A Unified Compiler Framework for Work and Data Placement*

F. Kuijlman, H.J. Sips, C. van Reeuwijk, and W.J.A. Denissen
Delft University of Technology, TNO-TPD
email: {F.Kuijlman,H.J.Sips,C.vanReeuwijk}@cs.tudelft.nl
den-wja@tpd.tno.nl

Abstract

In parallel programs the most important improvements in execution times can be achieved by the optimal placement of data and the optimal assignment of work to the processors in the system.

In a simple parallel programming environment this information is entirely specified by the user, but this places a heavy burden on the user. It is much more comfortable for the user if parts of the placement can be left unspecified.

However, this means that the user must be able to specify part of the data placement and work assignment, and it must be clear which data placements and work assignment are left to the compiler. The compiler must then find the optimal choices for the placements that the user has left unspecified.

In this paper we present a compiler framework that takes a program with partial work and data placement information, and transforms it into an explicit parallel program optimized for the amount of communication.

1 Introduction

Most parallel languages rely on user-specified parallelism in a program. This can take the form of parallel language constructs or annotations for the compiler. In some languages, parallel constructs explicitly specify work that can be done in parallel. However, in some languages parallel constructs only specify that operations can be done independently. For example the `forall` construct in HPF [5] and Fortran 95. The compiler must determine whether the operations can be done in parallel. Other parallel constructs specify that in addition each piece of work (usually an instance of an iteration) is to be assigned to an independent thread of execution like in OpenMP [2, 9].

The placement of data is another important aspect of parallel computation, especially on distributed-memory computers. Data placement is done by specifying data distributions, for instance by annotating the data with distribution functions.

*This research was supported by NWO (project 'Automap'), Esprit (LTR Project 'Joses' (#28198)), and Delft University of Technology (DIOC project 'Ubicom').

Ideally, the user should be able to use both data and work placement annotations in the same program, since that is the most flexible solution. However, both HPF and OpenMP impose restrictions, and the mixture of work and data distributions, and different semantic implications in various parallel languages, makes it hard for compilers to deal with all these kind of cases and to derive efficient parallel code.

To solve this problem, we have developed a generic compiler framework for dealing with work and data distribution in parallel languages in a uniform manner. This framework consists of two stages.

The first stage is the analysis phase in which partial work and data placement information is used to derive full placement information for every piece of code and data in the program. This includes determining possible placements of the code and data, and selecting an actual placement that minimizes the number of communication statements.

The second stage takes all the selected placement information and uses this, together with other hints provided through annotations, to transform a program into an explicitly parallel version that implements the computations and communication as efficiently as possible.

We have implemented this framework in our Spar/Java compiler [10, 12].

In Section 2, a small example will be used to illustrate how work and data placement can be specified in a uniform way and how these specifications can be combined. This will also be used as an introduction to the problems our compiler framework has to tackle. This is followed by Section 3 which lists the specifications for the annotations we use in parallelizing a program.

Section 4 describes how full placement information is derived for all code in the program. Section 5 describes the different transformations that need to be done to obtain an efficient parallel program.

2 The problem

In explicit parallel languages, like HPF, OpenMP and our own language Spar/Java, the user is responsible for specifying the parallelism. However, parallel constructs or compiler directives in parallel programs usually only partially specify all the parallel details a compiler requires to generate efficient code. Take for example the following program:

```
double A[*] <$on = (lambda (i) P[(cyclic i)])$> = new double [N];
double B[*] <$on = (lambda (i) P[(cyclic i)])$> = new double [N];
double C[*] <$on = (lambda (i) P[(cyclic i)])$> = new double [N];
foreach( i :- 0:N )
    A[i] = B[i]*C[i];
```

The `foreach` construct is an example of a parallel construct taken from the parallel language Spar/Java [13], which means that each iteration instance `i` of the loop body is to be executed atomically, but that the iterations can be executed in arbitrary order. Together with dependence analysis on the body of the loop, the construct is in this example equivalent to an HPF `forall` construct.

The data declarations define three one-dimensional arrays. The declarations contain annotations that specify that the arrays are distributed cyclically among the processors. To be able to execute the loop in the example, we need to know to which processor each iteration is assigned. In such cases compilers usually apply a default rule, like the *owner computes rule*[1]. In the owner computes rule, the processor on which the left-hand array element is located performs the evaluation of the right-hand expression. Although such a default scheme is easy for the compiler, it is not necessarily the most efficient scheme.

In the next example, only the distribution of work is specified:

```
double A[*] = new double [N];
double B[*] = new double [N];
double C[*] = new double [N];
foreach( i :- 0:N )
    <$on = P[(block @i 5)]$> A[i] = B[i]*C[i];
```

In this case, we assign the iterations of the loop block-wise with block size 5 to the processors. The distribution of the data structure `C[]` is left unspecified. This example is in the spirit of OpenMP type of parallel constructs. As long as data is located in a real shared memory, no further specification is necessary. However, if the parallel processors have local memories within a single address space, the compiler has come up with a suitable distribution scheme.

As a last example, both work and data distributions can occur in a single loop body:

```
double A[*] <$on = (lambda (i) P[(cyclic i)])$> = new double [N];
double B[*] <$on = (lambda (i) P[(cyclic i)])$> = new double [N];
double C[*] <$on = (lambda (i) P[(cyclic i)])$> = new double [N];
foreach( i :- 0:N )
    <$on = P[(block @i 5)]$> A[i] = B[i]*C[i];
```

Here the work distribution explicitly overrules the default rule implied by the given data distributions. Such constructs appear for instance in HPF when using the `on home` construct, although in HPF the distribution of iterations is specified indirectly through a data distribution.

The last example suggests that if both work and data distributions are properly specified in a users program, a compiler could do a straightforward translation. This turns out not to be true, because a compiler often needs to introduce temporary data structures in breaking down the computations and to take care of the communication. These temporaries need to be given a work and data distribution as well.

Another problem in distributed-memory architectures is to determine whether or not communication is necessary. Communication is needed when different data used in an expression or statement is located on different processors. This analysis is not always simple, especially when distribution parameters are not known at compile time (e.g. the `inherit` attribute in HPF). In the worst case, the compiler will generate communication statements that only send data to the same processor, leading to inefficient programs.

To deal with all of the above cases, we have developed a generic compiler framework for analyzing code and data distributions. The framework has the following properties:

- Work and data distributions are treated in a unified manner;

[1]Note that the HPF standard does not prescribe the owner computes rule.

- Work distributions can be defined at any level of detail: functions, blocks, statements, and expressions;
- Partial work and data distributions are supported;
- Minimal communication through placement of computations;
- Communication requirements are determined using symbolic comparison of distribution functions.

3 Annotations

In the Spar/Java implementation of our parallelization framework, we have used the generic Spar/Java annotations for the placement specifications. Also, in order to help the compiler to analyze a program it is also possible to give the compiler hints which will aid it in generating better code.

All annotations will be in one of two forms:

```
<$flag$>
<$var=expr$>
```

We will use the using the following simple example as illustration:

```
double A[*] = new double [N];
double B[*] = new double [N];
double C[*] = new double [N];
foreach (i :- 0:N)
    A[i] = A[i] + B[i] * C[i];
```

The iterations of the example can be done in parallel. In general this requires complicated analysis to conclude that this is the case. To aid the compiler we have introduced the `independent` annotation:

```
<$independent$> foreach (i :- 0:N)
    A[i] = A[i] + B[i] * C[i];
```

This is however not enough as the compiler also needs to know where to perform the computation. This can be achieved in two ways. The first is to specify where the data is located:

```
double A[*] <$on = (lambda (i) P[(cyclic i)])$> = new double [N];
double B[*] <$on = (lambda (i) P[(block i 5)])$> = new double [N];
double C[*] <$on = (lambda (i) P[(block i 5)])$> = new double [N];
```

The compiler will then have to derive where the compilation has to be done. This derivation may not always produce the result you want. In that case you also have the option of specifying directly where the computation must take place:

```
<$independent$> foreach (i :- 0:N)
    <$on = P[(cyclic @i)]$>A[i] = A[i] + B[i] * C[i];
```

The two forms of specifying `on` annotations shown above are very similar. The first one using `lambda` expressions is called a *owner function* and binds each parameter of the `lambda` expression to the corresponding dimension of the array.

This function is then applied to the actual indices of an array access expression to derive the actual *owner* of that expression. For example, the owner of the expression `A[j+5]` would be `P[(cyclic j+5)]`.

The second form is called an *owner expression* and is used to specify an *owner* directly, without the need for any function applications. However, as the namespaces of the annotations and the Spar/Java language are separate, we use the `@i` notation to bind to the loop variable. This means that for example the iteration with `i=4` has `P[(cyclic 4)]` as owner.

There are several work and data distributions that can be used in the current Spar/Java implementation of the framework:

```
cyclic <expr>                   == <expr>%nProcs
block <expr> <blocksize>        == <expr>/<blocksize
blockcyclic <expr> <blocksize>  == (<expr>/blocksize)%nProcs
_all                            == replicated on all nProcs processors
_                               == don't care, compiler makes decision
local <expr>                    == <expr>
```

When using a 1-dimensional processor array, `nProcs` is equal to the total number of processors in the system. When using a multi-dimensional processor array, `nProcs` is equal to the number of processors in the dimension for which the distribution is used.

4 Derivation of owners

In order to derive owners for statements which will generate the least amount of communication we go through a multi-stage process. In the first stage (see Figure 1) we use the owners attached to the variable declarations to derive the owners for all variable accesses. This uses the simple substitution rules for `lambda` expressions as described in the previous section. The result of this stage is that all leaf nodes of the parse tree will now have an owner attached to it.

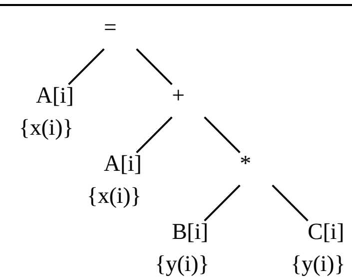

Figure 1: The parse tree with owners on variables

In Figure 1, `A[i]` is a variable access while `{x(i)}` denotes the owner attached to it.

By default, in Spar/Java arrays and scalars are replicated. It depends on the application whether or not this choice is the best one. If data is mostly read then no communication will be necessary. On the other hand, if data needs to be written an expensive broadcast operation must be used.

In some cases, notably nested array accesses, it is necessary to replicate intermediate expressions. This is taken into account by the compiler.

Following this we perform a depth-first traversal of the parse tree (see Figure 2) to derive all *possible* owner expressions for all the internal nodes of the parse tree.

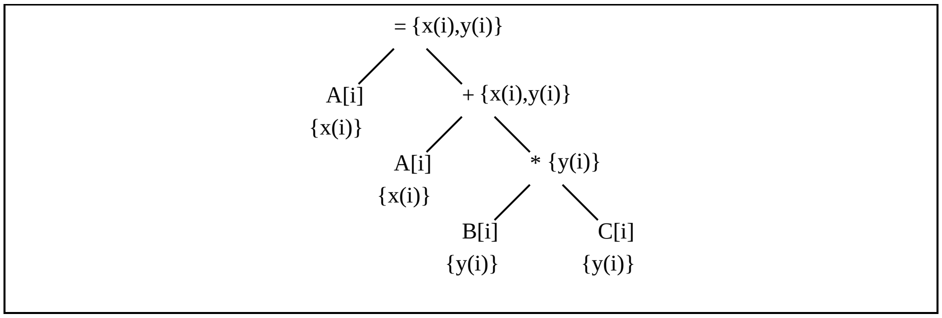

Figure 2: The parse tree with all possible owners

For each node in the parse tree with multiple possible owners we now want to select the one owner expression that will result in the minimal amount of communication. We assume that this can be achieved by introducing the minimal number of communication statements. This solution can be found by an exhaustive search that uses a cost function to determine the cheapest solution. All possible combinations of owners are considered using a cost function that sums the cost for all edges as defined by:

```
vertices have same owner      -> cost == 0
vertices have different owner -> cost == 1
```

Testing whether owners are the same or not is done symbolically using a recursive equivalence test. This test makes use of constant folding and reordering to determine equivalence.

In the case replicated owners are involved, the lists with multiple *possible* owners is first reduced before this search takes place. Owners for read operations (data in RHS of assignment) is minimized. This means that if there is a choice between a `cyclic` and a replicated owner, the replicated one is discarded. On the other hand, owners of assignments are maximized, and the `cyclic` one is discarded.

Figure 3 shows two possible selections of owners, with the solution with `cost=1` being the one our compiler will select and the other the one normally used in the *owner-computes scheme*. This means that our compiler will be able to generate code using only one communication action, while the *owner-computes* scheme would require 2 communications.

4.1 Algorithmic complexity

As previously mentioned the exhaustive search algorithm tries every possible solution. This can lead to a combinatorial explosion, which in turn can cause the compiler to take an exponential amount of time to find the best solution. Our initial implementation did this full search on complete functions. Given that a function has p statements, each with q internal nodes with r possible owners,

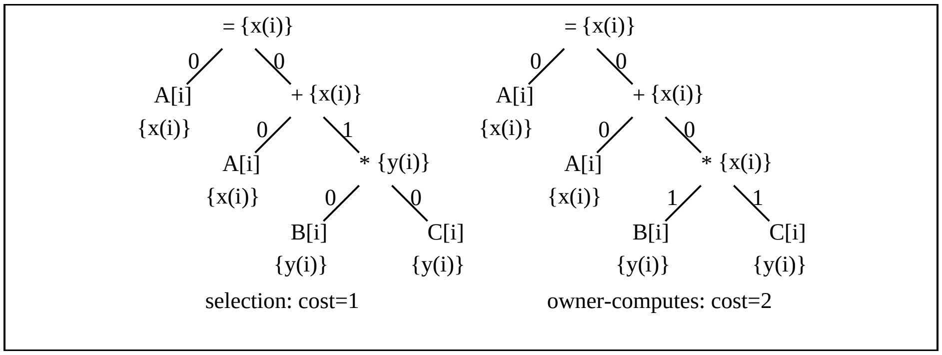

Figure 3: The two possible owner selections

this would mean that the compiler has to consider $r^{p \cdot q}$ possible solutions. This causes the search time to explode for all but the simplest toy problems.

To limit this explosion we have introduced two heuristics. The first one is to limit the search to single statements. This means that the number of possible solutions to be examined is reduced to $p \cdot r^q$.

We have also put a limit on the number of solutions in a single search. Thus, if the number of solutions in a parse tree exceeds this limit, we first solve a subtree that falls below the limit before solving the rest of the tree. So, if the subtree and the tree above it have the same size, this would reduce the search from $r^{2 \cdot q}$ to $2 \cdot r^q$ solutions.

The second heuristic can clearly generate suboptimal solutions. However, for the (numerical) programs we have considered to date, the first heuristic has been enough to prevent the exponential explosion. The solutions generated for these programs were the same for both the statement-wide and function-wide searches, which leads us to conclude that at least for these programs the owner solutions for consecutive statements are independent of each other.

For example, our compiler will need only seconds to process the NAS MG benchmark using the first heuristic, while without it we stopped the compiler after half an hour.

There is also a 2-sweep algorithm [8] that can produce a solution in $O(p \cdot q)$ time. However, this algorithm puts severe restrictions on the characteristics of the parse tree which would drastically reduce its effectiveness for real-life programs.

5 Transformations

Using the owner expressions selected in the previous section, the example program looks as follows:

```
<$independent$> foreach (i :- 0:N)
    <$on = P[x(@i)]$>A[i] = A[i] + (<$on = P[y(@i)]$>B[i] * C[i]);
```

The assignment statement has one subexpression with a different owner than the rest of the statement. This subexpression will lead to a communication statement, and the first step to make this explicit is by introducing temporary variables:

```
<$independent$> foreach (i :- 0:N){
    double tmp1;
    <$on = P[x(@i)]$>tmp1 = (<$on = P[y(@i)]$>B[i] * C[i]);
    <$on = P[x(@i)]$>A[i] = A[i] + tmp1;
}
```

The problem with this version is that if we do not do any further transformations, this will lead to element-wise communications which is not very efficient. What we would like is to be able to do aggregate communications. This is achieved by first performing scalar expansion (replacing the scalar temporary variables with array temporaries):

```
double tmp1[*] = new double [N];
<$independent$> foreach (i :- 0:N){
    <$on = P[x(@i)]$>tmp1[i] = (<$on = P[y(@i)]$>B[i] * C[i]);
    <$on = P[x(@i)]$>A[i] = A[i] + tmp1;
}
```

As the loop is annotated with `independent` this means that the loop iterations are independent of each other. We also know that the first statement in the loop has been automatically generated by the compiler. These two pieces of information allow us to put the first statement into a separate loop:

```
double tmp1[*] = new double [N];
<$independent$> foreach (i :- 0:N)
    <$on = P[x(@i)]$>tmp1[i] = (<$on = P[y(@i)]$>B[i] * C[i]);
<$independent$> foreach (i :- 0:N)
    <$on = P[x(@i)]$>A[i] = A[i] + tmp1;
```

We now have reached a form where the first loop allows us to perform aggregate communication. This means there will be at most one data packet being sent from a processor to any other processor.

After *communication aggregation* other optimizations such as *owner absorption*([11]) will be done. This basically means that owner tests using regular distributions like `cyclic` or `block` will be replaced by recomputed loop bounds and strides, resulting in smaller iteration volumes to be traversed by each processor.

6 Related work

Deriving work assignment from data layout specifications was first applied for SIMD processors. Gilbert and Schreiber [4] consider work distribution of array assignments on a fixed hardware topology (an SIMD like processor array). They show that an optimal work distribution can be found for a restricted class of problems (no common sub-expressions, only regular access functions, and known bounds), using a two sweep algorithm and a specific cost function with certain characteristics. Chatterjee et al. [1] extended the work of Gilbert et al. to array variables (user or compiler introduced), a larger set of array operations, and basic blocks as the scope of owner selection. Korstanje [8] removed some restrictions from Chatterjee's work by allowing reordering of associative operators, the usage of irregular owners, and true multi-dimensional owner comparison. The technique described in this paper is more general and can be applied to support the majority of parallel features in the parallel languages under consideration. Apart from the implementation in the Spar/Java compiler framework, it is shown in Denissen [3] that the method supports all features of the full HPF 2.0 language. The method integrates well with sequential optimizations and the distribution of work has the same two-level mapping as HPF data distributions. In addition, partially specified mapping information can be handled, ranging from 'aligned to an inherited mapping' to 'aligned to a distributed template'. Therefore, the exact distribution onto processors does not need to be known at compile-time (e.g. inherited mappings). The size of the search space in our approach is roughly the product of all owner options and is not related to the size of the processor array as in Chatterjee et al. [1]. Therefore, a full search algorithm can be applied in many more cases.

Another approach is presented by Kamachi et. al. [7]. They present methods for generating communication in compiling HPF programs. They introduce the concept of an iteration template, which corresponds to an iteration space. Their HPF compiler performs the loop iteration mapping through a two-level mapping of the iteration template in the same way as data mapping is performed in HPF. Making use of this unified mapping model of data and work, communication for non-local accesses is handled. This strategy is a limited form of alignment analysis as presented in this paper. Kamachi et al only allow communication based on data realignment. In our terminology, owner conflicts can only occur on edges where the child is a subscript operator. Only a single owner can be selected for the complete assignment. This single owner also has to be a mappable owner. The resulting two-owned assignments all have equally aligned left-hand side expressions, and a simple array subscript as right-hand side expression.

Josha and Bannerjee [6] describe a method to find out whether or not a parallel statement in HPF (or part of it) is communication free. Their method is based on Fourier-Motzkin elimination. The method described in this paper is much simpler and, we believe, just as powerful. The examples they use in their paper are handled properly by the technique applied in this paper.

7 Conclusion

We have created a flexible framework that supports data placement, work placement, and a combination of the two. Our compiler will take this specification and determines the missing placements, while searching for a solution that minimizes the amount of communication that is required. The compiler then uses a series of transformations to generate a parallel program with efficient communication where possible.

8 Future work

Currently the owner equivalence test does not compare constant distributions. We intend to add more aggressive constant folding so that if different distributions have constant indices such that they would evaluate to the same owner, the compiler will be able to determine this.

References

[1] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. *ACM TOPLAS*, pages 123–156, Jan. 1995.

[2] L. Dagum and R. Menon. OpenMP: an industrial standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January-March 1998.

[3] W. Denissen. *Design of an HPF Compiler: A compilation Framework for a Data-parallel Language*. PhD thesis, Delft University of Technology, 2000. ISBN 90-6464-197-8. Available at: pds.ewi.tudelft.nl/pubs/ph_d/Denissen.pdf.

[4] J. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, Sept. 1991.

[5] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edition, Feb. 1997.

[6] P. Joisha and P. Bannerjee. Exploiting ownership sets in HPF. In *Proc. of the 13-th International Workshop on Languages and Compilers for Parallel Computing*, Yorktown Heights, Aug. 2000. Published in LNCS 2017, pp. 259–273.

[7] T. Kamachi, K. Kusano, K. Suehiro, Y. Seo, M. Tamura, and S. Sakon. Generating realigment-based communication for HPF programs. In *Proc. of the 10th International Parallel Processing Symposium*, pages 364–372. IEEE Press, 1996.

[8] V. Korstanje. A new framework for compiling High Performance Fortran (HPF), decomposing parallel operations to a basic forall. technical report PDS-1998-005, Delft University of Technology, Feb. 1998.

[9] OpenMP fortran application interface, v 2.0. Technical report, OpenMP Organization, June 2000. www.openmp.org.

[10] C. v. Reeuwijk. Timber download page. www.pds.ewi.tudelft.nl/timber/-downloading.html.

[11] C. v. Reeuwijk, W. Denissen, H. Sips, and E. Paalvast. An implementation framework for HPF distributed arrays on message-passing parallel computer systems. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):897–914, Sept. 1996.

[12] C. v. Reeuwijk, F. Kuijlman, and H. Sips. Spar: an extension of Java for scientific computation. In *ACM Java Grande – ISCOPE Conference*, pages 58–67, June 2001.

[13] C. v. Reeuwijk, F. Kuijlman, H. Sips, and S. Niemeijer. Data-parallel programming in Spar/Java. In *Proc. of the Second Annual Workshop on Java for High-Performance Computing*, pages 51–66, May 2000.