PPAM'99/Kazimierz Dolny, Poland/September 14-17, 1999

Experience with Co-Array Fortran and Pfortran as Tools for Parallelization of a Large-Scale Scientific Application

– with revisions –

Piotr Bała
Interdisciplinary Centre for Mathematical and Computational Modelling,
Warsaw University,
Pawińskiego 5a, 02-106 Warsaw, Poland,
Institute of Physics, N. Copernicus University,
Grudziądzka 5/7, 87-100 Toruń, Poland,
bala@icm.edu.pl

and

Terry Clark
Department of Computer Science
The University of Chicago
1100 E. 58th Street
Chicago, IL 60637
USA
twclark@cs.uchicago.edu

Abstract

Parallelization of scientific applications remains a nontrivial task typically requiring some programmer assistance. Key considerations for candidate parallel programming paradigms are portability, and intuitive use, at times mutually exclusive features. In this study two similar parallelization tools, Pfortran and Cray's Co-Array Fortran, are discussed in the parallelization of Quantum Dynamics, a challenging scientific application. We found good performance results with both compilers, which we discuss relative to an HPF implementation we also performed.

1 Introduction

Today's parallel computers routinely provide computational resources which can permit large-scale scientific simulations. However, the parallelization of complicated applications resist automatic parallelization, while low-level methods of parallelization lead the applicationist into complex and fragile methods obscuring the algorithmic intent behind the code. Still, in many instances efficient parallelization must be performed at the algorithmic level – rather than simply transforming existing sequential code – and arguably, the applicationist needs some say in the parallelization process.

Well-established libraries such as MPI [10, 9], PVM [8, 6] and *shmem* [5] provide a minimal parallel code development environment. Implementations of these approaches are available on a wide range of hardware platforms, an important advantage. There remains, however, significant barriers to parallelizing complex scientific applications.

To fill the gap between high-level approaches such as HPF and low-level approaches such as MPI, intermediate-paradigms have been developed which address important issues of 1) efficiency, 2) concise notation, and 3) access to low-level details. In this paper we consider two of these: Cray's Co-Array Fortran [11, 12] and Pfortran [2, 3]. The aim of this work is to compare these intermediate tools and their efficacy for parallelizing large-scale scientific applications. As a point of comparison we have include HPF [13] in this study.[1]

2 Parallelization tools

Co-Array Fortran and Pfortran use the SPMD program model. All processes execute the same text, with parallelism exploited by explicit partitions of data and control flow, or some combination of the two. Operations involving arrays can be easily performed in parallel with the programmer distributing arrays and iterations across the processors, allocating only the memory required for the local part of the array if necessary. Ordinary variables are replicated, with the scalar parts of the code performed independently and redundantly on each processor.

2.1 Pfortran

Pfortran is the Fortran translator for the Planguages,[2] for which there is also a C translator, *PC*. Planguage extensions to Fortran and C are few and intuitive to apply. The Planguages extend FORTRAN77 and C with *fusions* using a variant of the guarded-memory model [1]. *Fusion objects* are distributed variables formed syntactically with the Planguage operators @ and {}. In a serial program, we can write

$$i = j.$$

The clear intention is to move the value at the memory location represented by j to the memory location represented by i. The Planguages allow the same type of assignment, however, the memory need not be local, as in the following example in a two-process system

$$i@0 = j@1$$

with the intention to move the value at the memory location represented by j *at process 1* to the memory location represented by i *at process 0*. In message passing terms, the statement is a "fusion of send and receive" and corresponds to a fusion of a put and get in Co-Array Fortran.

[1] We also report results using the Portland Group's Adaptor preprocessor for HPF code.

[2] We use *Planguages* and *Pfortran* interchangeably in this paper.

The other Pfortran fusion operator consists of a pair of curly braces with a leading function, $f\{\}$. This operator lets one represent in one fell swoop the common case of a reduction operation where the function f is applied to data across all processes. For example, suppose one wanted to find the sum of an array distributed across $nProc$ processes, with one element per process. One could write

$$sum = +\{a\}$$

where a is a scalar at each process, but can be viewed logically as an array across *nProc* processes. With @ and $\{\}$, a variety of operations involving off-process data can be concisely formulated.

In the Planguage model, processes can interact *only* through the same statement, with such statements containing one or more fusion objects.[3] Synchronization is implied along with the @ and $\{\}$. A consumer will obtain off-process data as it is defined at the point in the program where the access is performed, i.e., at the @ and $\{\}$. If there is uneven progress by the consumer and producer, the implementation could either buffer the data, or stall the producer.

Programmers have access to the local process identifier called *myProc*. With *myProc*, the programmer distributes data and computational workload.[4] The Planguage translators transform user-supplied expressions containing fusion objects into algorithms with generic calls to a system-dependent library using, for example, MPI [10], PVM [8], TCGMSG [14] and the Intel message-passing interface. To port a Pfortran code from one machine to another, one simply recompiles it with Pfortran, next compiling the output FORTRAN77 with the native Fortran compiler. Pfortran is currently used on the Cray T3E/T3D, IBM SP2, SGI workstations, SUN multiprocessor computers, as well as on clusters of workstations.

2.2 Co-Array Fortran

Cray Co-Array Fortran are the other parallelization paradigm considered in this study [11, 12]. Co-Array Fortran introduces an additional array dimension for arrays distributed across processors. For example, the Pfortran statement

$$a(i)@0 = b(i)@1$$

is equivalent with the Co-Array Fortran statement

$$a(i)[0] = b(i)[1].$$

We note that all Pfortran data fusion statements can be written with co-arrays, however, the converse is not true. Where the co-array and fusion constructs do support the same type of data communication algorithm, Co-Array Fortran generally requires more changes in the legacy code than does Pfortran; however Co-Array Fortran provides structured distribution of user-defined arrays. Co-Array Fortran does not supply intrinsic reduction operations syntax. These algorithms must be coded by the user using point-to-point exchanges.

[3] There are ways to circumvent this, but it is outside the model and can lead to unexpected results (see the Dining Philosopher's problem implementation in [3]).

[4] Computational workload typically translates to loop iterations in the targeted technical applications.

While Co-Array Fortran and Planguage models are very similar, they have crucial differences, namely, with Co-Array Fortran:

1. [ ] do not imply synchronization; the programmer must insert synchronization explicitly to avoid race conditions and data consistency.

2. Inter-process communication with co-arrays *can* occur between separate statements.

The communication underlying Co-Array Fortran is realized through the Cray's *shmem* library, providing high communication efficiency. Cray's parallel extensions to Fortran are available only on selected CRAY architectures limiting the portability of Co-Array Fortran applications.

2.3 I/O

A typical I/O strategy for SPMD codes is to use a designated process to open and read data, which is then communicated to all other processes. Similarly, for outputting, non-replicated data is sent to and then output from the designated process.

3 Quantum Dynamics Algorithm

We parallelized the quantum dynamics (QD) code described in [4]. In QD, the dynamics of the quantum particle is given by the time-dependent Schroedinger equation modeled with a discrete representation of the wavefunction on a uniform Cartesian grid. A Chebychev polynomial method is used for the propagation of the wavefunction [17, 16, 18]. Simulations consist of the evolution of the potential and wavefunction, which are evaluated at each timestep during the simulations.

The propagation of the quantum particle wavefunction requires evaluation of the Hamiltonian acting on the wavefunction. The Chebychev polynomial method is used to propagate the wavefunction, with the Hamiltonian evaluated $2N$ times for each time-step, where N is the length of the Chebychev polynomial expansion.

The evaluation of the kinetic part of the Hamiltonian $\frac{1}{2m}\Delta_x\Psi(x)$ is performed using a Fast Fourier Transform (FFT) [15]. We used a three-dimensional parallel Fast Fourier Transform, PCFFT3D, from the Cray T3E Scientific libraries for this. PCFFT3D requires a slicewise distribution of the transformed matrix over processes.

4 Code parallelization

The QD application consists of several different types of calculations which we addressed independently for purposes of parallelization. Most parallelization is concerned with the distribution of data and calculations for the wavefunction propagation and potential function. The most time-consuming part of the code computes the potential and propagates the wavefunction on a three-dimensional spatial grid. The evaluations of the potential and wavefunction propagation at each grid point require *only local* information, so all variables evaluated on the distributed grid, such as the potential $V(x, y, z)$ and the wavefunction $\Psi(x, y, z, t)$, are evaluated in parallel. This step does not involve any communication once arrays are distributed.

The $N_x \times N_y \times N_z$ grid on which the potential and wavefunction are defined are mapped onto a $np_x \times np_y \times np_z$ logical processor array. The mapping is such that processes hold equally sized parts of the grid within a modulo factor. In practice, three-dimensional arrays are linearized to one-dimensional arrays of length $N_{all} = N_x \times N_y \times N_z$. The workload is already balanced using the uniformly distributed grid since the processes perform identical operations at each grid point.

Reductions are required to evaluate different mean values characterizing quantum particles, such as the energy, position, momentum and norm of the wavefunction. These properties are computed only once per timestep, but, because of the communication involved, this step can significantly impact the code performance. For the HPF code, the reduction operation is performed using the intrinsic (parallel) SUM function. In the Pfortran and Co-Array implementation, the partial sums are performed at each processor with a subsequent summation of the partial sums across all processors. For this purposes the Pfortran global-sum fusion operator is used with the reduction algorithms generated by the translator; Cray's Co-Array Fortran requires honing an algorithm consisting of a collection of point-to-point communications.

The parallel I/O consists of inputting and outputting the wavefunction and potential-energy arrays. These quantities are stored in a file in some order, so that processes must output their data in appropriate order. This is done with processes accessing files directly, or by individually routing the data through a designated process. Input and output is performed only at the start and finish of the simulation. Since I/O is performed outside the computational kernel, it will not be here discussed. Other filesystem operations do not incur significant overhead.

5 Results

In this section we discuss the performance results of the QD code resulting from code produced by the various compilers in this study. The Co-Array Fortran and Pfortran implementations are compared to HPF. For the HPF implementation, the Portland Research Group compiler and Adaptor preprocessor was used along with the *dalib* library [7].

The total single-processor execution times for the array update are shown in Figure 1a. Algorithmically, the workload scales linearly with array size; the linearity is reflected in the Fortran90 compiled code performance. For the other compilers, arrays greater than 4096 elements resulted in a performance decrease departing from linearity, possibly due to memory incoherence or memory bank conflicts. In addition, the array update is most efficiently performed with code generated by the Fortran90 compiler, reflecting aggressive optimization of the scalar code.

The reduction algorithm execution times are given in Figure 1b. In this, the single processor case, the reduction simply consists of a sum over all vector elements. As in the array update, there is also a discontinuity at 4096 array elements in reductions using HPF, Co-Array

Fortran and Fortran90.

The scalability of the array update for the various implementations is presented in Figure 2, illustrating almost linear scaling for all compilers. Results obtained for 1024 element array (Figure 2a) are effectively identical in all cases. For 4096 element arrays (Figure 2b) the Adaptor does not scale well, while the differences in the execution time between Portland Group HPF, Pfortran and Co-Array Fortran are insignificant. Not surprisingly, better performance scaling is obtained for the longer 4096 element arrays.

Recall that the reduction algorithm interprocess data exchange consists of a single scalar from each processor that holds the processors accumulated partial sum. Consequently, the cost to perform the reduction is dominated by the partial summation at each process. However, interprocess communication costs become more apparent with short vectors. The data in figure 3 reflect this. For the 4096 element vector, costs are scaling similarly for all compilers, with time decreasing with P due to effective parallel summations of vectors. On the other hand, for the 1024 element vector case, the communication costs emerge to create greater variation in performances. Interestingly, the co-array reduction algorithm, even though naively written as $\mathcal{O}(P)$, outperforms the $\mathcal{O}(\log P)$ Pfortran compiler generated algorithm. The difference is likely measuring *shmem* and *MPI*, underlying Co-Array Fortran and Pfortran, respectively.

6 Discussion

Our results show that efficient parallelization of the QD code can be achieved using Pfortran, Cray's Co-Array Fortran, and HPF. The least encouraging results are obtained when the Adaptor preprocessor is used for precompilation of the HPF code. However, the Portland Group implementation of HPF on the Cray T3E performs acceptably. This suggests a need to further develop the Adaptor preprocessor and its communication library (*dalib*), rather than failure of HPF in general.

The Pfortran and Co-Array Fortran implementations scale well with the number of processors. The small number of extensions and intuitive application of Pfortran and Co-Array Fortran are important considerations; HPF on the other hand, is complex and at times difficult to understand. We found the limited portability of Co-Array Fortran a disadvantage. The lack of an intrinsic array-reduction operation in Co-Array Fortran is not intrinsic to the language, with extensions apparently possible. Pfortran had performance comparable to Co-Array Fortran, without the portability limitations, while providing built in reduction operations, and the facility for user-defined ones. The reduction experiments suggest that *shmem* performance is much better than *MPI* performance, a likely contribution to the performance differences between Co-Array Fortran and Pfortran reduction operations.

In general, we found Co-Array Fortran and Pfortran both to be improvements over MPI for engineering parallel applications. In our opinion the two methods are complimentary paradigms, suitable for different algorithmic necessities. We plan to explore this concept further. The QD calculations are representative of a wide range of large-scale computational chemistry and scientific applications, suggesting general relevance of our findings concerning the efficacy of the parallelization tools.

Acknowledgements We thank Ridgway Scott for his comments and suggestions. Piotr Bala was supported by the Polish State Committee for Scientific Research. Terry Clark was supported by the National Partnership for Advanced Computational Infrastructure, NPACI. The computations were performed using the Cray T3E at the ICM, Warsaw University, with Planguage compiler development performed in part at the San Diego Supercomputer Center.

References

[1] Babak Bagheri. Parallel programming with guarded objects. Research Report UH/MD, Dept. of Mathematics, University of Houston, 1994.

[2] B. Bagheri, T. W. Clark, and L. R. Scott. Pfortran: A parallel dialect of Fortran. *ACM Fortran Forum*, 11(3):20–31, 1992.

[3] B. Bagheri, T. W. Clark, and L. R. Scott. Pfortran (a parallel extension of Fortran) reference manual uh/md-119. 1991.

[4] P. Bala, P. Grochowski, B. Lesyng, and J. A. McCammon. Quantum–classical molecular dynamics. models and applications. In M. Field, editor, *Quantum Mechanical Simulations Methods for Studying Biological Systems*, Springer-Verlag Berlin Heidelberg and Les Editions de Physique Les Ulis, pages 115–196. Springer-Verlag Berlin Heidelberg and Les Editions de Physique Les Ulis, 1996.

[5] R. Barriuso and A. Kniesi. Shmem user's guide for Fortran. 1998.

[6] A. Beguelin, J. Dongarra, G. A. Geist, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Conmputing.* MIT Press, Cambridge, 1994.

[7] T. Brandes. Adaptor programmer's guide (version 6.1). technical documentation. 1999.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, and R. Manchek. PVM 3 User's guide and reference manual. 1994.

[9] W. Gropp. *Using MPI.* MIT Press, 1996.

[10] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[11] R. W. Numrich. F^{--}: a parallel extension to Cray Fortran. *Scientific Programming*, 6(3):275–84, 1997.

[12] R. W. Numrich, J. Reid, and K. Kim. Writing a multigrid solver using Co-Array Fortran. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Recent Advances in Applied Parallel Computing*, Lecture Notes in Computer Science 1541, pages 390–399. Springer-Verlag Berlin, 1998.

[13] High Performance Fortran Forum. High performance Fortran language specification (draft) 1.0 edition. 1993.

[14] Robert J. Harrison. harrison@tcg.anl.gov, 1992.

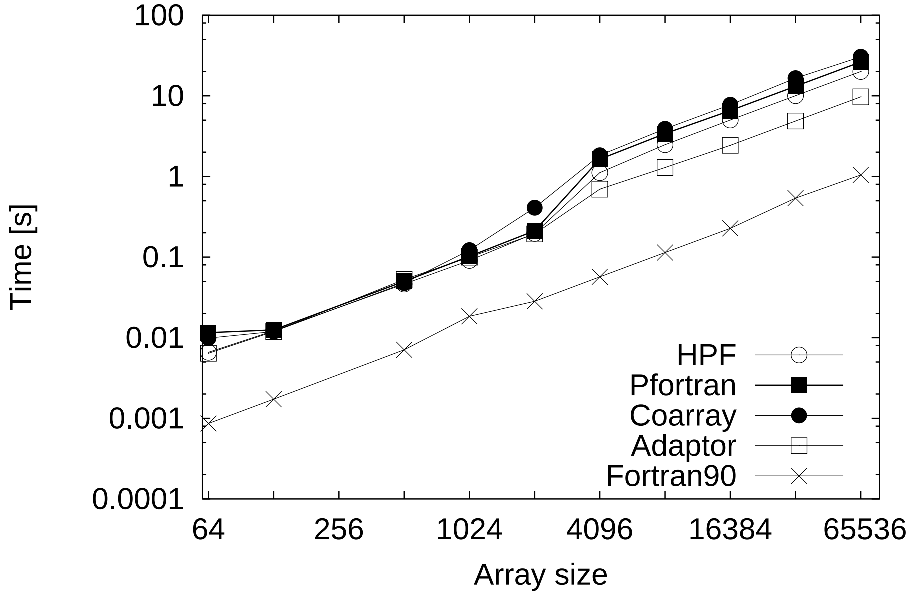
[15] D. Kosloff and R. Kosloff. A Fourier method solution for the time dependent Schroedinger equation as a tool in molecular dynamics. *J. Comput. Phys.*, 52:35–53, 1983.

[16] C. Leforestier, R. H. Bisseling, C. Cerjan, M. D. Feit, R. Freisner, A. Guldberg, A. Hammerich, G. Jolicard, W. Karrlein, H.-D. Meyer, N. Lipkin, O. Roncero, and R. Kosloff. A comparison of different propagation schemes for the time dependent Schrodinger equation. *J. Comput. Phys.*, 94(1):59–80, 1991.

[17] H. Tel-Ezer and R. Kosloff. An accurate and efficient scheme for propagating the time dependent Schroedinger equation. *J. Chem. Phys.*, 81:3967–3971, 1984.

[18] T. N. Truong, J. J. Tanner, P. Bala, J. A. McCammon, D. J. Kouri, B. Lesyng, and D. Hoffman. A comparative study of time dependent quantum mechanical wavepacket evolution methods. *J. Chem. Phys.*, 96:2077–2084, 1992.

a)



b)

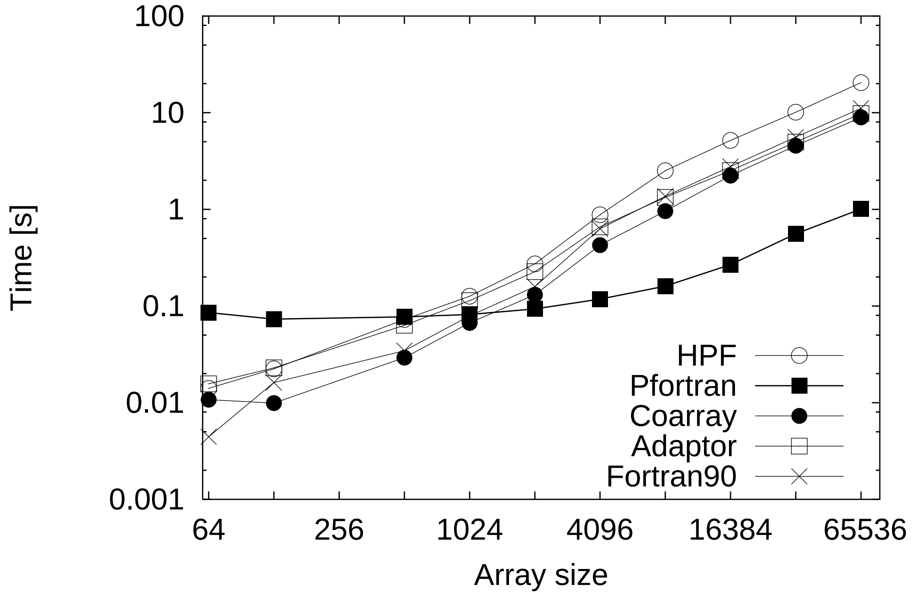


Figure 1: Single-processor performance for the (a) array update (potential energy calculation)) and (b) array reduction as function of array size.

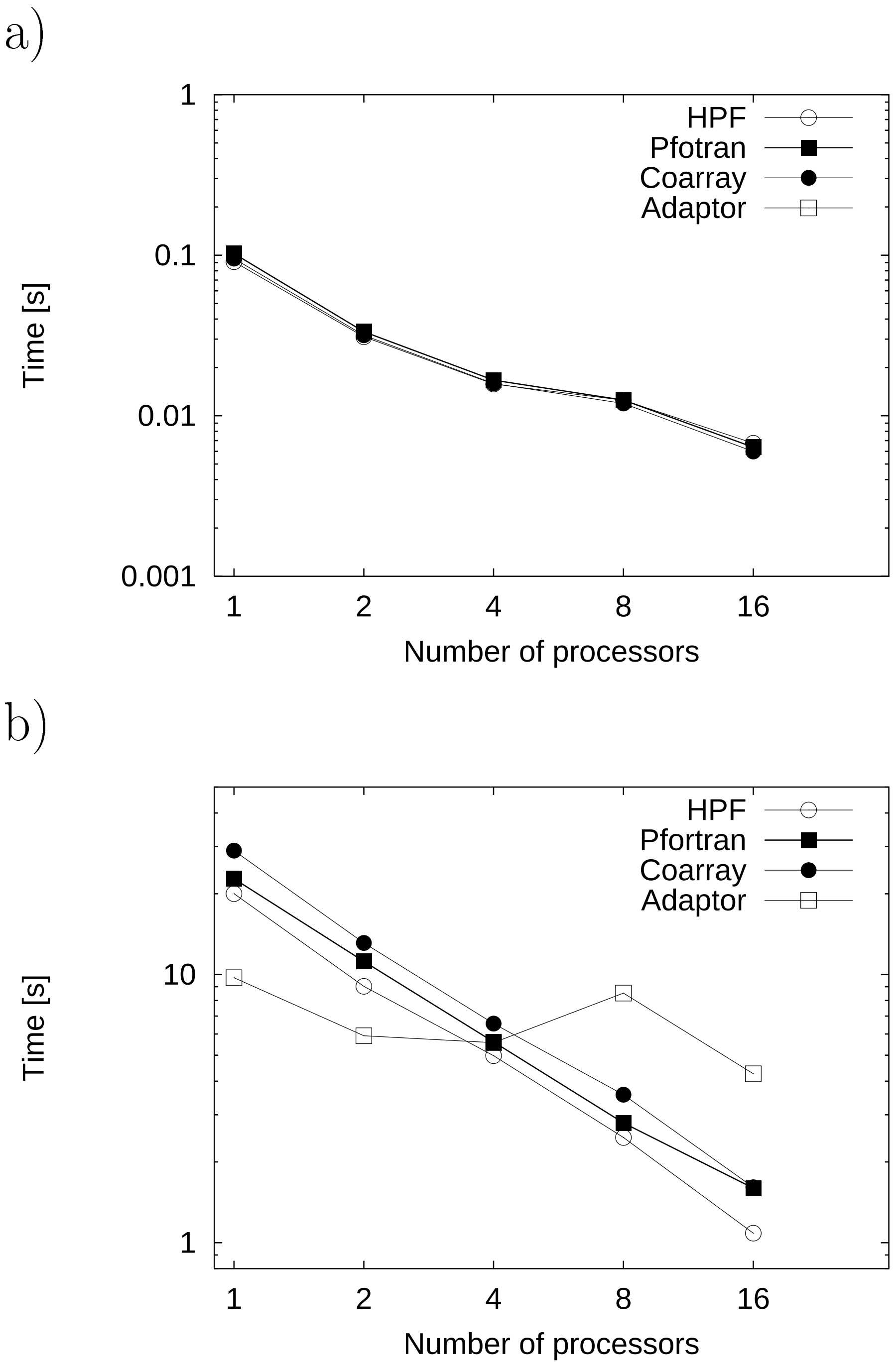


Figure 2: Performance of the array update calculations for (a) array length 1024 and (b) array length 65536.

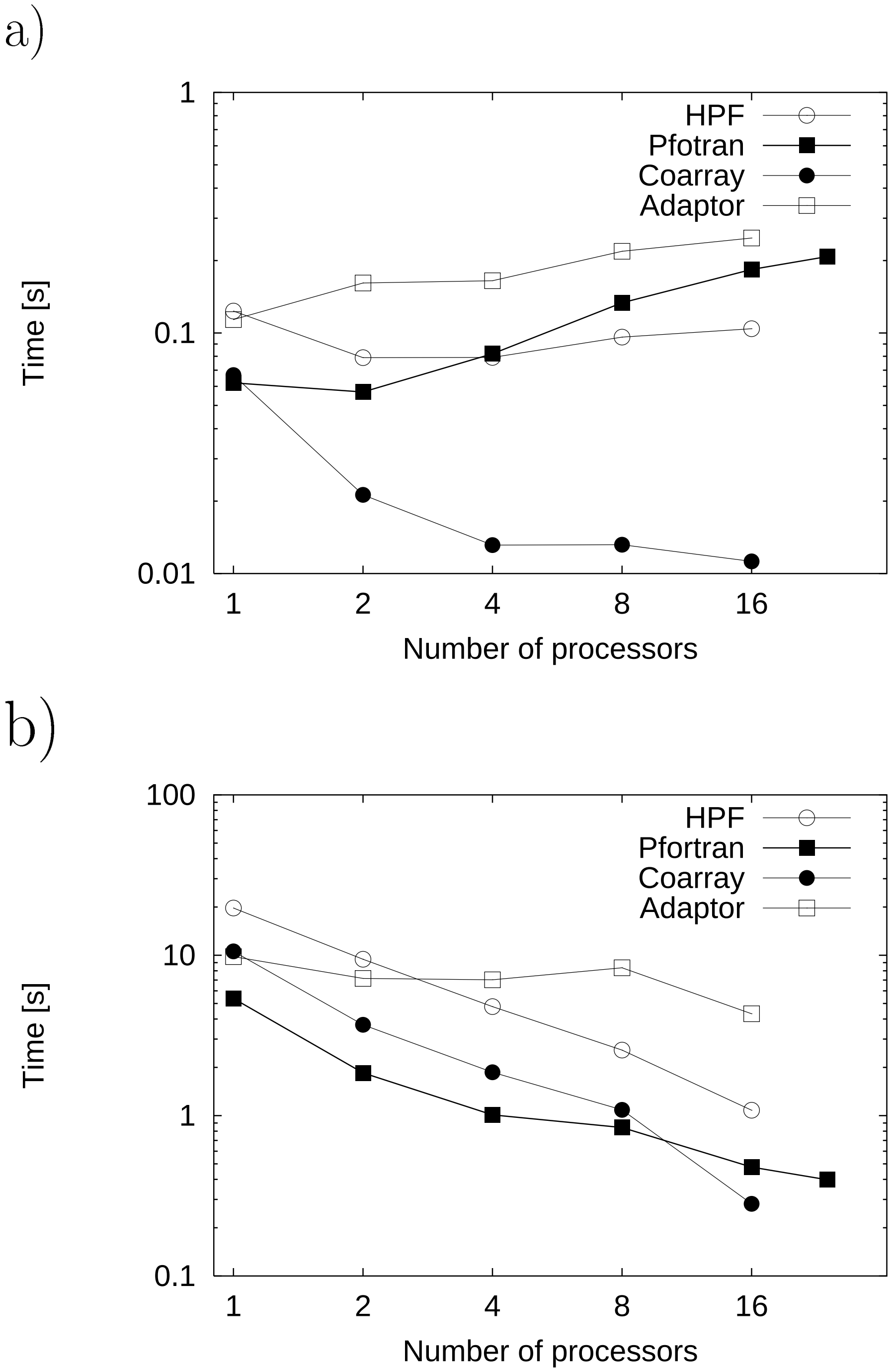


Figure 3: Performance of the array reduction for (a) array length 1024 and (b) 65536.