



Parallelization Using a PGAS Language such as X10 in HYDRO and TRITON

Marc Tajchman^{*a}

^a Commissariat à l'énergie atomique et aux énergies alternatives – CEA/DEN/DM2S/STMF/LGLS
91191 Gif-sur-Yvette, France

Abstract

In this whitepaper, after an introduction to X10, one of the PGAS languages, we describe the different parallelization paradigms used to write versions of two computing codes in this language. For HYDRO, a 2D hydrodynamics code, we started from the original sequential C version. We keep the global 1D alternating direction method, thanks to the logical global addressing scheme for distributed array in PGAS languages. Remote activities (or threads) of X10 were used to distribute work tasks between the different nodes. Local activities of X10 allow us to distribute local computations between cores on the same node. The only communication steps are the computation of the global time step and a global 2D array transposition. So we do not think that this scheme will be scalable on a large set of nodes. TRITON, a simulation platform that performs 3D hydrodynamics computations, was parallelized using a standard 3D domain decomposition method. We implement a specialized distributed array class, by extending the standard X10 array class to transparently handle ghost cells. Again, local and remote activities distribute computing tasks on all the cores. This scheme should show better scalability behaviour. These code porting actions show the flexibility and ease of programming of PGAS languages, even as the absolute performances of our PGAS implementations cannot rival the efficiency of current MPI implementations.

1. Introduction

We will only summarize here the main features of X10, the PGAS language used to port the two codes. For a more complete survey, reference [1] may be consulted.

The first feature is the possibility from one place (i.e. process) to start an activity (i.e. a thread) that runs on another place. An activity can also launch another activity on the same place. The examples in Figure 1 and Figure 2, show two algorithms that call the S function for every member q of a distribution D .

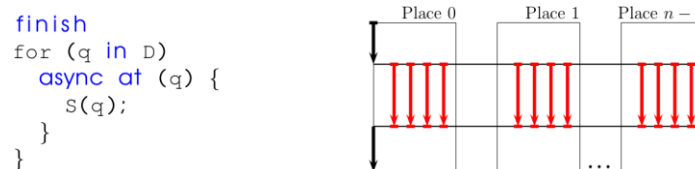


Figure 1: Single level parallelism: the initial activity (on place 0) launches a set of (local and remote) child activities (D is distributed on a set of places)

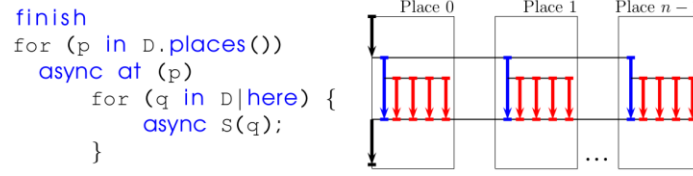


Figure 2: Multi-level parallelism: on each place, one activity is (remotely) launched and starts a set of local sub-activities

The language allows synchronous or asynchronous activities, and offers several ways to synchronize these. The compiler will statically determine local data needed to transfer before starting remote activities. This allows a greater flexibility when designing the parallel architecture of a computing code, but may imply hidden transfers between different places (processes). Optimizing these transfers is a key factor to increase the code performances.

The second feature is the possibility to define distributed arrays. Three steps are needed to build an array:

- define the set of global indices (i,j,k) in 3D space (or (i,j) in 2D space) where the array contains valid coefficients: *the region*
- partition the region between the places: *the distribution*
- reserve, at each place, a memory space where to put the local coefficients: *the array*

Each place (or process) contains some of the coefficients of a distributed array (possibly none), and the access, from this place to these coefficients, is standard: $z = A(i,j,k)$. The access from a place to a coefficient that doesn't belong to that place, must use the *at* keyword: $z = at(P) A(i,j,k)$ where P is a place. This expression can be interpreted as a remote activity launched at place P to read and retrieve a coefficient. The language allows transferring blocks of coefficients with a similar instruction. For performance reasons, block transfers must be used as much as possible instead of multiple scalar transfers.

2. Parallelization of HYDRO, the first code

We will not discuss here the physical, mathematical or numerical aspects of this algorithm (for more information, see reference [3]). The algorithm used in the sequential version of the first code, is, in broad outline, shown below:

- Apply the initial values to the array of computed values
- While the end simulation time has not been reached:
 1. Apply the boundary conditions in X direction
 2. Compute a time step suitable for numerical stability
 3. Use a 1D numerical scheme in the X direction
 4. Transpose the array
 5. Apply the boundary conditions in Y direction
 6. Use a 1D numerical scheme in the Y direction
 7. Transpose the array

Steps 3 and 5 contain most of the local computations (numerical scheme, physical laws application, etc.). They consist in a series of independent 1D computation (in the X direction for step 3, and in the Y direction for step 5). During step 3 (resp. 5), whole lines (resp. columns) are located at the same place.

So the data distribution of the computed values is different during steps 1-3 and 5-6:

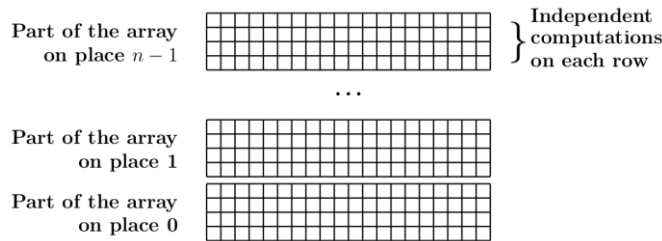


Figure 3: Distribution of arrays during the X-direction scheme

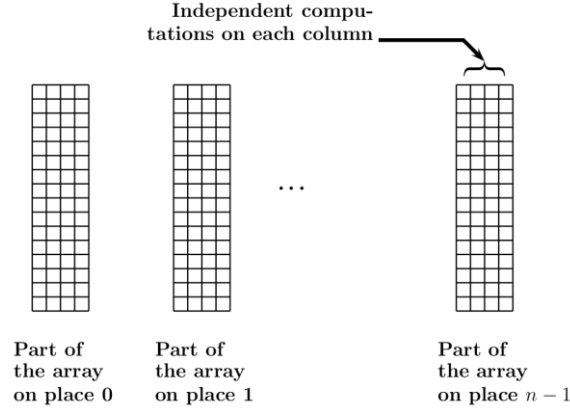


Figure 4: Distribution of the array during the Y-direction scheme

These steps are separated by a (global) transposition step.

3. Performance and scaling tests on Hydro

First, we compare the absolute performance of our X10 version vs. an existing MPI version (in C language), on a sequential test-case, and on 4 processes (places in X10). The X10 version was tested using socket- and MPI-based runtimes (both provided in the X10 distribution). Results are much better than in previous X10 compiler/runtime versions, but are still behind those of the MPI version, see Table 1, below.

Code version	1 process or place	4 processes or places
MPI	15,120 s	4.793 s
X10 (on a socket-based runtime)	121,292 s	31,907 s
X10 (on a MPI-based runtime)	37,355 s	9,958 s

Table 1: Time to solution comparison between MPI and X10 versions (1000x1000 points, 10 time steps).

These tests run on a cluster of our department (36 nodes, each containing 2 Intel Xeon X5667 processors with 4 cores). We first used this machine to develop and debug our code. The other tests presented in this document run on the CCRT Titane parallel machine.

On small sets of computing nodes (up to 25 processors, each containing 8 cores), strong scaling behaviour was reasonably good (see Figure 5). But, we consider that the transposition steps and the time step computation will not scale properly for larger sets. The latter operation is needed by the numerical scheme (to preserve the numerical stability of the solution). The first operation is a consequence of the parallelization method used. So we tested, for the second code, another parallelization method which is more like the standard implementations.

We didn't test the weak scaling behaviour of this code. The reason is that the entire internal loops will always be executed inside one computing node. And some of these loops will have the maximum 1D size of the domain. As the domain grows, it will be impossible to keep a bounded computing work in each computing node.

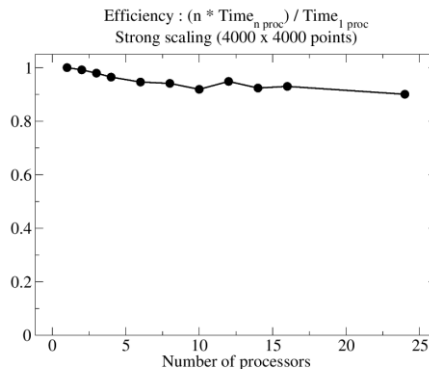


Figure 5: Strong scaling behaviour with Hydro (up to 25 8-cores processors)

4. Parallelization of the Triton platform

Triton is an in-house simulation platform for numerical CFD computations developed at DM2S, a department of simulation and modelling of CEA (see reference [2]). This software uses an alternating direction numerical scheme similar to the previous code, in 3 space dimensions.

We used a classical method of parallelization: after partitioning the global domain, we attribute each sub-domain to a different place (process). To organize the data exchanges, we maintain buffers of “ghost cells” that contains a copy of values located on neighbour places and synchronize these values between computation steps.

The oriented-object nature of the X10 language allows us to design an extended distributed array class. In this class, array values on ghost cells synchronization are taken care of by the array class (the number of ghost cells layers needed depends on the numerical scheme and is specified by the user at array creation), see Figure 6.

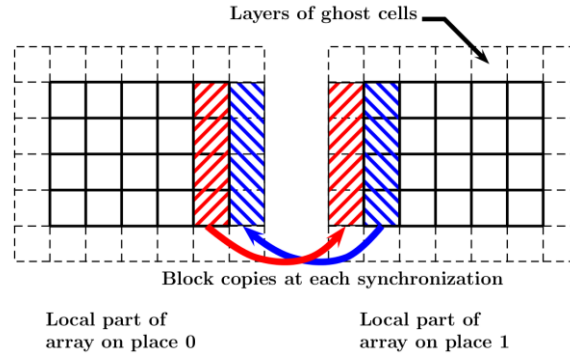


Figure 6: Synchronization of ghost cells on an extended distributed array

The new array class let the developer use global indexes with local-only accesses to data during the computation steps. Similar classes have been independently developed by several authors (see, for example [4]).

To illustrate the difference between a standard distributed array class in X10 and our extended class, let us show code snippets.

Using a standard class to represent the distributed array of unknowns U,

```
val U = new DistArray(...)
finish                                     // Put a barrier after the loop
for P in places()                         // Launch an activity on each place
  at (P) async {                          // to compute the local solution
    val R = U.dist() | here;              // R : local region (indexes of
                                          // components located at place P)
    for ((i,j,k) in R)
      U(i,j,k) += h*(at(U.dist(i+1,j,k)) U(i+1,j,k) +
                    at(U.dist(i-1,j,k)) U(i-1,j,k) + ...
  }
```

We have to use the “at” operator here because, for some components, neighbour components may be located at other places. The previous code snippet will induce a lot of scalar data exchanges across the borders places.

With the extended class, the code can be rewritten as:

```
val U = new ExtendedDistArray(...)

U.synchronize();                          // Update ghost values
finish                                    // Put a barrier after the loop
for P in places()                         // Launch an activity on each place
  at (P) async {                          // to compute the local solution

    val R = U.dist() | here;              // R: local region (indexes of
                                          // components located at place P)
```

```

for ((i,j,k) in R)
    U(i,j,k) = U(i,j,k) + h*( U(i+1,j,k) -
                               U(i-1,j,k) + ...
    }

```

These code snippets (and the Triton source code) read and modify local components, but only read components in ghost cells.

The internals of the new class will return local values or ghost values (it throws an error if the indexes are outside the union of local and ghost regions). It is mandatory to call the synchronize method of the array to be sure that ghost cells contain an updated copy of the remote values.

The “at” keyword is not needed any more for remote values that are mirrored in ghost regions, but must still be used for those located at greater distance. So, to have local-only access during computation steps, the depth of the ghost regions must depend on the numerical scheme, and will be specified by the user of the class. This class delegates all the remote accesses to the synchronization method (where remote data can be packed before transfer and unpacked after it).

For example, if A is distributed structure, of type ExtendedDistArray, and

- A(i,j,k) is located on a place p
- (i,j,k) is also a point of a ghost region q, neighbour of p

```

1  at (q) {
2      val x = A(i,j,k) // retrieves a ghost value located on place q
3      val y = at (p) A(i,j,k) // retrieve the original value on place p
4  }

```

Line 2 involves only local data movements, line 3, instead, will imply a remote data access. If, and only if, the synchronization method of the class has been called after the last modification of A(i,j,k), both operations will return the same value.

Also, ghost values are read-only: any attempt to modify a ghost value outside the synchronization method, will throw an exception.

A working version of the platform using this extended class has been written and tested. It needs to be better optimized (e.g. the tests to see if a coefficient is local, in a ghost region or remote region).

We expect better scalability behavior than for the first programming model tried, provided that the local regions (excluding ghost cells) remain large enough.

5. Performance and scaling tests on Triton

We run the X10 version of Triton on two test cases. The first one uses a fixed discretized domain (400 x 400 x 20 points). It’s a strong scaling experiment on a small set of computing nodes (between 1 and 25), see Figure 7.

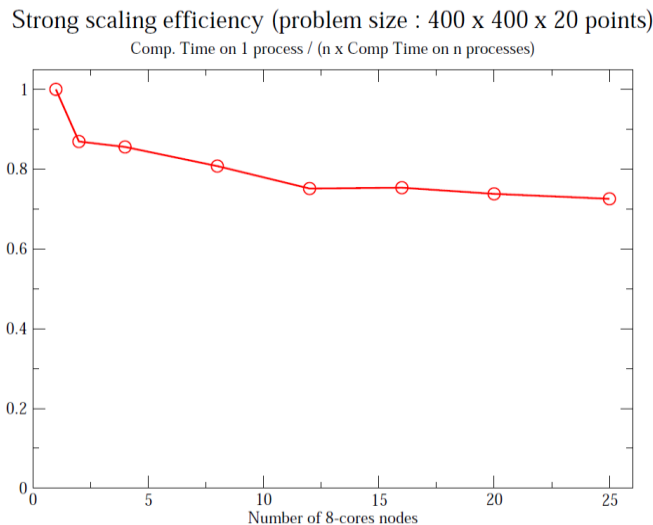


Figure 7: Strong scaling test with Triton

The domain size (and number of unknowns) in the second test case depends on the number of computing nodes (i.e. a fixed-size sub-domain is assigned to each node). It's a weak scaling experiment.

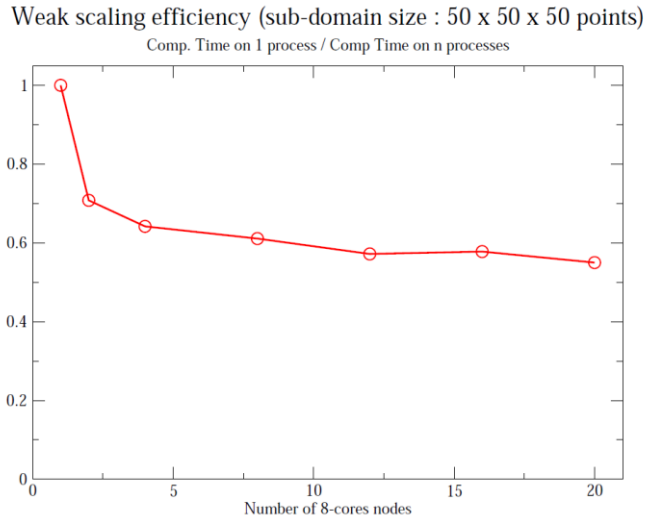


Figure 8: Weak scaling test with Triton

The efficiency gap between 1 and 2-nodes runs suggests that our implementation needs to be better optimized.

6. Conclusion and comments

Two mid-sized computer programs (1000 to 10000 lines for the sequential versions in C), have been ported to X10, one of the PGAS languages. We have tested them on use cases of the original versions, and on small sets of computing nodes. Results have been successfully compared to those of the original versions.

We list here what we have learned about programming with a PGAS language during the development of X10 code versions.

First, as general considerations, we can note that:

- The easiest and fastest way to build a code using a PGAS language is to start from a sequential version. The structure of the PGAS code versions is very similar to the original sequential version, especially if we use “transparent” accesses to remote data. But the last feature may lead to a very inefficient code, with a lot of hidden movements of small packets of data between processors. See next item of this list.
- To obtain more performance, we must pack as much as possible data exchanges between two places (processes). In practice, the developer must maintain buffers for data exchanges so that the code will not be as simple as we can expect.
- The X10 language (the same characteristics apply to Chapel, another PGAS language) offers a very fine control on the parallel tasks and data distributions. The ability to start threads in the same process or in a remote process facilitates the conception of the control flow during the execution. Other languages, Co-Array Fortran, UPC, or XcalableMP are less versatile, but also easier to learn for “PGAS programming style” beginners.
- A very interesting aspect of PGAS languages is that there is much less possibilities of “dead locks”. The PGAS runtime will be in charge of the “details” of organizing the communications.
- Still, the developer has to verify that there are no data races, e.g. by correctly using synchronization mechanisms provided by the language. In the codes we consider here, control flow is simple; we encounter no difficulties with data races. This may be more challenging for more complex codes.

Next, on the X10 version of Hydro:

- Our method of parallelization seemed, at first sight, interesting because all data exchanges (apart from time step computations), are grouped in the transposition steps. We can measure, for small sets of computing nodes, the expected efficiency.
- But, the method will be limited to cases where a whole line or column of the problem fits in a single computing node. This is not characteristic of PGAS languages; this limitations arise from the parallelization scheme used and does not depend on the chosen programming language.
- So, we do not expect good scaling behaviour for larger problems, and we decided to use another method with Triton.

And, on the X10 version of Triton:

- The extended array class simplifies a lot the algorithm implementation, by hiding accesses to values in ghost regions.
- Current implementation of the new class is functional but needs further optimization. This is clearly visible on Figure 7 and Figure 8 (see the gap between the efficiency on 1 and 2 nodes).
- This parallelization method should be more scalable (as the internal computations at each places can be lower- and upper- bounded), and the communication pattern is simple.

Measured time to solution shows fair scalability behaviour (on the limited sets of nodes) but remains behind MPI versions (see Table 1). There are probably multiple reasons for that, but, at least, our X10 source needs to be optimized and compiled properly (without assertions and checks). Also, next versions of the X10 compiler must and will probably generate more efficient intermediate C++ code.

The main advantages of PGAS languages for developers are the easiness to implement task and data parallelism paradigms, the flexibility of the remote data accesses and computations. The main drawback is the performances of the binaries generated by current PGAS tools. It remains us to check the scalability of our developments on larger sets of nodes.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° RI-283493.

We used a local cluster (36 nodes, each containing 2 Intel Xeon X5667 processors with 4 cores, i.e. a total of 288 cores) located at the Department of Modelling and Simulation Software, Nuclear Energy Division, CEA, France, and the Titane supercomputer of CCRT, France.

The sequential and MPI versions, in C language, of the first code were provided by IDRIS, an HPC centre of CNRS, France. The other software, Triton, is a proprietary code of CEA.

References

1. X10 language specification, <http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf>
2. S. Kokh, Simulation of Multi-Material Compressible Flows with Interfaces, in proceedings of the Congrès SMAI-Guidel, 2011 (<http://smai.emath.fr/smai2011/slides/kokh/Slides.pdf>)
3. R. Teyssier, The RAMSES code and related techniques (I Hydro solvers), in proceedings of HIPACC 2010, http://hipacc.ucsc.edu/html/HIPACCLecture/lecture_hydro.pdf
4. J. Milthorpe and A.P. Rendell (2012). Efficient update of ghost regions using active messages, (preprint, to appear), in proceedings of the 19th IEEE International Conference on High Performance Computing (HiPC), http://cs.anu.edu.au/~Josh.Milthorpe/publications/Milthorpe2012_HiPC.pdf.