SUPERB Support for Irregular Scientific Computations *

Peter Brezany[1] Michael Gerndt[2] Viera Sipkova[1] Hans P. Zima[1]

[1]Department of Statistics and Computer Science, University of Vienna
A-1210 Vienna, Austria

[2]Zentralinstitut fuer Angewandte Mathematik, Forschungszentrum Juelich (KFA)
D-5170 Juelich, West Germany

Abstract

Runtime support for parallelization of scientific programs is needed when some information important for decisions in this process cannot be accurately derived at compile time. This paper describes a project which integrates runtime parallelization with advanced compile-time parallelization techniques of SUPERB. Besides the description of implementation techniques, language constructs are proposed, providing means for the specification of irregular computations. SUPERB is an interactive SIMD/MIMD parallelizing system for the SUPRENUM, iPSC/860 and GENESIS-P machines. The implementation of the runtime parallelization is based on the Parti procedures developed at ICASE NASA.

1 Introduction

SUPERB (SUprenum ParallelizER Bonn) ([3, 17]) is a semi-automatic parallelization tool for distributed memory multiprocessors, e.g. SUPRENUM, iPSC/860 and GENESIS-P. It is a source-to-source transformation system which translates Fortran 77 programs into parallel programs written in the Fortran dialect available on the target machine.

In SUPERB parallelization is guided by a user defined data partition, specifying a set of processors, a set of distributed arrays and their individual distributions. Distribution of work results from this specification by the owner-computes-rule, i.e. a process executes all assignments to array elements mapped to it.

Accesses to non-local array elements are implemented via interprocess communication. In SUPERB the overlap concept is used to describe non-local variables accessed by a processor. The overlap area of a process are all non-local elements in an area around the rectangular section assigned to that process.

The overlap concept simplifies storage allocation as well as the optimization of the communication between processors. SUPERB performs an interprocedural analysis to determine the maximum overlap area for each distributed array in the program. This information is used to statically allocate storage for copies of non-local data.

The overlap concept is especially tailored to efficiently handle programs with local computations adhering to a regular pattern. For such programs, the set of non-local variables of a process can be described by a small overlap area around its local segment.

However, the overlap concept cannot adequately handle computations with irregular accesses as they arise in sparse or unstructured problems, for example. Here, subscript functions often depend on data available at runtime only. Because of the dependence on runtime data, worst case compile-time assumptions must be made by SUPERB in most of the cases mentioned above when determining an overlap description. This results in the allocation of memory for any potentially non-local variable and additional overhead for the resulting communication, part of which may be superfluous.

To effectively exploit distributed memory systems for irregular computations, techniques for runtime parallelization ([6, 7, 12, 13, 14]) have been developed. Besides these implementation techniques, languages have been designed, providing means for the specification of irregular distributions and to support the efficient compilation of codes from sparse or unstructured applications [2, 1, 9].

This paper describes a project which is an experiment in integrating these techniques with the advanced compile-time parallelization techniques of SUPERB. In Section 2 related approaches are presented. Section 3 presents an overview of the integrated parallelization approach in SUPERB. Section 4 introduces forall loops which are the target of the runtime techniques presented in Sections 5 and 6.

2 Related Work

There are several other research groups working on runtime parallelization for distributed memory multiprocessors. In our approach compile time techniques, such as interprocedural distribution analysis, are combined with runtime parallelization. We use the Parti routines developed at ICASE NASE by Joel Saltz and coworkers to implement runtime analysis and support parallel execution.

*The work described in this paper is being carried out as a part of the ESPRIT project "An Automatic Parallelization System for Genesis" funded by the Austrian Ministry for Science and Research (BMWF) and the research project "High-level Programming Support for Parallel Systems" funded by the Austrian Science Foundation (FWF). The authors assume all responsibility for the contents of the paper.

On top of Parti, Joel Saltz and coworkers developed a compiler for a language called ARF (ARguably Fortran). The compiler generates automatically calls to the Parti routines from distribution annotations and *distributed loops* with an *on clause* for work distribution [16].

The concept of processor arrays and distributing data across such arrays was first introduced in the programming language BLAZE [8] in the context of non-uniform access time shared memory machines.

The Kali programming language and compiler [9] allows the user to explicitly control data distribution, parallel execution and load balancing. The user specifies which loops are to be executed in parallel. Kali originated the *on clause* to specify work distribution for parallel loops. The compiler generates explicit communication statements where enough compile time information is available. If access patterns are runtime dependent or array accesses, array distribution, and work distribution do not fit into the compile time analysis, runtime parallelization is generated [5, 4].

At the Technical University of Zürich a compiler called OXYGEN has been developed for the K2 distributed memory multiprocessor [11]. The input language includes distribution annotations for data and loop iterations. The entire analysis of communication among nodes is done at runtime. No compile time techniques are supported.

3 Functional Overview

The various analysis and transformation techniques developed to parallelize and vectorize programs, have been implemented in SUPERB in an interactive environment. The user and system can work as a team to produce good parallel code. During the transformation process, the user is able to inspect the internal information, supply special information to the system and select transformations. For example, it is easy to find inefficiencies via the inspection of the computed overlap information, or to identify parallelization inhibiting factors by analysing data dependences.

Currently, MIMD parallelization in SUPERB is performed in eight steps. The steps are illustrated within a simple example.

Example 1:

```
      program Example
      real a(100), b(400), c(100), z
      integer map(100), index(100)
      read (*,*) b
c     ... initialization of map and index ...
c     ... by some user defined algorithms ...
      do i = 3, 90
            c(i+2) = b(i-2) + b(i+10)
      enddo
      .  .  .
      do i = iexp1, iexp2
            z = c(i+2) * b(i)
            .  .  .
            a(index(i)) = b(index(i)+1) + z
      enddo
      .  .  .
      end
```

Step 1: Program Splitting.
Program splitting transforms the input program into a host and a node program. All I/O statements are collected in the host program, and communication statements for the corresponding value transfers are inserted into both programs. In the resulting code, the host process is loosely synchronised with the node processes. Thus, the host process may read input values before they are actually needed in node processes.
Step 2: Marking Some Do Loops as Forall Loops.
The user has the possibility to mark interactively as **forall** loops those do loops which he wants to be processed by the runtime strategy. After the marking, the second do loop from Example 1 is transformed to a **forall** loop which has the following form:

Example 2:

```
forall i = iexp1, iexp2
      z = c(i+2) * b(i)
      .  .  .
      a(index(i)) = b(index(i)+1) + z
endall
```

Syntax of **forall**s does not introduce specification of the iteration step, because the frond-end of SUPERB transforms all do loops to a normalized form in which the iteration step value equals one.
The marked do loop is only transformed to a **forall** loop if it fullfills semantic requirements which are specified in Section 4. The sticking to these features is checked by services provided by SUPERB, or consulting the user.

Step 3: Data Partitioning.
The user determines individual distributions for selected arrays of the node program. Each distribution characterizes the decomposition of an array into contiguous rectangular parts called segments and the mapping of these segments to the processors. SUPERB provides flexible language means for the specification of arbitrary block sizes and regular mapping. The simplest way is to specify only the number of parts. Thus for the program in Example 1, the data decomposition specification

```
part a(4); part b(4); part c(4)
```

has introduced an implicit declaration of the set $PROCS of four (node) processors accross which the data structures are distributed and which will execute the program. Processors are logically arranged in a one-dimensional array according to the implicit declaration

```
processors $PROCS (0:3)
```

Step 4: Initial Adaptation.
Different kinds of processing are applied to program

parts that are enclosed by **forall** loops and to the rest of the node program.

For the program parts, not enclosed by the **forall** loops, the initial adaptation distributes the entire work assigned to these node program parts across the set of all node processors according to the given array distributions, and resolves accesses to non-local data via communication. The basic rule governing the assignment of work to the node processors is that a node processor is responsible for executing all the assignments to its local data that occur in the original sequential program (owner computes rule). The distribution of work is internally expressed by masks: A mask is a boolean guard that is attached to each statement. A statement is executed iff its mask evaluates to true. The mask of a statement is omitted if it is constant TRUE. After masking has been performed, the node program parts processed by this technique may contain references to non-local objects. For all references which may access a non-local variable a communication statement EXSR (see [3]) is inserted which update a private copy of the variable if necessary.

Various analysis techniques are applied to the **forall** loops. For each **forall** loop, the initial adaptation tries to derive the initial work distribution which we call as **automatic** work distribution. The specification of this distribution appears in the **dist** clause of the header of the **forall** loop (see Section 4). Lists of variables (distributed and undistributed arrays, and scalar variables) occurring in the loop body are constructed for each **forall** loop. These lists can be viewed by the user using information services of SUPERB. In SUPERB, distributed arrays are classified into distribution classes (see 5.1). A global list of distribution classes encountered in the **forall** loops in the whole program is also constructed in the initial adaptation phase. The information about the work distribution derived by the system, and the lists constructed are stored in the internal representation.

Example 3:

```
      program NODE
      real a(100), b(400), c(100), z
      integer map(100), index(100)
      processors $PROCS (0:3)
      receive b
c     ... initialization of map and index
      do i = 3, 90
            EXSR (b(i+10))
            EXSR (b(i-2))
            owned(c(i)) → c(i) = b(i-2) + b(i+10)
      enddo
      . . .
      forall i = iexp1, iexp2 dist on owner
                                    (a(index(i)))
            z = c(i+2) * b(i)
            . . .
            a(index(i)) = b(index(i)+1) + z
      endall
      . . .
      end
```

The **dist on owner** clause determines the computation on the processor where the specified data is stored.

Step 5: Work Distribution Specification.
The user can change the automatic work distribution derived by the system in the initial adaptation phase to any type of work distribution proposed in Section 4, e.g., he or she can prescribe that the iteration i of the **forall** loop will be executed on the processor whose number is stored in the integer array element map(i).

Example 4:

```
      program NODE
      real a(100), b(400), c(100), z
      integer map(100), index(100)
      processors $PROCS (0:3)
      . . .
c     ... initialization of map and index
      . . .
      forall i = iexp1, iexp2 dist on processor
                                    $PROCS(map(i))
            z = c(i+2) * b(i)
            . . .
            a(index(i)) = b(index(i)+1) + z
      endall
      . . .
      end
```

Specifying suitable work distribution, the user can minimize load imbalance for the **forall** loop.

Step 6: Private Variable Specification.
SUPERB **forall** loops can contain variable declarations, which declare scalar variables local to the loop body, with a separate copy of the variable for each loop iterations. The user can mark some scalar variables appearing in the **forall** body to be local. If, e.g., the variable z has been marked, the **forall** loop is transformed to the new form.

Example 5:

```
      program NODE
      real a(100), b(400), c(100), z
      integer map(100), index(100)
      processors $PROCS (0:3)
      . . .
c     ... initialization of map and index
      . . .
      forall i = iexp1, iexp2 dist on processor
                                    $PROCS(map(i))
            real z
            z = c(i+2) * b(i)
            . . .
            a(index(i)) = b(index(i)+1) + z
      endall
      . . .
      end
```

Step 7: Optimization of parts not enclosed by the forall loops.

The code resulting from the initial adaptation is usually not efficient, since the updating is performed via single-element messages and work distribution is enforced at statement level. In the optimisation phase, special transformations are applied to generate more efficient code. First, communication is extracted from surrounding do-loops resulting in the fusion of messages and secondly, loop iterations which do not perform any computation for local variables are suppressed in the node processes. Furthermore, SUPERB determines standard reductions, such as sum, product, maximum and minimum values of vector elements and dotproduct of two vectors, and treats them in an efficient way.

Step 8: Code Generation.
In SUPERB, the program that is being parallelized and all information collected at compile-time are given in an internal representation. In the last step, the back-end adapts the internal representation to the target Fortran language. Then the reconstructor produces files with the FORTRAN code of the host and node programs which can be passed to the sequential FORTRAN compilers which generate object codes for the host and node processors.

If a **forall** loop appears in the program unit processed, the back-end generates new data structures and statements for the runtime processing of this loop. All new constructs are generated on the syntax tree level using lists which describe the utilization of variables occurring in the program statements. These lists are available for every statement and are constructed in the previous parallelization steps.

A runtime processing strategy based on the inspector-executor paradigm is used. Each **forall** loop is evaluated in two phases. The inspector phase generates a description of the communication necessary for the loop. The executor uses this information to perform the communication and the execution of the loop body. Much of the complexity of our implementation is in the Parti procedures.

The most important part of code generated for the **forall** loop from Example 3 can be seen in Example 6. It will be discussed in Sections 5 and 6.

Example 6:

```
      program NODE

      parameter (ICYCL=1, IBLCYC=2,
     &   IBLOCK=3, IOWNER=4, IADRES=5)
      parameter (K1=2, K0=0, ISDCL=1,
     &   NODCL=0)

      real a(25*K1+K0), b(329*K1+K0),
     &   c(25*K1+K2), z

      integer a_left(150), b_right(300), c_right(150),
     &   sb(2,8), ob(2,6)
      integer map(100), index(100)
      integer execp(150), eact, tratab(2),
     &   iters(400), maxdc, block
      integer a_left_count, b_right_count,
     &   c_right_count
      integer a_left_sched, b_right_sched,
     &   c_right_sched

      common /runt/ iters, itnumb, execp, eact,
     &   tratab, maxdc
      common /syst/ sb, ob

      data a_left_count, b_right_count,
     &   c_right_count/0,0,0/
      data maxdc, tratab/2,ISDCL,ISDCL/

c     a, b and c are only distributed
      .  .  .

c     Inspector - generation of translation tables

      call tabgen
      .  .  .
c     ... initialization of map and index
      .  .  .
c     Inspector - work distribution
      (initialization of execp and eact)

      itnumb = iexp2 - iexp1 + 1
      ihelp = iexp1 - 1
      do i = 1, itnumb
         iters(i) = index(ihelp + i)
      enddo
      call local_iters(IOWNER, iexp1)

c     Inspector - precomputing globally indexed
c     references for all distributed arrays
c     occurring in the loop

      do j = 1, eact
         i = execp(j)
         a_left_count = a_left_count + 1
         a_left(a_left_count) = index(i)
         b_right_count = b_right_count + 1
         b_right(b_right_count) = i
         b_right_count = b_right_count + 1
         b_right(b_right_count) = index(i) + 1
         c_right_count = c_right_count + 1
         c_right(c_right_count) = i + 2
      enddo

c     Inspector - generation of schedules for c, b, and a

c        Generation of a schedule for the array c
      call flocalize (tratab(1), c_right_sched, c_right,
     &   c_right_loc, c_right_count, n_off_proc,
     &   sb(1,2)-sb(1,1)+1)
      call checkbuff(n_off_proc, 1, 'small buffer for c')

c        Generation of a schedule for the array b
      call flocalize (tratab(2), b_right_sched, b_right,
     &   b_right_loc, b_right_count, n_off_proc,
     &   sb(2,2)-sb(2,1)+1)
```

```
      call checkbuff(n_off_proc, 2, 'small buffer for b')

c         Generation of a schedule for the array a
      call flocalize (tratab(1), a_left_sched, a_left,
     a     a_left_loc, a_left_count, n_off_proc,
     a     sb(1,2)-sb(1,1)+1)
      call checkbuff(n_off_proc, 1, 'small buffer for a')

c     end of inspector

c     Executor

      call ffgather (c_right_sched, c(sb(1,2)-ob(1,1)+2),
     a     c(sb(1,1)-ob(1,1)+1))

      call ffgather (b_right_sched, b(sb(2,2)-ob(2,1)+2),
     a     b(sb(2,1)-ob(2,1)+1))

      a_left_count = 1
      b_right_count = 1
      c_right_count = 1

      do j = 1, eact
         z = c(c_right_loc(c_right_count)) *
     a       b(b_right_loc(b_right_count))
      .  .  .
         a(a_left_loc(a_left_count)) =
     a       b(b_right_loc(b_right_count+1)) + z

         a_left_count = a_left_count + 1
         b_right_count = b_right_count + 2
         c_right_count = c_right_count + 1
      enddo

      call ffscatter (a_left_sched, a(sb(1,2)-ob(1,1)+2),
     a     a(sb(1,1)-ob(1,1)+1))

c     end of executor

      .  .  .
      end
```

4 SUPERB Forall Loops

To be able to handle runtime dependent access patterns efficiently in SUPERB, we included forall loops in the source code which is subject to the distributed memory compilation techniques. The **forall** loops have been designed such that they resemble closely sequential loops but provide the user with a wide spectrum of possibilities for specification of work distribution.

The semantics of the **forall** loops is the standard semantics of parallel loops. The iterations of the loop have to be independent and thus can be executed in an arbitrary order.

Forall loops have the following structure:

```
forall i=lb,ub [<work distribution>]
       <declaration of local variables>
...
endall
```

Work distribution annotations determine the scheduling strategy of the loop. If no annotation is specified, SUPERB is free to select any pre-scheduling strategy. Self-scheduling is not supported. The work distribution annotation may be one of the following:

1. Cyclic work distribution

   forall i = low, high **dist by cyclic**

   The iteration set is distributed in a round robin fashion across the processors.

2. Block-Cyclic work distribution

   forall i = low, high **dist by cyclic** (iexp)

   The set of blocks of iterations is distributed in a round robin fashion across the processors. The block length is determined by the value of the integer expression *iexp*.

3. Block work distribution

   forall i = low, high **dist by block**

   A contiguous block of iterations is assigned to each processor.

4. Work distribution according to the owner of an array element

   forall i = low, high **dist on owner** (A(f(i)))

   Iteration i of the loop will be executed on the processor which ownes the array element denoted by A(f(i)), where f(i) is a function of the loop variable i and must not depend on the value of any variable changed in the loop body. Furthermore, array A has to be distributed.

5. Addressed work distribution

   forall i = low, high **dist**
   on processor $PROCS (P(i))

   The iteration i of the loop will be executed on the processor whose number is stored in the integer array element P(i). Elements of P are initialized by a user defined algorithm. It is supposed that in the environment of this loop a processor array declaration of the form
   processors $PROCS(0:(n$proc-1))
   exists, where n$proc indicates the number of processors available.

The first three iteration distribution mechanisms are refered to as **uniform** ([10]) because they spread the iterations uniformly across the avaiable processors without regard to data locality. However, it is a low overhead form of work distribution.

Inside a **forall** loop private scalar variables can be declared. Each iteration has an own copy of a private variable.

Forall loops do have the following restrictions:

1. No host/node communication is allowed in **forall** loops.

   These communication statements result from I/O statements in the original program. Execution

of such communication statements is mapped to the node processes according to the ownership of communicated variables. This scheduling may be very different from the scheduling given by the work distribution annotation.

2. Procedure calls are excluded from **forall** loops.

   Work distribution is determined for **forall** loops on the level of individual iterations whereas work distribution inside of subroutines may result from the owner computes rule. Furthermore, all communication in the subroutine would have to be executed before the **forall** loop.

In the current approach **forall** loops are generated from sequential loops by SUPERB. The user marks a loop to be transformed to a parallel loop and SUPERB tests whether this transformation is valid.

SUPERB determines private scalar variables of the loop iterations. Variables are declared as private variables if the following conditions are satisfied:

- The variable is defined in each iteration prior to any read access.
- No value computed inside the loop is used outside.

In general, sequential loops can be transformed to parallel loops iff they do not carry any dependence resulting from accesses to non-private data. In SUPERB we allow additionally carried anti dependences since our implementation ensures that these are satisfied in the parallelized code.

If SUPERB determines parallelization inhibiting dependences these may result from inaccurate information, e.g. this is a typical situation for runtime dependent accesses. Therefore, SUPERB does not in general prohibit the transformation in such cases, but prompts the user to make a decision whether the assumed dependences really exist at runtime.

Work distribution for forall loop is derived automatically by SUPERB in the initial adaptation step. SUPERB tries to determine a work distribution which is dependent on the distributions of arrays in the loop body, i.e. an "on owner" work distribution. It looks for the first occurrence of a distributed array on the left hand side of an assignment statement in the **forall** which fulfills the following restrictions:

1. The assignment is not guarded by a logical if.

   Otherwise, the array reference may not be a valid reference in all iterations and thus cannot be used in the on clause for the specification of the mapping of all iterations.

2. The array subscript expressions do not contain any variable defined in the **forall**.

   This restricition simplifies our implementation since work distribution can be evaluated at runtime without having to execute loop iterations before.

If such an array reference is found, it is used in the **dist** clause, otherwise, cyclic work distribution is applied. The user can change the work distribution provided by the system to any type mentioned above.

5 Inspector Generation

5.1 Construction of Translation Tables

In SUPERB, each distributed array is a member of a corresponding distribution class. Two distributed arrays are members of the same class if they have identical declarations and distributions.

Each element of a distributed array is assigned to a particular processor. In order for a processor to access a given non-local element of the array, it must know the processor in which it is stored, and its local logical address in this processor's memory. Therefore, a translation table is constructed for each distribution class whose members appear in a **forall** loop. The information about distribution classes of arrays occurring in the **forall** loops is collected in the initial adaptation phase.

The Parti procedure ifbuild_translation_table allows to construct a translation table for each distribution class. This translation table is distributed in a very regular manner. In our implementation, we use BLOCK type distribution. Each processor passes ifbuild_translation_table a list of the global indices which are assigned to it according to the given distribution class. At runtime this information is available on each processor in a special data structure. In Example 6, this data structure is denoted by the identifier *sb*, and is allocated in the common area *syst*. The extent of its first dimension equals two because there are used two distribution classes in our program; the distribution class number 1 for arrays *a* and *c*, and the distribution class number 2 for the array *b*. A given processor can obtain information about a specific distributed array or communicate elements of this array using other Parti primitives that consult the translation table of the corresponding distribution class. Pointers to translation tables returned by ifbuild_translation_table are stored in a one-dimensional integer array *tratab* (allocated in the common area *runt*) whose extent, denoted by *maxdc*, is determined by the maximum distribution class number that is used in the **forall** loops.

Currently, all translation tables required are generated in the subroutine *tabgen* by a sequence of ifbuild_translation_table calls at the start of the main program unit. This could cause storage problems in some situations. Therefore, the future optimization of our implementation will focus on the translation table generation and deletion at the **forall** loop level. Then, only translation tables that correspond to *live forall distribution classes* which may be calculated in a manner similar to *live variables* will exist.

SUPERB only supports regular static distributions. Therefore, construction and consulting the translation table might be more efficient by providing special Parti procedures for this kind of data distribution.

5.2 Construction of Local Iteration Sets

In this step each processor p computes a set

$exec(p) = \{i \in <low, high> |\ i \text{ is executed by } p\}$[1]

of its own iterations. Computation of this iteration set is determined by the work distribution specification in

[1] *low* and *high* denote the bounds of the **forall**

the **dist** clause of the **forall** loop that is being processed (see Section 4). The uniform and addressed work distribution specification enable to conctruct *exec(p)* very efficiently, without communication. For this construction, we have implemented functions defined in [18] for the construction of sets of distributed array elements stored on a given processor.

"On owner" work distribution type can often provide a higher degree of locality (references in the iterations are close together) than the uniform type, and is on a higher abstraction level than the addressed type, however, its most general forms (e.g., when indices of the array introduced in the "on owner' clause are functions of distributed arrays) may cause a high overhead due to extensive communications required to construct local iteration sets. For "on owner" type work distribution, our implementation is partly based on techniques proposed in [4].

In our example, the subroutine *local_iters* is used for the computation of the local iteration set which is implemented by the array *execp* and variable *eact*. The value of *eact* denotes the extent of this set. The first parameter of *local_iters* determines the work distribution type used. If the "on owner" work distribution is used, as it is in our case, values of the subscript function of the array in the "on owner" clause, are precomputed and stored in the array *iters* before the call of *local_iters*. These values are precomputed for the whole set of **forall** loop iterations. One clear research goal is the parallelization of this preprocessing itself. The subroutine *local_iters* is also generated by the back-end.

5.3 Generation of Schedules for Offprocessor Accesses

Parti subroutines that are called in the executor phase move data between processors. The work of each subroutine is controlled by a data structure called schedule whose contents specifies the locations in distributed memory from which data is to be acquired. On each processor, schedules are generated by the Parti procedure flocalize. In our example, to generate a schedule for a distributed array a, flocalize is passed:

1. *tratab(1)* - a pointer to the translation table that has been constructed for the distribution class of a.
2. *a_left* - a set of globally indexed references to a. All globally indexed references to a, which might be made by local iterations stored in the array *execp* are precomputed and resulting values are stored in the temporary array *a_left*.
3. *a_left_count* - number of global references.
4. *sb(1,2)-sb(1,1)+1)* - the size of the local data segment of a according to the distribution of a. Computation of this parameter value is determined by the distribution class of a and partitioning information stored in the array *sb*.

To build the globally indexed reference sets, the **forall** body is partially interpreted. Results of the dataflow analysis are used to reduce the overhead needed for this interpretation. If the same distributed array appears on the left and right hand sides of assignment statements and the reference patterns used are different, two reference sets are constructed for this array. The set that corresponds to the left hand side references will be used for the preparation of communication for Parti primitives of an fscatter type, and the other for Parti primitives of an fgather type (see below).

Besides a pointer to the schedule (*a_left_schedule* for the array a), the subroutine flocalize also returns the number of off processor data elements denoted by the variable *n_off_processor*, and a list of integers, *a_left_loc*, which stores the local reference string corresponding to the set of global references, *a_left*, passed to this subroutine. The list *a_left_loc* is used in the executor code.

The number of off processor elements returned by flocalize is used by the subroutine *checkbuff* to check whether a buffer allocated for these elements is large enough. When not, the message passed as the parameter to *checkbuff* is printed and an error state is indicated. The second parameter represents the distribution class of the array whose off processor elements will be stored in the buffer.

6 Executor Generation

The purpose of the executor is to perform the communications specified by schedules generated by flocalize and perform the actual computation prescribed by the **forall** body, however, for only the local iteration set. The semantics of the **forall** guarantees that the communication may be required only before the loop to acquire non-local array elements (this is supported by the Parti exchangers of a gather type), and after the loop to update or combine non-local data elements (this is supported by the Parti exchangers of a scatter type).

On each processor, array segments are declared as described by the following structure
(*left overlap area*; *local data*; *right overlap area*)

At runtime parameters of the segment structure can be obtained from data structures *sb* and *ob* on each processor.

This SUPERB conception has to be preserved, because a distributed array can appear in loops parallelized at compile time and runtime as well. However, the *right ovelap areas* of segments of all distributed arrays that appear in the **forall**s are extended, to allocate the necessary space for off processor data elements. These areas are used by Parti exchangers as buffers for off processor data elements. Therefore the segment size, denoted by *segmentsize*, whose value has been derived by the SUPERB compile time strategy, is extended to the size $K1 * segmentsize + K0$ for the needs of the executor. *K1* and *K0* are parameter constants that can be modified by the user, however, their initial values are chosen by the system. Before a call of an exchanger, the subroutine *checkbuff* is called to check whether the buffer size is large enough.

Each processor passes to an exchanger the starting address of the *local data* subsegment (the third pa-

rameter), the buffer address which is the starting address of the extended *right overlap area* subsegment (the second parameter), and the pointer to the scheduler (the first parameter). Gather type subroutines place copies of data value obtained from *local data* subsegments of other processors in the buffer. Before a call of a scatter exchanger, copies of data values to be scattered to other processors are placed in the buffer. Local and buffer data are referenced using the local reference strings produced by flocalize.

7 Conclusions

The implementation of the SUPERB extension described above has been completed. In this implementation we use the Parti version described in [15].

SUPERB performs an extensive interprocedural distribution analysis to determine which formal arrays are distributed and which distributions may occur at runtime. Due to this analysis, distributed formal arrays can be accessed in the **forall** loops.

After the first performance measurements of the runtime processing implemented had been done, we replaced some Parti procedure calls with calls to other specialized routines, because the SUPERB regular data decomposition strategy creates special conditions for the treatment of irregular computations by the compiler.

There is still much work to be done. Our future research focuses on the optimization of the translation table generation (time versus space), reduction of the number of schedule generations, optimization of precomputation of globally indexed references, and inclusion of parallel input/output statements into the **forall** loops.

Acknowledgement

The authors would like to thank Joel Saltz for his helpful discussions about the Parti procedures and for providing their research group with the Parti library.

References

[1] B.Chapman, P.Mehrotra, H.P.Zima. Vienna FORTRAN - A Fortran Language Extension for Distributed Memory Multiprocessors. Technical Report, ICASE, NASA Langley Research Center, 1991.

[2] G.Fox, S.Hiranadani, K.Kennedy, C.Koelbel, U.Kremer, C.Tseng, M.Wu. Fortran D Language Specification. Rice University, Technical Report COMP TR90-141, December 1990.

[3] H.M. Gerndt. Automatic Parallelization for Distributed-Memory Multiprocessing Systems. *Ph.D. Dissertation*, University of Bonn, 1989.

[4] C. Koelbel. Compiling Programs for Nonshared Memory Machines. *Ph.D. Dissertation*, Purdue University, West Lafayette, IN, November 1990.

[5] C. Koelbel. Compile time Generation of Regular communications Patterns. *Proceedings Supercomputing 91*, Albuquerque, 101-110.

[6] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Sytems*, October 1991.

[7] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 177–186, March 1990.

[8] P. Mehrotra, J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, Vol. 5, 339-361, 1987.

[9] P. Mehrotra, J. Van Rosendale. Programming Distributed Memory Architectures Using Kali. ICASE, Nasa Langley Reserach Center 1990.

[10] D.M. Pase. MPP Fortran Programming Model, Draft 1.0. Technical Report, Cray Research, October 1991.

[11] R. Ruehl, M. Annaratone. Parallelization of Fortran Code on Distributed-Memory Parallel Processors. *Proceedings of the 4th International Conference on Supercomputing 1990*, Amsterdam, 342-353.

[12] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors. Report 90-59, ICASE, 1990.

[13] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.

[14] J. Saltz, R. Das, R. Ponnusamy, D. Mavriplis, H. Berryman and J. Wu. Parti Procedures for Realistic Loops. *Proceedings of DMCC6, Portland OR*, 1991.

[15] R. Das, J. Saltz, H. Berryman. A manual for Parti runtime primitives - revision 1. *Interim Report 91-17*, ICASE, 1991.

[16] J. Wu, Joel Saltz, Harry Berryman, Seema Hiranandani. Distributed Memory Compiler Design For Sparse Problems. ICASE Report No. 91-13, January 1991.

[17] H. Zima, H.-J. Bast, and H.M. Gerndt. SUPERB - a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6, 1-18, 1988.

[18] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. In preparation, Austrian Center for Parallel Computation, University of Vienna, Vienna, Austria, 1992.