A Current Task-Based Programming Paradigms Analysis

Jérôme Gurhem[1,2][0000−0002−0741−9766] and Serge G. Petiton[1,2][0000−0002−8423−3682]

[1] Univ. Lille, CNRS, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
[2] Université Paris-Saclay, UVSQ, CNRS, CEA, Maison de la Simulation, 91191, Gif-sur-Yvette, France {jerome.gurhem,serge.petiton}@univ-lille.fr

Abstract. Task-based paradigm models can be an alternative to MPI. The user defines atomic tasks with a defined input and output with the dependencies between them. Then, the runtime can schedule the tasks and data migrations efficiently over all the available cores while reducing the waiting time between tasks. This paper focus on comparing several task-based programming models between themselves using the LU factorization as benchmark.
HPX, PaRSEC, Legion and YML+XMP are task-based programming models which schedule data movement and computational tasks on distributed resources allocated to the application. YML+XMP supports parallel and distributed tasks with XscalableMP, a PGAS language. We compared their performances and scalability are compared to ScaLAPACK, an highly optimized library which uses MPI to perform communications between the processes on up to 64 nodes. We performed a block-based LU factorization with the task-based programming model on up to a matrix of size 49512×49512. HPX is performing better than PaRSEC, Legion and YML+XMP but not better than ScaLAPACK. YML+XMP has a better scalability than HPX, Legion and PaRSEC. Regent has trouble scaling from 32 nodes to 64 nodes with our algorithm.

Keywords: Parallel and distributed programming paradigms · Task-based programming models · Supercomputers.

1 Introduction

Task-based programming models are an interesting alternative to the Message Passing Interface (MPI) [9]. MPI is widely used and very efficient on the current architectures. However, MPI may not be a solution efficient enough on exascale machines, especially in terms of fault tolerance and check-pointing [16]. Task-based approach can help in managing fault tolerance and check-pointing since the tasks could be restarted on another location and data from tasks saved at any moment. The goal of this paper is to analyze and compare the task-based programming models and languages between themselves.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50426-7_16

While MPI focus on exchanging data between processes, other programming models may be efficient to parallelize applications with good scalability without being held back by global synchronizations. For instance, Partitioned Global Address Space Languages (PGAS) programming models which let the users see the distributed memory as a global memory address space that is partitioned across each processing element [5] is an alternative to MPI. Task-based programming models which allows to define fine-grain tasks (computations) on a specific set of data (input and output) are also an alternative. Runtime systems which can optimize execution are another one. Moreover, task-based programming models (first level of parallelism) combined with coarse-grain tasks implemented in a PGAS language (second level of parallelism) can also be one of them. Usually, fine-grain tasks use one process (eventually multi-threaded) whereas coarse-grain tasks perform on several processes (eventually with distributed memory).

In Section 2, we will describe the languages we are using to implement the application and how we did it. We focus on a PGAS language, task-based programming models and a combination of the two. Furthermore, we will present the performed experimentations and the results we obtained in Section 3.

2 Programming Paradigms

In this section, we present and use several languages to implement a LU factorization. MPI, a message passing library, and XcalableMP, a PGAS language, are used to implement a regular and distributed LU factorization (non blocked). Legion, PaRSEC, HPX and YML+XMP are used to implement the block-based version in which tasks make matrix operations on the sub-blocks of the matrix with a subset of the processes allocated to the application. Legion, PaRSEC, HPX are programming models based on a graph of task and YML+XMP is based on a graph of parallel and distributed tasks. They will be succinctly described afterwards.

2.1 MPI

The Message Passing Interface (MPI) [9] is a standardized norm designed to work on a wide variety of parallel computers. It defines the syntax of the core library routines to write message-passing applications. The application uses several processes to make computation at the same time on different cores and uses MPI to send data from one process to another one. MPI has several implementations (OpenMPI, MPICH2, IntelMPI, . . .) which consist of a set of routines that can be called from C, C++ and Fortran.

The MPI library interface includes point-to-point send/receive operations, aggregate functions involving communication between all processes, synchronous nodes (barrier operations), one-way communication, dynamic process management and I/O operations. MPI provides synchronous and asynchronous routines as well as blocking and non-blocking operations. Collective routines involve communications between all processes in a group, for instance MPI_Bcast (broadcast that sends an array to all of the other processes).

MPI can be used with shared memory programming models like OpenMP or with libraries to send data and make computations on CPUs like CUDA. This hybrid model is called MPI+X.

2.2 XcalableMP

XcalableMP (XMP) [13] is a directive-based language extension for C and Fortran, which allows users to develop parallel programs for distributed memory systems easily and to tune the performance by having minimal and simple notations. XMP supports (1) typical parallelization methods based on the data-/task-parallel paradigm under the "global-view" model and (2) the co-array feature imported from Fortran 2008 for "local-view" programming.

The Omni XMP compiler translates an XMP-C or XMP-Fortran source code into a C or Fortran source code with XMP runtime library calls, which uses MPI and other communication libraries as its communication layer.

2.3 Legion (Regent)

Legion [1] is a data-centric parallel programming model. It aims to make the programming system aware of the structure of the data in the program. Legion provides explicit declaration of data properties (organization, partitioning, privileges, and coherence) and their implementation via the logical regions. They are the fundamental abstraction used to describe data in Legion applications. Logical regions can be partitioned into sub-regions and data structures can be encoded in logical regions to express locality describing data independence.

Regent [15] is a programming model which simplifies Legion. Regent compiler translates Regent programs into efficient implementations for Legion. It results in programs that are written with fewer lines of codes and at a higher level.

2.4 PaRSEC

PaRSEC [3] [2] (Parallel Runtime Scheduling and Execution Controller) is an engine for scheduling tasks on distributed hybrid environments.

It offers a flexible API to develop domain specific languages. It aims to shift the focus of developers from repetitive architectural details toward meaningful algorithmic improvements. Two domain specific languages are supported by Parsec, the Parameterized Task Graph [6] (PTG) and Dynamic Task Discovery [11] (DTD)

2.5 YML+XMP

YML [8] is a development and execution environment for scientific workflow applications over various platforms, such as HPC, Cloud, P2P and Grid with multilevel of parallelism. YML defines an abstraction over the different middlewares, so the user can develop an application that can be executed on different middlewares without making changes related to the middleware used. YML can be adapted to different middlewares by changing its back-end. Currently, the proposed back-end [17] uses OmniRPC-MPI [14], a grid RPC which supports master-worker parallel and distributed programs based on multi SPMD programming paradigms. This back-end is developed for large scale clusters such as Japanese K-Computer [17]. A back-end for peer to peer networks is also available.

For the experiments, we use XMP to develop the YML components as introduced in [17]. This allows two levels programming. The higher level is the graph (YML) and the second level is the PGAS component (XMP). In the components, YML needs complementary information to manage the computational resources and the data at best : the number of XMP processes for a component and the distribution of the data in the processes (template). With this information, the scheduler can anticipate the resource allocation and the data movements. The scheduler creates the processes that the XMP components need to run the component. Then each process will get the piece of data which will be used in the process from the data repository.

2.6 HPX

High Performance ParalleX (HPX) [12] is a C++ Standard Library for Concurrency and Parallelism. HPX API implements the interfaces defined by the C++11/14/17/20 ISO standard and respects the programming guidelines used by the Boost collection of C++ libraries. It also extends the C++ Standard APIs to the distributed case. It aims to improve the scalability of current applications. It also tries to expose new levels of parallelism which are necessary to take advantage of the future systems.

HPX is an open-source implementation of the ParalleX execution model. This model focuses on overcoming the four main barriers to achieve scalability (Starvation, Latencies, Overhead, Waiting for contention resolution).

3 Performance Experiences

3.1 Cluster description

The tests were performed on Poincare, the cluster of *La Maison de la Simulation* in France. It is an IBM cluster mainly composed of iDataPlex dx360 M4 servers, hosted at IDRIS, the CNRS supercomputer centre in Saclay, France. There is 77 compute nodes with 2 Sandy Bridge E5-2670 processors (8 cores each, so 16 cores per nodes) and 32 GB of RAM. The file system is constituted of two parts : a replicated file system with the homes of the users and a scratch file system with a faster access from the nodes. The network is based on QLogic QDR InfiniBand.

3.2 Experiments details

We performed performance tests on up to 64 nodes of Poincare with the LU factorization implemented via MPI, ScaLAPACK, XMP, YML+XMP, HPX, PaRSEC and Regent. We used several sizes of matrices : 16384×16384, 32768×32768 and 49512×49512. 16384×16384 is the largest size we can use to perform the tests on one node since YML+XMP cannot perform the LU factorization with greater sizes of matrices on one node.

In HPX, PaRSEC and Regent, the performances depends on the number of blocks in each dimension (thus, the size of the blocks). We used several values for the number of blocks. Table 1, Table 2 and Table 3 show the block parameters which obtained the fastest execution time for each size of matrix. The execution times shown here are the case in which we obtained the fastest time for each number of nodes. We performed those test several times and computed the execution times mean of the same case. We will compare the results of the task-based programming languages to those obtained with ScaLAPACK. We will also compare them to our MPI and XMP implementations. Tests were run on several number of nodes in order to extract strong scaling information which will be discussed in Section 3.4. We used 1, 2, 4, 8, 16, 32 and 64 nodes to factorize the 16384×16384 values matrix. Then, we used 4, 8, 16, 32 and 64 nodes for the 32768×32768 values matrix. And finally, we used 8, 16, 32 and 64 nodes for the 49512×49512 values matrix.

Table 1. Number of blocks for the fastest case on a 16384×16384 matrix with number of processes per tasks between parenthesis

	1	2	4	8	16	32	64
HPX	$90^2(1)$	$45^2(1)$	$80^2(1)$	$45^2(1)$	$45^2(1)$	$55^2(1)$	$55^2(1)$
PaRSEC	$150^2(1)$	$200^2(1)$	$70^2(1)$	$120^2(1)$	$210^2(1)$	$240^2(1)$	$250^2(1)$
Regent	$50^2(1)$	$50^2(1)$	$50^2(1)$	$35^2(1)$	$40^2(1)$	$35^2(1)$	$30^2(1)$
YML+XMP	$4^2(8)$	$8^2(8)$	$8^2(16)$	$8^2(32)$	$4^2(128)$	$4^2(128)$	$4^2(128)$

Our MPI application is MPI-only so we used MPI support for shared memory and used one MPI rank per core i.e. 16 processes per core.

ScaLAPACK has a MPI only distributed implementation so it is run with one MPI rank process per core.

Our XMP implementation only uses pure XMP directives which are converted to MPI calls. It is launched as a MPI only application with one MPI rank process per core.

Regent is a compiler that translates a Lua based code into Legion. Regent applications are launched by passing the MPI command to Regent launcher which will compile and run the application. It creates a Legion worker on each node. Each one of them spawns a process to manage the local tasks, a process to

Table 2. Number of blocks for the fastest case on a 32768 × 32768 matrix with number of processes per tasks between parenthesis

	4	8	16	32	64
HPX	90^2(1)	90^2(1)	90^2(1)	75^2(1)	81^2(1)
PaRSEC	70^2(1)	120^2(1)	270^2(1)	380^2(1)	420^2(1)
Regent	70^2(1)	70^2(1)	60^2(1)	50^2(1)	50^2(1)
YML+XMP	1^2(64)	4^2(64)	8^2(32)	8^2(128)	8^2(128)

Table 3. Number of blocks for the fastest case on a 49512 × 49512 matrix with number of processes per tasks between parenthesis

	8	16	32	64
HPX	148^2(1)	148^2(1)	148^2(1)	145^2(1)
PaRSEC	250^2(1)	250^2(1)	400^2(1)	420^2(1)
Regent	70^2(1)	70^2(1)	70^2(1)	70^2(1)
YML+XMP	1^2(128)	2^2(128)	4^2(128)	8^2(128)

manage data and a process to execute the tasks by default. Then, the user has to specify the number of processes on which the tasks will be executed by passing specific arguments to Legion runtime. We used 14 processes on each node to execute the tasks.

To launch our HPX application, we used the *mpirun* command to execute one instance of HPX runtime on each node as one would use MPI to execute one process per node. Then, HPX is able to infer the node configuration. HPX runtime spawns a worker process on each core of the node and tasks are run as light-weight threads on those processes. HPX is able to detect that there is two sockets on the node and manages them internally.

PaRSEC runtime depends on MPI and is used in the applications. Therefore, PaRSEC applications has to be run with the *mpirun* command. We created one MPI rank per core i.e. 16 MPI processes per node.

YML scheduler is launched with MPI on one core (the first one in the machine file) which launch XMP tasks with *MPI_Comm_spawn* routine on the leftover cores available.

3.3 Performances

Fig. 1 shows the performances obtained for the LU factorization with HPX, MPI, PaRSEC, Regent, ScaLAPACK, XMP and YML+XMP on three sizes of matrices 16384×16384 (top), 32768×32768 (middle) and 49512×49512 (bottom).

On a 16384 × 16384 matrix, MPI is close to XMP on a small amount of nodes. When the number of node increases, MPI becomes significantly faster than XMP.

Indeed, Fig. 1 middle and bottom charts show that MPI is significantly faster than XMP for each number of node. MPI and XMP applications share the same algorithm and a similar implementation but expressed with two different models. This may be due to an overhead from the PGAS description and access of the data in XMP compared to MPI.

Regent, HPX and PaRSEC are relatively close to one another on a small number of cores. However, we can outline tendencies. PaRSEC is faster than HPX on the lower number of node then HPX becomes faster when the number of node increases. It also seems that when the size of the matrix increases, HPX and PaRSEC performances are becoming closer and that HPX becomes faster than PaRSEC on the larger number of nodes. Indeed, HPX becomes faster than PaRSEC after 4 nodes for a matrix of size 16384×16384, after 16 nodes for a matrix of size 32768×32768 and after 64 nodes 49512×49512. For the later value, the difference between the two is very small (330s vs 331) so we expect HPX to become significantly faster for this size of matrix with a greater number of nodes.

Regent is a little bit behind HPX and PaRSEC on each number of nodes and size of matrices except for 2 and 4 nodes on a 16384×16384 matrix where Regent is very efficient. We can also notice that Regent is taking more time on 64 nodes than on 32. This may be related to the fact that Regent does not seem to be able to manage a large number of tasks on a large number of nodes since the number of sub-matrices is decreasing when the number of cores is increasing as Table 1, Table 2 and Table 3 are showing. However, other task based languages obtain better results when the number of sub-matrices they process increase with the number of cores. It creates more task and parallelism so that the runtime can use the resources most efficiently.

The YML+XMP applications are the slowest compared to the the applications implemented with the other models. However, YML+XMP is the only model where tasks are also parallel and distributed. Moreover, it also uses the file system to perform the communications between the tasks so the communications between tasks are not efficient.

Our last application uses the ScaLAPACK library to compute the LU factorization. It performs very well on large number of nodes but HPX, PaRSEC and Regent are faster on lower number of cores for each size of matrix. They are not using the same block based algorithm but ScaLAPACK is using a tiled algorithm that makes computations on rows and columns of the matrix [4]. Therefore, it is an interesting comparison to our block-based algorithms where the operations on the blocks are implemented with tasks. For a 16384×16384 matrix ScaLAPACK and HPX are close on 64 nodes but ScaLAPACK is faster for greater size of matrices. This may be due to the cyclic distribution of data in ScaLAPACK which induces a different communication pattern very efficient on this kind of machine and algorithm.

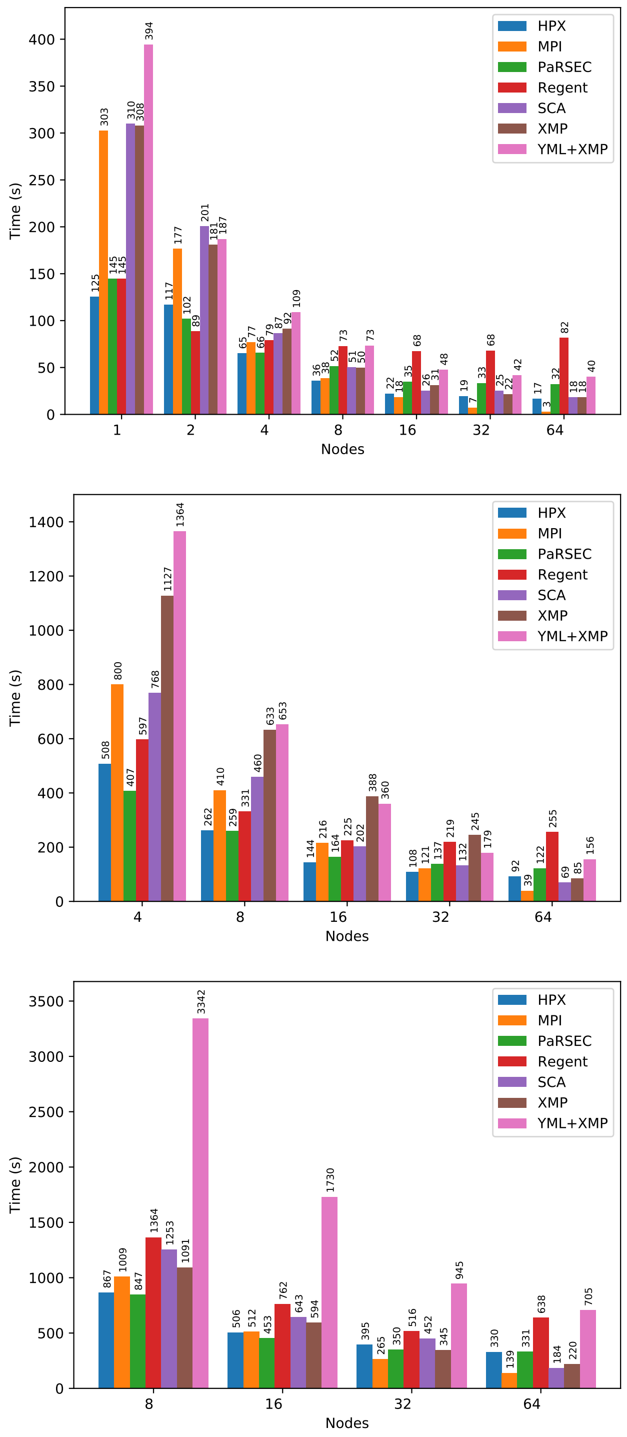

Fig. 1. Execution times obtained with the block-based LU factorization implemented with several task-based programming models on a 16384×16384 matrix (top), a 32768× 32768 matrix (middle) and a 49512 × 49512 matrix (bottom)

3.4 Strong scaling

Fig. 2 shows the speed-up extracted from the performances values from Fig. 1 for HPX, MPI, PaRSEC, Regent, ScaLAPACK, XMP and YML+XMP on three sizes of matrices 16384×16384 (top), 32768×32768 (middle) and 49512×49512 (bottom). The speed-up corresponds to the ratio t_S/t_N where t_N is the execution time for N nodes and t_F is the execution time of the first number of nodes considered in the test. In the top chart of Fig. 2, t_F is t_1 since the experiments start with 1 node. In the middle (bottom) chart, t_F corresponds to 4 (8). It translates how efficiently we are managing the addition of more resources to solve the same problem.

Our MPI regular LU factorization is scaling very well as we can see on the charts. It even exceeds the ideal speed-up with matrices of size 16384×16384 (Fig. 2 top chart) and 32768×32768 (Fig. 2 middle chart). We think that it may be due to processes not having enough computations to do on 32 and 64 nodes matrices of size 16384×16384. Indeed, when increasing the size of the matrix to 32768×32768, the strong scalability for our MPI application seems more reasonable. The same situation occurs for 64 nodes when increasing the size of the matrix from 32768×32768 to 49512×49512.

Our task based applications obtain better scalability with the increase of the data size and the number of tasks processed by the applications. Table 1, Table 2 and Table 3 show that the number of tasks for a given number of nodes increases with the size of the matrix for each task based programming model. It produces more parallelism and opportunities to optimize the scheduling of the tasks and improve the use of the computing resources.

Regent strong scalability decreases from 32 to 64 nodes for each size of matrix. We expect its strong scalability to decrease even more with the increase of the number of cores.

Our HPX application is scaling better than our PaRSEC application with matrices of size 16384×16384 and 32768×32768. It corresponds to the results we obtained in the previous section. We can also see that PaRSEC and HPX are very close with matrices of size 49512×49512 and that HPX is exceeding PaRSEC after 32 nodes. It seems that HPX may have a better scalability than PaRSEC on more than 64 nodes with matrices of size 49512×49512 if more nodes were available.

Finally, our YML+XMP application has the best strong scalability compared to the other task-based programming models. Therefore, we think that this programming model will be well adapted to larger machines with a distributed system and integrated schedulers.

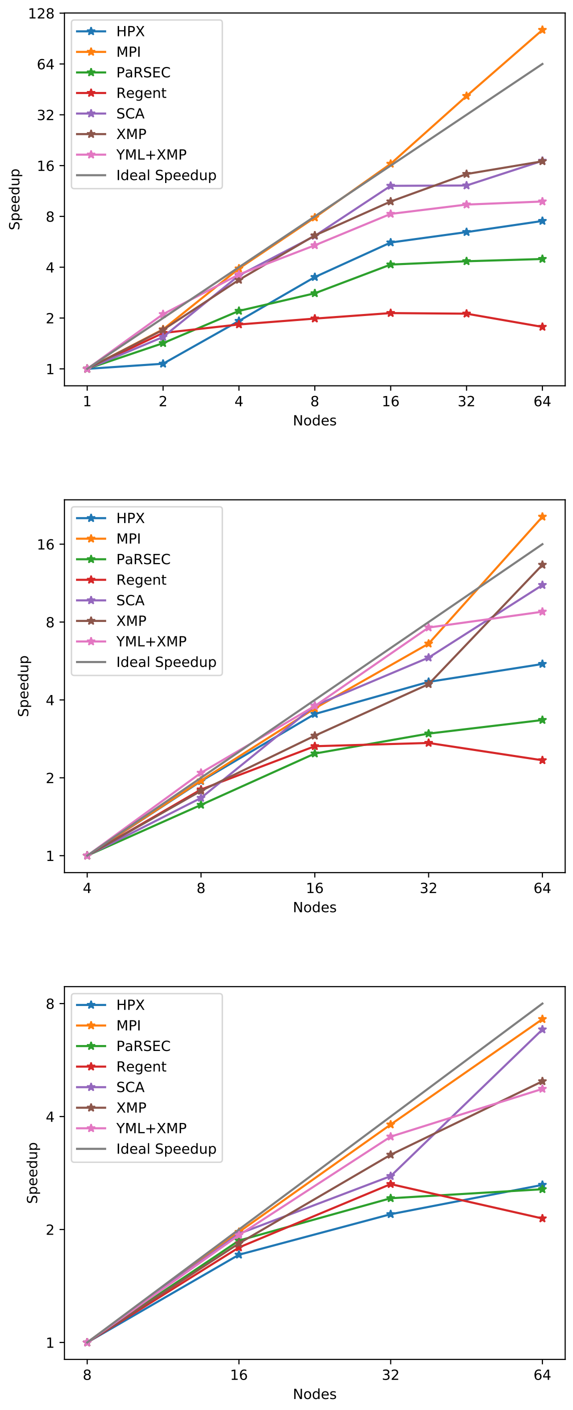

Fig. 2. Speed-ups obtained with the block-based LU factorization implemented with several task-based programming models on a 16384×16384 matrix (top), a 32768×32768 matrix (middle) and a 49512×49512 matrix (bottom) - log_2 scale for the y-axis

3.5 Results summary

As expected on a relatively small cluster, MPI has the best results and scalability on 64 nodes but the application does not use partial pivoting so it is not comparable to ScaLAPACK. It is also faster than XMP since MPI routines are highly optimized. Even though, XMP translates its directives into MPI code, the PGAS model used in XMP is not as efficient as using directly MPI.

ScaLAPACK is faster than the applications implemented with the task based programming models but it uses very efficient kernel routines to perform computations and communications internally whereas we are using unoptimized routines.

In term of task based programming models where we implemented everything (the description of the tasks and the computations executed by the tasks) with the language of the programming model, HPX is the most efficient on 64 nodes. However, PaRSEC also shows interesting performances in specific circumstances. Furthermore, Regent applications performances are not improving while increasing the number of nodes from 32 to 64. We think that the difference of performances between those programming models comes from their ability to manage the number of tasks, the dependencies between the tasks, tasks workload and the data migrations between nodes. Indeed, Regent performs best with a smaller amount of tasks than HPX and PaRSEC but its performances are behind them (see Table 1, Table 2 and Table 3 where we can see the number of tasks executed with the programming models to obtain their best performances). HPX and PaRSEC are performing better than Regent and YML+XMP with a larger number of smaller tasks since the number of tasks increase but not the data global size. HPX and PaRSEC seem to distribute very efficiently the tasks on the resources and optimize the data migrations between the nodes whereas Regent does not seem to be able to do so since the user has to reserve resources in Legion to manage the data and the computations. We would expect Legion to be able to reserve those resources by itself. Moreover, we used the default mapper for data and tasks provided by Regent and Legion. The default mapper may not be efficient enough to be used on production environment. Implementing a new mapper more adapted to our use may improve the performances of our Regent application. However, this means that the responsibility of implementing a good data and computation mapper is pushed to the user. Therefore, the optimization of the scheduling of the computations, the data positioning and data migrations is relegated to the user and not feature of the task based programming model anymore.

Finally, YML+XMP is the only programming model using tasks which are also distributed but the data migrations between the tasks are performed by reading/writing them on the file system which decrease its efficiency compared to node to node communications. YML+XMP performances may not be impressive on this number of nodes but with the increase of the number of nodes and its strong scalability higher than the other task-based programming languages, YML+XMP could be able to perform better than the other programming models on a very large scale as already experienced on the K computer [10]. Moreover, changing the use of the file system to make the communications between the tasks to in-memory communications could improve the performances even more.

4 Conclusion and Perspectives

We experimented several programming models on a cluster composed of 77 nodes. Indeed, we performed strong scalability tests on up to 64 nodes (1024 cores) with our implementation of the LU factorization in several programming paradigms. We implemented it with XMP, a PGAS language, with Regent (Legion), HPX and PaRSEC, three fine-grain task-based programming models and with YML+XMP, a coarse-grain task-based programming model combined with a PGAS language. We compared the performances we obtained with the different task-based programming models. We also compared to the ScaLAPACK library and our MPI implementation.

Our study has shown that ScaLAPACK performed better than task-based languages with a problem large enough. ScaLAPACK is expected to run better since it uses high performance libraries (e.g. BLACS and LAPACK) to perform the inner computations and the data migrations so it may also explain why it is faster. Moreover, we also showed that HPX is performing better than the other task-based languages on a large number of cores and that PaRSEC is more efficient than HPX on smaller number of cores. Unfortunately, Regent performances are close to HPX and PaRSEC but we encountered difficulties to make it scale from 32 to 64 nodes. However, we expect fine-grain task-based programming models them to get better performances with the increase of compute nodes and the use of optimized routines to implement tasks. Finally YML+XMP, is the less efficient one due to the communications between the tasks being held by the file system. Furthermore, coarse-grain task-based programming models with two levels of parallelism (the graph of tasks and the tasks implemented in a PGAS language) are not adapted to this kind of machine and number of nodes. They could possibly obtain better results with an adapted scheduler and a greater number of nodes as shown in [10].

As new perspectives, since, we only used a relatively small amount of nodes on an already old cluster, the number of cores could be increased as well as the size of the problem. We think that task-based programming models may get better performances than MPI+X when the size of the problem, the number of computing resources and the communication network involved in its solution will greatly increase. The task approach allows to describe the computations, the data migrations and the dependencies between them more precisely and at a finer grain. Therefore, the scheduler will be able to predict and anticipate the location where the data will be required. The scheduler could also optimize load balancing in the processes available as well as run different type of task at the same time compared to MPI where each process does almost the same thing at the same time. Moreover, the scheduler could be able to launch computations on resources where the data are stored and place the data in a way that reduces their movement during the execution. Other graph of tasks based frameworks like Pegasus [7] could also be studied.

Finally, our applications could get even better results by using existing and efficient libraries to perform the operations on the sub-matrices. Another improvement is to manage data sizes which are not divisible by the number of blocks and introduce pivoting to improve numerical stability.

Acknowledgment

Thanks to George Bosilca for his help with PaRSEC. Thanks to Harmut Kaiser and Mickael Simberg for their answers to my questions about HPX. Thanks to Elliott Slaughter for his help in installing and in using Regent properly. We would like to thank TOTAL SA which supported and allowed this work.

References

1. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 66:1–66:11. SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2388996.2389086
2. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J.J.: Parsec: Exploiting heterogeneity to enhance scalability. Computing in Science Engineering **15**(6), 36–45 (Nov 2013). https://doi.org/10.1109/MCSE.2013.98
3. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: Dague: A generic distributed dag engine for high performance computing. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. pp. 1151–1158 (May 2011). https://doi.org/10.1109/IPDPS.2011.281
4. Choi, J., Dongarra, J.J., Pozo, R., Walker, D.W.: Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In: [Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation. pp. 120–127 (Oct 1992). https://doi.org/10.1109/FMPC.1992.234898
5. Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y., Chavarra-Miranda, D.: An evaluation of global address space languages: Co-array fortran and unified parallel c. pp. 36–47 (01 2005). https://doi.org/10.1145/1065944.1065950
6. Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: Ptg: An abstraction for unhindered parallelism. In: 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing. pp. 21–30 (Nov 2014). https://doi.org/10.1109/WOLFHPC.2014.8
7. Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J.C., Katz, D.S.: Pegasus: a framework for mapping complex scientific workflows onto distributed systems. Scientific Programming Journal **13**(3), 219–237 (2005), http://pegasus.isi.edu/publications/Sci.pdf

8. Delannoy, O., Emad, F., Petiton, S.: Workflow global computing with yml. In: 2006 7th IEEE/ACM International Conference on Grid Computing. pp. 25–32 (Sept 2006). https://doi.org/10.1109/ICGRID.2006.310994
9. Forum, M.P.: Mpi: A message-passing interface standard. Tech. rep., Knoxville, TN, USA (1994)
10. Gurhem, J., Tsuji, M., Petiton, S.G., Sato, M.: Distributed and parallel programming paradigms on the k computer and a cluster. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. pp. 9–17. HPC Asia 2019, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3293320.3293330
11. Hoque, R., Herault, T., Bosilca, G., Dongarra, J.: Dynamic task discovery in parsec: A data-flow task-based runtime. In: Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems. pp. 6:1–6:8. ScalA '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3148226.3148233
12. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: Hpx: A task based programming model in a global address space. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. pp. 6:1–6:11. PGAS '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2676870.2676883
13. Lee, J., Sato, M.: Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In: 2010 39th International Conference on Parallel Processing Workshops. pp. 413–420 (09 2010). https://doi.org/10.1109/ICPPW.2010.62
14. Sato, M., Hirano, M., Tanaka, Y., Sekiguchi, S.: Omnirpc: A grid rpc facility for cluster and global computing in openmp. In: International Workshop on OpenMP Applications and Tools. pp. 130–136. Springer (2001). https://doi.org/10.1007/3-540-44587-0_12
15. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: a high-productivity programming language for hpc with logical regions. In: SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (Nov 2015). https://doi.org/10.1145/2807591.2807629
16. Snir, M., Wisniewski, R.W., Abraham, J.A., Adve, S.V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., Chien, A.A., Coteus, P., DeBardeleben, N.A., Diniz, P.C., Engelmann, C., Erez, M., Fazzari, S., Geist, A., Gupta, R., Johnson, F., Krishnamoorthy, S., Leyffer, S., Liberty, D., Mitra, S., Munson, T., Schreiber, R., Stearley, J., Hensbergen, E.V.: Addressing failures in exascale computing. The International Journal of High Performance Computing Applications **28**(2), 129–173 (2014). https://doi.org/10.1177/1094342014522573
17. Tsuji, M., Sato, M., Hugues, M., Petiton, S.: Multiple-spmd programming environment based on pgas and workflow toward post-petascale computing. In: 2013 42nd International Conference on Parallel Processing. pp. 480–485 (Oct 2013). https://doi.org/10.1109/ICPP.2013.58