 105 (2020), 196–209.

[3] B. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, P. Sahabu, and G. Titus. 2018. Chapel comes of age: Making scalable programming productive.

[4] Chapel 2022. *The Chapel Parallel Programming Language.* https://chapel-lang.org

[5] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, and L. Votta. 2004. Measuring HPC productivity. *International Journal of High Performance Computing Applications* 18 (2004).

[6] J. Gmys, T. Carneiro, N. Melab, E.-G. Talbi, and D. Tuyttens. 2020. A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. *Swarm and Evolutionary Computation* 57 (June 2020).

[7] E. Gomez. 2020. MPI vs OpenMP: A case study on parallel generation of Mandelbrot set.

[8] J. Kepner. 2004. HPC Productivity: An Overarching View. *The International Journal of High Performance Computing Applications* 18, 4 (2004), 393–397.

[9] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. 2013. Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages. In *Languages and Compilers for Parallel Computing*, Hironori Kasahara and Keiji Kimura (Eds.). Springer Berlin Heidelberg, 174–189.

[10] E. Lusk and K. Yelick. 2007. Languages for High-Productivity Computing: the DARPA HPCS Language Project. *Parallel Processing Letters* 17 (2007), 89–102.

[11] Open MPI 2022. *Open MPI: Open Source High Performance Computing.* https://www.open-mpi.org

[12] OpenMP API 2022. *The OpenMP API specification for parallel programming.* https://www.openmp.org

[13] M. Parenteau, S. Bourgault-Cote, F. Plante, E. Kayraklioglu, and E. Laurendeau. [n.d.]. *Development of Parallel CFD Applications with the Chapel Programming Language.*

[14] TOP500 ranking 2022. *TOP500.* https://www.top500.org

[15] K. Wheeler, R. Murphy, D. Stark, and B. Chamberlain. 2011. The Chapel Tasking Layer Over Qthreads.

[16] K. Wheeler, R. Murphy, and D. Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8.