An HPF Case Study of a Domain-Decomposition Based Irregular Application

C. Germain[1] ,J. Laminie[2], M. Pallud[2], D.Etiemble[1]

[1] Laboratoire de Recherche en Informatique
[2] Laboratoire d'Analyse Numérique
CNRS - Université Paris-Sud
Corresponding author: C. Germain LRI -bat 490 - Université Paris-Sud
91405 Orsay cedex - France
email cecile@lri.fr
fax 33 (0)1 69 15 65 86

Abstract. Data-parallel languages, in particular HPF, provide a high-level view of operators overs parallel data structures and hide the details of data partitioning and communication. One of the most difficult issues in compiling such languages is managing irregular data-dependent parallelism. This paper presents the study of a realistic, but non adaptive irregular application. We show that HPF can easily express the natural parallelism of the application. Experimental results and a detailed examination of the compiler process are presented.

1 Introduction

Data parallel languages, and especially High Performance Fortran (HPF) [1, 2] have been designed to make possible efficient and high-level parallel programming for distributed memory machines, in contrast to the low-level concurrent programming model based on explicit message-passing primitives. Even if many academic or proprietary data-parallel languages preceded it, HPF is very recent: the first language specification was completed in 93, and the first industrial compilers in 95. Although the experimentation time is very short, the trend in the end-user community is to question the usability of HPF for irregular applications.
Irregular applications are those where the references to distributed arrays are data-dependent, e.g via indirection arrays. This paper presents a detailed study of a moderately irregular application, in the field of Domain Decomposition applied to Finite Elements methods. Instead of a global sparse structure, one has a (parallel) collection of independent (sequential) sparse problems coupled by logically shared variables called *interfaces*. DDM have been widely experimented in message-passing programming, with excellent results.
Our work addresses two questions:

- is HPF appropriate to express static DDM? Static means thats the unstructured mesh and the decomposition are computed at run time, but do not change afterwards.

– from a theoretically optimized HPF code, do current commercial compilers generate efficient code?

The experimentation platform is a SP-2, and the compiler XL-HPF from IBM.
The rest of this paper is organized as follows: section 2 presents the application; section 3 is a very brief overview of HPF; section 4 describes the implementation and gives performance results; section 5 discusses the appropriateness to our application of recent work in compiler research.

2 The application

Finite Elements approximation of partial differential equations result in sparse linear systems. The sparse matrix describes the interactions of the neighbor degrees of freedom (DF), the neighborhood structure being determined by a spatial grid. We assume in the following that the grid is unstructured, that is, the neighborhood structure has no closed form expression and must be described extensively. DDM partition the initial grid into subsets called *domains*, whose frontiers are called *interfaces*.
Renumbering of the sparse matrix, so that grid points belonging to the same domains become contiguous, results in a block-structured matrix, where each block describes the interactions of DF either inside a domain, or between a domain and its interfaces, or between the interfaces. From the decomposition in subdomains, the initial problem can be casted into an interface problem (direct and dual Schur's methods), or kept unmodified, in order to apply iterative methods to the new matrix. Our application follows the second scheme, with a biconjugate gradient algorithm (DD-GBC in the following).
The main advantage of DD-GBC is that the global sparse system has been turned into a set of independent sequential sparse matrix-vector products inside each domain, followed by the gathering of partial components of the values at the interfaces.
The natural parallelism of DD-GBC is between domains. As the typical number of domains is less than 100, such parallelism is well adapted to moderately parallel machines, which are the most widely available. As the computation cost is proportional to the size of the domain and does not depend on the numerical properties of the matrix (which is false for more complex domain decomposition based algorithms such as Schur's complement [3]), load balancing depends only on the quality of the partition. Numerous versions of the basic partitioning algorithms [5, 13] have been proposed, and are out of the scope of this paper.

3 Overview of HPF

High Performance Fortran defines a limited set of Fortran 90 extensions for data-parallel programming on distributed-memory machines. The most important of these extensions are the *forall* construct and the mapping directives.

The *forall* construct extends the F90 array constructs to array assignments involving any reference pattern, with the doall semantics: the right-hand side is evaluated before the assignment. For instance, the instruction
`forall (i =1:N, j =1:N) A(i,j) = A(j, i)`
actually transposes the matrix `A` without user-defined temporary.
The mapping directives are either *align* or *distribute*. Alignment is used to cancel communications: if two arrays `A` and `B` are aligned, an assignment using the same reference function will not create communication. Moreover, duplicate or collapsed alignments can be defined. Collapsed alignments allow to specify which dimensions of an array should be treated sequentially, thus involving no communication. Distribution is used to achieve load balancing, and to minimize the volume of communication. The distribution function maps each point of an array to a processor. Thus for instance, if communication occurs, block distribution will create less communication that cyclic distribution, but cyclic distribution may ensure better load balancing if, as for instance in Gauss solver, only an array subset is involved in the computation.
From the data-parallel code and the distribution, the compiler has to partition the arrays and to generate SPMD code which will be executed on each processor; in this code, off-processor references to partitioned arrays are translated into calls to message-passing primitives. A survey about the compilation techniques can be found in [4]

4 HPF implementation

4.1 Mapping onto domains

The implementation should make visible the domain decomposition structure to the compiler, in the sense that data belonging to the same domain should be HPF-aligned. Thus, we have to define a basic template `Domain`. To ensure load balancing, the template `Domain` is distributed cyclically over a linear arrangement of the processor set. All data structures are then defined with two logical dimensions: one along the domain, which will be treated sequentially, and the second across the domains, which is aligned with the template. A generic mapping is thus :

```
!hpf$ distribute Domain (cyclic) onto procs
!hpf$ align (*, d) with Domain (d) :: mystruct
```

This mapping assumes that `mystruct` has the same size inside each domain. The basic data structures of DD-GBC are the sparse matrix, including the value array and the associated indirection arrays, and the vector of DF. If the partition is good, each of these have been roughly equally partitioned across the domains; thus the assumption of equal size will only result in small oversizing.
With this mapping, only the interface update should require communication. For instance, assuming a local Morse bidimensional format, the local sparse matrix-vector products can be written:

```
                  !hpf$ independent
do d =1, NbDomains
  do i = 1, NbLines(d)
    do j = indlin(i-1, d) + 1, indlin(i, d)
      rVect(i, d) = rVect(i, d) + &
      & mat(j, d) * xVect(icol(j, d), d)
    end do
  end do
end do
```

for non-symmetric storage and

```
                  !hpf$ independent
do d =1, NbDomains
  do i = 2, NbLines(d)
    do j = indiag(i-1, d) + 1, indiag(i, d)-1
      rVect(i, d) = rVect(i, d) + &
      & mat(j, d) * xVect(icol(j, d), d)
      rVect(icol(j, d), d) =rVect(icol(j, d), d) + &
      & mat(j, d) * xVect(i, d)
    end do
  end do
end do
```

for symmetric storage.
The directive `!hpf$ independent` specifies that the first loop has no data dependences. We have inspected the concurrent code generated by three compilers: the academic GMD-Adaptor, the industrial from PGI and IBM's XL-HPF. It turns out that the first loop is correctly compiled, but that the second generates communication of the whole the array `icol` to all processors. No theoretical reason can be found for this feature. However, as the size of `icol` is the number of non null coefficients of the sparse matrix, this precludes the use of a symmetric format.

4.2 Updating the interfaces

Updating the interface needs to communicate the partial results obtained by local operators for the grid points situated at the interfaces of the domains. The interface points have two numbering systems, one relative to the interface, and the other to the domain. The scattering pattern is described by an array `Intfa`, such that `Intfa(i, d, k)` is the local number inside domain `d` of the point which has number `i` relative to the interface. Hence, the interface descriptor should be aligned with its owning domain. The domain decomposition can be designed so that each point belongs to only one interface. Thus, using the `forall` construct, updating the interface seems very easy:

```
!hpf$ align (*, d, *) with domain(d) :: Intfa
```

```
forall (d =1:n, k=1:n)
  forall (i = 1:IntfaSize(d, k))
    xVect(Intfa(i, d, k), d) = xVect(Intfa(i, d, k), d) + &
    & xVect(Intfa(i, k, d), k)
  end forall
end forall
```

The semantics of the `forall` construct avoids a temporary, because right-hand side arrays are conceptually all read before the assignement. Figures 1 and 2 plot the timings of this parallel loops on a 4-processor configuration (labeled *dep*), compared with the timing of its sequential counterpart using a temporary (labeled *seq*). In order to asses the impact of the distribution of the interface descriptor, the domains are confined to their interfaces. The performance loss is four order of magnitudes between the sequential and the parallel code. Performance loss is natural when comparing communications against on-processor computations, but these figures are intolerable.

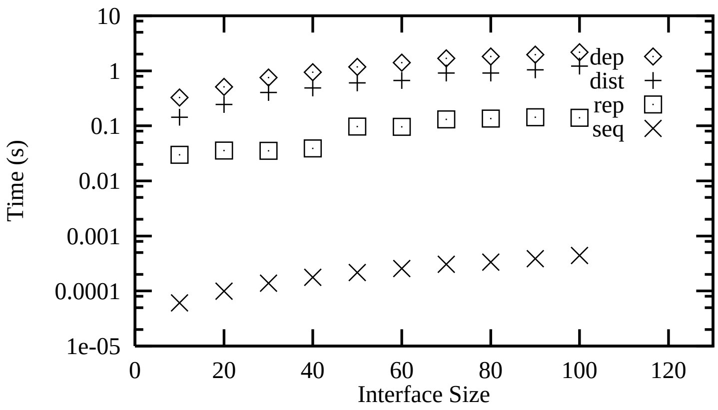

Fig. 1. Performance of Updating the Interfaces - 9 domains

The IBM XL-HPF compiler (and the other HPF compilers as well) provides a compilation option displaying a pseudo-SPMD code. Inspection of the concurrent code reveals the causes of this very poor performance: the interface descriptor `Intfa` and the vector of DF `xVect` firts go back to processor 0 two times; then, all the additions are computed on processor 0, with the result assigned to a temporary, and finally the result is written back to array `xVect`.
Faced with this very strange compiler behavior, we enter in a process of reverse engineering, in order to work out a typology of construct that the compiler can efficiently manage. It turned out that the computation is sequentialized (all on the same processor) by loop-carried dependencies: if the source and destination vectors are distinct, the forall loop

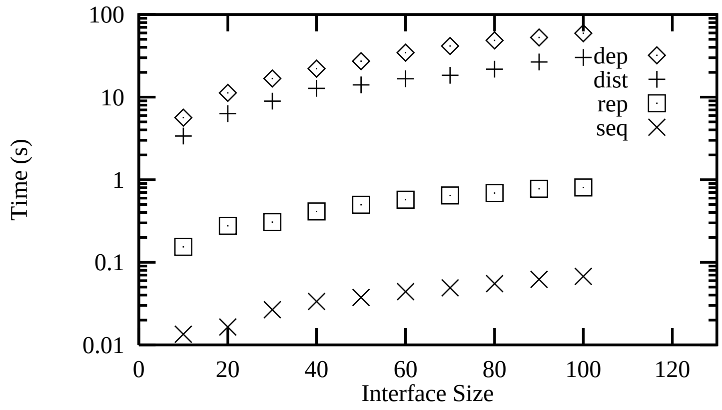

Fig. 2. Performance of Updating the Interfaces - 100 domains

is compiled in truly concurrent code. The timing of the corresponding code is labeled *dist* in fig. 1 and 2. Removing loop dependencies improves the performance. However, the slowdown with respect to sequential performance remains high. The basic reason is that there are no means to tell the compiler that the communication pattern is one-to-one, and not one-to-many. In fact, with a distributed array in the left hand side of an assignment, the compiler scans sequentially the values of this array and broadcasts them to all processors.

Finally, we tested the replicated distribution: each processor holds one instance of the interface descriptor (label *rep*). The performance is now reasonable, only one order of magnitude below the sequential one. With typical interface size, and moderate number of domains, such replication is possible.

Nevertheless, the compiler follows a very crude scheme: each processor sends to all other processors its section of the right hand side array. Thus, before actual updating, the interface values must be packed into a temporary. Obviously, the all-to-all scheme limits the speedup: as interface size is small, most of the time is spent inside the communication startup, nearly 100 μs in this experiment. Thus the communication cost will grow linearly with the machine size.

4.3 Performance results

Table 1 displays some preliminary results about the matrix-vector product, which is the most time consuming part of the DD-GBC algorithm. There are 4 subdomains, 3 interfaces per domain, and the code was executed on four processors. As can be expected, speedup can be obtained only when the ratio between the domain size and the interface size is low. For the smallest configuration, the ratio is nearly 25%, thus the communication cost dominates.

Grid Size	Matrix Size	Interface Size	Aspect ratio	Speedup
2,400	16,000	50	0.25	0.44
19,200	128,000	200	0.12	1.45
64,800	432,000	450	0.08	2.02
153,600	1,024,000	800	0.06	2.57
300,000	2,000,000	1250	0.04	

Table 1. Performance of the matrix-vector product

5 Discussion

Implementing parallel indirect addressing remains a challenging problem. Most of the research and commercial implementation works have targeted irregular problems which exhibit spatial and temporal locality. Temporal and spatial locality can be exploited through data-flow analysis [8], in order to eliminate redundant communications. The IBM-XLHPF compiler performs such an analysis: when possible, communications are hoisted out of the loops where they remain invariant. This optimization does not apply to our application: the matrix-vector product will be called many time inside the biconjugate gradient loop, but each time with a new input vector.

The inspector/executor scheme [6, 9, 11, 12] sets up at run-time data structures which describe the communication (*schedules*), which explicitly store all the global to local translation tables. Temporal locality allows to reuse already bought off-processor data. Spatial locality eliminates duplicate off-processor references by hash tables. Real-world applications displaying a high level of spatial locality have been proven to benefit extensively from the inspector/executor scheme [9, 12]. Setting up schedules is costly: [12] reports that the time to generate a schedule for a 14026-sized data array is in the range 0.4 to 1 second for 128 down to 16 processors on a iPSC/860, and that the time to build the translation tables is in the 1 to 7 second range. Thus, the cost of building the schedule must be amortized over multiple uses of the communication pattern.

Our application can really benefit from the inspector/executor scheme, because the communication scheme does not change across the biconjugate gradient loop. In the present state, all interface values are made available to all processors, most of them being useless: the volume of communication could be greatly reduced. However, the matrix-vector product should be encapsulated in a routine (this allows compact and portable programming, using the operator overloading feature of F90). Separate compilation rules out the visibility of the invariant scheme. Conciliating the inspector/executor scheme and modularity may be achieved either via interprocedural analysis, or timestamping of schedules [10].

Neither HPF1.1 nor HPF2.0 provide a statement to inform the compiler that an indirect mapping is one-to-one. This information cannot be inferred by the compiler. Even if this is not the case here, many applications require a one-to-one mapping that spans a large array, and efficient techniques based on sorting can be exploited [7] for such mappings.

6 Conclusion

Our initial questions concerned the appropriateness of HPF to DD-GBC, first in principle, and second when using state of the art compilers. To the first question, the answer is yes, even if only HPF 1.1 is considered, but there is still room for improvement. Yes, because the basic placement directives are enough to express the particular structure of DDM. Yes also because most of the initial F90 code, using the object-oriented features of F90, can be kept unmodified. Improvement, because there are no means to tell the compiler that some vector-valued subscripts describe one-to-one mappings.

To the second question, our study shows that in the current state of HPF compiler technology, performance is highly fragile: a minor difference in programming style can result in orders of magnitude variation in overall performance. Naive coding creates a massive slowdown. Moderate speedup is possible with the current tools, but needs to discover where and why inefficiencies appear, and to work out what can be efficiently compiled. However, HPF programming will become competitive with message-passing programming in the field of irregular applications only if the compiler technology is able to integrate the most advanced developments of research work.

References

1. High performance Fortran language specification. Technical report, Rice University Houston Texas, January 93. V1.0 Draft.
2. High Performance Fortran Forum. High Performance FOrtran Language Specification,V2.0 δ, Rice University Houston Texas, October 96
3. P.E. Bjorstad and O.B. Widlund. Iterative methods for the solution of elliptic problems on region partitioned into substructures. *SIAM J. Numerical Analysis*, 23:1097-1120, 1986
4. F. Coelho, C. Germain and J-L Pazat. Compiling HPF. In A. Darte G.-R. Perrin, editors, *The Data Parallel Programming Model*, pages 104–130. Springer Verlag, 1996. LNCS 1132.
5. E. Cuthill and J. Mackee. Reducing the band width of sparse symmetric matrices. In *ACM Nat. Conference*, pages 157-172, 1969
6. R. Das et al. Communication optimization for irregular scientific computations on distributed memory architectures. *Jal of Parallel and Distributed Computing*, (22):462–478, 94.
7. C. Germain and D. Gautier de Lahaut. Improving Irregular Parallel Communication through Sorting. *High Performance Computing and networking Conf, HPCN'97.* To appear in LNCS,
8. M. Gupta, E. Schonberg and H. Srinivasan. A unified data-flow framework for optimizing communication in data-parallel programs. *IEEE Trans. on Parallel and Distributed Systems*, 7(7):689–704, 96.

9. R. Ponnusamy, J. Saltz and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing 93*, pages 361–370. ACM, 93.
10. R. Ponnusamy et al. Suporting irregular distributions in Fortran90D/HPF compilers. Technical report CS-TR-3268 and UMIACS-TR-94-57, 94
11. M. Ranganathan et al. Runtime coupling of data-parallel programs. In *Supercomputing'96*. IEEE, May 96.
12. S. D. Sharma and al. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Supercomputing'94*, pages 99–106. IEEE Press, 94.
13. H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Eng.* 2(2/3):135-148, 91