Parallel GMRES with Futures and Promises

Antoine Tran Tan[1], Bryce Adelstein-Lelbach[2], Joel Falcou[1], Hartmut Kaiser[2], and Daniel Etiemble[1]

[1] LRI, Université Paris-Sud XI - Orsay, France
[2] CCT, Louisiana State University - Baton Rouge, USA

Abstract. The exponential increment of Flops and execution units implies to develop more and more sophisticated tools, but it is still difficult to take advantage of new architectures peak performances. In this context, new paradigms are investigated such as Parallex [?]: an execution model allowing asynchronous calculations with use of *Futures* and *Promises* semantic. In this paper, we will discuss an evaluation of Parallex by implementing the scientific calculating problem GMRES [?], and by examining performance in relation to conventional parallel programming models. This paper will thus expose one of the major interests of Parallex which is to better express asynchronous calculations while keeping a reasonable scalability.

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 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 2 introduces *Futures* and *Promises* mechanism in wider context. Section 3 describes the High Performance Parallex (HPX) runtime system [?], an experimental implementation of Parallex. Section 4 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.

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 a number of 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 1)

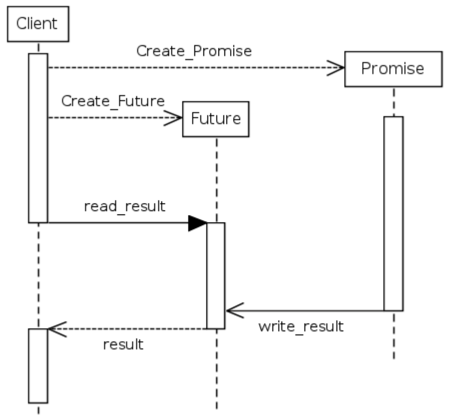


Fig. 1: Behavioural pattern of Futures and Promises

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:

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

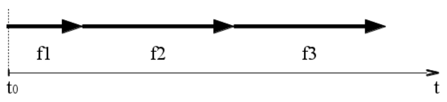
Listing 1.1: Sequential C++ example code

With Futures programming, the code is written like:

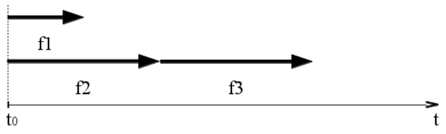
```
int a, b;
future<int> y1 = async(f1,a);
future<int> y2 = async(f2,b);
y = f3(y1.get() , y2.get());
```

Listing 1.2: C++ example code with Futures

In this example (cf. Listing 1.2), 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.



(a) Sequential program



(b) Program with Futures

Fig. 2: Timing diagrams of example 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.

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

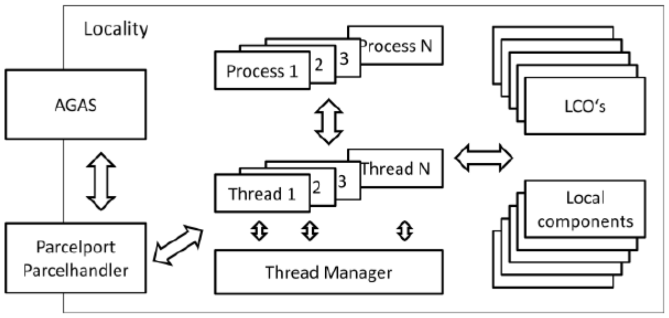


Fig. 3: Architecture of HPX runtime system

AGAS - The Global Address Space A single dress 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.

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 3.1. This will enable dynamic resource management and the use of distributed control flows.

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.

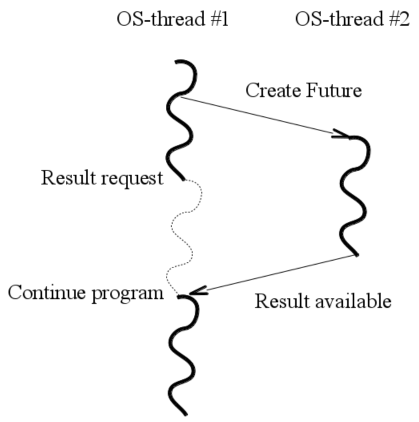


Fig. 4: Execution schematics of a Future in HPX

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 2 will thus enable event-driven thread creation, on-the-fly scheduling and effectively exploit the implicit parallelism of directed graph problems. Figure 4 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 a high level of parallelism: hence the idea to make an HPX implementation of GMRES algorithm. The GMRES method (Generalized Minimal Residual) [?] is an algorithm resolving the linear system below:

$$Ax = b$$

where A is an invertible square matrix, calculating the $x^{(k)}$ value which minimizes the Euclidian norm of the residual in each iteration. This method allows to find theoretically the x solution with a finite number of iterations.

4.1 The Gram-Shmidt Orthogonalization

To reach this solution, we have to construct the orthonormal vectors which will 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 construction of each vector requires to realize two global communications: one during the orthogonalization step and the other during the normalization step.

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 $x \leftarrow x_0 + V_m y_m$
```

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 of the algorithm, three kinds of operations are performed: matrix-vector product, dot product and vector addition. 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 5). 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.

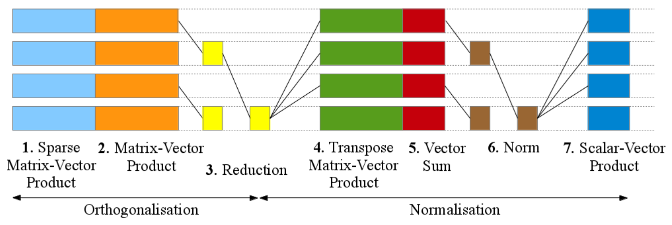


Fig. 5: Classical parallelization of GMRES

4.3 Performances evaluation

This Section confronts the performances of the HPX GMRES implementation with the one used in PETSC [**?**]. Recognized as a reference in HPC field, PETSC performs its parallel calculations using MPI.

Karp-Flatt Metric 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:

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

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.

5 Conclusion

References