Performance Analysis of HPF+ Kernels *

M. Calzarossa, L. Massari, A. Merlo
Dipartimento di Informatica e Sistemistica
Università di Pavia
via Ferrata 1, I–27100 Pavia, Italy

Abstract *Loops represent the core of most applications in that they contain the bulk of the computations. An efficient parallelization of loops leads to good overall performance of the applications. High Performance Fortran provides a set of directives to be used to exploit the potential parallelism of the code. Our studies focus on the analysis of the performance of* `INDEPENDENT` *loops with the aim of pointing out the efficiency of the compiler as a function of the various HPF directives. Performance improvements achievable by means of the* `REUSE` *directive are also investigated.*

Keywords: High Performance Fortran, kernel benchmarks, parallelizing compilers, performance evaluation

1 Introduction

The introduction of High Performance Fortran (HPF) [1], [2] has changed the programming approach adopted in distributed memory parallel systems. HPF provides the language support required by Single–Program Multiple–Data applications which deal both with regular problems as well as with irregular problems characterized by indirect data accesses.
The potential parallelism of the application is expressed by means of the HPF directives. Compilers have then to take advantage of these directives to choose the most appropriate strategies to distribute data and work among the allocated processors and to generate the parallel code, which will also contain explicit communications. In particular, compilers have to:

- partition the data sets into subsets, each of which is assigned to a processor defined as the owner of the subset itself;
- distribute the computations such that each processor only executes the operations onto the data it owns;
- insert, into the generated parallel code, explicit message passing communication statements in order for the processors to access non–local data.

As a consequence, the performance of HPF applications depends on the efficiency of the compiler and of its runtime system.

In this paper we focus on the study of the performance of a few kernels with particular emphasis to the characterization of `INDEPENDENT` loops. Loops represent the core of most applications, in that they contain the bulk of the computations. Moreover, depending on the directives associated to `INDEPENDENT` loops, different parallelization strategies can be applied by the compiler and different performance can be achieved. Our studies are aimed at the characterization of the performance of the inspector/executor strategy adopted by the compiler and at the analysis of the performance improvements achievable by means of the `REUSE` directive.
This work is part of the Esprit Long Term

*This work was supported in part by the European Commission under the ESPRIT IV Long Term Research project n. 21033 "HPF+", by the Italian M.U.R.S.T. and by the Italian Research Council (C.N.R.).

Research project "HPF+"[1] whose aim is to develop an HPF–like language (HPF+) and its compiler for optimizing advanced HPF applications.

The paper is organized as follows. Section 2 describes the main characteristics of `INDEPENDENT` loops in the framework of the HPF+ language and compiler. Experimental results dealing with the evaluation of these loops with respect to the inspector/executor strategy are presented in Section 3. The performance improvements obtained by applying the `REUSE` directive are discussed in Section 4. A few conclusions are finally drawn in Section 5.

2 INDEPENDENT loop characterization

As already pointed out, the efficient parallelization of loops is a crucial aspect for applications to achieve good performance on parallel systems.
In what follows we provide a brief overview of the directives specific for the parallelization of loops. Such directives, which are part of the HPF+ language [3], are also part of the kernels developed within the HPF+ project and used as a target for our performance studies.
The `INDEPENDENT` directive asserts that the iterations in the following `DO` loop may be executed independently (e.g., in parallel) without changing the semantics of the application. An `INDEPENDENT` loop, in turn, may contain a `NEW` clause which declares local variables used and updated within each iteration without causing loop–carried dependencies.
The use of the `INDEPENDENT/NEW` directives by themselves may not be enough for the compiler to generate an efficient parallel code. The `ON` clause provides compilers with an extra aid in choosing the most appropriate work distribution, that is, the most efficient mapping of loop iterations to processors. Indeed, this clause specifies exactly which processor is going to execute which iterations. Moreover, when a reduction operation is executed across a set of iterations to assign the computed result to a scalar variable, a dependence is introduced. This dependence is solved by using the `REDUCTION` directive.

The compiler of the HPF+ language is based on extensions of the Vienna Fortran Compilation System (VFCS) [4]. The parallelization of irregular `INDEPENDENT` loops adopts the inspector/executor strategy which combines compile–time and run–time analyses [5], [6]. Communications required to access non–local data are performed outside the loop. This means, for example, that non–local assignments within an iteration can be committed to the processors that own that particular data only at the end of the loop.
The inspector/executor strategy transforms the execution of an `INDEPENDENT` loop into five phases, namely, the work distributor, the inspector, the gather, the executor, and the scatter. The work distributor deals with the computation of the execution set, that is, each processor computes the set of loop iterations to be executed. The inspector performs a run–time analysis of the loop in order to generate the appropriate communication schedules. The gather is responsible of the communications required to access non–local data used within the loop. The executor represents the actual computation to be done on the execution set. Finally, the scatter deals with the communications required to update the non–local data modified in the loop.
Note that, in the current implementation of VFCS, both the work distributor and the executor do not contain any communication at all. Furthermore, the scatter phase is only required when the `REDUCTION` directive and/or the `ON` clause are used.

In such a framework, many performance issues need to be addressed. In particular, the influence on the performance of HPF+ directives and of the parallelization strategies adopted by the compiler has to be studied. All these issues are easily addressed within the HPF+ project because VFCS provides

[1] More information can be found at http://www.par.univie.ac.at/hpf+.

all the facilities for instrumenting and monitoring HPF+ kernels and because of the integration of VFCS with Medea performance tool [7], [8]. Medea provides quantitative and qualitative information about the behavior and the performance of these kernels. Moreover, such an integration allows a source–level analysis of their performance where measures about the various phases of the inspector/executor strategy applied by the compiler are also derived.

In the following sections, we discuss these performance issues by analyzing two kernels which represent the main functional units of the PAM–CRASH solver [9]. All experimental results are obtained by running the kernels on the Meiko CS–2HA, located at the VCPC in Vienna.

3 Performance issues of the inspector/executor strategy

In this section we study two `INDEPENDENT` loops which are part of a kernel developed by NEC Europe. Our objective is the characterization of the inspector/executor strategy (see Section 2), both from a quantitative and a qualitative viewpoint. The NEC kernel performs a shell element calculation; it contains the stress-strain calculations over 25600 elements, driven by changing nodal points, and the accumulation of the resulting forces as nodal quantities. The two parallel loops considered in our analysis represent the core of the kernel. They are executed only once, that is, the derived results represent a one–shot timestep.

Figure 1 shows the basic structures of the two loops (hereafter referred to as `loop1` and `loop2`) which differ in that the first one does not contain any `REDUCTION` directive. As a consequence, the parallel code generated by the compiler for `loop1` contains only four phases of the inspector/executor strategy since the final scatter does not need to be performed.

Figure 2 shows the overall execution times of the two analyzed loops, and the times spent in each of the phases originated by the compiler. The times refer to runs with 8, 16, and 32 processors. The times of the work distributor phase are negligible; the behavior of all the other phases but the executor does not reflect any particular trend. Indeed, in the case of `loop1` the execution time of the inspector phase shows a decreasing behavior with respect to the number of allocated processors. In our example, these times are 0.473, 0.269, and 0.209 seconds, respectively. In the second loop, the execution times of the inspector phase are 0.671, 0.301 and 0.345 seconds, for the three runs. We have also noticed that these times always exhibit a decreasing behavior when the number of allocated processors varies from 8 to 16, and an increasing behavior when the number of processors grows from 16 up to 32. The gather and scatter phases do not reflect any particular trend. This is a consequence of the communication activities that take place within these phases. When communications are involved to exchange information across all the allocated processors, the behavior of the underlying interconnection network is not deterministic.

On the contrary, the time spent by the executor exhibits a monotonic decreasing behavior (see Fig. 3). As can be seen, this time scales quite well as a function of the number of allocated processors. Indeed, the actual computations to be performed on each processor decrease as the number of processors increases; this is due to the reduction in the execution set. Since no communications are required within the executor, the time spent within this phase is proportional to the number of iterations executed by each processor.

The communication activities that result from the inspector/executor strategy have been analyzed from a functional viewpoint, that is, for each phase the type of the communications performed has been identified.

As already pointed out, the work distributor and the executor, where the actual com-

```
!HPF$ INDEPENDENT, NEW(....)
   DO IEL = 1,NUMEL2
      NN1 = IX(1,IEL)
      ......
      DO I = 1, 3
         XN(I,1) = X(1,NN1)
         ......
      END DO
      CALL MFORCE(....)
      DO I = 1, 6
         F(I,NN1) = F(I,NN1) + FORCE(I,1)
         ......
      END DO
   END DO
```

(a)

```
!HPF$ INDEPENDENT, NEW(....), REDUCTION(F)
   DO IEL = 1, NUMEL2
      NN1 = IX(1,IEL)
      ......
      DO I = 1, 6
         F(I,NN1) = F(I,NN1) + FORCE1(I,IEL)
         ......
      END DO
   END DO
```

(b)

Figure 1: Basic structures of `loop1` (a) and `loop2` (b) of the NEC kernel.

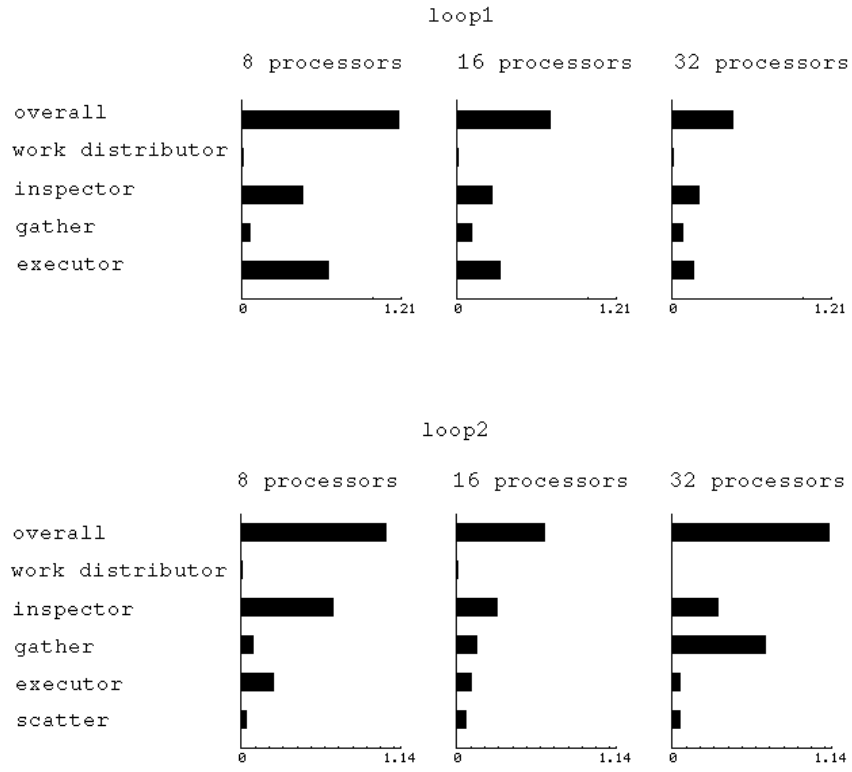

Figure 2: Overall execution times (in seconds) of the two loops and times of the phases of the inspector/executor strategy.

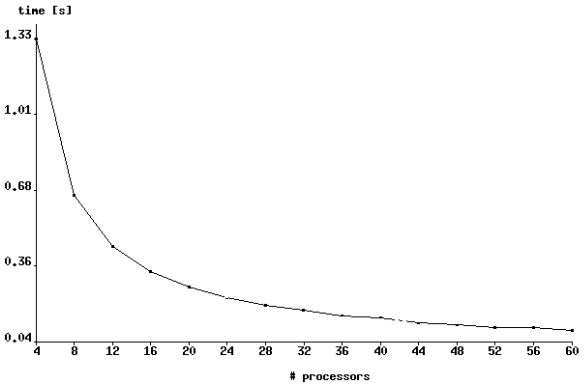

Figure 3: Times spent by the executor phase of `loop1` as a function of the number of allocated processors.

putations take place, do not contain any communication at all. The inspector phase results in six different communication types, namely, four point–to–point and two collective communications. The gather and scatter phases contain calls to one type of collective communications only. Regardless of the number of allocated processors, the communication times for both loops are dominated by collective communications, which always account for about 99% of the total communication time. Note that these communications highly stress the interconnection network since each processor has to exchange data with all the other processors. Figure 4 shows the times spent by each of the processors allocated to an 8–processors run while performing the collective communications within the inspector phase (`Allreduce`) and within the gather and the scatter phases (`Alltoall`), respectively. As can be seen, synchronization losses among the processors arise. For example, processor `p3` takes 0.095 seconds to perform the two collective communications issued within the inspector phase (`Allreduce`) and 0.039 seconds to perform the three collective communications within the gather and the scatter phases (`Alltoall`). On the other hand, processor `p1` takes 0.043 and 0.062 seconds, respectively.

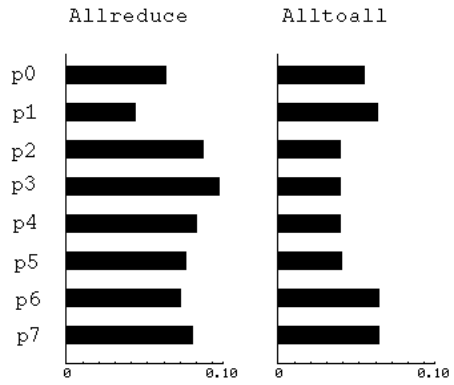

Figure 4: Times (in seconds) spent by each of the 8 allocated processors in collective communications within the inspector phase (`Allreduce`) and within the gather and the scatter phases (`Alltoall`) of `loop1`.

All these experimental results have shown that the performance of the inspector/executor strategy is highly influenced by the communication times which do not scale as the number of processors increases. Moreover, we have noticed that the overall cost of the inspector/executor strategy is quite large and typically dominates the actual computation time of the loop. As we will see, the use of HPF+ directives, such as `REUSE`, can drastically reduce the cost of this parallelization strategy.

4 Performance issues of the `REUSE` directive

In this section, the performance improvements achievable by means of HPF+ language support are presented. In particular, our analysis focuses on the `REUSE` directive, as applied in a kernel developed by ESI. This kernel contains an `INDEPENDENT` loop (see Fig. 5) which is iterated 125 times by means of a time marching scheme. At each iteration, the same patterns for accessing non–

local data are used. In such a case, the REUSE directive asserts that a schedule computation is not needed since a previously computed schedule can be reused [10]. Hence, the inspector phase is executed during the first iteration of the time marching scheme only.

```
!HPF$ INDEPENDENT, ON HOME(IY(2,I)), NEW(....)
   DO I = 1, N4NODE
        .....
        XL(1) = X(1,IY(2,I))
        XL(2) = X(2,IY(2,I))
        .....
        VL(1) = V(1,IY(2,I))
        VL(2) = V(2,IY(2,I))
        .....
        CALL FORCEI(....)
        .....
   END DO
```

Figure 5: Basic structure of the loop of the ESI kernel.

The achievable performance improvements have been tested on various runs of the kernel without and with the REUSE directive. In Figure 6, the overall execution times of the loop without and with the REUSE directive are presented as a function of the number of allocated processors. The attained reduction in execution times is always larger than 33%.

The overall execution time of the loop together with the times spent in the five phases of the inspector/executor strategy are presented in Table 1. These times, expressed in seconds, refer to runs with 32 processors. The relative performance improvement in execution time obtained by means of the REUSE directive is equal to 43.98%. As can be seen, the main gain is due to the reduction in the time of the inspector phase. This time accounts for 44% of the overall execution time of the loop executed without the REUSE directive. This percentage reduces to 1% with the REUSE directive.

The percentages of the times spent in the five phases of the inspector/executor strategy with respect to the overall execution time have also been analyzed. Figure 7 shows the

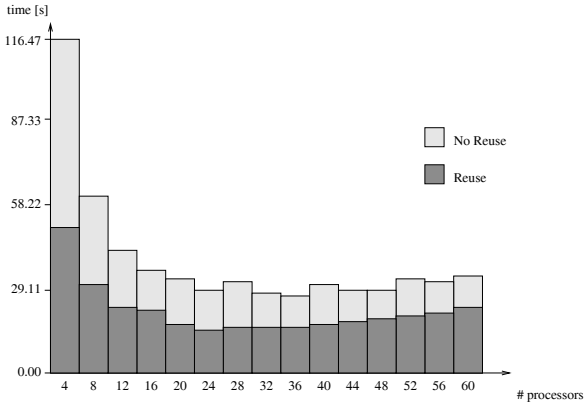

Figure 6: Execution times of the ESI loop without and with the REUSE directive.

contributions of each phase for runs with 16 processors, without and with the REUSE directive, respectively. Execution times are equal to 36.09 and to 22.03 in the two cases. As can be seen, there is a sharp decrease in the time spent in the inspector phase, which is executed just once. With the REUSE directive, the executor phase accounts for 52.93% of the overall execution time; this percentage decreases to 32.22% when these directives are not used.

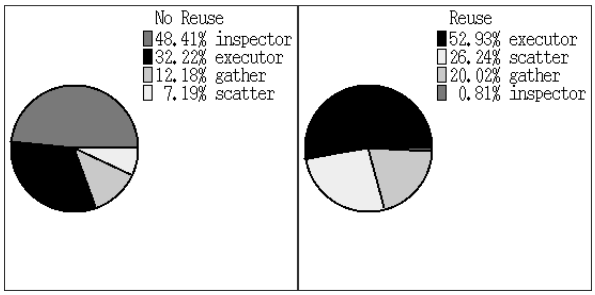

Figure 7: Breakdown of the overall execution time of the ESI loop, without and with the REUSE directive, in runs with 16 processors.

In Figure 8, the behavior of the times spent by the inspector, by the gather, by the executor and by the scatter, varying the number of allocated processors from 4 to 60 and using

Table 1: Overall execution times (in seconds) of the ESI loop and times of the five phases of the inspector/executor strategy in runs with 32 processors.

		inspector/executor phases				
	overall	work distributor	inspector	gather	executor	scatter
No Reuse	28.991	0.012	12.720	8.088	5.486	2.685
Reuse	16.240	0.001	0.163	7.987	5.258	2.831

the `REUSE` directive, is presented.
The actual computation time scales quite well when increasing the number of processors. As can be seen, the time spent to gather non–local data increases with the number of allocated processors. Similar behavior has been recognized in runs without the `REUSE` directive.

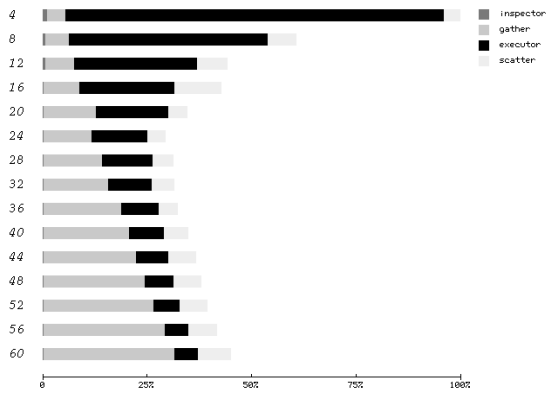

Figure 8: Breakdown of the overall execution time of the ESI loop, with the `REUSE` directive, varying the number of allocated processors.

5 Conclusions

Irregular `INDEPENDENT` loops represent the core of most parallel applications. Their overall performance are determined by the efficiency achieved by the inspector/executor strategy adopted by compilers when parallelizing such loops. This strategy may result in unoptimized code when appropriate directives are not provided as part of the language. The experimental studies presented in this paper have focused on the characterization of the "costs" associated to the inspector/executor strategy with the objectives of pointing out the influence of the various HPF+ directives and the overhead introduced by the compiler. The communication costs highly influence the performance of `INDEPENDENT` loops, since the communication times do not scale as the number of allocated processors increases.
A large reduction of the overhead associated to the inspector/executor strategy can be achieved by using the HPF+ `REUSE` directive. Future work will be dedicated to the study of the performance of new language directives as implemented by the compiler and used within kernels. This analysis will be made possible because of the integration between a parallel compiler (VFCS) and a performance tool (Medea).

References

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

[2] 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/.

[3] S. Benkner and H. Zima. HPF+ Initial Language Definition. Technical Report D2.1a, Institute for Software Technology and Parallel Systems, University of Vienna, June 1996.

[4] S. Benkner and H. Zima. Extension of VFCS for the compilation of project benchmarks Vers.1. Technical Report D2.2a, Institute for Software Technology and Parallel Systems, University of Vienna, January 1997.

[5] 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.

[6] C. Koelbel and P. Mehrotra. Compiling global name–space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[7] 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.

[8] M. Calzarossa, L. Massari, A. Merlo, and D. Tessera. Parallel Performance Evaluation: the MEDEA Tool. In H. Lidell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 522–529. Springer–Verlag, 1996.

[9] 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.

[10] S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima. High-Level Management of Communication Schedules in HPF–like Languages. Technical Report TR 97–5, Institute for Software Technology and Parallel Systems, University of Vienna, April 1997.