International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Parallel Gmres with Futures and Promises

A. Tran Tan · B. Adelstein-Lelbach ·
J. Falcou · H. Kaiser · D. Etiemble

Received: date / Accepted: date

Abstract The exponential growth of available FLOPS requires the development of more sophisticated programming tools enabling significantly improved application efficiency and scalability on current and future architectures. We investigate the new execution model ParalleX [?], which defines a work queue based, message driven, fine grain parallelization scheme based on a global address space for highly asynchronous calculations. This paper uses HPX [?], a C++ library implementing the ParalleX execution model, to realize the Gmres operation from BLAS [?]. *Futures*, a mechanism of asynchrony [?] [?] supported by ParalleX, are used to expose parallelism in our implementation. We examine the performance of this new HPX Gmres code, relative to conventional parallel Gmres implementations.

Keywords C++, Futures, Parallel Programming, High Performance Computing

1 Introduction

High Performance Computing (HPC) is taking a new direction due to challenges provided by multicore and large scale systems. Indeed, the announced emergence of thousand billion core computers will significantly increases parallelism. The aim, therefore, will be no longer to make faster programs but to make programs that efficiently scale.

Clusters of multicore processors which are systems commonly used in HPC, use two programming models: shared memory model for SMP nodes and message passing model for the whole cluster. But using these models, parallel programs tend to follow the same programming rule consisting in implementing global barriers. ANL (Argonne National Laboratory) noticed the critical

LRI, Université Paris-Sud XI - Orsay, France · CCT, Louisiana State University - Baton Rouge, USA

cost of barriers because of latencies, system noises and non-uniform workloads especially in massively parallel machines. [?]

To solve this problem, the idea is to integrate asynchrony, a concept already existing in MPI but still difficult to implement. This paper thus introduces new concepts by means of ParalleX model which get away from conventional parallel programming models allowing to maximize asynchrony in calculations.

Section **??** introduces *Futures* and *Promises* mechanism in more general context. Section **??** describes the High Performance ParalleX (HPX) runtime system [?], an experimental implementation of ParalleX. Section **??** defines the Generalized Minimal Residual algorithm (Gmres) [?] and discusses the results of its HPX implementation.

2 Futures and Promises

Parallel programming models currently provide different ways to protect data shared between multiple threads (mutual exclusions, global barrier). But sometimes, the requirement is not necessarily to protect data but to simply synchronize concurrent operations encapsulated in different threads. One example is a thread which needs to wait that another thread finishes its work before running itself. To address this demand, object oriented programming research have proposed an alternative so called *Futures* and *Promises* [?] [?] which fully implements this concept.

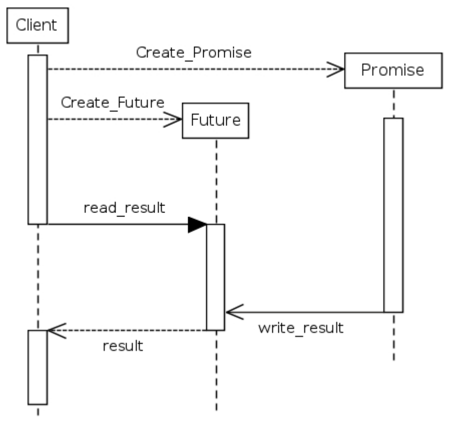


Fig. 1: Behavioural pattern of Futures and Promises

2.1 Definition

In one hand, *Futures* are objects encapsulating a value which will be available later. On the other hand, *Promises* are objects intended to solve these *Futures* using a function and some arguments. Programming with Futures is different from classical synchronous programming in the fact that tasks run in non-deterministic manner in time. Futures will then make it possible for applications to invoke asynchronous method without being interrupted. (cf. Figure **??**)

2.2 Example

Here we illustrate the mechanism of Futures with an example. Suppose we want to realize the following calculation:

$$y = f3(f1(a), f2(b))$$

With sequential programming, the code could be written like:

```cpp
int a, b;
int y1 = f1(a);
int y2 = f2(b);
y = f3(y1, y2);
```

Listing 1: Sequential C++ example code

With Futures programming, the code is written like:

```cpp
int a, b;
future<int> y1 = async(f1,a);
future<int> y2 = async(f2,b);
y = f3(y1.get() , y2.get());
```

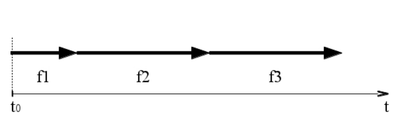
Listing 2: C++ example code with Futures

In this example (cf. Listing **??**), Futures y1 and y2 each encapsulate one asynchronous function result. Required results are obtained by calling Futures `get()` method which lead to suspension of consumer threads until these results are available. The following timing diagrams illustrate the interest of Futures in terms of parallelism: with a syntax close to the sequential program, we had written a parallel code.

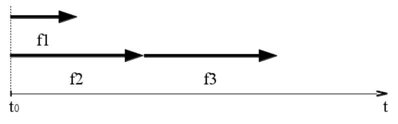
2.3 Implementations

The various implementations of Futures/Promises model are either directly integrated in languages (Alice ML, Scala) or integrated in specific libraries (OCaml, Java). A class implementation is also integrated in the last C++ standard [**?**]. From this point, Stellar group (Systems Technologies, Emergent Parallelism and Algorithms Research) of CCT (Center for Computation and

Technology - Louisiana State University) has adapted this model in particular for distributed computing and grid computing. This adaptation, fully written in C++, consists of a runtime system named HPX (High Performance ParalleX). The next section will review the main elements of this system.



(a) Sequential program



(b) Program with Futures

Fig. 2: Timing diagrams of example code

3 HPX Runtime System

HPX runtime system, designed by Stellar group, is an effective implementation of ParalleX paradigm that aims to increase performances by considering all obstacles encountered by conventional programming models. Thus, ParalleX combines several concepts that we will develop below.

3.1 General Design

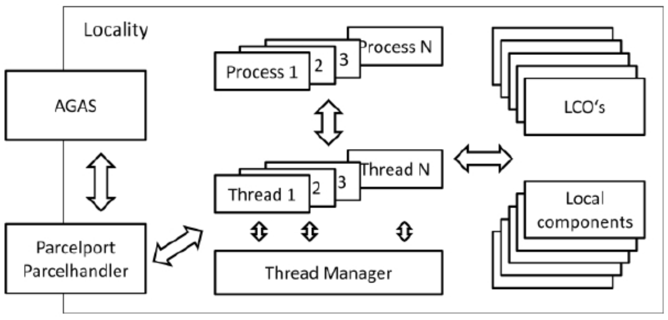


Fig. 3: Architecture of HPX runtime system

3.1.1 AGAS - The Global Address Space

A single adress space will open new perspectives like dynamic load balancing and dynamic graph related problems implementation. The PGAS model [**?**]

partially solves this problem but don't allow objects to keep their name moving from a cluster node to another. Thus, Stellar group proposed a PGAS based model so called AGAS (Active Global Address Space) [?] which assigns global names (128 bits integers) to objects. While maintaining no coherence between cluster nodes, AGAS dynamically provides the correspondence between this global id and a local virtual address (LVA) [?] which includes the node id, the type of the entity and its local memory address. This will help inter-node data migration.

3.1.2 Parcels - Message Oriented Programming

Message passing is an essential step when we work in a distributed memory. But the kind of message may vary with applications. On the one hand, we can move the data to the nodes which know in advance the work to accomplish; this is the solution of MPI. On the other hand, we can move the work to the nodes which already have their data; this is the solution of ParalleX. Thus, ParalleX incorporates the notion of *Parcels* [?] which are active messages including the destination address (provided by AGAS), the work to accomplish, some needed arguments and control elements for *LCOs* that we will describe in section **??**. This will enable dynamic resource management and the use of distributed control flows.

3.1.3 PX-threads - User Level Threads

Thread migration between nodes is theoretically feasible but is still a costly operation in particular in heterogeneous parallel architectures. The preferred solution in ParalleX is to simply send requesting Parcels leading to the creation of a continuation thread in a certain locality. These continuation threads so called *PX-threads* [?] are user level threads, scheduled non-preemptively by a local thread manager (one per node) which knows in advance the corresponding node topology. This thread manager provides a PX-threads queue to each *OS-thread* [?] following the "first come first served" scheduling rule. Note that in conventional systems, an OS-thread is physically assigned to a processor core. Like a classical thread, a PX-thread may be in four different states which are: *pending*, *running*, *suspended* and *terminated*. The state of the PX-threads is controlled by the LCOs that we will describe below.

3.1.4 LCOs - Thread Synchronisation Mechanism

As we have seen in Section 1, global barriers have become problematic. Thus, ParalleX integrates a lightweight thread synchronisation mechanism in the form of *LCOs* (Local Control Objects) [?]. The LCOs are a class of objects which may locally create or reactivate (in a node) a PX-thread as a result of one or more events (ex: receipt of a Parcel). The LCOs may take several forms: *semaphores*, *mutexes*, *Futures*. The Futures semantics described in section **??**

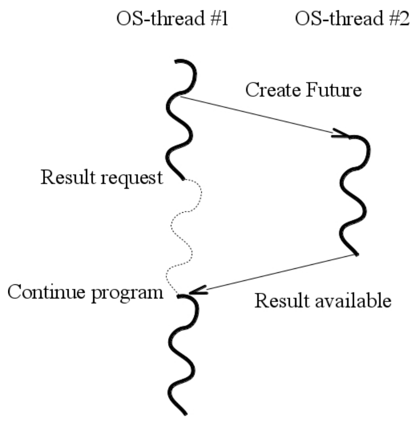


Fig. 4: Execution schematics of a Future in HPX

will thus enable event-driven thread creation, on-the-fly scheduling and effectively exploit the implicit parallelism of directed graph problems. Figure **??** illustrates the use of Futures in the HPX system.

4 Gmres Algorithm

In the HPC field, linear systems resolution is increasingly done by iterative methods [**?**] (Jacobi, Gauss-Seidel, ...). These have indeed the advantage to provide high levels of parallelism in particular for large sparse systems: hence the idea to make an implementation of Gmres algorithm into HPX system. The Gmres method (Generalized Minimal Residual) [**?**] is an iterative algorithm solving the linear system $Ax = b$, where A is an invertible square matrix, from an initial guess x_0.

At each iteration, this method calculates $x_k \in K_k$ where K_k is a k-dimensional Krylov subspace, minimizing the Euclidian norm of the residual. But the amount of calculations and storage work increases with the iteration. Thus, the algorithm is usually restarted every m iterations using current approximate solution as initial guess before each restart step. (Gmres(m)) [**?**]

4.1 The Gram-Shmidt Orthogonalization

To reach this solution, we need to construct orthonormal vectors which constitute a Krylov space basis (Gram-Shmidt Orthogonalization) and then to perform several rotations to get a trivial system (upper triangular system). The main problem of Gmres is in the orthonormalization step. Indeed, the

Algorithm 1: Gmres Algorithm

```
1  $r_0 \leftarrow b - Ax_0$
2  $v_0 \leftarrow r_0/\|r_0\|_2$
3  for $i \leftarrow 0$ to $m-1$ do
4      $z \leftarrow Av_i$
5      $h_{j,i} \leftarrow \langle z, v_j \rangle, j = 0, ..., i$
6      $\tilde{v}_{i+1} \leftarrow z - \sum_{j=1}^{i} h_{j,i} v_j$
7      $h_{i+1,i} \leftarrow \|\tilde{v}_{i+1}\|_2$
8      $v_{i+1} \leftarrow \tilde{v}_{i+1}/h_{i+1,i}$
9      # Apply Givens rotation to $H_{:,i}$
10 $y_m \leftarrow argmin\|(H_{m+1,m}y_m - \|r_0\|_2 e_1)\|_2$
11 $x \leftarrow x_0 + V_m y_m$
```

construction of each vector requires to realize two global communications: one during the orthogonalization step and the other during the normalization step.

A first solution, that we won't deal with in this paper, is to minimize the inter-processes communications by adding redundancy in calculations. This approach so called "communication-avoiding" [?] results to the CA-Gmres version of the algorithm. A second solution is to change the synchronization model in order to reduce the communication overhead; this is the solution that we will examine below.

4.2 Implementation with Futures

In this study, we started from the original algorithm [?] developed by Y. Saad and M.H. Shultz. In each iteration, four kinds of operations are performed: sparse matrix-vector product, dot product, vector addition and vector scale. The most naive way to parallelize these various operations is to slice matrices and vectors and to attribute each pieces to a process (cf. Figure **??**). With HPX, the idea is to invoke several Futures in each iteration so that to perform an asynchronous calculation by domain. The second step including Givens rotations is mostly sequential and spend lower time than orthonormalization step. Then, we didn't take care of its optimization knowing its little influence on the performances.

4.3 Experimental results

This Section confronts the performances of HPX Gmres implementation with the one provided by Petsc [?]. Recognized as a reference in HPC, Petsc performs its parallel calculations using MPI. These performance tests have been done for 5 problems from Matrix Market Collection and using the same initial guess x_0 and right-hand side vector b. We fixed the restart parameter to 30 to keep reasonable storage (Krylov basis vectors) and the maximum number of iterations to 1000, just enough for problems at hand. The used machine was a

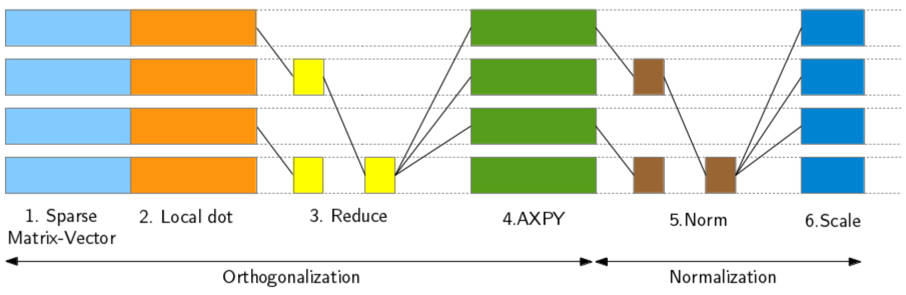


Fig. 5: Classical parallelization of Gmres

2*6 core NUMA node. At this time, despite the common use of this architecture and existing tools to manage it, HPX doesn't take care in giving NUMA domain affinity to PX-thread. This could be problematic when applications are memory bound: this is the case for Gmres when it uses BLAS interface. We will see below the impact on the performances.

To make a quantitative evaluation of performances, we used the Karp-Flatt metric: a parallelism measuring instrument developed from Amdahl's law and giving more informations about parallelism cost. The Karp-Flatt formula is:

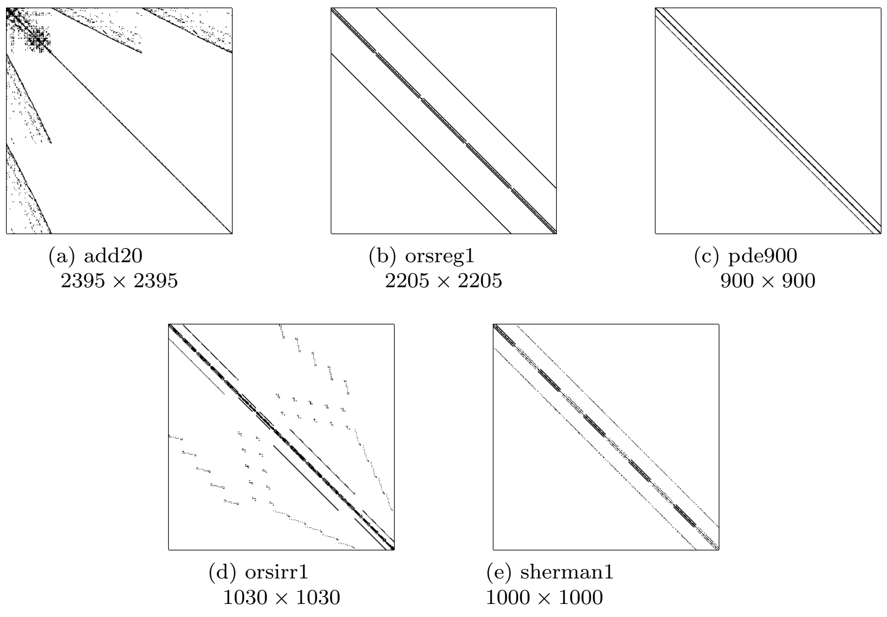
$$\alpha(p) = \frac{1/\gamma(p) - 1/p}{1 - 1/p} \tag{1}$$

with
γ: Speedup of parallelism
p: Number of processors (or cores)

The major interest of this metric is to be representative of the evolution of parallelism cost. If $\alpha(p)$ increases with p, the declining efficiency of an algorithm is due to parallelism cost. If not, this is due to sequential part of the algorithm.

Globally, HPX and Petsc codes scale relatively well while used cores are located in the same NUMA domain (from 1 to 6 cores). But when we use more cores (from 7 to 12), performances degrade because of remote memory access. We can notice that this degradation is worst for HPX because all data are located in the same NUMA domain. The distributed nature of Petsc code in which every core owns their data makes its scalability less impaired. From here, we'll restrict our focus on one NUMA domain (from 1 to 6 threads).

Structure of matrices (cf. Figure **??**), in terms of non-zero values repartition, has a significant influence in performances particularly in SpMv step because more the values are near to the diagonal more Petsc can exploit the quasi-diagonal structure of matrices whereas HPX code simply uses the same


(a) add20
2395 × 2395

(b) orsreg1
2205 × 2205

(c) pde900
900 × 900

(d) orsirr1
1030 × 1030

(e) sherman1
1000 × 1000

Fig. 6: Structure of example matrices

blocking strategy using BLAS. Then, it is important to take account the standard deviation of each problem in case of performance gap between HPX and Petsc implementations.

Figures **??** and **??** show performances for less sparse systems. In this case, each thread owns sufficient data to confer some scalability. Although execution times are equivalent, Karp-Flatt measuring show clearly that communication overhead has a more negative impact in HPX than in Petsc. Looking beyond the code structure, classical Gmres is largely synchronous and so it remains difficult to gain performance with asynchrony.

Figures **??** and **??** show performances for quasi-diagonal systems. In this case, Petsc optimize the sparse matrix location using lower storage and optimizing communication scheme of common operations. Then, Petsc sequential code is faster than HPX one but scalability remains equivalent.

Figure **??** show performances for very sparse systems. The overhead treshold appears clearly seeing HPX results. Here, the SpMV step scales less because of overlap of calculation by communications (starting from 4 threads). This overhead treshold is approximatively $25 * 10^6$ cycles with 2.4 Ghz frequency.

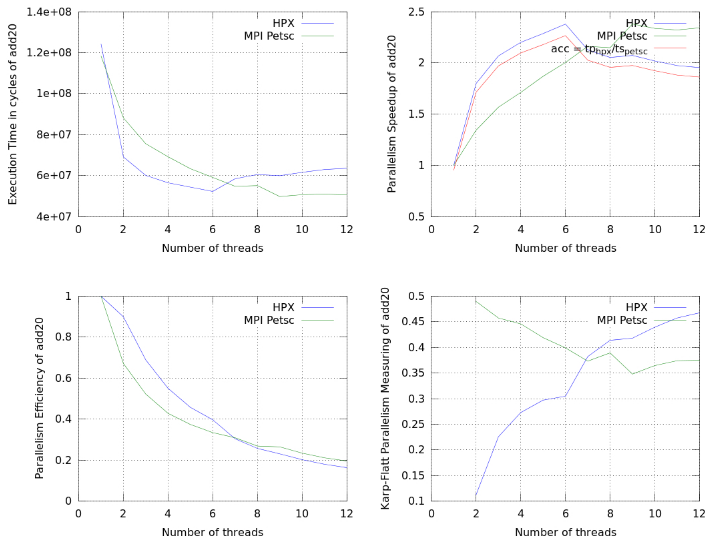


Fig. 7: Performance measures of add20 (2395×2395, 17319 entries)

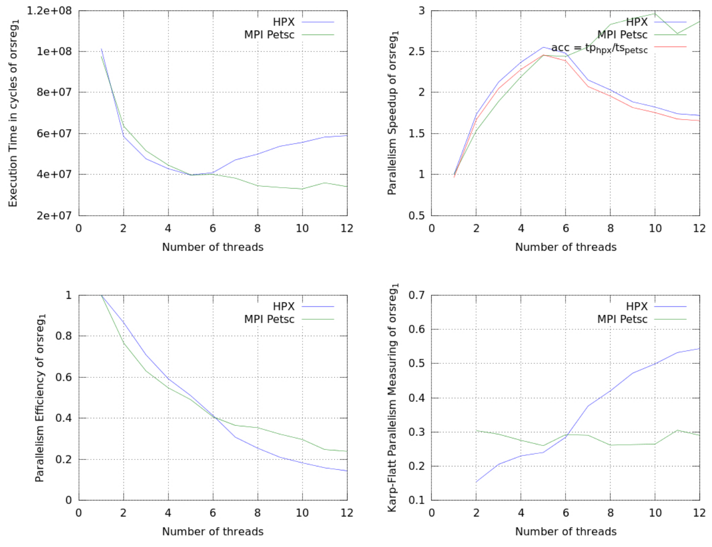


Fig. 8: Performance measures of orsreg$_1$ (2205×2205, 14133 entries)

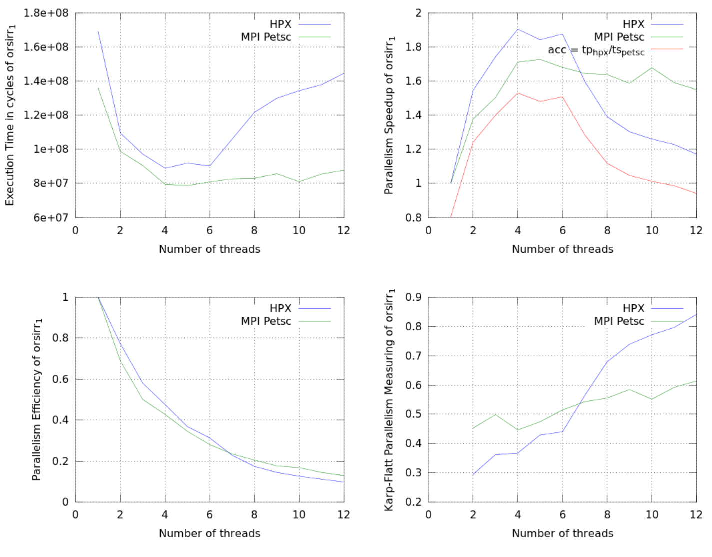


Fig. 9: Performance measures of orsirr$_1$ (1030×1030, 6858 entries)

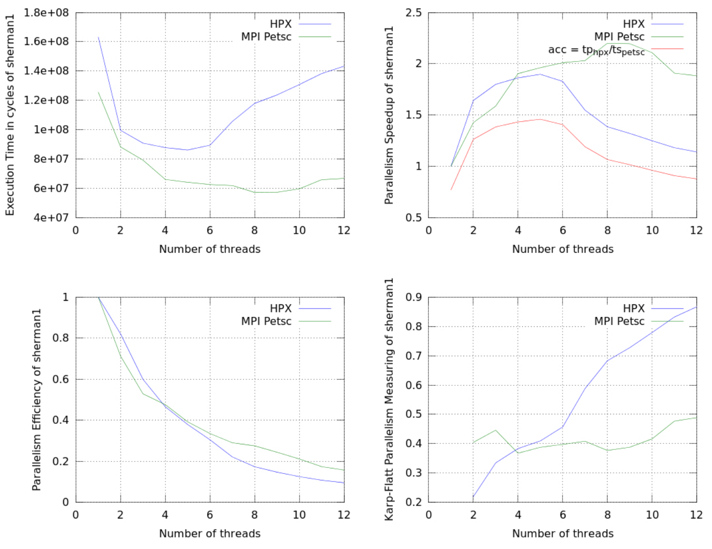


Fig. 10: Performance measures of sherman$_1$ (1000×1000, 3750 entries)

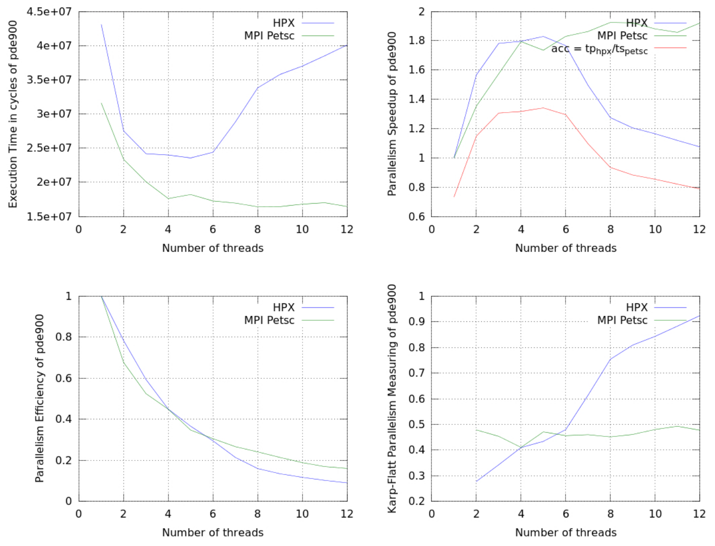


Fig. 11: Performance measures of pde900 (900 × 900, 4390 entries)

5 Conclusion

In this paper, we exposed the main concept of *Futures* and *Promises* in order to improve the expression of asynchronous calculations in parallel applications. By using Parallex model and its corresponding implementation, we compared performance of our Gmres implementation to Petsc in NUMA architecture. Results show equivalent scalability for the two implementations while used cores are located in the same NUMA domain. Performances can be improved by making it possible to launch asynchronous calculations in specific NUMA domain. Otherwise, the classical Gmres algorithm is naturally synchronous and it remains difficult to take advantage with an asynchronous model. But other versions of the algorithm exist like Pipelined Gmres (p(l)-Gmres) [**?**] in which some of calculation steps are interlaced so that to hide global communications. This version integrating more asynchronous calculations could be investigated with HPX and compared to the actual Petsc implementation.