Extending High Performance Fortran for the Support of Unstructured Computations

Andreas Müller Roland Rühl

Swiss Scientific Computing Center (CSCS), ETH Zurich
Via Cantonale, CH-6928 Manno, Switzerland
E-mail: {mueller,ruehl}@cscs.ch

Abstract

We have extended an existing HPF compiler with language features designed for the parallelization of unstructured computations on multicomputers. The language extensions include block-general distributions and dynamic data distributions specified through user-defined mapping arrays and functions. A prototype compiler has been implemented which features different run-time preprocessing mechanisms and also allows clean integration of explicit message-passing primitives. The compiler is developed as part of a complete multicomputer programming environment being used by a group of application developers in the framework of the Joint CSCS–ETH/NEC Collaboration in Parallel Processing. As such, it is supported by a high-level debugger and performance monitor, and the usability and efficiency of generated parallel programs is validated by the application developers.

In this paper, we summarize the programming paradigm implemented through HPF extensions, and detail the respective compiler directives. We describe the implemented run-time preprocessing mechanisms and evaluate the efficiency of compiler-generated code on an NEC Cenju-3 multicomputer.

1 Introduction

Recent progress in VLSI and communication technologies enable the construction of large-scale Distributed Memory Parallel Processors (DMPPs or simply *multicomputers*), which offer a better price-performance ratio than traditional shared-memory vector supercomputers. A wide range of applications have already been parallelized on such machines and experience shows that sustained high performance can be achieved provided the underlying system follows some basic architectural constraints, such as balanced communication and computation resources. Various research groups have not only implemented "embarrassingly parallel" programs, but have also parallelized less structured computations like finite-element and irregular mesh based solutions of partial differential equations. Most of these parallel codes have been written using message-passing libraries, similar in functionality to the Message-Passing Interface standard (MPI) [1].

It has also been recognized, however, that the use of message passing is tedious and error-prone. This is particularly true for unstructured computations. In addition, programming tools offered by vendors hardly give the required support for the development of message-passing programs for massively parallel machines. A multicomputer debugger, for instance, would ideally feature race-condition and deadlock-detection mechanisms, as well as the possibility to deterministically replay erroneous message-passing programs.

Several new programming languages have been proposed to simplify program development by providing a global name space along with a single-threaded program image. With these languages, inter-processor communication is handled by the compiler and underlying run-time system transparently to the user. The compiler generates deadlock and race-condition free parallel programs which are easier to develop and easier to debug. Currently, the most promising such language for data-parallel programming is High Performance Fortran (HPF) [2]. The user of HPF can expect portability across a variety of platforms because HPF has been accepted as a language standard by most DMPP vendors.

Experience with the first HPF compilers has shown that high parallel efficiency can only be achieved with regular computations, such as dense linear algebra or image processing. The High Performance Fortran Forum is currently discussing extensions for better support of unstructured computations. The language features considered base on the run-time preprocessing and irregular data distributions featured by some research compilers, such as Arf [3], Fortran D [4], Kali [5] and Oxygen [6].

As part of the *Joint CSCS-ETH/NEC Collaboration in Parallel Processing* [7], we have extended an existing HPF subset compiler (built by NEC Corporation) with language primitives for the support of unstructured computations. The language extensions are on the one hand based on our previous experience with the Oxygen compiler. On the other hand, we have taken into account the requirements of a group of application developers who work on the parallelization of unstructured problems as an integral part of the collaboration (see for instance [8]). Our language extensions include three dynamic data distribution methods, and the compiler features a general mechanism for run-time preprocessing.

After summarizing the background of our work, we describe our language extensions and the above mentioned pre-processing method. Then, we demonstrate the efficiency of generated parallel code for a simple test algorithm and for a full application. Performance numbers were collected on a NEC Cenju-3.

ICS '95 Barcelona, Spain © 1995 ACM 0-89791-726-6/95/0007..$3.50

2 Background

2.1 The Annai Tool Environment

The software described in this report is developed as part of the integrated multicomputer programming environment *Annai* [9]. Since our compiler accepts as input not only extended HPF, but also Fortran and C with message-passing primitives, it serves as the main language processor for Annai, and is generically called the Parallelization Support Tool (PST). Annai also includes a Performance Monitor and Analyzer (PMA), a Parallel Debugging Tool (PDT), and a common graphical User Interface (UI). As a component of Annai, PST follows the environment's general design objectives, while support for both high-level performance monitoring and debugging is also included in PST. More details on PMA and PDT can be found in [10] and [11], respectively. Below, we only summarize Annai's design objectives:

- *Design and implementation of tools for the development of parallel programs in a high-level data-parallel MIMD language and also with low-level message passing.* To ease portability of the tool environment we use standards: HPF at the high level and MPI at the low level. Both levels can be mixed since PST allows integration of message passing into high-level data-parallel code.
- *Parallelization, monitoring, and debugging support for applications considered today difficult to parallelize on DMPPs.* We believe that only extensive support for unstructured problems, such as finite-element based solutions of partial differential equations using irregular meshes, can make DMPPs as versatile as today's common scientific-computing platforms, the shared-memory vector supercomputers. Since HPF lacks suitable language features, we defined a set of HPF extensions and implemented them in PST.
- *Application-driven and user-oriented tool design:* The tools are developed as a sequence of prototypes; a team of application developers use and test these prototypes and provide feedback. This objective is particularly important for PST: the application developers strongly influence PST's development during the design of the HPF language extensions as well as through critical evaluation of the parallel efficiency of compiler-generated code.

2.2 Related Work

As part of the K2 project [12] we developed the Oxygen compiler for DMPPs. The compiler supports a global name space at run-time, through a mechanism called *run-time data consistency analysis*. Oxygen was ported to a variety of platforms, among others the Intel Paragon and iWARP, Fujitsu AP1000 and Parsytec SC256 [13]. We used Oxygen to parallelize PILS [14, 15], a *Package of Iterative Linear Solvers*. That library was applied to systems of equations stemming from two and three-dimensional finite-element based semiconductor device simulations and performance results were collected on Intel Paragon and Fujitsu AP1000.

Our application developers currently use PST to implement PLUMP [8], a *Parallel Library for Unstructured Mesh Problems*. For this class of problems, PLUMP provides a high-level interface to basic linear algebra operations on several different data formats. The library also supports local refinement and dynamic repartitioning of meshes on DMPPs.

From the requirements of our application developers, and from the experience with Oxygen and PILS, we can draw several conclusions for the design of HPF language extensions for unstructured computations like the operations on sparse matrices implemented in PILS and PLUMP:

Replicated Variables and Shared-Memory Semantics HPF enforces sequential semantics on any Fortran statement. That is, unless the user explicitly specifies that two statements are independent (which is only possible in HPF if the statements are surrounded by a loop), any data dependence between two statements in HPF code enforces either execution of the two statements on the same processor, or inter-processor communication. For this purpose, the first HPF compilers we have experience with, from NEC and Applied Parallel Research (APR) replicate sequential code execution and introduce many broadcasts. In contrast to this technique, Oxygen supports a programming paradigm more like typical shared-memory paradigms: by default data are *private* and can only be accessed by *one* processor. Explicit declarations allow sharing of data between processors, and only interprocessor consistency of shared data is ensured.

User-Defined Data Distributions PILS' main data structure (the matrix) is stored in a "colored jagged diagonals" format [16] and no assumption can be made about its structure. PLUMP supports several different data formats and mesh refinement is one of its major features. Typically an application adds elements to a mesh for local refinement until load imbalance becomes large enough to justify the costs of data repartitioning. Local refinement increases the local partitions of the main data structures on only a few processors. For both PILS and PLUMP efficient distributions of matrices and vectors are irregular and depend on run-time information. Therefore they cannot be expressed using HPF `BLOCK` or `CYCLIC` primitives.

Redistribution and Remapping of Dynamically Distributed Data PLUMP's mesh refinement and repartition mechanisms require additional support from the compiler: repartitioning has to be supported by extending HPF's `REDISTRIBUTE` directive to dynamic distributions, and by adding a generic data permutation feature.

User-Defined Loop Distributions In both PILS and PLUMP, the most compute-intensive loops access matrix and vectors indirectly, and compile-time distribution strategies (such as the "owner computes rule") cannot be applied for loop parallelization. Therefore, distribution of loops is best left to the user, and the compiler must support both remote data fetches and updates in the global name space.

Run-Time Data Consistency Analysis Since data distributions can be irregular, communication patterns must be computed at run-time. As is typical for the iterative solvers supported by both PILS and PLUMP, the most compute intensive code segments are *start-time schedulable* [17], i.e. access patterns depend on run-time information but never change. In such cases, run-time global name space maintenance is not expensive, since communication patterns need to be computed only once. In PILS, the same compute-intensive code segments (matrix-vector operations) are applied to different data structures ("colors" of the matrix). Hence, several communication patterns must be saved for the same code segment.

Since detection of run-time schedulability at compile-time is difficult, we leave specification of start-time schedulable code segments to the user.

3 Input Language

PST supplements an NEC HPF compiler by providing an extended input language, and a different underlying programming paradigm and compilation technique. To achieve a clear separation, routines that use PST extensions and rely on PST's programming model have

to be declared `EXTRINSIC(PST_LOCAL)`. The two compilers are integrated by a driving program that selects routines and uses either the HPF compiler or PST for compiling the respective routines. `PST_LOCAL` routines can be called from HPF programs and vice-versa, and data conversion is handled transparently at subroutine boundaries.

Although PST extends HPF, the programming model is slightly different and similar to a shared-memory model: by default data are private, i.e. the compiler does not enforce consistency of non-distributed data across processors. Distributed data are part of the global name space which is supported by run-time analysis. Also code is replicated and, by default, executed by all processors. Where users want only selected processors to execute parts of the code, they must specify that explicitly by using loop distribution directives, or by executing code depending on the processor identifier (that is, by using the Single Program Multiple Data paradigm). Since replicated execution is part of the semantics of PST programs, routines of the underlying message-passing library can be called. PST uses MPI *communicators* to avoid that user-inserted message-passing primitives interfere with compiler-generated code.

By default, a global name space is not enforced, but distributed array elements can be accessed as long as they are allocated locally. The user can specify certain code segments to be *public*, and inside such segments remote data accesses to elements of distributed arrays are supported through data consistency analysis: for public code segments, PST generates a run-time pre-processing phase, called *symbol handler*, and an executions phase, called *executor*. The symbol handler initializes the data transfers necessary to execute statements with references to distributed arrays in the executor.

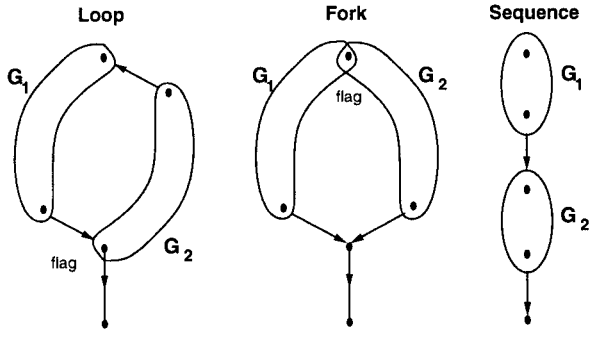

Figure 1: *The three possible ways of decomposing a bin-graph with $n+1$ nodes into two bin-graphs with $\leq n$ nodes.*

We have designed and implemented a more complicated preprocessing method than the so-called *inspector/executor* mechanism [18]. We compile public code segments into *symbol-handler* and executor. Depending on the nesting of distributed-array accesses, the symbol-handler consists of one or more *slices* of the segment (see also Section 4). Such a preprocessing strategy was first implemented and described with Oxygen [12], and has also been investigated recently by other research groups [19, 20].

PST cannot generate symbol handlers for arbitrary input code: a public block's control-flow graph must be a *bin-graph*. A bin-graph can be recursively decomposed into a sequence, branch or loop of smaller bin-graphs as shown in Figure 1. This makes recursive generation of program slices possible.

An executor consists of computational chunks and communication checkpoints in between. Remote data are fetched or updated in the checkpoints and the computational chunks operate on buffers to access remote data. These buffers are loaded by receives in a previous checkpoint for fetches and they are used in a send in the next checkpoint for data updates. To define the ordering of computation chunks, a virtual time stamp (so-called *serial time*) is introduced. By default this time stamp is initialized to zero and increased by one after every checkpoint. However, a directive exists which allows the user to control the serial time. This feature can be used to parallelize loops with data dependencies.

3.1 Directives

3.1.1 Subroutine Specifiers

```
EXTRINSIC(PST_LOCAL) subroutine ...
```

As mentioned above, routines to be compiled by PST instead of the HPF compiler are identified as `EXTRINSIC` routines. Inside `EXTRINSIC(PST_LOCAL)` routines, the PST programming paradigm can be used. The following directives are part of the declarations in subroutine heads:

```
!PST$ SUB_SPEC LOCAL
!PST$ SUB_SPEC PUBLIC { SAVECOM(key) }
```

By default, PST compiles its input to local code, i.e. no symbol handler is created. Code segments that contain remote accesses and therefore require a symbol handler have to be enclosed in separate routines, which are declared `PUBLIC`.

With `SAVECOM(key)`, an `EXTRINSIC(PST_LOCAL)` subroutine can be declared to be start-time schedulable and run-time generated communication patterns are saved and reused in later invocations of the routine. The communication patterns are saved symbolically, i.e., even if different routine arguments are used, the execution will be correct, as long as the arguments have the same *shape* (i.e., same size and distribution) as the arguments used for the routine's first invocation. The integer expression `key` is used for the generation of multiple communication patterns for the same start-time schedulable `EXTRINSIC(PST_LOCAL)` subroutine.

3.1.2 Data Distributions

Arrays can be distributed using all regular HPF distribution and alignment directives. In addition, the user has the choice to distribute an array `var` dynamically with the following three directives:

```
!PST$ DISTRIBUTE var(BLOCK_GENERAL(BGMapArray))
!PST$ DISTRIBUTE var(DYNAMIC(MapArray))
!PST$ DISTRIBUTE var(DYNAMIC(G2L, L2G, G2PE, Sz))
```

The first directive defines a block-general distribution. That is, the distributed array is partitioned into contiguous blocks of possibly different sizes. In contrast to a similar distribution described by Chapman, Mehrotra and Zima [21], PST `BLOCK_GENERAL` distributions permit *gaps*: i.e. extra space is added when such an array is allocated, to provide dynamic support for an increasing number of array elements during program execution. This feature is used in PLUMP to implement mesh-refinement. Array `BGMapArray` contains $2 \times p$ integers (where p is the number of processors) which define the start and size of each processor's block. Global Fortran indices of multi-dimensional arrays are mapped (or "linearized") into a single integer by using reverse lexicographic ordering (for instance column-wise order for two-dimensional arrays).

The second directive defines the distribution of an array via a mapping array (`MapArray`). This array has as many elements as the distributed array, each element defining the processor owning the respective element of `var`. `MapArray` must be allocated and initialized explicitly by the user in the program.

The third directive uses mapping functions as an alternative to mapping arrays, which introduce significant memory overhead. `G2L`, `L2G`, `G2PE` and `Sz` are integer valued functions which

respectively map global to local indices, local to global indices, global indices to processors, and define the size of the local array which may be different on each processor.

3.1.3 Data Redistribution and Remapping

Two features of PST support PLUMP's mesh refinement: data redistribution and data remapping. Data redistribution corresponds to the HPF `REDISTRIBUTE` primitive expanded to irregular distributions. That is, the distribution of an array can be changed by an executable statement to any other distribution allowed by the extended language. Data remapping is unique to PST. Any distributed array can be remapped if the user defines a permutation array of global indices. Consider for instance the following statements:

```
      integer p(m)
      double precision a(m,n)
!HPF$ DISTRIBUTE (BLOCK) :: p
!HPF$ DISTRIBUTE (BLOCK,*) :: a
!PST$ REMAP_1D (a,p,1)
```

The array `a` is remapped along its first dimension using a user-defined permutation array `p`. Effectively `a` is mapped into a new array `a'` with `a'(i) = a(p(i))`. Note that `p` is distributed identically to the dimension of `a` which is to be permuted.[1]

3.1.4 Loop Alignments

```
!PST$ ALIGN L1,L2,... WITH var(i1,i2,...)
      do L1 i1=min1,max1
         do L2 i2=min2,max2
              :
              ... = var(i1,i2,...)
              ... = var(j1,j2,...)
              var(k1,k2,...) = ...
              :
L2       end do
L1    end do
```

Figure 2: *Aligning loops with variables using PST.*

Nested Fortran `DO` loops can be parallelized using the `ALIGN` directive as shown in Figure 2. The semantics of aligned loops are defined such that the access to array element `var(i1,i2,...)` inside the loop nesting (using the loop indices as array indices) is local.

Aligned loops are compiled into new loops with different index minima, maxima, and strides. When aligning with statically distributed HPF arrays (using combinations of `BLOCK` and `CYCLIC`), the problem of computing these minima, maxima, and strides can be reduced to the solution of linear Diophantine equations [22]. Loops aligned with dynamically distributed arrays are transformed into loops over the local index range. The first statement in their body computes the respective global indices using the local to global index mapping.

3.1.5 Checkpoints

The semantics of public code segments depend on the ordering of the computational chunks in the executor, and how the serial time is defined. The compiler generates a checkpoint at the beginning and end of a public code segment as well as before and after every aligned loop. To account for data dependences inside parallel loops, additional checkpoints can be inserted with the following directive:

```
!PST$ CHECKPOINT
```

By default, the serial time is set to zero when a public code segment is entered and incremented by one after every checkpoint. In due course, we plan to add a directive to explicitly set the serial time to any Fortran integer expression.

3.2 An Example Code Segment

We explain some of PST's annotations with one of the basic linear algebra operations supported by PLUMP: the sparse matrix-vector product shown in Figure 3 operates on a sparse matrix data format similar to the ITPACK format [23] and has been parallelized using PST directives.

```
      EXTRINSIC (PST_LOCAL) subroutine matvec(
     &     mx_nrcols, mx_nrrows, cols, mat,
     &     x, y, nrcols, vector_map, mat_map)
!PST$ SUB_SPEC PUBLIC

      integer mx_nrcols, mx_nrrows
      double precision mat(mx_nrrows, mx_nrcols)
      double precision x(mx_nrrows), y(mx_nrrows)
      integer cols(mx_nrrows, mx_nrcols)
      integer nrcols(mx_nrrows), map(*)

!PST$ DISTRIBUTE x(BLOCK_GENERAL(map))
!HPF$ ALIGN WITH x :: y, nrcols

!PST$ DISTRIBUTE mat(BLOCK_GENERAL(map),*)
!HPF$ ALIGN WITH mat :: cols

!PST$ ALIGN 5 WITH y(i)
      do 5 i = 1, mx_nrrows
         y(i) = 0.
 5    end do

!PST$ ALIGN 10 WITH y(i)
      do 10 i = 1, mx_nrrows
         do 20 j = 1, nrcols(i)
            y(i) = y(i) + mat(i,j) * x(cols(i,j))
 20      end do
 10   end do

      end
```

Figure 3: *A sparse matrix-vector multiplication parallelized with PST HPF extensions.*

The matrix is stored in three arrays: `mat` contains the non-zero elements, `cols` the column indices, and `nrcols` the number of non-zeros in each column of the matrix. `x` and `y` are the input and output vectors of the product, respectively.

We assume that for an efficient parallelization of the product, both `x` and `y` need identical distributions. Figure 3 shows how PST is used to assign block-general distributions to the matrix and vectors used in the product. Columns of the matrix are aligned with `x` and `y` to reduce the amount of communication. The index space of the main loop `10` is distributed and aligned with the distributed vectors and the columns of the matrix. As a consequence, accesses to `y` are local while accesses to `x` may be non-local, depending on the structure of the matrix. The allocation *gaps* in the matrix and in the vectors (which reserve space for future mesh refinements) are not visible to the implementor of the parallel matrix-vector product: the distributed iteration space of loop `10` only includes iterations which correspond to defined elements of vector `y`.

[1] Note also, that in the current PST version, data remapping is currently not supported through the above directive but through a similar subroutine call. We plan to add the `REMAP_1D` directive to the PST front-end in the near future.

4 Symbol-Handler Code Generation

The symbol handler preprocesses code segments to determine where non-local data are accessed, and prepares executor data transfers on both data requesting and owning processors. So-called *envelopes* are set up, which are later interpreted in executor checkpoints to perform actual data exchanges. An envelope consists of (1) a logical time stamp which identifies the checkpoint in which the envelope is used, (2) a destination- or source-processor identification, (3) a flag specifying whether remote data are fetched or updated, and (4) an identification of the data item to be communicated. This identification is not an address but *symbolic information*, i.e. an array symbol and a (linearized) array index.

The executor consists of the parallel code as specified by the user (including explicit parallelism in the form of aligned DO loops) but with all references to non-local elements of distributed arrays replaced by references to the *communication cache*. The communication cache buffers data communicated in previous checkpoints (for remote data fetches) or updates of non-local data to be communicated in future checkpoints. All numbers presented in Section 5 were collected using communication caches to avoid the expensive data replication often employed by HPF compilers.

In simple cases, symbol handler code looks similar to executor code with references to distributed data replaced by macros which construct envelopes. However, the contents of an envelope may depend on non-local data itself. Such a dependence is either *explicit*, for instance, when a non-local data reference is enclosed in an `IF` statement with a non-local reference in the controlling boolean expression, or *implicit*, when the index expression in the reference depends on non-local data. In both cases, PST resolves non-local data dependences by generating multiple symbol handler iterations. The first iteration computes envelopes for remote data accesses with no further dependence on non-local data. All statements (and control flow constructs) with nested dependences are ignored. The second iteration uses envelopes computed in the first iteration to resolve first order dependences and compute another level of envelopes. This process continues until envelopes for all non-local references are computed. Such preprocessing is similar to the program *slicing* [24] typically used for static performance prediction. That is, each symbol handler iteration computes envelopes for another *slice* of the executable code, where each slice adds a nesting level of non-local data dependences.

For the computation of non-local data dependences, the data flow in the executable code must be analyzed. This can be done at run time by adding tags (or *guards*) to each data element which store the number of symbol handler iterations required for the complete computation of a respective envelope. Alternatively, a less accurate data-flow analysis can be performed at compile time. Symbol handlers that are generated using compile-time analysis are typically faster and more memory-efficient, but there are also cases where the guard-based run-time mechanism is superior. For instance, for the subroutine in Figure 3 an analyzer which is based on compile-time analysis consists of two iterations because the compiler will assume that the upper bound of `20` (`nrcols(i)`) is non-local. An analyzer which performs guard-based analysis, however, detects that an access to `nrcols(i)` is always local within loop `10` and analyze the whole subroutine including loop `20` in only one iteration.

PST supports both symbol-handling mechanisms and allows the user to choose (with a command line flag) between the more precise guard-based (or *dynamic*) symbol handler and the faster and less memory consuming *static* symbol handler, generated with compile-time data-flow analysis. For the generation of static symbol handlers, PST uses the same algorithms as Oxygen; for a detailed explanation, the interested reader is referred to [6]. In Section 4.1, we describe how dynamic symbol handlers—which are original to PST—are generated. In Section 4.2, we explain how adding of compiler optimizations and a simpler data-flow analysis can lead to a more memory-efficient hybrid preprocessing strategy.

4.1 Generating the Dynamic Symbol Handler

```
Void SYMBOLHANDLER(CFG c)
 1:  Integer iteration;
 2:  Boolean cont := true;
 3:  while cont
 4:    INITIALIZEITERATION();
 5:    cont := PARTIALEXECUTECFG(iteration, c);
 6:    ENDOFITERATION();
 7:  end while
```

Figure 4: SYMBOLHANDLER *repeatedly executes symbol-handler slices.* INITIALIZEITERATION *mainly allocates copies of data referenced in the respective iteration.* ENDOFITERATION *interprets envelopes to initialize actual data transfers in the executor.*

```
Boolean PARTIALEXECUTECFG(Integer iteration, CFG c)
 1:  if c is assignment a
 2:    return PARTIALEXECUTEASGN(iteration, a);
 3:  else
 4:    Symbol flag;
 5:    CFG c1, c2;
 6:    Boolean r1, r2;
 7:    decompose CFG into two sub-graphs c1 and c2,
 8:      as shown in Figure 1.
 9:    if (loop(c1, c2, flag) or fork(c1, c2, flag)) and
10:      GUARD(flag) > iteration
11:      return true;
12:    end if
13:    r1 := PARTIALEXECUTECFG(iteration, c1);
14:    r2 := PARTIALEXECUTECFG(iteration, c2);
15:    return r1 or r2;
16:  end if
```

Figure 5: PARTIALEXECUTECFG *recursively decomposes a control-flow graph into subgraphs. As an example, in Figure 6 we depict how this decomposition is performed for the matrix-vector product of Section 3.2.*

In Figures 4, 5, 7 and 8 we formally describe the structure of the dynamic symbol handler generated by PST. The structure is outlined as a set of algorithms written in a PASCAL-like language. The following data types are used in the description:

CFG: this type describes the control-flow graph of the subroutine, for which a symbol-handler is generated. Note that although algorithms SYMBOLHANDLER and PARTIALEXECUTECFG (Figures 4 and 5) include parameters of type CFG, for any given control-flow, the two algorithms are inline-expanded by PST; the parameters are included in the figures to maintain generality of the description.

Expression: a valid Fortran expression. It may contain arithmetic operators, and side-effect-free function calls.

Assignment: a valid Fortran assignment statement.

Symbol: a variable or a routine parameter specifier.

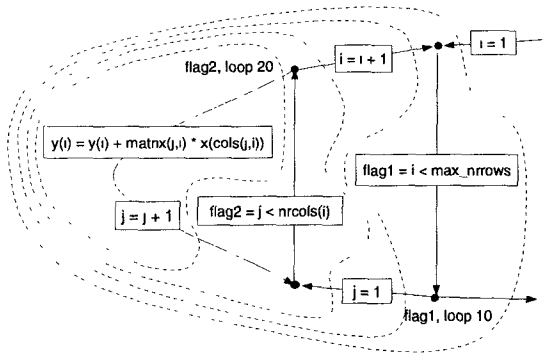

Figure 6: *Control-flow graph of the matrix-vector product of Section 3.2, recursively decomposed into loops, forks and sequences.*

Integer, Boolean: equivalent to the respective PASCAL types.

Algorithm SYMBOLHANDLER in Figure 4 describes the iterative execution of symbol-handler slices in PST-generated code. Each iteration executes statements which depend either on local information, or on non-local data fetched in previous iterations. INITIALIZEITERATION performs initializations required for each symbol-handler iteration: for instance copies of data written in the symbol-handler must be allocated. The main task of ENDOFITERATION ("the *router*") is to route envelopes from data-requesting to data-owning processors. Both the symbol handler and the executor run in parallel, and therefore a data owner must be notified about data fetches or updates on another processor, such that send/receive communication pairs can be executed on both processors at the respective checkpoint in the executor.

PARTIALEXECUTECFG (Figure 5, "the *analyzer*") recursively decomposes a given control-flow graph and computes communication envelopes. Depending on the symbol-handler iteration number (*iteration*), the control-flow of a loop or fork with two subgraphs is only entered if all necessary information is available (*flag* in Figure 5, line 10). As an example, we show in Figure 6 how the control-flow graph of the PLUMP matrix-vector product (see also Figure 3) is decomposed into subgraphs.

Given the iteration number, PARTIALEXECUTEASGN decides (Figure 7) whether data referenced on either the left or the right-hand side of an assignment are available to a processor (because they are allocated locally, or because corresponding envelopes were created in the previous iteration). If data are not locally available, either an update or fetch envelope is created with routines CRTUPDATEENVELOPE (Figure 7) or CRTFETCHENVELOPE (Figure 8), respectively. If all data are available, the assignment is executed with routine EXECUTEASSIGNMENT.

In contrast to the *depth vectors* computed by PST to generate the static symbol handler [6] the dynamic symbol handler allocates a *guard* for each referenced data element. A guard defines the earliest iteration number at which the respective data element becomes available on a given processor. In PARTIALEXECUTEASGN and in GUARDEXP (Figure 8), guard values are accessed via the function GUARD. Note that for arrays, a guard for each element is allocated, as well as a collective guard for the whole array (which is equal to the maximum of the element guards). For a given array x, the latter is denoted as GUARD($x(*)$). Routine SETGUARD is used to set guards for arrays and their elements. At a given iteration, the guard of the left-hand side is determined by both the guard of the right-hand side (computed with GUARDEXP) and by the explicit and implicit control-flow dependences on non-local data. A processor

```
Boolean PARTIALEXECUTEASGN(Integer iteration, Assignment a)
 1:   Integer g_l, g_r;
 2:   Symbol x;
 3:   Expression lhs, rhs;
 4:   decompose a into lhs = rhs;
 5:   g_r := GUARDEXP(iteration, rhs);
 6:   if lhs is array x with index expression, lhs = x(index)
 7:     Integer g_i := GUARDEXP(iteration, index);
 8:     g_l := max(g_i, g_r);
 9:     if x is local array
10:       if g_i ≤ iteration
11:         SETGUARD(x(index), g_l);
12:       else
13:         SETGUARD(x(*), g_l);
14:       end if
15:     else x is distributed
16:       if g_i ≤ iteration
17:         if ISREMOTE(x(index))
18:           g_l := max(g_i + 1, g_r);
19:           if g_l ≤ iteration + 1
20:             CRTUPDATEENVELOPE(x(index));
21:           end if
22:         end if
23:         SETGUARD(x(index), g_l);
24:       else
25:         g_l := max(g_i + 1, g_r);
26:         SETGUARD(x(*), g_l);
27:       end if
28:     end if
29:   else lhs is scalar symbol x
30:     g_l := SETGUARD(x, g_r);
31:   end if
32:   if g_l ≤ iteration
33:     EXECUTEASSIGNMENT(a);
34:     return false;
35:   else
36:     return true;
37:   end if
38:   return g_l;
```

Figure 7: PARTIALEXECUTEASGN *computes the guards of both the left and the right-hand side of an assignment. If no non-local dependences exist, the assignment is executed.*

computes whether or not a data item is locally accessible with function ISREMOTE. If both guards of left and right-hand sides are less than or equal to the iteration number, the assignment can be executed.

4.2 Additional Compile-Time Optimizations

The above described symbol-handler introduces fairly large overhead both in execution time, because complete data-flow analysis is done at run-time, and in memory consumption, because a guard is allocated for each array element. To reduce overhead, we implemented two classes of optimizations: compile-time optimizations which do not change the semantics of the symbol-handler, and optimizations which result in a less accurate data-flow analysis.

```
Int GUARDEXP(Integer iteration, Expression e)
 1:   Integer ret := 0;
 2:   if e is operation
 3:     foreach operand expression op in e
 4:       ret := max(ret, GUARDEXP(iteration, op));
 5:     end foreach
 6:   else if e is array access x(index)
 7:     Integer g_i := GUARDEXP(iteration, index);
 8:     if x is local array
 9:       if g_i ≤ iteration
10:         ret := max(g_i, GUARD(x(index)));
11:       else
12:         ret := GUARD(x(*));
13:       end if
14:     else x is distributed array
15:       if g_i ≤ iteration
16:         ret := max(g_i, GUARD(x(index)));
17:         if ISREMOTE(x(index))
18:           ret := ret + 1;
19:         end if
20:       else
21:         ret := GUARD(x(*)) + 1;
22:       end if
23:       if iteration + 1 ≥ ret
24:         CRTFETCHENVELOPE(x(index));
25:       end if
26:     end if
27:   else e is scalar x
28:     ret := GUARD(x);
29:   end if
30:   return ret;
```

Figure 8: GUARDEXP *computes the guard of an expression. If non-local data accesses are encountered during this recursive guard computation, a communication envelope is created.*

Semantic-Preserving Optimizations The compiler can assist the above described run-time mechanism mainly through static data-flow analysis. The described symbol-handler executes all assignments (if the operands are available) and computes guards for every symbol in the critical code segment. However, only assignments to variables on which implicit (index expressions of distributed arrays) or explicit control-flow (if statements and loop expressions) depends have to be performed; and only for those variables guards have to be computed. In addition, the compiler can predict that guards of some variables are zero or remain constant in a certain code segment. For instance, Fortran does not allow a loop index to be changed during the execution of a DO loop, and therefore guards for a loop index need only be computed and checked before a loop is entered. The compiler can also predict locality of data-accesses to eliminate calls to ISREMOTE. For instance, all accesses to `var(i1,i2,...)` in Figure 2 are local. Many other well-known compile-time optimizations such as constant propagation and common subexpression elimination are also applicable for accelerating the symbol-handler.

Since the symbol handler performs assignments, the values of variables must be saved before and restored after each symbol-handler iteration. Here, the compiler can assist by statically computing a list of symbols which are not written in the symbol-handler and need not to be saved.

Relaxing the Accuracy of the Data-Flow Analysis In Figure 4, we associate one guard with each array element. Both memory and execution-time overhead can often be significantly reduced by allocating only *one* guard for the whole array. Practically this can be done by replacing expressions like SETGUARD($x(index), g$) with generic SETGUARD($x(*), g$) calls (Figure 7, lines 11 and 23, and Figure 8 lines 10 and 16). For many simple loops, the compiler needs to generate only symbol-handler code for one iteration.

In general, the above simplifications may also result in less efficient code, because for some expressions the guard computation may be too conservative and cause additional symbol-handler iterations. In many practical cases even with conservative and completely static data-flow analysis, the number of symbol-handler iterations is not affected. Our strategy is to also build optimizations of the second type into PST, but to leave their choice to the user through compiler command-line flags. In addition to the fully static and dynamic symbol handlers, the flags also allow for choosing *hybrid* mechanisms, which combine the virtues of the two approaches.

5 Evaluation of Compiler-Generated Code

The NEC Cenju-3 installed at CSCS is configured with 128 processing nodes, each comprising a 75 MHz VR4400SC RISC processor, 32 Kbytes primary on-chip cache, 1 Mbyte of secondary cache and 64 Mbytes main dynamic memory. Processors communicate via a packet-switched multi-stage interconnection network. For more details see [25]. We use the GNU `gcc` compiler version 2.5.7 as back-end for PST.

For the remainder of this section we collected performance results of two PST-compiled parallel programs: LAPLACE and PILS. The two examples serve different purposes. With LAPLACE we illustrate the overhead introduced by a user-defined dynamic distribution in contrast to a static distribution. With PILS however, the emphasis is on presenting a full application which could not be efficiently parallelized without PST's language extensions and run-time support. In both cases, sequential execution times were obtained by compiling with PST for one processor, i.e., all parallelization and data distribution directives were ignored.

5.1 LAPLACE

LAPLACE is based on a simple Successive Over Relaxation (SOR) algorithm that solves a discretized two-dimensional second order elliptic equation of the form

$$\frac{\partial u}{\partial t} = \Delta u - \rho.$$

As test problem we consider the two-dimensional function

$$\rho(x, y) \quad = \quad sin(x) * sin(y)$$

over $[0; 1] \times [0; 1]$ with periodic boundary conditions discretized on a 1024×1024 grid; u_0 is initialized as 0. In every iteration all points of the underlying grid $U_{i,j}$ are recalculated using a five-point stencil operator. To respect data dependencies, a "red-black" coloring strategy is used: in a first step all ("red") grid points with even $i + j$ are computed; the second step computes all ("black") points with odd $i + j$. Both steps are enclosed in the same public PST `SUB_SPEC PUBLIC` routine. The routine's symbol-handler runs twice and generates and saves two communication patterns.

The algorithm described above can be efficiently parallelized using standard HPF features. For instance, the matrices u and ρ can be distributed in blocks of columns. A mature HPF compiler is able to parallelize all loops statically using the "owner computes rule" and derive the communication pattern. Our point is however, to measure the overhead introduced by dynamic data distributions in a

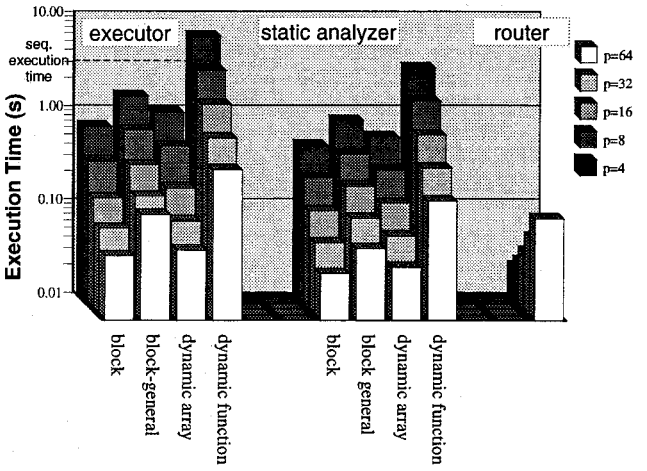

Figure 9: *Execution time of static analyzer, executor and router on several Cenju-3 configurations accumulated over the first SOR iteration for different data distributions, all emulating a block-wise distribution. One iteration consists of two steps, which all call the same routine to first compute "red" and then "black" grid points. The router is based on an MPI all-to-all global broadcast and its performance is independent of the actual distribution directive. Analyzer and router are only invoked during the first iteration. For the computation of speedups executor performance can be compared to the sequential execution time of the main routine.*

	block	block-gen.	dyn. array	dyn. function
static	0.06	0.11	0.07	0.37
dynamic	0.20	0.24	0.21	0.49

Table 1: *The performance of static and dynamic analyzer on 32 Cenju-3 processors is compared by showing analyzer execution times in seconds during the first LAPLACE iteration.*

well-known iterative algorithm. For this purpose, we implemented LAPLACE in four different ways: using a standard HPF BLOCK distribution, using PST's BLOCK_GENERAL distribution and using DYNAMIC distributions (using both mapping array and mapping functions). In all four cases the physical distribution of the matrices u and ρ is the same. Figure 9 summarizes our LAPLACE measurements on several Cenju-3 configurations.

Performance of both executor and analyzer strongly depend on the chosen data distribution. For instance, on 64 Cenju-3 processors, the use of mapping functions results in more than eight times slower executor code compared to the BLOCK distribution. This is primarily due to the implementation of distributed arrays. Because we want to avoid data replication and use the communication cache, every access to an array element requires the calculation of it's owner and local index. For the fastest distribution (BLOCK) this involves the computation of some shift and mask operations. For the slowest (dynamic functions), mapping functions must be called. In addition, PST detects that parts of ownership and local index calculations are loop-invariant with the BLOCK distribution, whereas no optimizations can be applied in all other cases.

As shown in Table 1, on 32 processors, depending on the data distribution, the dynamic analyzer is between 2.4 and 4.3 times slower than the static analyzer. In all cases, however, it consumes less time than three SOR iterations. Since convergence is typically achieved after many more iterations, symbol-handler overhead hardly influences the overall execution time.

5.2 PILS

PILS was originally designed for vector supercomputers, and it is mainly implemented in C++. However, the most time-critical, vectorizable parts of the package are implemented in 50 Fortran subroutines which handle all linear algebra operations. In a typical application, they account for more than 99% of the run-time. In the parallelized version, the C++ part is executed sequentially on processor zero. For the measurements presented in this paper, we have used our HPF extensions to parallelize the Fortran routines.

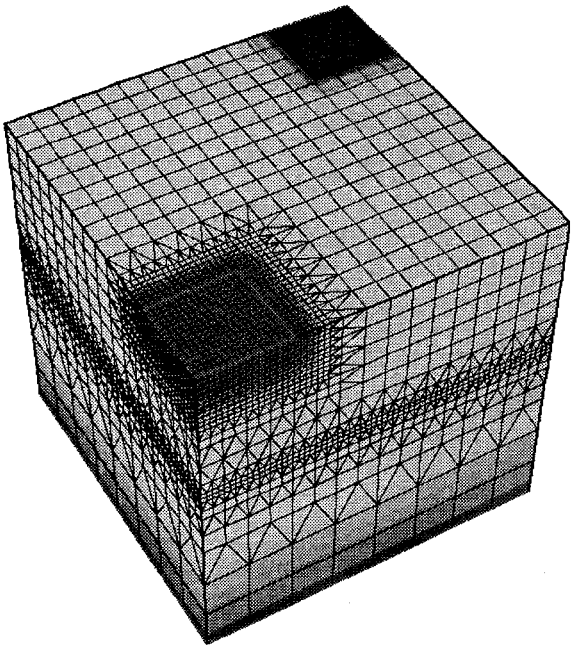

Figure 10: *Finite-element discretization grid for the 3-D model of a bipolar transistor* (BIPOL).

The parallelized library is applied to solve equation systems originating from the 3D finite-element simulation of two semiconductor devices. The sparse matrix of the first system (BIPOL, see Figure 10) stems from a bipolar transistor and has 20,412 rows, 263,920 non-zeros and, on the average, 13 non-zeros per row. The matrix of the second system (DRAM) stems from the coupled solution of a sub-micron DRAM cell with 46,692 matrix rows, 986,042 non-zeros, and on the average 21 non-zeros per matrix row. We use BiCGSTAB as the iterative solution method, preconditioned by a D-ILU preconditioner in split position. Note that the solution of both systems with the same solution method and preconditioner were among the benchmark problems presented in [15].

Parallelization Strategy We have chosen two different mechanisms to distribute the vectors and matrix used in PILS. Vectors are declared as distributed arrays but the matrix is declared local since no changes occur once the array elements have been initialized at the beginning of the iteration. All vector-vector operations (such as `daxpy` and `ddot`) are parallelized in local PST routines. In some vector-vector operations, MPI global reductions are added for the efficient parallelization of reduction loops (such as in `ddot`).

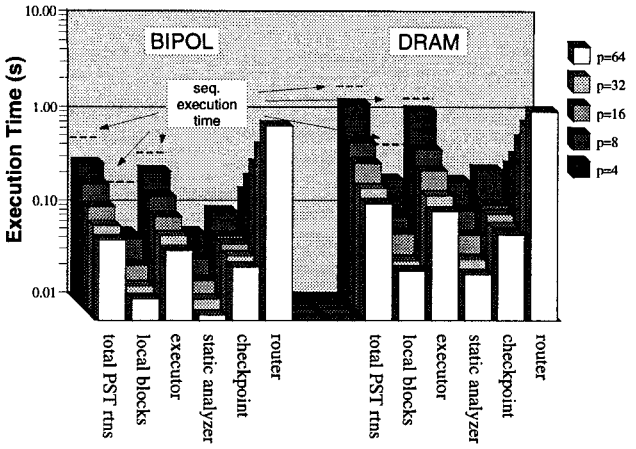

Figure 11: *Execution times on several Cenju-3 configurations of different PILS components during one preconditioned BiCGSTAB iteration of the solution of* BIPOL *(left) and* DRAM *(right). The leftmost columns show total time spent in PST routines in the iteration's steady state. This is broken down into time spent in PST local routines and in time spent in the executor of public routines. In the first iteration, analyzer and router account for additional execution time overhead and several communication patterns are saved (20 for* BIPOL *and 32 for* DRAM*). For large machine configurations, the major performance bottleneck is the execution of communication checkpoints in the executor.*

Matrix-vector operations are declared public and their read and write accesses to distributed vectors are managed by the compiler's global name space. The vectors are distributed using mapping arrays (see also Section 3). These arrays are initialized at the beginning of the iteration using a two-dimensional geometric mapping heuristic [26].

PILS includes coloring heuristics for the extraction of parallelism when solving triangular systems (which arise due to preconditioning). Depending on the number of colors, the same Fortran routines implementing matrix-vector operations are applied to different parts of the linear system. Hence, for the respective PST routines, several different communication patterns must be stored.

Results Figure 11 summarizes our measurements on the Cenju-3 when solving the two equation systems BIPOL and DRAM. A closer look at the figures let us make the following observations:

- The comparison of execution times on one and four processors allows us to estimate the cost of introducing an additional level of indirect addressing when accessing vectors in main memory via the mapping array and communication cache (which is not necessary in sequential code). We estimate that on our machines, accessing vector elements mapping arrays and cache accounts for a performance loss of approximately a factor two to three.
- Both the costs of analyzer and router are negligible if one considers that for achieving a relative norm of the residual of 10^{-10} for BIPOL around 80 and for DRAM 65 iterations are required. If the computation of several linear systems of the same structure is required (which is the case for typical PILS client applications), the overhead is reduced even more.
- With increasing numbers of processors, the analyzer's performance scales almost as well as the local vector-vector operations. The router, however, becomes more expensive because in our current implementation, it requires the execution of an MPI all-to-all global communication operation.
- The main performance bottlenecks when running PILS on large DMPPs, are the executor checkpoints. Detailed measurements show that most of the checkpoint execution time is spent in MPI send and receive primitives. Since message packaging is done by PST optimally at run-time, we conclude that without major changes in the underlying algorithm or data structures, performance could only minimally be improved through additional compiler optimizations. Note that we expect higher performance on DMPPs with smaller communication startup latency because messages are short (for instance, for DRAM and 16 processors in the average 34 data items are communicated in each checkpoint).

analyzer	$P = 4$	8	16	32	64
static	0.17	0.10	0.06	0.04	0.03
dynamic	0.66	0.52	0.43	0.36	0.49

Table 2: *Execution times of static and dynamic analyzer (in seconds) for the solution of* DRAM *running on several Cenju-3 configurations.*

Table 2 provides a performance comparison of static and dynamic analyzers measured when solving DRAM. For large machine configurations, the static analyzer outperforms the dynamic analyzer by roughly an order of magnitude.

6 Summary and Conclusions

Based on the collaboration with application developers, we have identified weaknesses in the current definition of HPF which inhibit the efficient parallelization of similar codes on DMPPs. We have defined language extensions to HPF and built a prototype compiler for these extensions which has been integrated into an existing HPF compiler. The compiler features three directives for dynamically distributing data, and several mechanisms for the run-time preprocessing of critical code segments.

We have put emphasis on the clean integration of our new directives and programming paradigm at the high level into HPF and at the low level with MPI. The programmer specifies code segments which include PST directives as HPF extrinsic routines, with all regular HPF distributions allowed in these routines, and parameter passing handled transparently independent of parameter shapes. On the other hand, and in contrast to HPF, the explicit replication of local computations as defined in the PST programming paradigm, enables clean integration of message-passing primitives into PST code segments.

PST has been integrated into the DMPP programming environment Annai, which also features parallel debugging and performance monitoring support. The parallel debugger assists the PST programmer for instance through visualization of dynamically distributed arrays, and we envisage performance monitoring support such as relating communication operations back to movement of non-local array elements. The combined high-level data-parallel and low-level message-passing support of all three tools gives both comfort and access to the full potential power of the underlying machine.

The existence of our compiler prototype and its usability and performance experienced by our pilot users show that HPF extensions like the ones described in this paper are both indispensable for the parallelization of important applications, and efficiently implementable. We are currently including further optimizations into

PST, increasing the support of the other components of Annai, and considering the port of PST to other platforms. In addition we plan to open access to PST to other users of our computing center, to receive more feedback and work towards making PST a powerful and versatile program development tool for DMPPs.

Acknowledgements

The design of PST and its integration into the NEC HPF compiler only advanced to its current state with the help of T. Nakata, S. Sakon and Y. Seo who are coordinating a group of NEC staff responsible for the HPF compiler development at NEC Tokyo. The development of the Annai tool environment is a joint effort with C. Clémençon, A. Endo, J. Fritscher, and B. Wylie. Other project members, N. Masuda, W. Sawyer, E. de Sturler and F. Zimmermann, are continuously evaluating our prototypes. During the parallelization of applications they have given numerous suggestions which influenced the development of PST. We are grateful to K. M. Decker, C. Pommerell and the anonymous reviewers for careful proofreading and many valuable comments.

References

[1] MPIF (Message Passing Interface Forum). Document for a Standard Message-Passing Interface. Technical Report, Oak Ridge National Laboratory, TN, Apr. 1994.

[2] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr, and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[3] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory architectures. *Concurrency: Practice and Experience*, 3(3), June 1991.

[4] S. Hirandani, K. Kennedy, and C. W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. Supercomputing 91*, Albuquerque, NM, November 1991. ACM-IEEE.

[5] C. Koelbel, P. Mehrotra, and J. van Rosendale. Supporting shared memory data structures on distributed memory architectures. In *Second Symposium on Principles and Practice of Parallel Programming*, page 177, Seattle, WA, March 1990. ACM SIGPLAN.

[6] R. Rühl. *A Parallelizing Compiler for Distributed-Memory Parallel Processors*. PhD thesis, ETH-Zürich, 1992. Published by Hartung-Gorre Verlag, Konstanz, Germany.

[7] C. Clémençon, K. M. Decker, A. Endo, J. Fritscher, G. Jost, N. Masuda, A. Müller, R. Rühl, W. Sawyer, E. de Sturler, B. J. N. Wylie, and F. Zimmermann. Application-Driven Development of an Integrated Tool Environment for Distributed Memory Parallel Processors. In R. Rao and C. P. Ravikumar, editors, *Proceedings of the First International Workshop on Parallel Processing (Bangalore, India, December 27–30)*, 1994. (to appear).

[8] I. Beg, W. Ling, A. Müller, P. Przybyszewski, R. Rühl, and W. Sawyer. PLUMP: Parallel Library for Unstructured Mesh Problems. In *Proceedings of the IFIP WG 10.3 International Workshop and Summer School on Parallel Algorithms for Irregularly Structured Problems, Geneva, Switzerland, August 30, 1994*. Kluwer Academic Publishers, Aug. 1994.

[9] C. Clémençon, A. Endo, J. Fritscher, A. Müller, R. Rühl, and B. J. N. Wylie. The "Annai" Environment for Portable Distributed Parallel Programming. In H. El-Rewini and B. D. Shriver, editors, *Proceedings of the 28th Hawaii International Conference on System Sciences, Volume II (Maui, Hawaii, USA, 3–6 January, 1995)*, pages 242–251. IEEE Computer Society Press, Jan. 1995. ISBN 0-8186-6935-7.

[10] B. J. N. Wylie and A. Endo. Design and realization of the Annai integrated parallel programming environment performance monitor and analyzer. Technical Report CSCS-TR-94-07, CSCS, CH-6928 Manno, Switzerland, Aug. 1994.

[11] C. Clémençon, J. Fritscher, and R. Rühl. Execution Control, Visualization and Replay of Massively Parallel Programs within Annai's Debugging Tool. Technical Report CSCS-TR-94-09, CSCS, CH-6928 Manno, Switzerland, Nov. 1994.

[12] M. Annaratone, M. Fillo, M. Halbherr, R. Rühl, P. Steiner, and M. Viredaz. The K2 distributed memory parallel processor: Architecture, compiler, and operating system. In *Proc. Supercomputing' 91*, Albuquerque, NM, Nov. 1991. ACM-IEEE.

[13] R. Rühl. Evaluation of compiler generated parallel programs on three multicomputers. In *Proc. ACM International Conference on Supercomputing*, Washington DC, July 1992.

[14] C. Pommerell. *Solution of Large Unsymmetric Systems of Linear Equations*. PhD thesis, ETH-Zürich, 1992. Published by Hartung-Gorre Verlag, Konstanz, Germany.

[15] C. Pommerell and R. Rühl. Migration of Vectorized Iterative Solvers to Distributed Memory Architectures. In *Colorado Conference on Iterative Methods (Breckenridge, Colorado, April 1994)*, 1994. Preliminary proceedings, accepted for publication in SIAM J. Sci. Comput.

[16] Y. Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci. Stat. Comput.*, 10(6):1200–1232, Nov. 1989.

[17] J. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *Journal of Parallel and Distributed Computing*, April 1990.

[18] K. Crowley, J. Saltz, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. Technical Report 89-7, ICASE, NASA Langley Research Center, January 1989.

[19] R. Das, J. Saltz, and R. Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993.

[20] C. Chase and A. Reeves. Data remapping for distributed-memory multicomputers. In *Scalable High Performance Computing Conference*, pages pp 137–144, 1992.

[21] B. Chapman, P. Mehrotra, and H. Zima. Extending HPF for advanced data parallel applications. Technical Report TR 94-7, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, 1994.

[22] U. Banerjee. *Speedup of ordinary programs*. PhD thesis, University of Illinois at Urbana–Champaign, October 1979.

[23] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computation. Technical Report CSRD Report no. 1029, University of Illinois, CSRD, August 1990.

[24] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Proc. POPL 95, 22nd Symposium Principles of Programming Languages*. ACM SIGPLAN-SIGACT, 1994.

[25] C. Clémençon, K. M. Decker, A. Endo, J. Fritscher, T. Maruyama, N. Masuda, A. Müller, R. Rühl, W. Sawyer, E. de Sturler, B. J. N. Wylie, and F. Zimmermann. Architecture and Programmability of the NEC Cenju-3. In *Proceedings of the 16th SPEEDUP Workshop*, 1994. SPEEDUP Journal 8(2).

[26] C. Pommerell, M. Annaratone, and W. Fichtner. A set of new mapping and coloring heuristics for distributed-memory parallel processors. *SIAM J. Sci. Stat. Comput.*, 13:194–226, 1992.