Promise Plus: Flexible Synchronization for Parallel Computations on Arrays

Amaury Maillé, Ludovic Henrio, Matthieu Moy

Promise Plus: Flexible Synchronization for Parallel Computations on Arrays

Amaury Maillé[1], Ludovic Henrio[1], and Matthieu Moy[1][0000−0002−6054−8882]

Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France.
first.last@ens-lyon.fr

Abstract. Parallel applications make use of parallelism where work is shared between tasks; often, tasks need to exchange data stored in arrays and synchronize depending on the availability of these data. Fine-grained synchronizations, *e.g.* one synchronization for each element in the array, may lead to too many synchronizations while coarse-grained synchronizations, *e.g.* a single synchronization for the whole array, may prevent parallelism. We propose PromisePlus, a synchronization tool allowing tasks to synchronize on chunks of arrays with a granularity configurable by the programmer.

Keywords: Promises · Programming Models · Parallel Programming · High-Performance Computing

1 Introduction

In parallel programming, a promise is a synchronization tool. Initially, a promise is *unresolved*. Then, one task produces a value that resolves the promise; another task consumes that value through a `get` operation on the promise. `get` blocks if the promise is still unresolved. When exchanging arrays between tasks, promises can be used in two ways: either as a promise of array, where a promise holds a whole array; or as an array of promises, where there are as many promises as there are elements in the array.

The contribution of this paper is *a synchronization tool "PromisePlus" that works as a trade-off between an array of promises and a promise of array.* It allows programmers to *specify the granularity of synchronization and to stream data between tasks.* Unlike usual streaming frameworks [9] which are typically implemented using FIFOs [6] with support for bulk insertion and removal, PromisePlus allows access to elements out-of-order without removing them from its internal buffer, allowing several consumer entities to access the buffer's elements. Specifying a fine-grained (resp. coarse-grained) granularity makes PromisePlus akin to an array of promises (resp. a promise of array), and yields similar results in terms of performance. Moreover, there is a guarantee that a request to an element of the PromisePlus will unblock after at most n values have been produced, where n is the granularity of the synchronization. Finally, a request to an element of the PromisePlus will never produce an undefined value.

We begin this article with some context on HPC kernels and how they could benefit from data streaming and fine-tuned synchronizations. We subsequently present PromisePlus, benchmarks, related and future work, before concluding.

2 Context

HPC applications often perform heavy operations on huge amount of data stored in arrays. A kernel is a function applied to an array that typically produces another array as a result (*e.g. map*). Streaming data between kernels allows one kernel to start working on a partial output of another kernel; this is achieved by adding synchronization points between the two kernels. In order for the streaming to be efficient, the synchronization needs to be smart, *i.e.* synchronizations must be performed at the right points to fully exploit parallelism without wasting too much time in synchronizations.

Parallelism in HPC application usually comes from frameworks such as MPI (processes) or OpenMP (threads). These frameworks provide building blocks to perform streaming, which requires programmers to either write smart synchronization patterns on top of OpenMP/MPI or rely on third-party libraries (*e.g.* [9]) to achieve streaming. Moreover, primitives from HPC frameworks are low level, not always easy to use, and do not necessarily provide safety guarantees by construction.

As stated above, efficient streaming requires smart synchronization patterns. Ideally, these patterns should be reusable in different contexts (OpenMP threads, MPI processes) with different parameters (*e.g.* the granularity of the synchronization). Moreover, they should make the relation between data dependencies and the synchronization explicit, in order for a programmer to understand the concurrency problem solved. Finally, they should be optimized for the underlying architecture, thoroughly tested. Writing such a synchronization pattern takes time, time that programmers may not always have, forcing them to write patterns well suited for the problem at hand, but that cannot be reused as-is elsewhere and that may not always be comprehensible to outside observers.

This motivates us to propose a new synchronization tool for streaming arrays between tasks, that meets the following criteria:

- It makes the data dependency explicit: the programmer explicitly writes which data is shared between threads;
- It is configurable in a way that allows programmers to specify the granularity of the synchronization;
- It is not tied to a specific problem or a specific setup, and so is reusable;
- When the granularity is configured to the minimum (resp. maximum) value, it behaves like an array of promises (resp. a promise of array), providing the same guarantees and similar performance; moreover, every request for the i-th value of an array unblocks after an amount of values equals to the granularity has been produced;
- Contrarily to classical streaming solutions it supports the existence of several consumers for the same data, and the access to produced data in any order.

3 PromisePlus: Flexible Synchronization for Arrays

We present PromisePlus: a flexible synchronization pattern for arrays and matrices that allows data streaming between tasks.

A PromisePlus works like a standard promise, with additional support for integral indices. Both `get` and `set` work on indices; the programmer associates a value to an index through `set`, and gets access to the value associated with an index through `get`. Unlike an array of promises, performing a `get` on a given index may not immediately return once the index has received a value. PromisePlus is tied to two integral values: the *step* referred to as S that is configured at instanciation-time the PromisePlus, and *last*. *last* is the index passed to the last `set` call that triggered an unblock, initialized to -1 meaning the whole array is unresolved.

During a call to `set` with index i, if $i - last \geq S$, *last* becomes i and all calls to `get` with an index $i' \leq i$ are unblocked. This construct ensures that at most S elements have to be produced to unblock a `get`. Changing the value of S changes the granularity of synchronization.

Finally, a `set_immediate` primitive triggers an immediate synchronization, unblocking the synchronization on all previous elements of the array.

API PromisePlus exposes three functions: `set`, `get`, and `set_immediate`. `Index` is the type used for indices, `T` is the type of the values stored in the PromisePlus. A call to `set(i, v)` associates value `v` with index `i`, and if `i` $- last \geq S$, *last* becomes `i`. A call to `get(i)` blocks until $last \geq$ `i`; then it returns the value associated with index `i`. A call to `set_immediate(i, v)` associates value `v` with index `i`. The value of *last* becomes `i` without checking the step S. This function is particularly useful to signal that the last element has been produced and no further `set` operation should be expected. Calls to `set` and/or `set_immediate` must be performed on consecutive indices.

Notes on Implementation. While blocking a consumer thread in a `get(i)`, we use busy waiting as passive waiting induces too much overhead for HPC applications. All threads must share reading access to the value of *last* while the producer thread can write it too. Enforcing that all threads see an up-to-date value when reading *last* is cost-heavy, therefore each thread has a local index that caches the value of *last*. This local index is updated with the value of *last* when it is not sufficient to unblock a call to `get(i)`. In such a case, the thread synchronizes on the shared index. Finally, the producer threads stores a local copy of *last* to avoid some cost-heavy reads. Annex A presents algorithms for `get` and `set`.

4 Benchmarks

We benchmarked two things: how PromisePlus compares to the two naive approaches "array of promises" and "promise of array", and how the average time required to solve a problem using PromisePlus changes as the step changes.

Chosen problem Our problem is inspired by the LU program in the NPB [5], it reproduces the same data dependencies. Given a 4D matrix, we run a function f

on every element of the matrix, excluding boundary values, in parallel through several OpenMP threads. In order to update certain values, thread T_n, $n > 0$, requires values computed by thread T_{n-1}.

Matrix shapes We consider a work matrix of two billion values, excluding boundary values. We consider three different shapes for this matrix, leading to three different amounts of synchronization, while keeping the amount of computation constant: $101 * 161 * 62501 * 2$ (62500 synchronizations), $101 * 126 * 80001 * 2$ (80000 synchronizations) and $101 * 101 * 100001 * 2$ (100000 synchronizations).

Environment These tests were performed on a machine equipped with four Intel(R) Xeon(R) CPU E5-4620 0 @ 2.20GHz, with 96 threads without hyper-threading. Applications were built using GCC 8.3, C++17, and the $-O2$ flag in Release mode.

Comparison of patterns We compare the performance of the different patterns: array of promises, promise of array, and PromisePlus with three steps: 1, *max* that only synchronizes upon a `set_immediate`, and *opt* that is the step that achieves the best performance for a given shape. The promise of array uses home-made promises, with a `get` operation that performs a busy wait. The array of promises is tested using these same home-made promises, as well as C++ promises[8].

Figure 1 shows the results. In the legend, "P[Arr]" designates the promise of array, "Arr[P]" designates the array of promises, "Arr[SP]" designates the array of promises using C++ promises and "P+X" designates PromisePlus with a step of X. Figure 2 show the performance of PromisePlus as the step grows.

As expected, PromisePlus with a step of 1 performs similarly to the array of home-made promises with a slight overhead. Similarly, PromisePlus with a maximum step performs like the promise of array, again with a slight overhead. Both overheads are due to additional checks performed in the function `set` for PromisePlus. Also, there is an optimum step for PromisePlus that performs better than all others, with a performance gain of up to 12.63 % reached in the second shape with a step of 82 compared to array of home-made promises. Finally, home-made promises are well optimized: they perform up to 45.58 % better than C++ promises (Arr[P] compared to Arr[SP]).

Regarding the evolution of the average time as the step grows, an optimum step exists at which PromisePlus performs better than both the array of promise and promise of array.

5 Related Work

Streaming futures. In [1], promises are used as a way to stream data: they can be resolved multiple times, and each `get` returns the next resolving value when available. Unlike PromisePlus, streaming futures work as FIFOs with support for multiple producers at the cost of performance, and allow theoretically infinite streams . In PromisePlus, we chose to focus on performance and flexible granularity of synchronization for a better applicability to HPC. This flexibility could be ported to streaming futures, and potentially improve their performance.

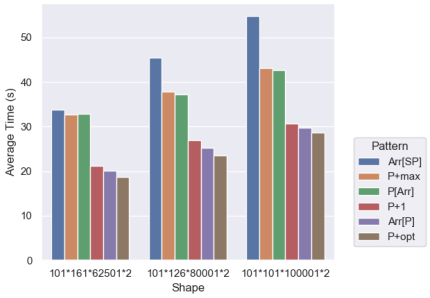

Fig. 1. Execution time for each pattern on different matrix shapes

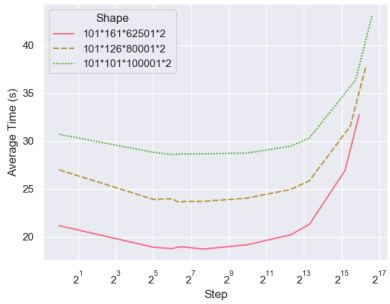

Fig. 2. Execution time for different matrix shapes and different steps

Distributed futures. Distributed futures [7] provide efficient data transfer when coupling data-parallel kernels in a task-parallel way. Spawning a task creates a distributed entity that works as a future that can be partially resolved by each process of the data-parallel computation. Once the result is computed, any process may request any chunk of the data directly from the process that holds it. Like PromisePlus, distributed futures slice the data and grants access to any computed slice, however unlike PromisePlus distributed futures require the whole computation to be over before allowing access to slices, preventing streaming. The fact that we have a less strict synchronization allows us to perform more optimizations and to find the optimal synchronization granularity.

OpenStream. OpenStream [9] is an extension of OpenMP that adds stream-like faculties to tasks. An OpenStream task may produce one or more streams, and a task may consume elements from one or more streams. A task is launched only when the elements requested on each input stream are available, this adds scheduling dependencies between tasks. A task requesting N elements from a stream before being able to be launched is akin to a PromisePlus with step N.

WeakRB. WeakRB [6] is an efficient implementation of a FIFO queue, using atomics in C, with a proof of correction.

The key differences between streams/FIFO queues and PromisePlus are the destructiveness of streams, which can be read only once and in a predefined order, and support only a single consumer in WeakRB. On the contrary, PromisePlus enforces an interaction pattern inherited from promises, where values can be read multiple times once they are ready, in any order, and by multiple consumers.

Message sets & Join patterns Message sets[3] and join patterns[4] allow for a declarative way of defining synchronizations. Operations wait for messages sent by other operations before executing. This is similar to get and set in PromisePlus. The key difference is that PromisePlus was thought first and foremost for efficiency with a conditional jump and a store at worse, while join patterns and message sets typically wait passively for a message.

SkePU SkePU[2] is a framework for algorithmic skeletons. Internally SkePU performs agressive optimizations to reduce the amount of synchronizations inside a thread, and could benefit from an optimized tool like PromisePlus to perform synchronizations between threads.

6 Future Work & Conclusion

In this paper we presented PromisePlus, an abstraction over promises that allows parallel computations to synchronize on slices of arrays with a granularity chosen by the programmer. This allows them to express different synchronization patterns through the use of a single tool. Moreover, PromisePlus is not tied to a specific framework, and as such can be used in multiple contexts. PromisePlus also features performance improvements without requiring programmers to extensively refactor their code. Finally, PromisePlus offers the same guarantee as classic promises: a call to `get` never produces an undefined value.

In the future we want to design static or dynamic analyses to optimize the granularity of the synchronization, allowing the programmer to focus only on where to put the synchronization points in their program.

References

1. Azadbakht, K., Boer, F., Bezirgiannis, N., Vink, E.: A formal actor-based model for streaming the future. Science of Computer Programming **186**, 102341 (12 2019). https://doi.org/10.1016/j.scico.2019.102341
2. Enmyren, J., Kessler, C.: Skepu: A multi-backend skeleton programming library for multi-gpu systems. pp. 5–14 (09 2010). https://doi.org/10.1145/1863482.1863487
3. Frølund, S., Agha, G.: Abstracting interactions based on message sets. In: Ciancarini, P., Nierstrasz, O., Yonezawa, A. (eds.) Object-Based Models and Languages for Concurrent Systems. pp. 107–124. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
4. Haller, P., Van Cutsem, T.: Implementing joins using extensible pattern matching. In: Lea, D., Zavattaro, G. (eds.) Coordination Models and Languages. pp. 135–152. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
5. Jin, H., Van der Wijngaart, R.F.: Performance characteristics of the multi-zone nas parallel benchmarks. Journal of Parallel and Distributed Computing **66**(5), 674 – 685 (2006). https://doi.org/https://doi.org/10.1016/j.jpdc.2005.06.016, iPDPS '04 Special Issue
6. Le, N., Guatto, A., Cohen, A., Pop, A.: Correct and efficient bounded fifo queues. pp. 144–151 (09 2013). https://doi.org/10.1109/SBAC-PAD.2013.8
7. Leca, P., Suijlen, W., Henrio, L., Baude, F.: Distributed futures for efficient data transfer between parallel processes. pp. 1344–1347 (03 2020). https://doi.org/10.1145/3341105.3374104
8. Liskov, B., Shrira, L.: Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. SIGPLAN Not. **23**(7), 260–267 (Jun 1988). https://doi.org/10.1145/960116.54016
9. Pop, A., Cohen, A.: Openstream: Expressiveness and data-flow compilation of openmp streaming programs. ACM Trans. Archit. Code Optim. **9**(4) (Jan 2013). https://doi.org/10.1145/2400682.2400712

A Algorithms of `get` and `set`

In this annex we present algorithms for the `get` function of PromisePlus in Algorithm 1, and for the `set` function of PromisePlus in Algorithm 2.

Algorithm 1 Algorithm of `get`

```
1: function GET(index)                          ▷ Get value associated with index
2:     while index > local_index do   ▷ The local index avoids a cost-heavy read of
   last if it is greater than the requested index
3:         local_index ← last
4:     end while
5:     return value associated with index
6: end function
```

Algorithm 2 Algorithm of `set`

```
1: procedure SET(index, value)   ▷ Associate value with index. If enough calls have
   been made, unblock calls to get with a lower index
Require: index is the next integer compared to the last call to set
2:     if (index - local_last) ≥ step then ▷ Reading from a local copy of last in the
   producer thread avoids cost heavy accesses to last
3:         last ← index                              ▷ Unblock get(i), i < index
4:         local_last ← index
5:     end if
6:     Associate value with index
7: end procedure
```