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

Piotr Bała[1,2] and Terry W. Clark[2]

[1] Interdisciplinary Centre for Mathematical and Computational Modelling, Warsaw University, Pawińskiego 5a, 02-106 Warsaw, Poland,
bala@icm.edu.pl
[2] Institute of Physics, N. Copernicus University, Grudziądzka 5/7, 87-100 Toruń, Poland,
[3] Department of Computer Science
The University of Chicago and Computation Institute
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, efficiency, 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 complex scientific application.

1 Introduction

Today's parallel computers routinely provide computational resources permitting simulations based on complex scientific models such as the models for describing the dynamics of quantum systems [1]. This area has experienced heightened activity with the recent progress in experimental physics, especially ultrafast optical spectroscopy and quantum electronics. Because analytical tools available for the description of quantum dynamical systems are limited, computational models must be used. Quantum dynamics simulations are usually based on numerical propagation of a quantum wavefunction obeying the time-dependent Schroedinger equation. This task can be easily performed for one-dimensional systems, but in most cases, the size of the system is limited by the available computational resources, i.e., computer memory and speed. However, this obstacle can be overcome with parallel computers.

For the task of parallelization, well-established libraries such as MPI [2], PVM [3,4] and *shmem* [5] can be used. An important advantage is availability of implementations of these on a wide range of hardware platforms. There remains, however, significant barriers to parallelizing complex scientific applications because complicated applications resist automatic parallelization, while low-level

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 511–518, 2000.

methods of parallelization lead the applicationist into complex and fragile methods obscuring the algorithmic intent behind the code.

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 efficiency, concise notation, and access to low-level details. In this paper we consider two of these: Cray's Co-Array Fortran [6,7] and Pfortran [8,9]. The aim of this paper is to compare these intermediate-level tools and their efficacy for parallelizing large-scale scientific applications starting from both specially-defined test cases and production code.

2 Quantum Dynamics Algorithm

We parallelized the Quantum Dynamics (QD) code described in [10]. 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 [1,11]. Simulations consist of the evolution of the wavefunction, which is evaluated at each timestep during the simulations.

The propagation of the quantum-particle wavefunction requires at each time step several evaluations of the Hamiltonian acting on the wavefunction. In practical calculations, both wavefunction and potential are represented on the discrete grid and all variables are calculated numerically on the grid.

The evaluation of the potential energy part of the Hamiltonian is a relatively lightweight computation consisting of the multiplication of a wavefunction by the value of the potential. The evaluation of the kinetic part $\frac{1}{2m}\Delta_x\Psi(x)$ is performed using a Fast Fourier Transform (FFT) [12]. 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.

3 Parallelization Tools

Co-Array Fortran and Pfortran were used in the parallelization of the QD code. Both languages fall into the SPMD program model with all processes executing the same program. Parallelism is 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.

3.1 Pfortran

Pfortran extends Fortran with several operators designed for intuitive use and concise specification of off-process data access. *PC* is the C counterpart to Pfortran; both are implementations of the Planguages. The crux of Pfortran and

PC are *fusions*, a variant of the guarded-memory model [13]. *Fusion objects* are distributed variables formed syntactically with the Planguage operators @ and {}.

In a sequential program the assignment statement specifies a move of a 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*.

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. 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. The Planguage translators transform user-supplied expressions containing fusion objects into algorithms with generic calls to a system-dependent library using, for example, MPI [2], PVM [3], 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. This allows for mixing Pfotran code code with traditional Fortran77 subroutines and functions which allows for easy parallelization of large parts of the code.

Pfortran currently targets on the Cray T3E/T3D, IBM SP2, SGI workstations, SUN multiprocessor computers, as well as on clusters of workstations.

3.2 Co-Array Fortran

Cray Co-Array Fortran is the other parallelization paradigm considered in this study [6,7]. 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. Also, variables used in Co-Array Fortran statements must be explicitly declared as co-arrays. While the co-array and fusion constructs 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-operation syntax. These algorithms must be coded by the user using point-to-point exchanges.

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

1. [ ] does 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.
3. Co-array variables must be explicitly defined in the code.

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.

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 evaluation. 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.

Evaluation of different mean values characterizing quantum particles, such as the energy, position, momentum and norm of the wavefunction requires various reduction operations. These properties are computed only once per timestep, but, because of the communication involved, this step can significantly impact the code performance. In Pfortran and Co-Array implementations, 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 point-to-point exchanges.

Parallel I/O consists of inputting and outputting the wavefunction and potential-energy arrays. These quantities are stored in a file in some order required for other applications, 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. Other filesystem operations do not incur significant overhead.

5 Results

We have explored performance of the simple tasks such as array reduction and array exchange as well as performance of QD application as a whole.

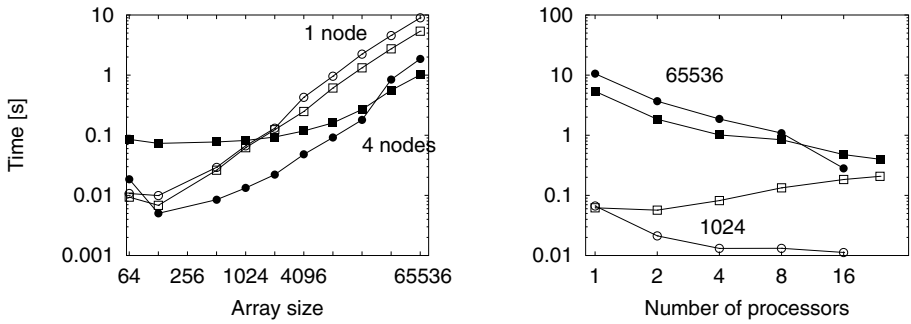

Fig. 1. Performance of array reduction for different array lengths and parallel execution as a function of processing elements. Pfortran results are denoted as squares and Co-Array results as circles.

The reduction algorithm execution times are given in Figure 1. In the single processor case, 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 Co-Array Fortran and Fortran90. Recall that the reduction algorithm interprocess data exchange consists of a single scalar from each processor accumulating a processor's 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 as shown in figure 1. Interestingly, the Co-Array Fortran 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.

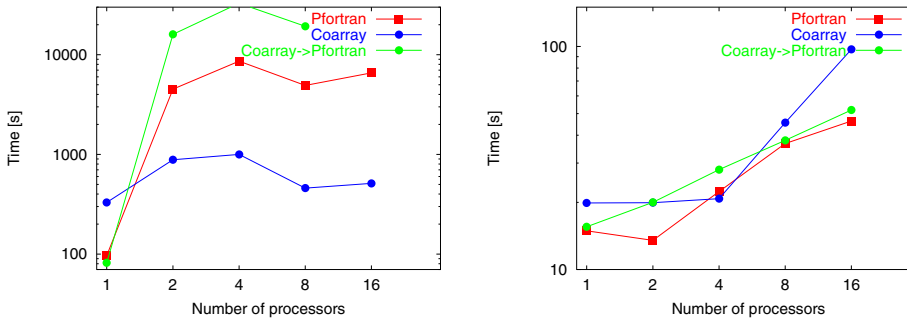

Fig. 2. Performance of the array exchange using short (left) and long (right) messages for different array length. Pfortran results are denoted as squares and Co-Array as circles. Full circles denote results for automatic translation of Co-Array code into Pfortran.

During QD code array exchange, the non-replicated data is sent to all processes in the order needed to obtain all data available at each processor. This task is performed using two different communication approaches. In the first, the arrays are sent element by element, which results in exchanging small portions of data. In the second model, communication is performed by a single exchange. In both communication approaches, the amount of exchanged data is the same therefore the of the exchange performance is dominated by the communication efficiency. Co-Array Fortran and Pfortran results in significant differences in performance due to the cost of communication initialization and process synchronization (Figure 2). The communication overhead is smaller while using Co-Array Fortran resulting in better performance for large numbers of short messages. Where large arrays are exchanged, Co-Array execution time increases almost linearly for number of processors grater than 2. The Pfortran code exhibits a slower ($O(log(P))$) increase of execution time with increasing numbers of processors as result of the underlying communication algorithm.

The overall performance for the 100 step propagation of quantum particle represented on the $32 \times 32 \times 32$ grid. presented in Figure 3 confirms high efficiency of parallelization. Both Pfortran and Co-Array codes scale linearly with the number of nodes, illustrating the viability of both parallelization tools. Small differences originate in differences in the performance of elementary array operations.

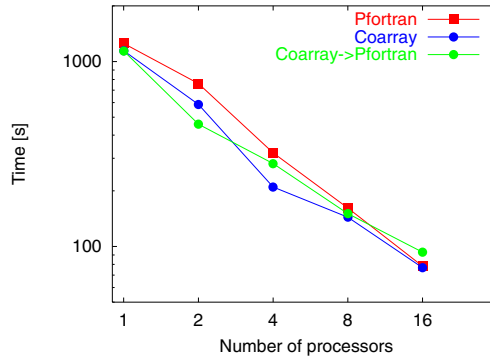

Fig. 3. Performance of the Quantum Dynamics code. Pfortran results are denoted as squares and Co-Array results as circles. Full circles denote results for automatic translation of Co-Array code into Pfortran.

6 Discussion

Our results show that efficient parallelization of large-scale scientific applications such as the QD code can be achieved using Pfortran and Cray's Co-Array Fortran.

The Pfortran and Co-Array Fortran implementations scale well with the number of processors, however, differences were observed in communication performance which results from the respective approaches to interprocess data movement. The small number of extensions and intuitive application of Pfortran and Co-Array Fortran are important considerations; HPF on the other hand, is more complex, resulting in code at times difficult to understand. We found the limited portability of Co-Array Fortran a disadvantage. Pfortran had performance comparable to Co-Array Fortran, and is without the portability limitations. In addition, the builtin Pfortran reduction operations along with the facility for user-defined ones are a definite plus for developing the Quantum Dynamics code. The Pfortran array exchange data reflects the communication algorithm used which results in logarithmic scaling. This could be implemented in Co-Array code, however this requires some additional programming effort.

In general, we found Co-Array Fortran and Pfortran both to be marked 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. 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.
2. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.
3. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, and R. Manchek. PVM 3 User's Guide and Reference Manual. 1994.
4. A. Beguelin, J. Dongarra, G. A. Geist, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Paralle Virtual Machine, A User's Guide and Tutorial for Networked Parallel Conmputing*. MIT Press, Cambridge, 1994.
5. R. Barriuso and A. Kniesi. Shmem User's Guide for Fortran. 1998.
6. R. W. Numrich. F^{--}: A Parallel Extension to Cray Fortran. *Scientific Programming*, 6(3):275–84, 1997.
7. 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.
8. B. Bagheri, T. W. Clark, and L. R. Scott. Pfortran: A parallel dialect of Fortran. *ACM Fortran Forum*, 11(3):20–31, 1992.
9. B. Bagheri, T. W. Clark, and L. R. Scott. Pfortran (a parellel extension of fortran) reference manual uh/md-119. 1991.
10. 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, 1996 pages 115–196.
11. 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.
12. 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.
13. Bagheri Babak. *Parallel programming with guarded objects*. Research Report UH/MD, Dept. of Mathematics, University of Houston, 1994.
14. Robert J. Harrison. harrison@tcg.anl.gov, 1992.