Privatization and Distribution of Arrays
A Preliminary Proposal

Peng Tu

Abstract

In today's high performance NUMA (Non-Uniform Memory Architecture) multiprocessors with memory hierarchy or distributed memory, the partition and distribution of data associated with parallel computations affect the amount of parallelism that can be exploited and the amount of data movement in the system. The objective of this research is to study and evaluate compile time data management techniques to enhance parallelism and to improve locality of memory reference for large scientific programs written in Fortran.

Our first step is to reduce the amount of shared data through *privatization*. Privatization is a technique that allocates a separate copy of a shared variable in the private storage of each processor such that each processor can access a distinct instance of the variable. Privatization can enhance inherent parallelism of a program by eliminating memory-related anti- and output dependences. It can also improve the locality of references since accessing a private variable is inherently local and communication free. We present our algorithm for array privatization and the result of our experiment on the effectiveness of the algorithm.

For the remaining shared data, we introduce a new concept: placement matrix, and show its application in deriving data alignment and data decomposition to reduce communication. We also incorporate the ratio of communication to computation in our evaluation of different data partitions. The work is continuing on heuristics for data distribution and the implementation of the tools.

1 Introduction

Non-Uniform Memory Architecture (NUMA) multiprocessors promise to be a cost efficient way to deliver high performance by exploring parallelism in scientific applications. While they have advantages in scalability and cost, the lack of uniform access global memory makes it difficult to program and manage. The difference in speed between local access and non-local access can range from one order of magnitude for a machine with shared memory hierarchy such as Cedar to two orders of magnitude for a machine with distributed memory such as CM-2. Communication cost can limit their performance. Software support for proper management of data distribution and load balancing become crucial for these machines to achieve high performance and efficiency at a reasonable programming cost.

1.1 SPMD Programming Model

Two programming models have been used for parallel multiprocessors. In the *fork-join* model, a program is executed sequentially by one processor until it enters a parallel section.

When entering a parallel section, the sequential thread *forks* into a number of parallel threads to be executed on several processors. At the end of the parallel section, the parallel threads *join* into one sequential thread with a single processor proceeds. In the SPMD model, all processors will execute the same program. Each processor can execute different piece of work by operating on different piece of data. In the *private section* of a program, each processor computes redundantly on its *private data.* In the *shared section* of a program, each processor cooperates on a portion of parallel work on *shared data.* One task is created for each processor at the beginning and killed at the end of a program execution. The number of concurrent tasks is fixed for a program and the system overhead for spawning and switching is minimized. In the following SPMD program,

```
SHARED A(100),B(100)
PRIVATE X

PARALLEL REGION
X = ...
DOALL I = 1, 100
   A(I) = B(I)/X
ENDDO
END PARALLEL REGION
```

each processor redundantly computes for private variable X and then cooperates in computing shared array A. If X is not redundantly computed, it will have to be broadcasted to all the processors. The *doall* loop is shared, different iterations of the loop can be executed in parallel by different processors.

The assignment of the iterations to processors is generally determined by the distribution of data involved in the computation for each iteration. For instance, if the array A and B is distributed in such a way that A(I) and B(I) are in the same processor, the ith iteration of the loop can be scheduled to be executed on the processor where A(I) is located.

Since computation and data are closely related, many compilers for distributed memory machines use *owner computes* rule to determine which processor shall execute a particular piece of a shared section[ZBG88] [CK88][RP89]. According to the data distribution specified by user or compiler, each data element is assigned to a *owner* processor which is the one that stores that data element in its local memory. The owner processor of a data item executes all the instructions that modify its value. Communication command is generated to fetch the non-local data to the owner from the computation. Parallelism is realised by partitioning the data, hence the computation is indirectly partitioned. Data dependence is enforced through communication of data value from the producer to consumer. Since SPMD model can provide better efficiency, it has been supported in Cray MPP Fortran [?] and Parallel Computing Forum Fortran.

1.2 Issues in Data Distribution

The task of compilers for today's high performance multiprocessors is to explore inherent parallelism in the program, and to explore the parallelism in the machines. The parallelism in the program can be explicitly specified by user or implicitly embedded in it data and control dependence structure. The parallelism in the machine is constrained by its number of processors, memory organization and underlining interconnection network. A compiler maps the program parallelism onto the machine parallelism to achieve performance and efficiency. This mapping includes data partition, task partition and matching task with data in the target machine. To obtain a good mapping, the following fundamental issues and tradeoffs must be taken into consideration.

Locality and privatization. Private data are allocated in the local memory of each processor, it cannot be accessed by other processors. Access to private data is inherently local, privatization can improve the spatial locality of reference of a program. In the SPMD model, redundant execution of program can be performed on private data. This provides an opportunity for computing required data locally instead of getting the data from remote processors.

Load balance and communication highlights the tradeoffs in the compiling process. Maximum load balance requires that computations be spread across all the processors and hence parallelism in the computations can be achieved. But then the data involved in the computations may not be local to all the processors, communication overhead can be expensive for bringing non-local data to the processors. Minimum communication requires that all the data involved in a computation to be in the same processors. Then some computations that could be spread into several processors may have to be executed on a single processor to preserve the data locality and hence parallelism in the program will suffer. Data replication and privatization can be used in some case to reduce communication while preserving parallelism.

Partition data and partition iteration space. Most compilers that generate message passing code from a shared memory program and a user specified data decomposition take the data partitioning approach. Once the owner of the data is determined, message passing code is generated according to *owner computes* rule. Since the owner of the variable on the left hand side of an assignment statement is to compute its new value, messages are generated to get values of variables owned by other processors for the computation. Communication can be reduced if the data decomposition keeps most of the data in the computation local. However, since the computation is indirectly partitioned, load balance and parallelism will suffer if data decomposition does not distribute the owners of the computations evenly across the processors. Partitioning of the iteration space has been used by automatic parallelizing compilers to exploit parallelism on shared memory machines. The advantage is that it can ensure load balance and parallelism. But the communication overhead to bring proper data to the site of each iteration may be large since the data are indirectly decomposed.

Data alignment determines which portions of two arrays shall be in the same processor for a particular data partition. The objective of data alignment is to keep portions involved in the same computation together so as to maximize local access and reduce communication. Most regular alignments are determined by three parameters: *orientation*, *offset*, and *stride*. Alignment constraints can be derived from subscript expression. Finding optimum alignment is difficult. The orientation problem has been formulated as the problem of *component affinity graph* and shown to be NP-complete[LC91]. As different computations require different alignments, it may not be feasible to keep a fixed alignment in the entire program. *Dynamic alignment* may provide a more efficient program.

The **communication pattern** can be classified in various ways. We identify two classes, *uniform communication* and *non-uniform communication*. Uniform communication happens where the relationship between the sender and receiver is invariant of their locations. It is important for distributed memory multiprocessors with regular interconnection, since a uniform communication is usually conflict free, i.e. each sender has a conflict free communication path to its receiver. Uniform communication usually costs less on real machines since the direction of data flowing through the communication network is uniform and network congestion is less severe than non-uniform communication. Techniques like vector and block data transfer can be used in uniform communication to explore the regularity of machine's interconnection network.

Pre-fetching is a way to hide the communication latency. By pre-fetching data and storing them in local temporary locations before they are used, computation and communication can be overlapped. In the case of dynamic alignment, data can be pre-aligned, where alignment is changed before computations are carried out on the data.

We will expand on the issues in the rest of this proposal as we explain our plan for this research. The rest of this proposal will be divided into two broad sections. In the first section, we will discuss our work on *automatic array privatization* [?] which can enhance program's inherent parallelism and improve locality of reference on target machines by reducing the amount of shared data. In the second section, we will discuss our framework on *automatic array distribution* for exploiting parallelism and reducing communication.

2 Privatization for Parallelism and Locality of Reference

Enhancing parallelism, balancing load and reducing communication are among the major tasks of compilers for distributed memory systems. Privatization is a technique that allows each concurrent process to allocate a variable in private storage such that each process accesses a distinct instance of the variable. Privatization of scalars has been used for many years in parallelizing compilers and is well understood [BCFH89]. In this section, we will focus on techniques for the privatization of arrays.

By providing distinct instance of a variable to each processor, privatization can eliminate memory related dependences and enhance parallelism. It reduces communication since access to a private variable is local to the processor. Previous studies on the effectiveness of automatic program parallelization show that *array privatization* is one of the most effective transformations for the exploitation of parallelism [EHLP91]. Also, privatization can improve load balancing under some translation systems targeted at distributed memory machines. To illustrate this point, consider the loop:

```
S0: DO I = 1, N
S1:   DO J = 1, N
S2:     A(J)   = B(I,J) + 1
S3:     C(I,J) = A(J) * D(I)
S4:   END DO
S5: END DO
```

Here, array `A` is involved in a cross iteration anti-dependence which prevents the outer loop to be executed in parallel. In a distributed memory system using the *owner computes* rule, load imbalance will happen due to extensive computation at the owner of `A(J)`. If `C(*, J)` is not distributed with `A(J)`, communication will be necessary from the owner of `A(J)` to the owners of `C(*, J)`.

These problems can be relieved by *array expansion* [PW86] [Fea88]. For the loop above, this means expanding A into a two dimensional array as shown next:

```
S0: DO I = 1, N
S1:    DO J = 1, N
S2:       A(I,J) = B(I,J) + 1
S3:       C(I,J) = A(I,J) * D(I)
S4:    END DO
S5: END DO
```

In this case, array expansion resolves the anti-dependence, exposes more parallelism and improves load balance. However, expansion makes the array references more complicated. Also, distribution and alignment pattern has to be specified in conjunction with the expansion. Otherwise, if data are not distributed in such a way that A(I, J) is located with C(I, J), communication from the owner of A(I, J) to the owner of C(I, J) would be necessary.

Array privatization together with a *owner stores* rule [Bal91] may result in a more efficient program. By allocating a private variable to each processor, the value of A(J) is computed locally, load is evenly distributed. Communication happens only for storing A(J) to its owner.

```
S0: DO I = 1, N
S1:    DO J = 1, N
S2:       PRIVATE X
S3:       X = B(I,J) + 1
S4:       C(I,J) = X * D(I)
S5:       IF (I.EQ.N) A(J) = X
S6:    END DO
S7: END DO
```

It can be completely eliminated if the compiler supports dynamic data redistribution or if the compiler determines statically that A(J) should be distributed with C(N, J). The dynamic nature of privatization also makes it easier to adapt the program to different physical machines.

The benefit of array privatization is similar to that of scalar replication in distributed memory systems [GB92][HKT91]. Instead of statically partitioning and distributing all the data, the compiler can dynamically allocate those private data to generate more efficient codes. In the rest of this section, we first give some definitions and an algorithm for identifying privatizable arrays within loops and go through an example. Then we explain our method to evaluate the effectiveness of its use on enhancing parallelism and give some preliminary performance results.

2.1 Array Privatization

For an array A to be private to a loop L, A must satisfy two conditions: (1) the array A must be written by some statements in L, otherwise there will be no benefit of making A private to L; (2) if there is a use of an element of A in any iteration, that element must be written in the same iteration before the use. The first condition prevents unnecessary privatization, and the second condition ensures that every use of A refers to values computed in the same iteration.

Note that it is possible to privatize a subsection of an array. Privatizable scalars are special cases where the array contains only one element. The definition can also be generalized from iterations of a loop to threads of a program section. An array is privatizable to a program section when every use of the array in the section is first written by a statement in the same thread. Later, when we talk about interprocedure analysis, we will use subroutine invocations as threads and find privatizable arrays for subroutines.

The problem of identifying privatizable arrays is therefore to determine for each use of an array in a loop if there are definitions of the array in the same iteration which *must reach* the use, and no other definitions of the array outside the iteration can reach the use. This problem can be formalized in a data flow framework as discussed next.

Consider a control flow graph for a loop body, which consists of basic blocks and control flow in one iteration. For each basic block `S`, we compute a set of *must-define* (subscripted) variables, `DEF(S)`, and a set of *possibly exposed-use* variables, `USE(S)`. By *exposed*, we mean that the variable used is not defined by any preceding statement in `S` and hence it is exposed to definitions outside of `S`. Define `IN(S)` as the set of variables that are always defined upon entering `S`, and `Pred(S)` as the set of immediate predecessors of `S` in the control flow graph of the loop body. `IN(S)` can be computed using the following equation

$$\mathtt{IN(S)} = \cap_{\mathtt{t}\in\mathtt{Pred(S)}}(\mathtt{IN(t)} \cup \mathtt{DEF(t)})$$

The evaluation of the data flow equations listed above starts at the innermost loop. It traverses the loop nests following the loop nest tree in post order. Intuitively, the algorithm determines privatizable arrays for the inner loop first and propagate the *define-use* information of the inner loop to the outer loop.

For each loop `L`, all the IN sets are initially set to be empty sets ($\{\}$). Data flow information is computed for each statement and propagated through the subgraph. It is then summarized for one iteration of `L`.

Summary information for one iteration of `L` is obtained as follows. `DEF(L)`, the set of *must-defined* variables for one iteration of `L`, is the set of *must-defined* variables upon exiting the iteration, i.e. $\mathtt{DEF(L)} = \cap(\mathtt{IN(t)} \cup \mathtt{DEF(t)}) : \mathtt{t} \in \mathtt{exits(L)}$. `USE(L)`, the *possibly exposed-use* variables are the variables which are used in some statements of `L`, but do not have a *reaching must-defined* in the same iteration, i.e. $\mathtt{USE(L)} = \cup(\mathtt{USE(t)} - \mathtt{IN(t)}) : \mathtt{t} \in \mathtt{L}$. The *privatizable variables* are the variables which are defined and not exposed to definitions outside the loop iteration, i.e. $\mathtt{PRI(L)} = \mathtt{DEF(L)} - \mathtt{USE(L)}$.

The summarized `DEF(L)`, `USE(L)` and `PRI(L)` are then aggregated across the iteration space of `L`. The aggregation process computes the region spanned by each array reference in `USE(L)`, `DEF(L)` and `PRI(L)` across the iteration space. Since the aggregated data contain all the information for the analysis of outer loop, the inner loop `L` is collapsed into one single node in the analysis of outer loop. The algorithm is shown below.

```
Algorithm Privatize
  privatize := func(body, L)
  Input: body for loop L
  Output: DEF(L), USE(L), PRI(L)
Phase 1: Collect local information
  foreach $S \in$ body do
     if S is a DO loop then
```

```
            ! S is an inner loop, visit S first
            [DEF(S), USE(S)] := privatize(body(S), S)
            collapse all nodes in body(S) into S
        else
            calculate local DEF(S), USE(S)
        endif
    endfor
Phase 2: Propagate flow information
    forall S ∈ body initialize IN(S) := {}
    foreach S ∈ body in topological order do
        IN(S) := $\cap_{t \in Pred(S)}(IN(t) \cup DEF(t))$
    until fix-point
Phase 3: Compute PRI(L), DEF(L), USE(L) as a function of the loop index
    DEF(L) := $\cap_{t \in exits(body)}(IN(t) \cup DEF(t))$
    USE(L) := $\cup_{t \in body}(USE(t) - IN(t))$
    PRI(L) := DEF(L) − USE(L)
Phase 4: Return aggregated DEF(L) and USE(L)
    aggregate PRI(L),DEF(L) and USE(L) in L
    annotate L with PRI(L), return [DEF(L), USE(L)]
```

Using the inner loop in our example at the beginning of this section, we illustrate the steps taken by our algorithm in determining privatizable arrays. In phase 1–2, we compute local USE and DEF for each statement, propagate the information in topological order through the loop body and compute the IN sets. The result for the original loop:

```
    DEF      USE              IN
S1: {J}      {N}              {}
S2: {A(J)}   {B(I,J),I,J}     {J}
S3: {C(I,J)  {A(J),D(I),I,J}  {A(J),J}
S4: {}       {}               {A(J),C(I,J),J}
```

In phase 3, we summarize the information for one iteration of L:

```
DEF(L) = IN(S4) = {A(J),C(I,J),J}
USE(L) = (USE(S1)-IN(S1)) + (USE(S2)-IN(S2)) +
         (USE(S3)-IN(S3)) + (USE(S4)-IN(S4))
       = {B(I,J),D(I),I,N}
PRI(L) = DEF(L) - USE(L) = {A(J),C(I,J),J}
```

In phase 4, we aggregate the summary information across the iteration space to get corresponding information for all the iterations of loop L:

```
PRI(L) = {(A(J),J=1,N),(C(I,J),J=1,N),J}
DEF(L) = {(A(J),J=1,N),(C(I,J),J=1,N),J}
USE(L) = {(B(I,J),J=1,N),D(I),I}
```

PRI(L) and DEF(L) can be simply expanded across the iteration space. Since $(\mathtt{A(J)}, \mathtt{J} = \mathtt{1}, \mathtt{N}) \in$ PRI(L) and there is no free variable in $(\mathtt{A(J)}, \mathtt{J} = \mathtt{1}, \mathtt{N})$, $\mathtt{A(1:N)}$ is privatizable to L.

USE(L) can also be expanded in the same way but the result may not be precise since one iteration's use may only be exposed to the definitions in some previous iterations of the same loop. This kind of use is not truly exposed to the outside of the loop body and shall be excluded from the aggregated USE set. For instance, in

```
   DO I = 2, N
S1:   A(I) = A(I-1) + B(J)
   END DO
```

the information for one iteration is $\mathtt{USE(L)} = \{\mathtt{A(I-1)}, \mathtt{B(J)}, \mathtt{J}\}$, and $\mathtt{DEF(L)} = \{\mathtt{A(I)}\}$, the region aggregately defined in all iterations prior to the ith iteration is $\mathtt{A(2:I-1)}$, $\mathtt{A(I-1)}$ is exposed to definitions outside the loop only in the first iteration, i.e. $\mathtt{USE(L)} = \{\mathtt{A(1)}\}$.

Now, we can collapse the loop body into one node L with the aggregated DEF(L) and USE(L) as its local information and run the same procedure for the outer loop.

Live analysis is needed to determined if a privatizable variable is live on exiting the loop. If it is live on exit, last value assignment will be necessary to preserve the semantics of the original program. We have developed a method to statically determine which iteration will assign the last value. Last value assignment can also be resolved at run time using iteration number tag.

The algorithm can generalized to deal with subroutine calls. All what is needed is to substitute subroutine body for loop iteration in the algorithm, summary information is computed for each subroutine. An analysis of the subroutines in their inverse calling order together with mappings from formal parameters to actual parameters and global variables give us the abilities to determine privatizable variables in loops with subroutine calls. We have implemented the generalized algorithm and it is used in the next section to evaluate effectiveness.

2.2 Effectiveness Evaluation

To evaluate the effectiveness of the algorithm, we implemented the algorithm using the Delta Program Manipulation System [Pad89]. The evaluations are made by comparing a program's optimal *loop level* parallel execution times with and without array privatization. Speedups are computed for each program with array privatization and without array privatization assuming loop level parallelism. An upper bound of the optimal loop level speedup for a program is also computed by ignoring all the anti-dependences. In all the calculations, we use a strategy introduced by Kumar [Kum88] to measure a program's execution time on an ideal machine where only the arithmetic operations consume time and an unlimited number of processors are available.

A program is instrumented such that run-time reference information about memory locations is used to detect the *flow* dependences and *anti*-dependences [Che89][PP92]. The optimal parallel execution time ignoring anti-dependences is obtained by calculating the maximum length over all flow dependence chains. The parallel execution time of the program without privatization is the maximum length over all flow dependence or anti-dependence chains. Privatization will break some of the anti-dependences, hence a chain involving anti-dependences may be cut into several shorter chains by privatization. The parallel execution time of the program with privatization is the maximum length over all flow or anti-dependence chains after the privatization breaks some of those chains.

The instrumentation is implemented by introducing a *read shadow variable* and a *write shadow variable* for each program variable. A read shadow variable records the latest time when a variable was read, and a write shadow records the earliest time when a variable was assigned. The earliest time a variable can be assigned is when the lastest read for the variable was finished, all the operands used in calculating new value for the variable were written and the computation for the new value was finished. To measure loop level parallelism, control dependences are added from every statement instance to its successors in the same iteration, which ensures the sequential execution of statements in the same iteration. A read shadow for a private variable is created locally for a loop iteration, different iterations have different shadows for a private variable, hence the anti-dependences are broken across iterations. Optimal execution time is computed by ignoring all the read shadow variables.

Six programs in the Perfect Benchmark Club [CKPK90] were instrumented. The loop level speedup results are reported in the table below. The results without privatization show that memory related dependences can severely limit parallelism in programs. After array privatization, we get speedups within for five cases within a factor of two of the optimal speedups. The differences between optimal speedups and privatization speedups are due to several factors: (1) Our algorithm is effective when subscript expressions are functions of loop indices. Its effectiveness is limited by the system's ability to forward substitute induction variables and to propagate constants. Another difficulty is when array references are used in subscript expressions. (2) Although we can handle symbolic loop boundaries, there are cases where two symbolic boundaries have the same value but we cannot establish that they are equivalent. (3) There are also cases where the privatizability of arrays depends on some conditions. We are currently conducting experiments on other programs in the Perfect Club. It is our intention that by testing more applications, we can identify difficulties and new techniques to improve the performance of our algorithm.

Program	Optimum	With Privatization	Without Privatization
MDG	7.8	6.0	2.2
TRACK	5.0	2.3	1.5
TRFD	547.8	536.7	1.0
FLO52Q	219.6	193.6	3.8
OCEAN	621.4	11.1	1.5
QCD2	2.4	1.6	1.6

Table 1. Loop Level Speedup Ratios

3 Data Distribution

Data distribution determines the layout of data in the system to minimize communication and to maximize parallelism. Data distribution can be divided into two phases: *data alignment* and *data decomposition*. Data alignment determines the relative position of different arrays such that array elements involved in the same computation are often in the same memory module to increase local access. Data decomposition determines the partition of arrays to explore parallelism and limit communication.

There are two classes of data object in the SPMD programming model: *private variables* and *shared variables*. Private variables are replicated on all the processors. Each processor allocates private variables in its local storage. Private variables with the same name may have different values on different processors. Shared variables are accessible to all the processors. Shared arrays can be distributed across the processors. Only one copy of each array element exists in the whole system. Arrays can be distributed dimensionally where each dimension being distributed independent of all other dimensions. A dimension is called *fully parallel* if different column in that dimension is allocated on a different processor. It is called *sequential* if all the columns in that dimension is allocated on a single processor.

When there is parallel computation on a dimension of a shared array, the dimension generally shall be distributed in parallel such that different processors can work on different columns allocated in their local memory in parallel. When there is only sequential computation on a dimension of a shared array, the dimension should be allocated on a single processor such that communication and synchronization can be minimized. Hence, the distribution of the shared arrays and the sharing of the parallel work have to be treated integrately to achieve parallelism and locality of reference.

In this section, we will present some techniques that will be useful in this work. No algorithm exists yet. Since the general problem is NP complete, our objective is to find some good heuristics to exploit parallelism and reduce communication. We first discuss data alignment. We introduce a *placement matrix* to represent how the array is distributed in a virtual processor array and give an equation to compute the placement matrix and hence the distribution. The objective is to find a placement such that all the elements accessed in an iteration of a parallel loop are located on the same virtual processor. Then we will discuss data decomposition when communication is necessary. The objective it to find a data decomposition which can fully use the parallelism on a target machine and the program while reducing the ratio of communication to computation.

3.1 Data Alignment

Several factors determine data alignment. There are *orientation*, *offset*, and *stride*. Consider the following loop:

```
DO I = 1, N
   DO J = 1, N
      A(I,J) = B(J,I)
   END DO
END DO
```

To have local access of both $A(I,J)$ and $B(I,J)$ for each iteration, array A and B shall be transposed placed with each other. That is, the first dimension of B shall be oriented with the second dimension of A, and the second dimension of B shall be oriented with the first dimension of A. Let *(Tr)(B)* be a transposely placed array B, then the following loop is equivalent to the above one but has all the elements aligned.

```
DO I = 1, N
   DO J = 1, N
      A(I,J)  = (Tr)(B)(I,J)
   END DO
END DO
```

Note that *(Tr)* is a notation for the original placement of B, it is not a run time operation to transpose array B.

3.1.1 Array Occurrence and Array Placement Matrix

Array alignment depends on the access pattern of the arrays in the computation. Matrix representation of array occurrence is a convenient way to represent the array references in the program. For instance:

$$(I,J) = (I,J)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad (J+1,I) = (I,J)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + (1,0)$$

Note that a subscript matrix consists of two part. The first part is a matrix product reflecting that the subscript in each dimension can be a linear combination of loop index variables. The second part is a vector reflecting the offset along each dimension. We will use (C,S) to denote an array occurrence with *combination matrix* as C and *shifting vector* as S.

For an n-dimensional loop $(I_1, I_2, \ldots, I_n)$, one can define a one-to-one mapping from the iteration space to an n-dimensional virtual processor network with exactly one iteration per processor. Array placement in the virtual processor network can also be represented in a matrix form. Placement matrix specifys how an array is placed in a multi-dimensional virtual processor network. For instance, we can place array B transposed in the virtual processor network using the placement matrix

$$O_B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

To calculate virtual address for an array occurrence, we multiply the occurrence matrix with its placement matrix. For an occurrence like $B(J,I)$, the virtual address can be calculated as

$$(J,I) * O_B = (I,J)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = (I,J)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Displacement in placement can also be represented by an displacement vector. To bring the array reference $A(J+1,I)$

$$(J+1,I) = (I,J)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + (1,0)$$

into alignment with loop index (I,J), we need a transpose O and a D displacement as:

$$O = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad D = (0,-1)$$

The virtual processor address is then:

$$(I,J)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + (1,0)\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + (0,-1) = (I,J)\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

We will use (O, D) to denote a placement with *orientation matrix* as O and *displacement vector* as D. To compute the virtual processor address for array occurrence X, we can use

$$V(X) = X * O + D$$

In general, for array occurrence (C, S),

$$X = \mathcal{I} * C + S$$

where $\mathcal{I} = (I_1, I_2, \ldots, I_n)$ is the loop index vector, its virtual processor address can be computed as

$$(\mathcal{I} * C + S) * O + D$$

Now, we can define a pair of array occurrences are *perfectly aligned* if and only if their virtual processor address are the same. That is

$$\mathcal{I} * C_1 * O_1 + S_1 * O_1 + D_1 = \mathcal{I} * C_2 * O_2 + S_2 * O_2 + D_2$$

This equation can be solved separately as:

$$C_1 * O_1 = C_2 * O_2$$

$$S_1 * O_1 + D_1 = S_2 * O_2 + D_2$$

We will call the above equations as *alignment equation*. In the example at the beginning of this section, we have

$$C_A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \; C_B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$S_A = (0, 0) \;\; S_B = (0, 0)$$

The solution for the equations are:

$$O_A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} O_B \;\; D_A = D_B$$

Hence for the references to A and B to be perfectly aligned, the orientation of A and B shall be transposely placed and the displacement of A and B shall be the same.

If there is no solutions to the alignment equations, there is no perfect alignment for the array occurrences. In the following program,

```
DO I = 1, N
   DO J = 1, N
      DO K = 1, N
         A(I,K)  = B(K,J)
      END DO
   END DO
END DO
```

If A is placed to align with the virtual processors.

$$O_A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad C_A = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$$

$$O_B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \quad C_B = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Solve the alignment equation

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

we have

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ c & d \\ a & b \end{pmatrix}$$

Since the first row on both sides of the equation are all constants and they are not equal, there is no solution to this equation. There is no perfect alignment for the array A and B.

The alignment equation can also be used to determine when a dimension of an array shall be *sequentialized* or *replicated*. For instance, if the placement of both array A and B is not determined, let them be:

$$O_A = \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \quad O_B = \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix}$$

Solve the alignment equation:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix}$$

we have

$$\begin{pmatrix} a_1 & b_1 \\ 0 & 0 \\ c_1 & d_1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ c_2 & d_2 \\ a_2 & b_2 \end{pmatrix}$$

i.e.

$$O_A = \begin{pmatrix} 0 & 0 \\ c_1 & d_1 \end{pmatrix} \quad O_B = \begin{pmatrix} c_1 & d_1 \\ 0 & 0 \end{pmatrix}$$

It says the first dimension of A and the second dimension of B shall be sequentialized since in the placement matrices the corresponding rows are zero vectors. A zero vector in a placement matrix can be interpreted as either mapping (sequentializing) all columns of the corresponding dimension to the single column 0 of the virtual processor arrays or mapping (replicating) all columns of the corresponding dimension in all the columns of the virtual processor arrays. The solution also specifys that the second dimension of A shall be aligned in the same way as the first dimension of B. In this case a pair of array occurrences are *partially aligned* when some projections of their virtual processor address are equal.

In the cases of partial alignment, the aligned dimensions shall be distributed in parallel and the sequentialized dimensions can be either sequentialized or replicated. As in our example, the dimension 2 of A and the dimension 1 of B shall be distributed in parallel and aligned so the loop K can be executed in parallel. The dimension 1 of A can be sequentialized and hence the dimension 2 of B and only the parallelism of loop K is explored. To explore the parallelism of loop I, the dimension 1 of A shall be distributed in parallel, then the dimension 2 of B will have to be replicated.

We end this section with an example shows that the placement matrix's application in deriving communication free data decompositions. This work is parallel to the work of Ramanujam and Sadayappan [RS91]. We solve the problem here with the same framework discussed above.

```
DO I = 1, 1000
   DO J = 1, 1000
      X(I,J) = Y(I,J-1) + Y(I,J)
   END DO
END DO
```

In this loop, we have two references to Y which are conflict. Our objective is to find a communication free decomposition while sacrificing some parallelism. The alignment equations are:

$$C_X * O_X = C_{Y_1} * O_Y$$

$$C_X * O_X = C_{Y_2} * O_Y$$

$$S_X * O_X + D_X = S_{Y_1} * O_Y + D_Y$$

$$S_X * O_X + D_X = S_{Y_2} * O_Y + D_Y$$

Solve the first two equations and equal the right hand side of the last two equations, we have

$$O_X = O_Y$$

$$S_{Y_1} * O_Y = S_{Y_2} * O_Y$$

i.e.

$$(0, -1) \begin{pmatrix} a & b \\ c & d \end{pmatrix} = (0, 0) \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

The solution for it is

$$c = 0, d = 0$$

Hence we shall sequentialize the second dimension of A and B. The communication free decomposition is to make the first dimension of A and B parallel distributed and their second dimension sequentialized. Note that communication free decomposition may not fully use the parallelism of the target machine when the number of parallel iterations is smaller than the number of processors, we will propose another criterion to evaluate it in a later section.

3.1.2 Spreading and Blocking in Aligned Dimension

In the previous section, we discussed using alignment equations to compute placement matrixes in order to achieve perfect alignment or partial alignment. Once the aligned dimension is determined, one has to determine how the aligned dimensions shall be placed. In the following loop,

```
DO I = 1, N
   DO J=1, N
      A(I,J)=B(J,2*I)
   END DO
END DO
```

The placement matrixes for A and B must satisfy the alignment equations for them to be aligned, i.e.

$$O_A = \begin{pmatrix} 0 & 1 \\ 2 & 0 \end{pmatrix} O_B$$

Hence, the stride of the first dimension of A shall be the double of the strides of B, i.e., if $B(J, I)$ is mapped to address (J, I), then $A(I, J)$ shall be mapped to address $(J, 2 * I)$. Or, if we blocking the B and map both $B(J, 2 * I)$ and $B(J, 2 * I - 1)$ to address (J, I), then $A(I, J)$ can also be mapped to (J, I). We call the first approach *spreading* and the second approach *blocking*. Spreading has the advantage that all the column are distributed while blocking will grouping some column onto same processor. The disadvantage of spreading is that the array allocation is not continuous which may result in load imbalance in the later phase when mapping the data into limited number of physical processors.

When using blocking, we would like to have the block size as small as possible. For example, if we have $B(J, 6 * I)$ and $A(4 * I, J)$. We can blocking the B by size 6 and A by size 4, and map both $B(J, 6 * I)$ and $A(4 * I, J)$ to address (J, I). But we can also blocking the B by size 3 and A by size 2 and map both $B(J, 3 * I)$ and $A(2 * I, J)$ to (J, I), and hence $B(J, 6 * I)$ and $A(4 * I, J)$ to $(J, 2 * I)$. The second approach results in more parallel blocks. In general, if we have $a * I$ and $b * I$, then the minimum block sizes for them are $a/GCD(a, b)$ and $b/GCD(a, b)$. These results can be easily generalized to cases with more than two arrays.

The blocking scheme can be viewed as creating a new dimension out of the original array. For example, $A(2 * I)$ can be treated as $A'(1, I)$, and $A(2 * I - 1)$ as $A'(0, I)$, where $A'(0 : 1, 1 : N/2)$ is an equivalence of $A(1 : N)$. Using this technique, we can make adjustments to the shifting matrixes of the array occurrences.

Since there can be multiple occurrences for an array in a program, placement matrixes in different occurrences can be different. To resolve the conflict placement requirement, machine characteristics must be taken into account. This leads to the topic of our next section on data decomposition.

3.2 Data Decomposition and Communication

The quality of a particular data distribution depends on the communication requirements of computations on the shared data. We will study the communication property of data decomposition in parallel loops since the loop level parallelism provides much opportunity for exploring program parallelism.

A data decomposition is *parallel communication free* in a loop, if each iteration of the loop can be executed in parallel and all the data elements accessed in each iteration are located in the same processor (fully aligned). For instance, the following loop has a communication free fully parallel data decomposition,

```
DO I = 1, 1000
   X(I) = Y(I) + Z(I)
END DO
```

if we map $X(I), Y(I), Z(I)$ to processor I. The loop can be executed in parallel and the computation in each iteration involves only local memory access. Parallel communication free data decomposition when achievable, can provide maximum parallelism with no communication overhead. It is the most desirable case in NUMA multiprocessors, since it has excellent scalability. With 10 processors, the above loop takes 100 units of time, with 1000 processors, it takes 1 unit of time. We also call it *parallel fully scalable* decomposition.

Some programs do not have parallel fully scalable data decomposition. In the loop,

```
DO I = 1, 1000
   X(I) = Y(I-1) + Y(I)
END DO
```

there are conflict placement requirement for array Y, and hence there is no communication free parallel decomposition for array Y. Requiring communication free will result in both array be sequentialized and the loop be sequentially executed. If we fully distribute both arrays, the communication overhead will be a dominate factor in its execution time.

An important measurement for communication overhead of a NUMA multiprocessor is the ratio of the communication rate for non-local access to the rate of local computation:

$$M_c = \frac{Communication\ Rate}{Computation\ Rate}$$

In most of current NUMA systems, M_c ranges from 0.001 to 0.1. It means the system's ability to deliver non-local data is ten to thousand times slower than local computation. For discussions in this section, we assume our target machine has $M_c = 0.1$. Most current machines have DMA ability to perform communication and computation simultaneously, communication latency can be hidden if there is enough local computation to execute while waiting for the non-local data.

Each loop has its own computation and communication characteristics. We define it as the ratio of the number of communication to the number of operations in one iteration of the loop:

$$L_c = \frac{Number\ of\ Communication}{Number\ of\ Operation}$$

It characterizes average communication requirement per local computation when each processor gets one iteration of the loop. In the above example, $L_c = 1.0$, i.e., average one communication per local computation.

Since $L_c > M_c$ in the above loop, it is *communication bounded*. The execution time will be restricted by communication overhead if both arrays are fully parallel distributed since the communication system cannot keep up with the remote data demand.

In the worst case, simply distribute X, Y and share the communication bounded loop may actually slow down the loop due to communication overhead. To achieve balance between parallelism and communication, we shall change the partition of the task to reduce L_c In the above loop, it can be done by increase the grain size of the task. Task grain size can be increased by blocking or stripmining X, Y, i.e., by allocating more than one iteration to one processor. Parallelism will suffer by a factor. But since the communication per processor in this case doesn't increase, the execution time will be the same as using all the available processors assuming communication and computation can be overlapped.

If we choose block size B, we have

$$L_c = \frac{1}{B}$$

The balance between communication and parallelism can be achieved when $L_c = M_c$, i.e. $B = 10$ in the loop. Since the communication per computation in the above loop can be reduced by a factor of B with block size of B, we call the data decomposition *parallel B scalable.* The parallelism in a parallel B scalable loop is decreased by a factor of B. When B is greater than the array size, the whole array shall be sequentialized and the loop be sequentially executed. Or the whole array be replicated and the loop be executed redundantly.

When $L_c < M_c$, the loop is *computation bounded.* There is an opportunity to reduce the grain size and exploit more parallelism.

It is not always possible to decide at compiler time if a loop has parallel scalable decomposition. In the following loop,

```
DO I = 1, N
   X(I) = Y(A(I)) + Y(I)
END DO
```

one can not find a decomposition for Y without knowing the value of A to have balanced communication and computation. In this case, we call it *parallel* ∞ *scalable.* Parallel execution is possible but the communication overhead is unpredicatable, or we can call it unscalable.

The above discussion can be generalized to nested loops and multidimensional arrays.

```
DO I = 1, 1000
   DO J = 1, 1000
      X(I,J) = Y(I,J) + Z(I,J)
   END DO
END DO
```

The above loop is two dimensional parallel fully scalable. The next loop is parallel fully scalable along I and parallel B scalable along J.

```
DO I = 1, 1000
   DO J = 1, 1000
      X(I,J) = Y(I,J-1) + Y(I,J)
   END DO
END DO
```

If the system has no more than 1000 processors, then the optimal decomposition for the above loop is parallel distribution of arrays along I dimension and sequentialization of the arrays along J. Since it fully utilizes all the resources and is communication free. There is no need to distribute the arrays along the J dimension.

It is not always possible to find a parallel fully scalable projection. The following program is not parallel communication free for either dimension:

```
DO I = 1, 1000
   DO J = 1, 1000
      X(I,J) = (Y(I,J-1) + Y(I,J+1) + Y(I+1,J) + Y(I-1,J))/4.0
   END DO
END DO
```

The L_c for this loop is 1.0, since each iteration access 4 non-local data and has 4 computation. By blocking along one dimension, be it `I` or `J`:

$$L_c = \frac{2 + 2 * B}{4 * B} = \frac{1}{2} + \frac{1}{2 * B}$$

Hence L_c is greater than 0.5 for any block size, it is ∞ *scalable* in either dimension. But if it is simultaneously blocked in both dimension with same block size B:

$$L_c = \frac{4 * B}{4 * B * B} = \frac{1}{B}$$

It is B^2*scalable* by blocking in both dimensions. In this case, since communication per processor increases to $4 * B$, the execution time will be 3 times longer than fully distribute the loop. When the number of processors in the system are more than the number of blocks, they are potential candidates for breaking up as we will discuss in the next section.

In the situation where there is no communication free decomposition and the communication is not scalable, the array may have to be sequentialized in one processor and the loop be executed sequentially on that processor. Or the array be replicated across all the processors and the loop can be executed in parallel. The choice will depend on whether it is possible to pre-fetch the array to all the processors.

The feasible block size B for scalable communication can be used to determine when a dimension shall be sequentialized. When the block size is greater than the array size in that dimension or the loop span, the array shall be sequentialized and the loop shall be executed sequentially in that dimension of the iteration space.

Privatizable scalar and array are communication free in the loop where they are private. They can be partitioned and distributed communication free in those loops.

Previous studies for efficient solution of partial differential equations on multiple processor systems noted that the efficiency of parallel algorithm is not determined by the amount of communication but the ratio of communication to computation [FO84]. As pointed out by Reed *etal.*[RAP87]: the stencil type, partition shape and machine organization all affect the performance of resulting parallel computation. In our studies, we try to apply the ratio of communication to computation to a broad class of problems. We limit the partition shape to rectangle since it is easier to partition the iteration space into rectangles.

3.2.1 Evaluation of Data Decomposition

Different feasible data decompositions can be compared by the degree of parallelism they can preserve. We can use the number of iterations in a nest loop to represent the potential parallelism in the loop. We will use $P = l_1 * l_2 * \ldots * l_n$ to represent the number of iterations in a nested loop $(l_1, l_2, \ldots, l_n)$, where l_i is the number of iterations for the ith loop.

For a data decomposition, we define it scaling factor along the ith dimension of the loop nest as,

$$s_i = \begin{cases} 1 & \text{if communication free} \\ B & \text{if } B \text{ scalable} \\ \infty & \text{if } \infty \text{ scalable} \end{cases}$$

The preserved parallelism for a decomposition can be computed as,

$$\max(l_1/s_1, 1) * \max(l_2/s_2, 1) * \ldots * \max(l_n/s_n, 1)$$

The above formula can easily be generalized to accommodate data decompositions which need simultaneously blocking of two or more dimensions.

Data distribution can be evaluated statement by statement inside a loop. For a specific target machine, we can use the formula to derive a data distribution which preserves enough parallelism to be running on that machine. When the preserved parallelism is smaller than the number of processors in the system, reducing the block size may be useful to create more parallelism. As we seem before, in the case of B scalable when the amount of communication per processor doesn't increase with blocking, reducing the block size won't reduce execution time. Hence, the candidates for breaking up are those that can reduce communication per processor, for instance, the B^2 scalable dimensions. We can also determine the level of parallelism preserved by most data decomposition and reserve the same amount of resource from the target machine.

The decisions made on different loop can then be combined or altered to achieve satisfactory parallelism level for the whole program. Data redistribution and data replication can be introduced to resolve conflict data distribution requirement when more parallelism is needed. Then we need to quantify the computation and communication requirements for data replication. The success of data redistribution and data replication in a large extent is determined by whether the communication latency can be hidden by prefetching or pre-distributing, which in turn is influenced by the control flow of the program. We will investigate the control flow problem in our future work.

3.3 Conclusion and Proposed Research

In this proposal, we have considered the problem of shared data distribution for NUMA machines. We have described two approaches for this task. One approach is privatization which trys the reduce the amount of shared data. I have implemented a module in DELTA system to identify privatizable array in the context of loop and procedures and conducted preliminary analysis on its effectness. Another approach is through proper data alignment and data decomposition. We have proposed the placement matrix and placement equations for deriving communication free data distribution. To map the parallelism of a program to the parallelism of a target machine, we proposed balancing communication and computation as an objective for data decomposition. We propose to use preserved parallelism for different data distribution to quantify the effect of different data partition schemes. Using the preserved parallelism as a measurement, we can compare different data distributions and find a combination of array distributions that preserves as much parallelism as the the target machine can provide.

For my Ph.D work, in the array privatization part, I plan to complete the work on array privatization by implementing the module for static last value assignment, and evaluate its effectiveness on the programs in the Perfect Benchmark Club. For the data distribution part, the task is to apply the placement equations and data decomposition scheme to real applications. Since different loops in the same program may place conflict data distribution requirement, we must derive a way to make compromise among conflict requirement. At this time, we do not have a clear solution to this problem. I shall focus on solving this problem during my Ph.D work. We will investigate the feasibility of using heuristics for deriving a global optimal distribution. We will also investigate the effect of using local optimal distribution for each loop and using data redistribution when the data distribution is different from one loop to other loops. A mechanism to evaluate the effectiveness of this work similar to that described for array privatization will be implemented.

References

[Bal91] V. Balasundaram. Translating control parallelism to data parallelism. In *Proc. 5th SIAM Conf. on Parallel Processing for Scientific Computing*, 1991.

[BCFH89] M. Burke, R. Cytron, J. Ferrante, and W. Hsieh. Automatic generation of nested, fork-join parallelism. *Journal of Supercomputing*, pages 71–88, 1989.

[Che89] Ding-Kai Chen. MAXPAR: An execution driven simulator for studying parallel systems. MS thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, October 1989. CSRD Report 917.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks. In *Proceedings of ICS, Amsterdam, Netherlands*, pages 162–174, March 1990.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In *Proc. 4-th Workshop on Programming Languages and Compilers for Parallel Computing*. Pitman/MIT Press, August 1991.

[Fea88] P. Feautrier. Array expansion. In *Proc. 1988 ACM Int'l Conf. on Supercomputing*, July 1988.

[FO84] G. C. Fox and S. W. Otto. Algorithms for concurrent processors. *Phys. Today*, 37:50–59, May 1984.

[GB92] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[HKT91] S. Hiranandani, K. Kennedy, and Ch.-W. Tseng. Compiler support for machine-independent parallel programming in Fortran D. Technical Report Rice COMP TR91-149, Department of Computer Science, Rice University, January 1991.

[Kum88] M. Kumar. Measuring parallelism in computation-intensive science/engineering applications. *IEEE Transactions on Computers*, 37(9):5–40, 1988.

[LC91] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

[Pad89] David A. Padua. The Delta Program Manipulation system — Preliminary design. CSRD Report 808, University of Illinois at Urbana-Champaign, Center for Supercomp. R&D, June 1989.

[PP92] Paul Petersen and David Padua. Machine-Independent Evaluation of Parallelizing Compilers. In *Advanced Compilation Techniques for Novel Architectures*, January 1992.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[RAP87] Daniel A. Reed, Loyce M. Adams, and Merrell L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. on Computers*, 36(7):845–858, July 1987.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. the SIGPLAN '89 Conference on Program Language Design and Implementation*, June 1989.

[RS91] J. Ramanujam and P. Sadayappan. Compiler-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[ZBG88] H. Zima, H.-J. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.