Performance Issues of an HPF–like compiler [1]

Maria Calzarossa, Luisa Massari [2], Daniele Tessera

Dipartimento di Informatica e Sistemistica, Università di Pavia, via Ferrata 1, I–27100 Pavia (Italy)

Abstract

The performance of High Performance Fortran (HPF) applications depends on their inherent parallelism and on the strategies adopted by compilers to distribute work and data among the processors. The evaluation of the performance of these applications has to consider all the aspects which in different ways determine such performance. The experimental approach presented here focuses on performance analysis of HPF kernels with the objectives of characterizing the preprocessing and communication overheads associated with the work and data distributions and of highlighting the benefits from reusing communication schedules.

Keywords: Performance evaluation; Measurements; High Performance Fortran; Compiler.

1 Introduction

The introduction of high–level data parallel languages, such as High Performance Fortran (HPF) [10, 7], has significantly eased the programming activity of distributed memory parallel systems. Parallel applications become easier to implement, maintain and port to different architectures. Application developers do not have to bother with any explicit call to low–level message passing primitives. They simply have to insert into their source codes HPF directives which specify the potential parallelism of the applications. The compilers then choose the most appropriate parallelization/optimization strategies and determine the communications required to distribute data and work among processors. The result of the compilation process is an explicit message passing

[1] This work was supported in part by the European Commission under the ESPRIT IV Long Term Research Project "HPF+" and the Working Group "APART", and by the Italian Research Council (CNR).
[2] Corresponding author. Email: massari@mafalda.unipv.it

data parallel code which contains the appropriate calls to the communication primitives. This code is then executed on the parallel system by each of the processors allocated to the application.

Apart from the parallel systems, the performance achieved by HPF applications strongly relies on the efficiency of the code generated by the compilers. Depending on the target of the evaluation process, the analysis of the performance has to focus on different aspects. For example, communication patterns and times are of little or even no interest for the developers of HPF applications, whereas compiler developers must rely on specific measures about the communications for investigating the efficiency of their design and implementation. Indeed, as already pointed out, application developers have no control over the communications which are automatically inserted by the compilers into the generated parallel codes.

In this framework, the evaluation of the performance of HPF applications becomes a multi-faced problem. In this paper, we discuss the performance issues to be addressed in the evaluation of these applications and of the corresponding compilers. Our studies focus on the HPF+ language, an optimized version of HPF for advanced industrial applications, which includes functionalities of both HPF–1 and HPF–2, HPF–2 Approved Extensions, and new features [2]. The performance of a set of kernel benchmarks representative of advanced applications, is analyzed with the aim of addressing the expressiveness of HPF+ constructs and the efficiency of the related compilation technology. Application developers benefit from these analyses to assess the efficiency of their codes. Compiler developers use these results to evaluate the strategies to distribute work and data.

The paper is organized as follows. Section 2 presents the performance analysis approach. An extensive set of experimental results, aimed at investigating the performance of selected kernels and the benefits from the use of some HPF+ constructs, is presented in Section 3. A brief summary of related work is described in Section 4. Finally, in Section 5 we draw a few conclusions.

2 Performance analysis approach

The analysis of the performance achieved by parallel applications has to focus on all the aspects which, in different ways, are responsible of such performance. In the case of HPF applications, the analysis has to include the evaluation of the various types of overhead associated with the distribution of work and data among the allocated processors. These overheads, which refer to the code generated for this purpose by the compilers, are “costs” which influence the efficiency and the performance of the applications. Hence, it is necessary to characterize and quantify these overheads in terms of their memory, computation and communication requirements.

To address these performance issues, an experimental approach, which allows the analysis of the performance at the level of the source–code, has to be applied. In this framework, a tight integration of compilers and performance analysis tools is required. Compilers identify and instrument the individual components of the code (e.g., loops, subroutines) and, within each of them, the various phases associated with work and data distributions. Performance analysis tools have to recognize, within the measurements collected at run–time, the measures related to the various components of the code and quantify their performance and the corresponding overheads. In order to obtain a detailed evaluation of these overheads, a breakdown of the performance of each component of the code according to the phases associated with work and data distributions has to be derived.

The approach toward performance analysis consists of various steps dealing with instrumentation and monitoring of the applications and analysis of the collected measurements. In our case, this approach is based on the integrated use of the VFC compiler [3] and of the Medea performance tool [4]. This integrated environment [5] allows us to derive a detailed characterization of the behavior of HPF+ applications and evaluate the preprocessing and communication overheads introduced by VFC to cope with work and data distributions.

At compile time, the instrumentation system embedded within VFC inserts the appropriate instrumentation into the explicit message passing parallel codes being generated. Measurements are collected during the execution of these codes. Medea performs a postprocessing of these measurements and derives performance metrics and parameters able to explain the behavior and the performance of the applications.

From the analysis of the measurements, the overall performance together with the overheads experienced by the applications is evaluated. The analysis is based on profiling techniques which derive the performance (e.g., execution time, communication time) of the various components of the applications and provide a breakdown according to the preprocessing and communication overheads experienced within each component. These breakdowns assess the efficacy of the HPF+ directives and clauses used to specify the parallelism (e.g., `INDEPENDENT`, `ON HOME`) and the distribution of the data (e.g., `BLOCK`, `GEN_BLOCK`). Moreover, our approach, as most experimental approaches, allows us to easily address scalability issues within HPF applications, by characterizing the scalability of the preprocessing and communication overheads introduced by the compilers. Indeed, as the number of allocated processors increases, the overheads might dominate the execution time of the applications.

3 Experimental results

As already pointed out, HPF codes might experience significant overheads because of work and data distribution performed by the compilers. The aim of our performance studies is to evaluate the preprocessing and communication overheads introduced by compilers, and the impact of the HPF+ features used to reduce these overheads. In particular, our experiments focus on `INDEPENDENT` loops, that is, loops whose iterations are independent and may be executed in any order. The choice of these loops is two–fold. These loops represent the core of most advanced scientific applications and are a valuable source of parallelism.
The compilers parallelize these loops by distributing work and data among the allocated processors. Because of the irregular data accesses characterizing most of `INDEPENDENT` loops, appropriate strategies have to be applied. The VFC compiler adopts a generalization of the inspector/executor strategy [14] for the parallelization of `INDEPENDENT` loops. Five phases, namely work distributor, inspector, gather, executor, and scatter, can be identified in the parallel code generated by VFC. Among these five phases, the executor only represents the actual execution of the loop. The remaining phases represent the preprocessing and communication overheads introduced by VFC for distributing work and data among the allocated processors. The work distributor deals with the computation of the execution set, that is, the set of loop iterations to be executed on each processor. The inspector performs a run–time analysis of the loop in order to determine the access patterns of distributed data and generate the corresponding communication schedules for non local accesses. Then, non local data is gathered according to the computed schedules, and the transformed loop is executed, during the executor phase. Finally, in case non local data is updated during the execution of the loop, a scatter is performed.

Our experimental results focus on two HPF+ kernels which are representative of the main functional units of the PAM-CRASH solver [6]. These kernels are a subset of the finite element solver and implement the stress–strain calculations of a typical element formulation [11]. The kernels have been parallelized by means of VFC. Through the instrumentation system embedded within VFC, `INDEPENDENT` loops have been instrumented, in order to measure the associated overheads. The kernels have been executed on a Meiko-CS2 platform. Then, measurements collected during their execution have been analyzed by means of the Medea tool.

The preprocessing and communication overheads introduced by VFC to distribute work and data are evaluated on two `INDEPENDENT` loops. These loops are the core of an HPF+ kernel benchmark which performs a shell element calculation, and contains the stress–strain calculations over 25,600 elements, driven by changing nodal points, and accumulating the resulting forces as nodal quantities. In particular, an `INDEPENDENT` loop (`loop1`) is responsible for gathering the nodal coordinates, and computing forces. The other `INDEPENDENT` loop (`loop2`), after gathering the coordinates, performs a sum scatter of the computed forces. Both loops contain the `NEW` clause used to define variables that are private to each loop iteration. `Loop2` contains also a `REDUCTION` clause, for performing a sum reduction across all loop iterations. The arrays used by both loops are distributed onto a logical array of processors according to the `(*,BLOCK)` directive.

Figure 1 shows, for each of the two loops, the overall execution and communication times and the times of the phases of the inspector/executor strategy. The times refer to runs with 8 and 16 processors. As can be seen, the time

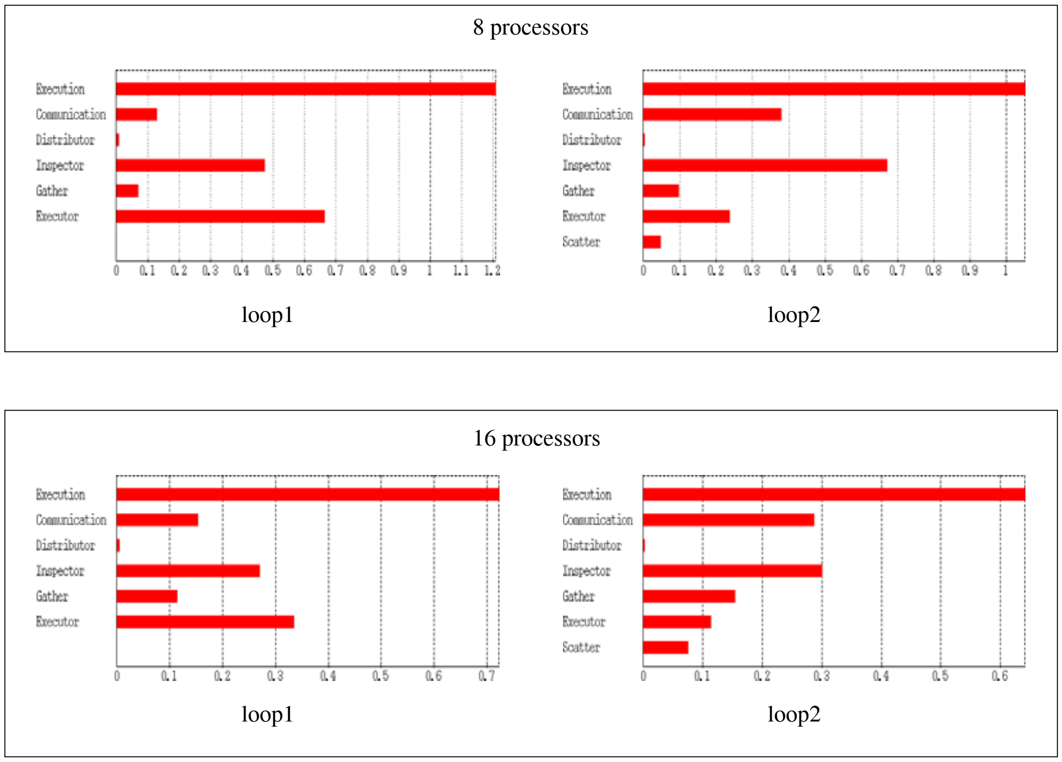

Fig. 1. Execution and communication times of the two loops and times of the phases of the inspector/executor strategy in runs with 8 and 16 processors. Times are expressed in seconds.

spent in the work distributor phase is always less than 0.007 seconds, whereas the time of the inspector phase is very large and may dominate the executor time. `Loop1` does not require the scatter phase since it does not update any non local array element. Moreover, the analysis of the communication times shows that `loop2` is characterized by a larger communication time. This is due to the sum scatter operations induced by the `REDUCTION` clause and to communications required to access non local array elements. In the run with 8 processors, the communication times for `loop1` and `loop2`, which are equal to 0.1277 and 0.3783 seconds, account for 11% and 36% of the execution times of the two loops, respectively. These percentages increase up to 21% and 44% in the run with 16 processors. This is a consequence of the imbalanced conditions that arise when a larger number of processors is involved in communication activities.

In order to better analyze the costs due to the parallelization strategies of VFC, we have subdivided the execution time of the two loops into the time spent by the executor for the actual execution of the loop and into the preprocessing and communication overheads.
Figure 2 presents, for `loop1` and `loop2` in a run with 16 processors, the fractions of the overall execution time of each loop spent for the actual execution and for the preprocessing and communication overheads. These overheads globally account for about 54% and 83% of the overall execution times of `loop1` and `loop2`, respectively. Indeed, as already stated, `loop2` is characterized by a larger communication overhead.

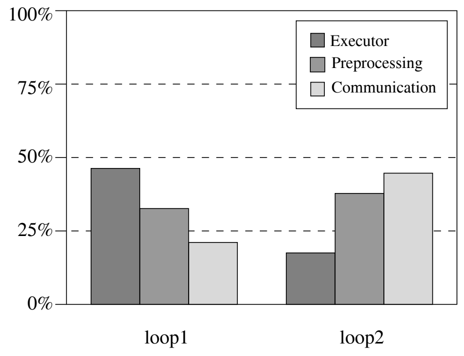

Fig. 2. Fractions of the overall execution time due to actual execution, preprocessing and communication overheads of `loop1` and `loop2` in a run with 16 processors.

The scalability of the preprocessing and communication overheads has been analyzed by looking at their behavior as a function of the number of allocated processors. Figure 3 shows the scalability of `loop1`. As can be seen, the executor time and the preprocessing overhead decrease as the number of processors increases, whereas the communication time always increases. This is due to the reduction of the execution set of each processor. When the number of iterations to be executed by each processor becomes too small, the communication times dominate the execution times and the preprocessing overhead becomes comparable to the executor time.

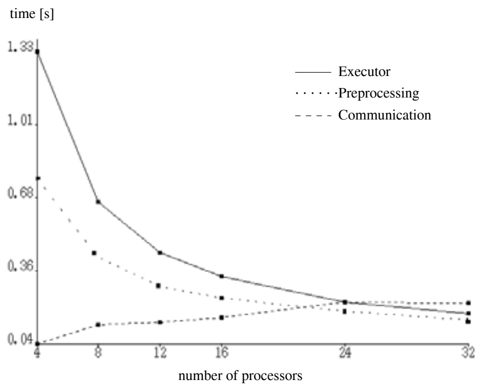

Fig. 3. Executor time and preprocessing and communication overheads for `loop1` as a function of the number of allocated processors.

In order to predict, for an arbitrary number of allocated processors, the overheads due to work and data distributions, we have applied fitting techniques to derive analytic models of the preprocessing and communication overheads. Figure 4 shows, for both `loop1` and `loop2`, the preprocessing overheads measured for a number of processors ranging from 4 up to 60. The corresponding analytic models are also shown. These models are given by $0.059 + \frac{2.830}{p}$ and $0.075 + \frac{2.746}{p-0.125}$, where p denotes the number of allocated processors.

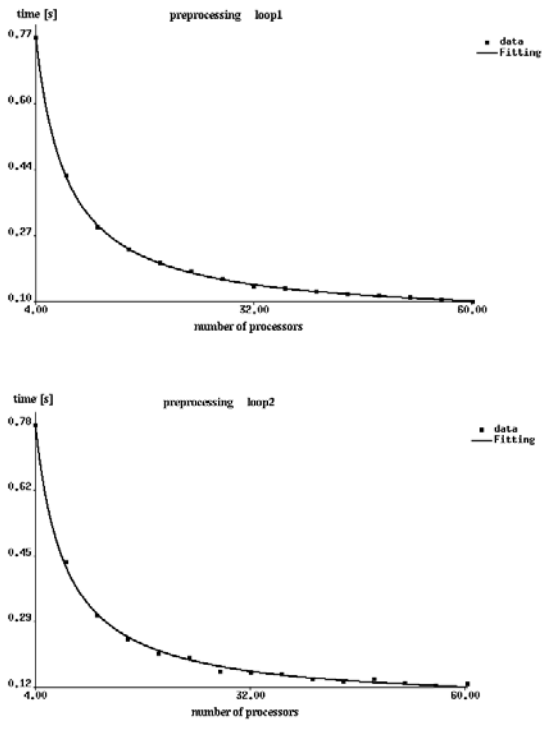

Fig. 4. Fitting of the preprocessing overhead for `loop1` and `loop2`, as a function of the number of allocated processors.

The analysis of each phase of the inspector/executor strategy from a functional view point highlights the types of communication primitive issued within the two loops. The work distributor does not contain any communication at all. The inspector phase results in six types of MPI communication primitives (i.e., `Probe`, `Isend`, `Recv`, `Get_count`, `Waitall`, and `Allreduce`). The gather and scatter phases contain `Alltoall` communication primitives only. The time spent in `Alltoall` and `Allreduce` collective communications accounts for more than 99% of the total communication time.
Figure 5 shows for `loop2` the times spent by each of the 8 allocated processors in `Allreduce` and `Alltoall` communications. As can be seen, even though all processors exchange the same amount of data, the time spent by processor `p2` in both types of communication is the smallest. This means that processor `p2` is not well synchronized with the other processors. Indeed, collective communications highly stress the interconnection network of the parallel system and may result in losses of synchronization among the processors.

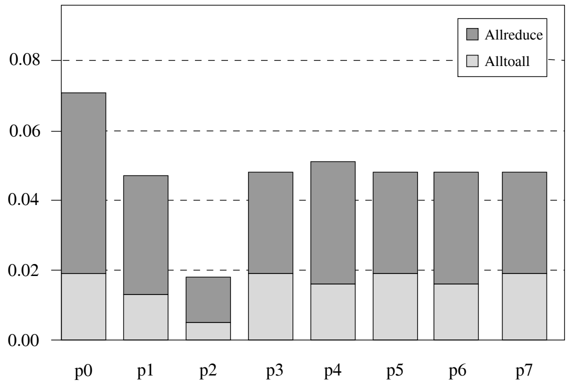

Fig. 5. Times (expressed in seconds) spent by each of the 8 allocated processors for the `Allreduce` and `Alltoall` communications of `loop2`.

3.2 Schedule reuse analysis

The characterization of the overheads has shown that the inspector phase may be particularly “expensive” and, in some cases, may dominate the actual execution time of the loop. It is then important to amortize this cost across subsequent executions of the loops, as in the case of loops nested into a time–step loop, or across different loops with the same access patterns. The `REUSE` clause works for this purpose, in that it asserts that communication schedules previously computed in the inspector phase are invariant throughout the execution of the application and can be reused.

The performance benefits attained from `REUSE` clause have been studied by analyzing two `INDEPENDENT` loops. These loops are part of an HPF+ kernel which deals with stress–strain calculations over 35,000 shell elements and

35,571 nodal points. The loops are responsible to compute the forces for each element and to transfer these forces to the nodal points of the element itself.

One `INDEPENDENT` loop (`loopA`) is characterized by `NEW` and `ON HOME` clauses. The `ON HOME` clause asserts that iterations have to be executed on the processor that owns the specified array elements only. The other `INDEPENDENT` loop (`loopB`) contains `NEW`, `ON HOME` and `REDUCTION` clauses and uses four arrays to compute the internal forces; indirect addressing is used together with a scatter operation. All the arrays used by both loops are distributed according to the `BLOCK` directive.
The two loops are nested in a time marching scheme iterated 125 times. Since communication schedules used in the loops are invariant for all the iterations of the loops, the use of the `REUSE` clause should improve the performance. The benefit from using this clause has been investigated by running the kernel without and with the clause itself.

Figure 6 shows the breakdown of the execution times of `loopA`, in runs with 8 and 32 processors, without and with the `REUSE` clause. The breakdown refers to the times spent by the inspector, gather, executor, and scatter phases. Note that the time spent by the work distributor has not been plotted because it is always negligible. As can be seen, thanks to the `REUSE` clause, the execution time decreases to 50% and 56% in the runs with 8 and 32 processors, respectively. This is a consequence of the large decrease of the time spent in the inspector phase, which is executed only once, that is, during the first iteration.

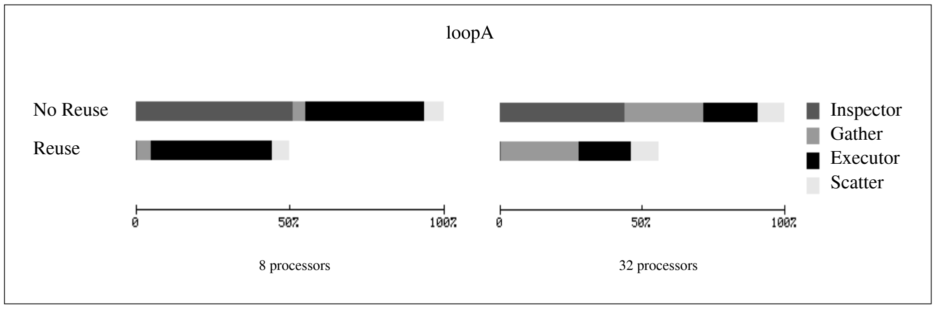

Fig. 6. Breakdown of the execution time of `loopA` without and with the `REUSE` clause in runs with 8 and 32 processors.

Figure 7 shows, as a function of the number of allocated processors, how the overall execution times of the two loops scale in runs without and with the `REUSE` clause. For `loopA`, the execution time decreases to 44%, in a run with 4 processors, and to 67%, in a run with 60 processors.
The analysis of the execution times of `loopB` without and with the `REUSE` clause shows that the benefit from the `REUSE` is bigger for this loop than for `loopA`. The execution times reduce to 19% and 42% in runs with 12 and 60 processors, respectively.

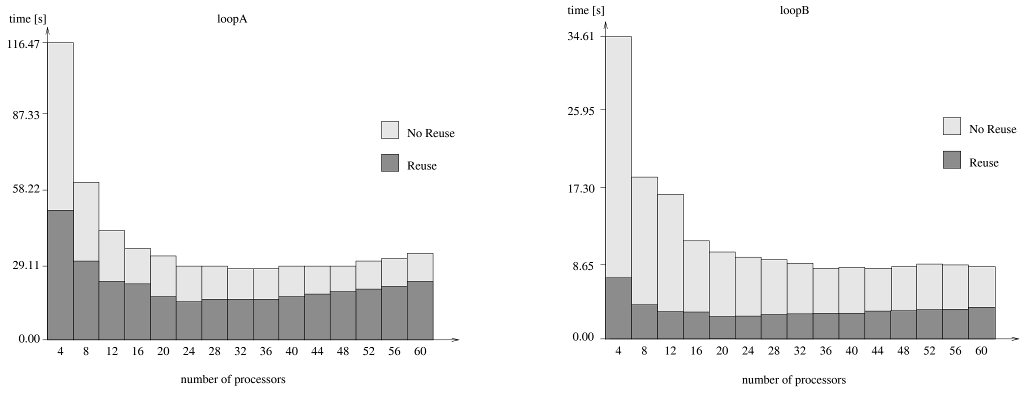

Fig. 7. Execution times of `loopA` and `loopB` without and with the `REUSE` clause, as a function of the number of allocated processors.

To gain better insights in the behavior of `loopB`, the breakdown of its execution time has been analyzed. This loop does not require any gather of the data because each processor accesses local arrays only. As can be seen from Fig. 8, the use of the `REUSE` clause reduces the execution time to 22% in a run with 8 processors and to 34% in a run with 32 processors. In particular, the times due to the inspector phase decrease from 14.23 to 0.14 seconds and from 5.61 to 0.08 seconds, respectively. Hence, it is very profitable to use this clause, whenever possible.

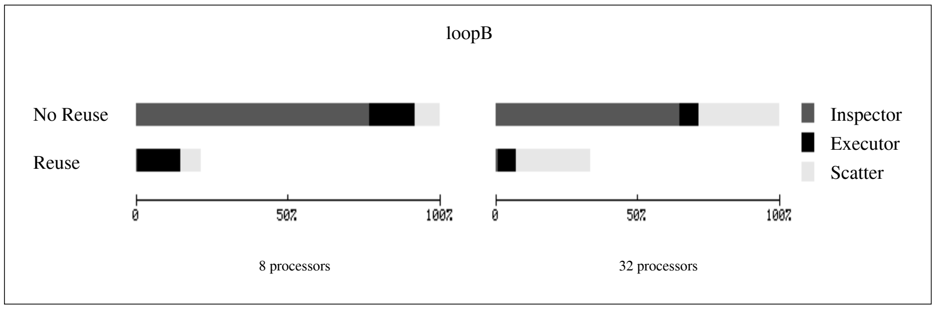

Fig. 8. Breakdown of the execution time of `loopB` without and with the `REUSE` clause in runs with 8 and 32 processors.

4 Related Work

A large variety of tools for performance analysis of parallel applications exists [8, 12, 13]. Most of these tools deal with very detailed analysis of the behavior of message passing codes.
The focus for HPF applications has to be quite different. Although their evaluation is typically carried out by looking at timestamps and speedups (see e.g., [16]), such an approach fails to provide details about the behavior of the codes. Hence, performance evaluation has to be based on a source–level analysis.

Few tools deal explicitly with HPF applications. In [1], the integration of the Pablo performance tool with a Fortran D compiler is presented. The Fortran D compiler records information describing the relationships between performance instrumentation and the original source code. Then, the Pablo tool correlates the dynamic behavior of the application with the source code.
The MPP Apprentice [15] is a tool integrated with C++, Fortran 90 and HPF compilers. It reports time statistics for the whole application, as well as for `DO` loops across all the processing elements.
The approach followed in [9] deals with the visualization of data placement within HPF applications. Various types of view show the communication activity in the context of the array distribution. This visualization provides important insights to understand the behavior of the code.

5 Conclusions

The important lesson we have learned from our studies is two–fold. The interests and the requirements of application and compiler developers are very different. Performance tools have to take into account such requirements. Profiling is a good figure to evaluate the performance of HPF applications and to investigate their behaviors. This profiling has to be strictly coupled with the source code in order to highlight the contributions due to the actual execution of the code and to preprocessing and communication overheads. Such an objective requires a good integration between performance tools and compilers.

There are several open issues to be addressed; one major issue deals with supplying into the performance tools mechanisms able to provide application developers with a better aid for tuning and optimizing their codes. These types of approach will require an even tighter integration between compilers and performance tools.

Acknowledgment

The authors wish to thank Viera Sipkova of the Institute for Software Technology and Parallel Systems of the University of Vienna for her invaluable assistance in running VFC.

References

[1] V.S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J. Wang, and D. Reed. Integrating Compilation and Performance Analysis for Data–Parallel Programs. In M.L. Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, editors, *Debugging and Performance Tuning for Parallel Computing Systems*, pages 25–51. IEEE Computer Society, 1996.

[2] S. Benkner. HPF+: High Performance Fortran for Advanced Industrial Applications. In P. Sloot, M. Bubak, and B. Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 797–808. Springer, 1998.

[3] S. Benkner, K. Sanjari, V. Sipkova, and B. Velkov. Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC. In P. Sloot, M. Bubak, and B. Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 816–827. Springer, 1998.

[4] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Medea: A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology*, 3(4):72–80, 1995.

[5] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Integration of a Compilation System and a Performance Tool: The HPF+ Approach. In P. Sloot, M. Bubak, and B. Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 809–815. Springer, 1998.

[6] J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, and M. Holzner. Performance Issues of the Parallel PAM–CRASH Code. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(1):3–11, 1997.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification - Version 2.0. Technical Report, Rice University, January 1997. Available at: http://www.crpc.rice.edu/HPFF/.

[8] A. Hondroudakis. Performance Analysis Tools for Parallel Programs. Technical Report, Edinburgh Parallel Computing Center – University of Edinburgh, July 1995. Available at: http://www.epcc.ed.ac.uk/epcc-tec/documents.html.

[9] D. Kimelman, P. Mittal, E. Schonberg, P.F. Sweeney, K. Wang, and D. Zernik. Visualizing the Execution of High Performance Fortran (HPF) Programs. In *IPPS'95: 9th International Parallel Processing Symposium*, pages 750–759. IEEE Computer Society Press, April 1995.

[10] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, 1994.

[11] G. Lonsdale, S. Meliciani, G. Mozdzynski, and H. Schiffermueller. Report on Project Benchmarks Rel. 1. Document HPF+ D1.1a, April 1997. Available at: http://www.par.univie.ac.at/hpf+/unsecure/d11a.ps.gz.

[12] C.M. Pancake, M.L. Simmons, and J.C. Yan, editors. *Computer*, Special

Issue on *Parallel and Distributed Processing Tools*, 28(11), 1995.

[13] C.M. Pancake, M.L. Simmons, and J.C. Yan, editors. *IEEE Parallel and Distributed Technology*, Special Issue on *Performance Evaluation Tools*, 3(4), 1995.

[14] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run–Time Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[15] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D. In K.M. Decker, editor, *Programming Environments for Massively Parallel Distributed Systems*, pages 333–345. Birkhauser Verlag, 1994.

[16] H.W. Yau, G.C. Fox, and K.A. Hawick. Evaluation of High Performance Fortran Through Application Kernels. In B. Hertzberger and P. Sloot, editors, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 772–780. Springer, 1997.