A Tracing Protocol for Optimizing Data Parallel Irregular Computations

Thomas Brandes[1] and Cécile Germain[2]

[1] Institute for Algorithms and Scientific Computing (SCAI)
GMD, Schloß Birlinghoven, D-53754 St. Augustin, Germany
e-mail: brandes@gmd.de

[2] Laboratoire de Recherche en Informatique (LRI-CRNS)
Université Paris-Sud, F-91405 Orsay Cedex, France
e-mail: cecile@lri.fr

Abstract. High Performance Fortran (HPF) is the de facto standard language for writing data parallel programs. In case of applications that use indirect addressing on distributed arrays, HPF compilers have limited capabilities for optimizing such codes on distributed memory architectures, especially for optimizing communication and reusing communication schedules between subroutine boundaries.
This paper describes a dynamic approach for optimizing unstructured communication in codes with indirect addressing. The basic idea is that runtime data reflecting the communication patterns will be reused if possible. The user has only to specify which data in the program has to be traced for modifications. The experiments and results show the effectiveness of the chosen approach.

1 Introduction

Data parallel languages as High Performance Fortran [9, 10] have been designed to make possible efficient and high-level parallel programming for distributed memory machines. As calculations in computationally intensive codes mostly occur in loops or intrinsic calls involving loops, a challenging problem is to handle efficiently loops with indirectly referenced distributed arrays. For irregular parallel loops, complex data structures, called *schedules*, are required for accessing remote items of distributed arrays and for communication optimization. The schedules cannot be worked out at compile time, but only at run-time, when the values of the indirection arrays are known. The code design which first builds the schedule, then uses it to carry out the actual communication and computation, has been coined as the *inspector/executor* scheme that was pioneered very early [16, 12], and developed in the PARTI [6, 2] and CHAOS [17] libraries. It is the state-of-the-art design for academic compilers [13, 11, 18, 4] as well as commercial ones [15].

The preprocessing associated with the inspector creates a large run-time overhead, because it involves a large amount of computation and all-to-all communications. Hence the inspector/executor scheme targets irregular loops which exhibit especially temporal locality where the schedule can be reused. The schedule reusing problem can be addressed at compile-time, through data-flow analysis [6, 8]. The main drawbacks of this approach are the limits of inter-procedural analysis and the wall of separate compilation. More precisely, real codes with indirect addressing often require inter-procedural analysis [1, 7, 5].

At the other extreme, language support makes the concept of schedule visible to the end-user. The HPF+ project [3] proposes *gather* and *scatter* clauses referring to explicit *schedule variables*, or *reuse* clause for independent loop. The user is responsible for tracking the modifications, but through high level conditional constructs. Reusing schedules across procedure calls relies on the Fortran 90 `save` attribute for the schedule variable.

The seminal paper [14] proposes a schedule reusing method based on time-stamps. Each modification of an array is mirrored by an update of the time-stamp, and the time-stamps are recorded at inspector invocation. A schedule may be reused if all relevant arrays have the same time-stamps at the point of reuse as the recorded ones. As no directive is provided, all array modifications have to be tracked. Moreover, array descriptors are shared through all arrays having the same structure parameters, so that an indirection array may be considered modified without necessity.

In this paper, we propose a new protocol to handle temporal locality. It has been implemented and evaluated in the framework of the ADAPTOR HPF compilation system [4] that already supported the inspector/executor scheme. The protocol is called URB, because the run-time system can either Use, Refresh or Build a schedule for each parallel loop. With some help from the user, URB is able to track changes in the irregular loops exactly: a new schedule is built only when either an irregular parallel loop is structurally different from the previously executed ones, or when the indirection arrays have been modified. The structural information includes the distribution, shape and size of the arrays, and is completely handled by the compiler and the run-time system. The intervention from the user is limited to provide an attribute for each indirection array, through a directive. This attribute indicates that all modifications of this array have to be recorded at run-time, and allows to handle procedure calls without inter-procedural analysis. Finally, the tracking scheme is conservative in the sense that, when tracing directives are missing, the generated code simply rebuilds a new schedule for each execution of an irregular loop. Hence, codes using the trace directive and other codes (e.g. library routines) can securely be mixed.

The rest of the paper is organized as follows. Section 2 describes the realization of indirect addressing of distributed arrays via the inspector/executor scheme. Section 3 introduces the protocol that is needed for tracking indirection arrays and how it can be realized. Section 4 presents performance results for kernels and a real application, and we conclude in section 5.

2 Unstructured Communication

In the following, we introduce a formal definition of the *communication schedule* that is built up during the inspector phase. We discuss it for a typical gather operation. The array T specifies the home of the iteration where the data is needed.

$$\text{forall } (I \in I_T, M(I))\ T(I) = A(L(I))$$

Such an indirect addressing will be noted as $\mathcal{I}_{T,A,L,M} = \{I_T, d_T, I_A, d_A, L, M\}$, the inspector info, where I_T and I_A specify the index space of the arrays T and A, d_T and d_A their distributions on the available processors P. M is a predicate defining the subset S_T that is the iteration space of the parallel loop. L stands for the indirect addressing of the array A. All indices and arrays can be multi-dimensional ($I = I_1, \ldots, I_n$), and L stands for a set of integer arrays $L_1, \ldots, L_m$.

$$\begin{array}{rc}
d_T : I_T \rightarrow P & d_A : I_A \rightarrow P \\
M : I_T \rightarrow \{true, false\}, & S_T := \{I \mid M(I) = true\} \subseteq I_T \\
L : S_T(\subseteq I_T) \rightarrow I_A, & I \mapsto L(I)
\end{array}$$

We assume that the function L is specified by an integer array or by a set of integer arrays L_j and that they are aligned with the array T. In other words, $L(I)$ is available on the processor that owns $T(I)$. Also the predicate M, if available as a logical array, will not involve any communication.

For every processor pair (p, q), $p \neq q$, the set $C_{p \leftarrow q}$ specifies which elements of I_A processor p needs from processor q:

$$C_{p \leftarrow q} := \{(I, L(I)) | I \in S_T, d_T(I) = p, d_A(L(I)) = q\}$$

$C_{p \leftarrow q} \subseteq d_T^{-1}(p) \times d_A^{-1}(q)$, so for every $(I, K) \in C_{p \leftarrow q}$, I is a local index of T on processor p and K is a local index of A on processor q. Data movement from processor q to processor p will be necessary if $C_{p \leftarrow q}$ is not empty. Processor q has to send the corresponding data and processor p must receive it. In the following, the collection of sets $C_{p \leftarrow q}$, $\mathcal{S}_{T,A,L,M} = \{C_{p \leftarrow q}\}$, is called the *communication schedule*. Due to the fact that L is not a function, the computation cannot be done by closed formulas as it might be possible in case of structured communication.

for $q \in P$: $C_{p \leftarrow q} := \{\}$ end for
for I in $d_T^{-1}(p) \cap S_T$
    $q := d_A(L(I))$
    $C_{p \leftarrow q} := C_{p \leftarrow q} + \{(I, L(I))\}$
    end for
end for
	for $q \in P$, $p \neq q$
	    send $C_{p \leftarrow q}$ to processor q
	    recv $C_{q \leftarrow p}$ from processor q
	end for

Fig. 1. Schedule building algorithms

for $q \in P$, $p \neq q$, $C_{q \leftarrow p} \neq \{\}$
  compute $V_{q \leftarrow p}$ (local on p)
  send $V_{q \leftarrow p}$ to processor q
end for
for $q \in P$, $p \neq q$, $C_{p \leftarrow q} \neq \{\}$
  recv $V_{p \leftarrow q}$ from processor q
  for $(I, v) \in V_{p \leftarrow q}$ set $T(I) = v$
end for
	for $q \in P$, $p \neq q$, $C_{p \leftarrow q} \neq \{\}$
	  compute $W_{p \leftarrow q}$ (local on p)
	  send $W_{p \leftarrow q}$ to processor q
	end for
	for $q \in P$, $p \neq q$, $C_{q \leftarrow p} \neq \{\}$
	  recv $W_{q \leftarrow p}$ from processor q
	  for $(v, K) \in W_{q \leftarrow p}$ set $A(K) = v$
	end for

Fig. 2. Executor algorithms

Each processor p has to compute the sets $C_{p \leftarrow q}$ for all processors q, as shown in Fig. 1, left part. By an all-to-all communication, the processors exchange the information which data they need from each other (*index exchange*, Fig. 1, right part). This is necessary in order to make sure that the other processors know which values they have to send. The inspector info together with its schedule builds the inspector data.

In the *executor* phase, the processors use the communication schedule of the inspector data to carry out the communication for gathering or scattering data. Therefore the values of T and A belonging to a domain X are transferred.

$$val_A : I_A \rightarrow X \qquad V_{p \leftarrow q} := \{(I, val_A(K)) | (I, K) \in C_{p \leftarrow q}\}$$

Figure 2 (left part) shows the algorithm which is executed on each processor p to gather the values of the distributed array A. For a scatter operation, $forall(I \in I_T, M(I)) A(L(I)) = T(I)$, the communication schedule can be used in the same way to realize the necessary communication (Fig.2, right part).

$$val_T : I_T \rightarrow X \qquad W_{p \leftarrow q} := \{(val_T(I), K) | (I, K) \in C_{p \leftarrow q}\}$$

3 The URB Protocol

3.1 Dynamic Reuse of Communication Schedules

Let be $\mathcal{I}_{T,A,L,M} = \{I_T, d_T, I_A, d_A, L, M\}$ an indirect addressing for which a communication schedule $\mathcal{S}_{T,A,L,M}$ has been computed. This communication schedule can be reused if the sets $C_{p \leftarrow q}$ have not changed. This is the case if I_T, I_A, d_T, d_A, L and M have not changed. This model points out that the inspector does not depend on the actual values of the arrays T and A, but only on their shape and distribution. Thus, an inspector has the potential to be reused for instance for different sections of the same array, or for aligned arrays.

Our URB protocol is based on run-time analysis. For this dynamic approach, it will be decided at runtime whether a communication schedule can be reused or not. Therefore, inspectors are kept in a database. Two problems must be addressed: at first, tracing the modifications of the indirection and mask arrays, and secondly, avoiding the explosion of the data base while keeping useful information.

The penalty of this approach is that modifications of the indirection arrays must be traced. In the presence of a mask, this is also necessary for the corresponding arrays within the mask.

3.2 Principle of Dynamic Tracing

The previous formulation shows the inspector database management as an instance of a cache management problem. The URB protocol is based on this analogy.

To allow tracing, the descriptor of an indirection array L includes a *dirty flag*. Each write to L sets its dirty flag. The schedule can thus be reused if and only if the dirty flag of L is clear. The approach has to guarantee that every update or every possible update of L will set its dirty flag.

Furthermore, if the dirty flag of an array L is set, all inspector for which this array is involved, become invalid. The entries for these inspectors can be freed avoiding the explosion of the database.

In fact, the previous scheme fails if an indirection array L_j is involved in more than one indirect assignment. The reason is simply that the flag of the L_j may be shared by many inspectors, but that their status with respect to these inspectors should not be shared. Hence, the descriptor of a traced array actually includes not only one dirty flag, but a *dirty mask*, with a flag for each possible inspector. Each write to the array sets *all* flags of the dirty mask, while each inspector update clears only the corresponding flag in the dirty mask.

The overall run-time algorithm for building or retrieving schedules is sketched in Fig. 3. Even when the schedule is reusable, the actual schedule can need refreshing. This means that all the hard work of computing ownership and local addresses is done, but that a translation term from the base address has to be taken into account. The inspector data then is updated with the refreshed schedule.

```
Schedule GetSchedule (InspectorInfo 𝓘)
begin
var InspectorData : 𝓟        /* InspectorData = InspectorInfo + Schedule */
        𝓟 := DBSearch (𝓘)
        L := IndirectionArrayDescriptor(𝓘)
        if (𝓟 == NULL) or DirtyMask(L, 𝓟) then
          DBFree (𝓟)
          𝓟 := InspectorBuild (𝓘)
          DirtyMaskClear(L, 𝓟)
        elseif not EqualBaseAddress (𝓟, 𝓘) then
          ScheduleRefresh (𝓟, 𝓘)
        endif
        return (Schedule(𝓟))
end
```

Fig. 3. The URB protocol

3.3 Tracing of Arrays

Tracing clearly belongs to the compiler, which has to insert the code setting the dirty flags as a side-effect of a write of some arrays. The requirement for tracing is described by the user directive `!ADP$ TRACE`. This directive has a Fortran attribute semantics, and follows Fortran 90 rules of scoping. By default, arrays are not traced. To insure correctness, this case must lead to the conservative scheme where the schedule is computed at each access. If at least one indirection array is not traced, no entry in the data base is created. The schedule is built, used and destroyed immediately after its use.

An array must be flagged dirty when it is changed. The syntactic constructs that can modify an array are limited, and can be analyzed at compile-time: assignments with this array as a left-hand-side, redistributions, allocations and deallocations. Thus, the trace attribute is compatible with the `DYNAMIC` and `ALLOCATABLE` attributes, but not with the `POINTER` or `TARGET` attribute, because the compiler will not be able to record modification of such arrays.

3.4 Procedure Handling

We want the URB protocol to be compatible with the general HPF framework that does not include the tracing directive at the present time. Thus, all combination of traced and not traced for actual and dummy arguments must be considered.

The run-time system copies information from the descriptor of the actual argument into the descriptor of the dummy argument at the entry of the subprogram, and vice versa when leaving the subprogram. Four cases must be considered, following the fact that the actual and the dummy arguments have or have not the trace attribute. If none has, we are done. If the actual and the dummy arguments both have, the modifications of the array will be traced through the subroutine boundaries, by copying forth and back the dirty mask. If the actual does not have the attribute and the dummy has, the modifications of the dirty mask of the dummy inside the procedure have no impact on the actual: reusing of schedules where this array is involved is only possible during the lifetime of this subroutine call.

Finally, procedures without explicit or implicit interface, or with an explicit or implicit interface that does not give the trace attribute to an output dummy argument, are assumed to modify the actual argument. Thus, on the calling side, the compiler will set the dirty flag of the corresponding array when the procedure returns. With this scheme, the case of library subroutines that modify an indirection array, and that are not aware of the trace attribute, is correctly handled.

4 Performance Results

All experiments were performed on an IBM SP2 equipped with P2SC nodes, and a 80MB/s maximal bandwidth per node. We used the user space protocol, and the hardware timer of the Power2 architectures. Measurements are averages.

4.1 Basic Performance

We studied the basic performance of the URB protocol for a typical gather operation, $A(I) = B(L(I))$, in order to make evident the various components of the inspector-executor scheme. Three schemes of indirect addressing L have been considered: identity (i), a cyclic shift (c), and uniform random values (r). The random and shift distributions are standing here for two typical degrees of irregularity, heavy and light. Identity is useful to measure pure overhead. It also describes *data-dependent locality*, where no communication is needed because the indirection array operates inside disjoint index subsets. Domain decomposition methods are a typical example of this situation.

First, a breakdown of the various components of the inspector and of the executor, for the random scheme, on an 8 processors configuration and a 4K array of real (Table 1) shows that the only potential overhead of the URB protocol, namely searching the database, is negligible. It also highlights the cost of the inspector step.

	inspector step	executor step
no reuse	0.16 sec	0.07 sec
reuse	0.25e-4 sec	0.07 sec

Table 1. Timing of unstructured communication (seconds, 8 processors).

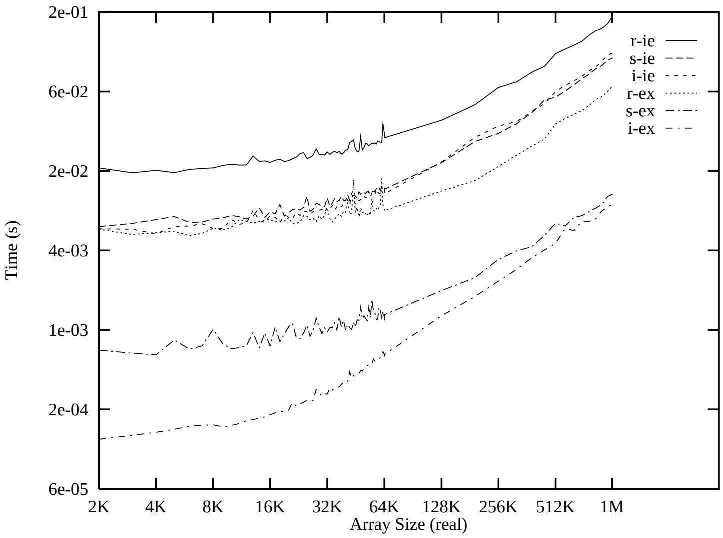

Fig. 4. Performance effects of schedule reusing: 16 processors. Curves labeled *ie* are the non-reuse case, and curves labeled *ex* are the reuse case.

Figure 4 presents a detailed comparison of timings with and without schedule reuse, on a 16 processors configuration. The random scheme displays a nearly linear difference (constant speedup) between the reuse and non reuse case, because of the index exchange, which in this case has exactly the same volume as the data movement. The identity scheme, in the reuse case, describes the minimal cost of an indirect operation, which includes reading the schedule and the local copy between the user arrays. The shift scheme, in the reuse case, has the only additional overhead of each processor sending one 4-bytes message. This overhead is very significant for small arrays, but becomes negligible for large arrays, where the performance converges with the one of the identity case. The shift and identity schemes have very similar performance in the non-reuse case, and display the overhead of building the schedule (ownership and offset computations).

4.2 Performance Results for a Real Application

The AEROLOG computational fluid dynamic software developed by MATRA is devoted to the study of compressible fluid flows around complex geometries. The kernel of the numerical solver (called CLASS for Conservative LAws Systematic Solver) is common to all physical models and is based on a second order accurate time and space finite volume scheme. Within the European Esprit project PHAROS, this code has been ported to HPF [5].

The HPF parallelization strategy is based on the coarse grain parallelism given by the decomposition of the mesh data into sub-domains. In one time step, the time increments for the physical variables are computed. The routines called within one time step can be sorted into two groups, the *local* routines and the *boundary* routines. The local routines are called independently over sub-domains. This group represents typically up to 90 % of the total CPU time of the serial code. The local computations scale well as no communication is necessary. The boundary routines perform the boundary conditions. These routines make an intensive use of indirect addressing and involve dependences between data belonging to different sub-domains.

	P=1	P=2	P=4	P=8	P=16
local	190.46	95.84	46.68	23.43	12.86
boundary					
replicated	11.24	35.99	38.53	42.97	52.19
compress	12.03	11.49	11.56	13.24	14.05
notrace	165.65	96.88	52.43	28.66	21.12
trace	34.42	18.72	12.16	7.73	5.88

Table 2. Execution times in seconds (304200 mesh points, 20 iterations).

In the initial HPF version, the whole mesh data involved in the boundary computations as well as the boundary computations itself are replicated on all processors. This avoids unstructured communication but the replication is very expensive. In the tuned HPF version, the mesh data is packed before the replication. This reduces the communication volume, but the scalability is still limited.

The third version distributes the mesh data thus requires unstructured communication. Performance results are very poor if the communication schedule is not reused (notrace). Only the reuse of communication schedules (trace) gives acceptable and scalable performance results. With ADAPTOR, the user gets this performance gain only by inserting some `TRACE` directives in his HPF code. Table 2 shows the performance results for a medium industrial test case with 16 sub-domains of 304200 total mesh points executing for 20 iterations.

5 Conclusions

In this paper, we have presented a strategy to implement effectively an inspector-executor paradigm where communication schedules can be reused. It requires no more data-flow and inter-procedural analysis at compile time. It will also reuse communication schedules in situations where even best compile time analysis might fail, e.g. if the indirection arrays are modified under a mask. Although the method described here requires a new language construct, the `TRACE` directive, the user does not have to track the multiple execution paths. This should be task of the compiler and of the run-time system.

The realization of our concept in the ADAPTOR HPF compilation system showed that it can be easily integrated in an existing HPF compiler that supports the inspector/executor scheme. The compiler itself has only to deal with the new `TRACE` attribute. The runtime system requires the support of reusing communication schedules and the implementation of a data base for these schedules. For the tracing of arrays with the `TRACE` attribute, the array descriptor has to be extended by a dirty mask. Tracing can be tracked along subroutine boundaries if an array or a section of an array is passed via such a descriptor. This is generally the case for ADAPTOR, but tracking fails if the HPF compiler passes an array only by a pointer.

The work described here demonstrates two new ideas. The first is that a schedule can be associated with more than one parallel indirect assignment, allowing aggressive optimization even in dynamic applications. The second is the analogy to schedule reusing with cache management.

We believe that the ideas of the URB protocol might also be useful for other optimizations in an HPF compiler, especially in the context of indirect and dynamic distributions, and for the efficient support of `DOACROSS` loops with irregular dependencies. In particular, the fact that the URB protocol is very efficient on the identity case has an important application: the HPF 2.0 Approved Extensions deal with data-dependent locality through the coupling of the `ON HOME` and `RESIDENT` directives. Schedule reusing is simpler, for the user and the compiler, and more versatile, because the same code can be kept unmodified whether data-dependent locality exists or not. The improvement of the current strategy as well as the investigation of benefits for other optimizations will be part of our future work.

Acknowledgments

We acknowledge the CRI and the CNUSC for access to their IBM SP2.

References

1. G. Agrawal and J. Saltz. Interprocedural communication optimizations for distributed memory compilation. In *Language and Compilers for Parallel Computing*, pages 1–16, Aug. 1994.
2. G. Agrawal, A. Sussman, and J. Saltz. Compiler and run-time support for structured and block-structured applications. In *Supercomputing*, pages 578–587. IEEE, 1993.
3. S. Benkner and H. Zima. Definition of HPF+ Rel.2. Technical report, HPF+ Consortium, 1997.
4. T. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In K. Decker and R. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, Apr. 1994.
5. T. Brandes, F. Zimmermann, C. Borel, and M. Brédif. Evaluation of High Performance Fortran for an Industrial Computational Fluid Dynamics Code. In *Proceedings of VECPAR 98, Porto, Portugal*, June 1998. Accepted for publication.
6. R. Das et al. Communication optimization for irregular scientific computations on distributed memory architecturess. *Journal of Parallel and Distributed Computing*, (22):462–478, 1994.
7. C. Germain, J.Laminie, M. Pallud, and D. Etiemble. An HPF Case Study of a Domain-Decomposition Based Irregular Application. In *PACT'97*. LNCS, Sep 1997.
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, 1996.
9. High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice Univ., Nov. 1994. *Version 1.1*.
10. High Performance Fortran Forum. *High Performance Fortran Language Specification*, Oct. 1997. *Version 2.0*.
11. S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed-Memory machines. *CACM*, 35(8):66–80, Aug. 1992.
12. S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12:415–422, 1991.
13. C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd. Symp. on Principles and Practice of Parallel Programming*, pages 177–186. ACM, 1990.
14. R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *ACM Int. Conf. on Supercomputing*, pages 361–370, 1993.
15. Portland Group, Inc. PGHPF User's Guide. Manual Release 2.2, PGI, 1997.
16. R. Mirchandaney et al. Principles of run-time support for parallel processing. In *ACM Int. Conf. on Supercomputing*, pages 140–152, 1988.
17. S. D. Sharma and al. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Supercomputing'94*, pages 99–106. IEEE, 1994.
18. H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, Feb. 1993.