Compiling Data Parallel Programs to Message Passing Programs for Massively Parallel MIMD Systems

Thomas Brandes
German National Center for Mathematics and Computer Science (GMD)
P.O. Box 1319, 53731 St. Augustin, Germany
e-mail: brandes@gmd.de

Abstract

The currently dominating message-passing programming paradigm for MIMD systems is difficult to use and error prune. This is due to the lack of the shared memory and due to race condition errors that can hardly be debugged, especially for massively parallel systems.

One approach to avoid explicit communication is using the data parallel programming model. This model stands for single threaded, global name space, and loosely synchronous parallel computation. It is easy to use and data parallel programs scale usually very well. With the definition of High Performance Fortran one can expect that this programming model will get more importance in the next future.

Based on the experiences of a real existing compilation system for data parallel Fortran programs it will be shown how to design such a compilation system and which optimization techniques are required to make data parallel programs competitive to their hand-written counterparts with message-passing.

1 Introduction

MIMD (multiple instruction, multiple data) architectures with distributed memory are the kind of parallel machines that are scalable and can be used for a wide range of scientific applications. Future massively parallel systems will be based on this kind of architecture.

But today there is a great confusion about how to write parallel programs for MIMD distributed memory architectures that will not only run efficiently on one architecture but also run efficiently on other architectures and scale with the number of processors.

Indeed, it is a correct approach to write a parallel program that is based on a portable message passing library and that really makes no assumptions about the number of processors that will be available. But due to the difficulties with processor synchronization and debugging, this approach is very expensive and very error prune. The explicit control of data partitioning by the user prevents him from experimenting with different data distributions.

A number of attempts have been made to simulate a shared memory on such machines [4, 1]. Together with a user specified data distribution, the code with global data references is translated into a program with local and non-local references, where the latter are satisfied by automatically inserting message-passing statements. The efficiency of the generated code is mainly dependent on the ability of the system to recognize local computations and to optimize communication for the underlying hardware architecture.

When using data parallel languages or SIMD (single instruction, multiple data stream) programming languages like C* or Connection Machine Fortran [2, 9], it is simpler to use the inherent parallelism of the array operations and to generate efficient communication as the operations have mostly regular communication patterns (e.g. global reductions, shift and spread operations). Experiments with many sequential programs and their Fortran 90 counterparts have shown [3] that automatic methods could not parallelize the sequential version where it is possible for the data parallel SIMD version.

With the introduction of High Performance Fortran [7] that supports the data parallel programming style, many large scientific applications are expected to be programmed in this model. For MIMD architectures with distributed memory it is necessary to compile such programs to MIMD message passing programs that can run efficiently on these architectures.

In the following such a compilation system for data parallel Fortran programs is described. The design of the runtime system and of the transformation system will be presented. Useful optimization techniques are pointed out that will be necessary to make data parallel programs competitive to hand-written message passing programs.

2 Adaptor

Adaptor (Automatic Data Parallelism Translator) is a system developed at the GMD for compiling data parallel Fortran programs to equivalent Fortran 77 message passing programs. Most features of Connection Machine Fortran (CMF) and many features of High Performance Fortran (HPF) are supported.

2.1 Overview of Adaptor

The data parallel program will be transformed by an interactive source-to-source transformation (fadapt) to an equivalent SPMD program (single program, multiple data) that runs on all available nodes (`node.f`). Optionally, a host program (`host.f`) can be generated that takes care of all I/O operations on the front end. Essential idea of the translation is to distribute the arrays of the source program onto the node processors where the parallel loops and the array operations are restricted to the local part that is owned by the processor. Communication statements for exchanging non-local data will be generated. The control flow and statements with scalar code are replicated on all nodes.

Adaptor generates also a `Makefile`. Compiling and linking of the generated message passing programs will be done by the Fortran compiler that is available on the parallel machine.

The DALIB (distributed array library) realizes functions for global reductions, transposition, gather and scatter operations, circular shifting, replication and redistribution of distributed and local arrays. Timing and tracing facilities as well as a random number generator are also part of this library. The DALIB is the runtime system of Adaptor and will be linked with the generated message passing program.

The source files of Adaptor, documentation files in PostScript and a number of example programs are available via 'anonymous ftp'. The address of the ftp-server is `ftp.gmd.de` (129.26.8.90), the files are in the directory `gmd/adaptor`.

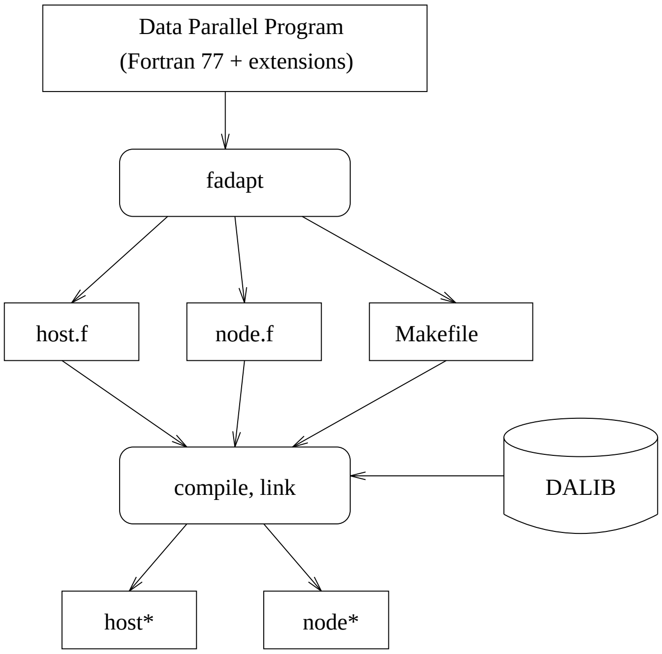

Figure 1: Overview of Adaptor

2.2 Results

For testing the Adaptor tool the HPF benchmark suite has been utilized [8] where many data parallel programs are given in different versions. This benchmark suite has been composed at the Syracuse university, New York. The CMF version was slightly modified to make a translation with Adaptor possible.

Furthermore there are experiences with some larger application programs, like a hydrodynamics code, a three-dimensional atmospheric circulation model, a two-dimensional weather forecasting system, and solvers of an interactive fluid dynamics package.

After some slight and sometimes more serious changes, all the codes have been translated successfully by Adaptor and run now on the supported message-passing architectures. The results gave especially the following insights:

- If the data parallel program scales on a Connection Machine, the generated message passing programs will also do and will run efficiently on all machines.
- In nearly all cases there were hand-coded message passing programs for the same problem available that could be compared with the automatically generated message-passing programs. The latter ones achieved mostly the same performance as the former ones. In some cases only about 60 to 90 % of the performance was seen, but also in some other cases the Adaptor version outperforms the hand-written version.
- Array operations and parallel loops are the potential of parallelism for Adaptor. It has been shown with Adaptor that this potential can also be used for optimizations of the node program (e.g. vectorization or utilizing local caches).

But the experiences have also revealed where the data parallel programming model cannot be used well and where additional optimizations are required.

3 Design of the Runtime System

The runtime system has to support all the data movements that occur when partitioning the global arrays to local parts on the processor nodes.

3.1 Basic Properties

The following basic decisions have been made for the runtime system:

- If an array is distributed, every processor will know which data is located on which processor. Therefore it is never necessary that one processor has to ask for the locations of the needed data. Dynamic distributions are not supported.
- The home of data is computed by functions that are simple and efficient. There are no tables that hold the information which data is on which processor. These tables would need too much memory on every node, especially when using massively parallel systems.
- The low level communication is based on non-blocking send and receive. The receiving processor will always know from which processor it gets the data. The required data has to be sent explicitly by the owner, no processor can access the memory of other processors.

3.2 Distributions

Let A be an array variable of the program and I_A the index set of possible subscripts, P the set of node processors. Then a *distribution*, where every element has exactly one home, can formally be described by a function:

$$d_A : I_A \rightarrow P$$

$d_A^{-1}(p)$ is the local part of the array A for processor p.

In the following the distribution function for one-dimensional block distribution is discussed. Though this kind of distribution is the only one supported in the current release of Adaptor, other distributions can be handled in the same way.

Let A be a one-dimensional array of N elements that is mapped to M processors. Then the block distribution of A to rectangular segments can be defined is as follows:

$$I_A = \{1, ..., N\} \qquad P = \{1, ..., M\} \qquad d_A \quad : \quad \mathrm{i} \mapsto \mathrm{(i*M\text{-}1)\ div\ N + 1}$$

The local part can be computed as follows:

$$d_A^{-1}(p) = \{ \text{ (p-1)*N div M + 1, ..., p*N div m } \}$$

The following example shows a block distribution of an array with 17 elements onto 5 processors. It should be noted that not every processor has the same number of elements.

```
 N = 17  M = 5   A(1:3)  A(4:6)  A(7:10)  A(11:13)  A(14:17)
                  P(1)    P(2)    P(3)      P(4)      P(5)
```

When using the above functions for block distributions it is not necessary to hold any partitioning tables. Every processor can efficiently compute its local range and the home of any data.

3.3 Data Movements and Schedules

Let A and B be two distributed arrays.

$$d : I_A \rightarrow P \qquad d : I_B \rightarrow P$$

Now let us consider an array assignment where values of B should replace values of A. Such a *data movement* can be described by a function m.

$$m : S_B(\subseteq I_B) \rightarrow I_A$$

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

$$C_{p \leftarrow q} = m^{-1}(d_A^{-1}(p)) \cap d_B^{-1}(q)$$

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.

Alls the sets $C_{p \leftarrow q}$ together are called the *schedule* of the data movement. If all sets are empty, no communication will be required and the operation is a local one.

3.3.1 Direct Data Movements

A data movement m will be called a *direct data movement* if every processor knows this function and can compute its local schedule without any communication. This means every processor p knows the values of $C_{p \leftarrow q}$ and $C_{q \leftarrow p}$ for all processors q.

Arrays assignments with array sections are a typical kind of direct data movements. The following example shows which schedule is generated for a simple array assignment when three processors are used. Only non-empty values within the schedule are of interest.

```
REAL A(1:20), B(1:13)
...
A(6:13) = B(3:10)
```

$$
\begin{array}{lll}
d_B^{-1}(1) = \{1, ..., 4\} & d_B^{-1}(2) = \{5, ..., 8\} & d_B^{-1}(3) = \{9, ..., 13\} \\
d_A^{-1}(1) = \{1, ..., 6\} & d_A^{-1}(2) = \{7, ..., 13\} & d_A^{-1}(3) = \{14, ..., 20\} \\
m^{-1}(d_A^{-1}(1)) = \{3\} & m^{-1}(d_A^{-1}(2)) = \{4, ..., 10\} & m^{-1}(d_A^{-1}(3)) = \{\}
\end{array}
$$

As in many other situations only some values of the sets $C_{p \leftarrow q}$ are not empty.

$$C_{2 \leftarrow 1} = \{4\}, \qquad C_{2 \leftarrow 3} = \{9, 10\}$$

3.3.2 Indirect Data Movements

A data movement m will be called *indirect data movement* if communication is required to compute the schedule. Therefore such a data movement has two steps of communication.

Indirect addressing of an array is a typical situation for data parallel Fortran programs that implies an indirect data movement.

```
REAL A(N), B(M)
INTEGER P(N)
...
A = B(P)
```

Every processor p can compute the values $C_{p \leftarrow q}$, that is the data required from other processors. But it does not know the values $C_{q \leftarrow p}$, that is the data needed by other processors. Therefore every processor p computes $C_{p \leftarrow q}$ and sends a description of this set to the processor q to inform it about the needed values. The actual data values can be sent and received in a second step.

3.4 Optimizations for the Runtime System

The DALIB is implemented in C. Most part of this library is portable between the different machines. Only the low level message passing commands, the timing functions and the random number generator have to be adapted to the hardware architecture.

Though the realization of the high level communication routines for data movements is based on the low level routines, these functions should be tuned for the underlying hardware architecture.

In the current release of the DALIB a data movement is a whole operation. Therefore the schedule of a data movement is always computed before it is used. It cannot be saved. As many application programs use a communication pattern more than once, there should be the possibility to reuse schedules. This is especially useful for array assignments with indirect addressing. It should also be possible to combine two schedules to one schedule to combine messages and to avoid additional communication start-up time.

Furthermore, for overlapping computation and communication it shoould be possible that data movements can run simultaneously with local computations.

4 Design of the Translation System

The following steps are done during the source to source transformation of Adaptor:

1. The source program is parsed and an abstract syntax tree will be generated.
2. Symbol tables are created and used for a semantic analysis.
3. The program (abstract syntax tree) is normalized as far as possible to reduce the complexity of the translation system.
4. The real translation on the internal abstract syntax tree and symbol tables has four phases:
   (a) analysis to verify that the program is suitable for translation with Adaptor,
   (b) splitting up statements in local and non-local operations and creation of the necessary temporary variables,
   (c) initial translation and serialization,
   (d) final translation with generation of calls to the DALIB
5. The new internal abstract syntax tree is unparsed back to source text.

All these phases of the source-to-source transformation are generated with a toolbox for compiler construction [6]. These tools have a great flexibility and can generate very efficient code. As intermediate language, abstract syntax trees will be used where the program module that defines the structure of the abstract syntax trees and provides general tree manipulating procedures is also generated by a tool. For the analysis and transformation components the new compiler tool *Puma* is utilized [5]. This tool cooperates with the generator for abstract syntax trees and supports the transformation and attribution of attributed trees. It is based on pattern-matching, unification and recursion. The flexibility of this tool allows not only to have a modular design but also to extend or modify the system in a very short time.

In the following the four phases of the real translation on the internal abstract syntax tree are described in more detail.

4.1 The Analysis Phase

In the first phase most checks are made to verify that there are no violations of the current restrictions. With some exceptions, there are possible work-arounds for the restrictions. Experiences have shown that many of the required changes in the data parallel source program help to think about more data parallelism and better data locality.

4.2 Temporary Variables

An array assignment is called a *non-local array assignment* if it needs a data movement or communication. In this case Adaptor splits up the assignment in primitive array assignments with communication and local array assignments. Sometimes new temporary arrays have to be inserted.

```
real A(N), B(N), C(N)     ! all distributed by default
...
A(1:K) = A(1:K) * B(1:K) - C(K+1:N-K)
```

This code needs at least one temporary array. Only one data movement for array C is required, the other data is local.

```
real A(N), B(N), C(N), A_TMP1(N)
...
A_TMP1(1:K) = C(K+1:N-K)                    ! requires communication
A(1:K) = A(1:K) * B(1:K) - A_TMP1(1:K)    ! local operation
```

In the same way parallel loops are handled. The loop will be split up into loops for data movements and loops that require no communication A loop with data movement is translated to a corresponding array operation.

```
real A(N), B(N)                 REAL A(N), B(N), A_TMP1(N)
...
forall (I=2:N-1)                A(2:N) = B(3:N)
  A(I) = B(I+1) + B(I-1)        A_TMP1(2:N) = B(1:N-2)
end forall                      FORALL (I=2,N=1)
                                   A(I) = A(I) + A_TMP1(I)
                                END FORALL
```

Due to the memory overhead attention should be paid by the user itself that not too many temporary arrays will be created.

4.3 Initial Translations

In this phase static and automatic arrays that are distributed become allocatable arrays. By this way it will be possible that the generated message passing programs can run for different numbers of processors.

Local array operations and forall loops will be translated to parallel loops. All these parallel loops do not require any communication.

The following program part shows the results of an input program after this phase:

```
                              INTEGER*4 N
                              PARAMETER ( N = 100 )
 integer N                    REAL*4 A (:), B (:), A_TMP1 (:)
 parameter (N=100)            INTEGER*4 I_1
 real A(N), B(N)              ALLOCATE (A(1:N), B(1:N), A_TMP1(1:N))
 a = b                        DO I_1=1,N  !parallel
 forall (i=2:n-1)                A(I_1) = B(I_1)
  a(i) = b(i) + a(i+1)        END DO
 end forall                   A_TMP1(2:N-1) = A(3:N)    ! array movement
                              DO I=2,N-1  ! parallel, independent
                                 A(I) = B(I)+A_TMP1(I)
                              END DO
                              DEALLOCATE (A_TMP1, B, A)
```

4.4 Final Translation

Now all statements are classified whether they are local statements or communication statements. After the final translation it is guaranteed that

- a parallel loop is restricted to the part of the arrays that is owned by the node process,

```
DO I = 1, N                          DO I = A_LOW, A_HIGH
   A(I) = B(I) * 2.0                    A(I) = B(I) * 2.0
END DO                               END DO
```

- data movements are translated to corresponding subroutine calls of the DALIB.

```
A(1:N-1) = B(2:N)   ->   call dalib_move_define (N,1,N-1,N,2,N)
                         call dalib_move_source1 (B,4,N,2,N)
                         call dalib_move_target1 (A,4,N,1,N-1)
```

By this way, the generated program alternates between phases of local computations and more synchronous phases of local and global communication.

4.5 Optimizations for the Translation System

In the current version of Adaptor no real optimizations are implemented. Some first experiments have shown that the following optimizations issues are the most important ones:

- loop fusion increases the potential of register optimizations,
- overlap analysis to reduce memory transfer,
- minimizing the use of temporary arrays,
- combination of messages to avoid communication overhead,
- and overlapping computation and communication.

Especially the introduction of overlapping areas makes it possible to reduce the number of temporary arrays. As overlapping areas decrease also the amount of local data movements, speed-ups of a factor about two have been pointed out in some of the considered codes.

Most of the mentioned optimization techniques will be investigated in the next future. This work is part of the Esprit project PPPE (Portable Parallel Programming Environment).

5 Summary

In this paper the design of a compilation system for data parallel programs has been presented. Though in the current release the translation itself is rather straightforward, the design has been proven to be useful and efficient.

The experiences with Adaptor have shown that the data parallel programming style can be used efficiently for massively parallel systems as data parallel programs can be compiled to efficient message passing programs. This allows users to write more quickly and more comfortable new applications for massively parallel machines. These applications are easy to understand and easy to maintain, they will run on nearly all architectures.

Though existent Fortran 77 applications might require recoding into array syntax there is a good hope that this can be supported also with tools.

Based on experiences with real application codes the most valuable optimization techniques have been pointed out. When integrating these optimization techniques in future compilers for data parallel languages, it is possible to be competitive to hand-written message passing programs.

Furthermore, language features are necessary to support sparse data structures, load balancing and functional parallelism in existent data parallel languages. Due to the high flexibility of the Adaptor system proposed features can be tested for usefulness in real scientific application programs.

Acknowledgements

The following people have influenced the Adaptor work by valuable discussions: James Cownie (Meiko, Bristol), Clemens-August Thole (GMD), Rolf Hänisch (GMD), Michael Gerndt (ZAM, Research Center Jülich), John Merlin (University of Southampton), and Dave Watson (NA Software, Liverpool).

References

[1] D. Callahan and K. Kennedy. Compiling Programs for Distributed-Memory Multiprocessor. *Journal of Supercomputing*, 2(2):151–169, October 1988.

[2] Thinking Machines Corporation. Connection Machine Model CM-2. Technical Summary Version 6.0, TMC, November 1990.

[3] G. Fox. Achievements and prospects for parallel computing. *Concurrency: Practice and Experience*, 3(6):725–739, December 1991.

[4] H.M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, 1989.

[5] J. Grosch. Puma - A Generator for the Transformation of Attributed Trees. Compiler Generation Report 26, GMD, Forschungsstelle an der Universität Karlsruhe, 1991.

[6] J. Grosch and H. Emmelmann. A Tool Box for Compiler Construction. *Lecture Notes of Computer Science*, 477:106–116, October 1990.

[7] High Perforamnce Fortran Forum. High Performance Fortran Language Specification. Final Version 1.0, Department of Computer Science, Rice University, May 1993.

[8] A. Mohamed, G. Fox, G. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Applications Benchmark Set for Fortran-D and High Performance Fortran. Technical Report 327, Northeast Parallel Architectures Center, 1992.

[9] Thinking Machines Corporation. CM Fortran Programming Guide, Version 1.0. Manual, TMC, January 1991.