Kemari: A Portable High Performance Fortran System for Distributed Memory Parallel Processors

T. KAMACHI,[1] A. MÜLLER,[3] R. RÜHL,[2] Y. SEO,[1] K. SUEHIRO,[1] AND M. TAMURA[3]

[1]*C&C Research Laboratories, NEC Corporation, Tokyo, Japan*
[2]*CSCS-ETH, Section of Research and Development (SeRD), Centro Svizzero di Calcolo Scientifico, CH-6928 Manno, Switzerland*
[3]*1st Computers Software Division, NEC Corporation, Tokyo, Japan*

ABSTRACT

We have developed a compilation system which extends High Performance Fortran (HPF) in various aspects. We support the parallelization of well-structured problems with loop distribution and alignment directives similar to HPF's data distribution directives. Such directives give both additional control to the user and simplify the compilation process. For the support of unstructured problems, we provide directives for dynamic data distribution through user-defined mappings. The compiler also allows integration of message-passing interface (MPI) primitives. The system is part of a complete programming environment which also comprises a parallel debugger and a performance monitor and analyzer. After an overview of the compiler, we describe the language extensions and related compilation mechanisms in detail. Performance measurements demonstrate the compiler's applicability to a variety of application classes.

1 INTRODUCTION

The two most popular paradigms for programming distributed memory parallel processors (DMPPs) are data parallelism and message passing. For both paradigms, standards were defined, namely the data-parallel High Performance Fortran (HPF) [1] and the message-passing interface (MPI) [2]. These standards have been accepted by most vendors which will enable programming DMPPs in a portable way. With MPI primitives, the programmer explicitly controls communication in a portable way at high efficiency. An HPF program may run less efficiently, but communication is implicit and programming less cumbersome and error prone.

Several HPF compilers are already commercially available (for instance from Digital Equipment, and from Applied Parallel Research and the Portland Group), and similar research systems have been developed [3–10]. However, the user of such compilers still faces problems: HPF is a complex language and the sophisticated technology required to generate efficient parallel programs is far from being established.

Ideally, the compiler hides communication from the user. However, when parallelizing an existing Fortran application, the average user often does not achieve

Received May 1995
Revised January 1996

The current address for A. Müller and R. Rühl is ISE Integrated Systems Engineering AG, Technoparkstr.1, 8005 Zurich, Switzerland.

Scientific Programming, Vol. 6, pp. 41–58 (1997)
CCC 1058-9244/97/010041-18

the performance expected. To reduce communication costs, the source code must be restructured manually, guessing which message-passing code will be generated. This is even more difficult since appropriate debugging and performance monitoring tools are often not available.

In addition, HPF only poorly supports some important application classes, namely scientific applications which require unstructured computations. Only well-structured computations can be efficiently parallelized by using the static HPF `BLOCK` and `CYCLIC` distributions.

In the framework of the Joint CSCS/NEC Collaboration in Parallel Processing [11], we have developed a compilation system called "Kemari"* to overcome the above-mentioned problems. Kemari is integrated into a tool environment [12] which supports symbolic debugging and performance monitoring of both high-level data-parallel and low-level message-passing programs. The whole tool environment and in particular the compiler and its language extensions are continuously evaluated by a team of application developers.

We extend HPF so that the user can explicitly define the mapping of computation to processors. The mapping of loop nests can be specified via template and distribute and align directives, in a similar way to the distribution of arrays. With these directives, the user gains more control over the compilation process and the compiler can generate optimized communications. We have also extended HPF with directives to support unstructured computations via dynamic data distributions and run-time preprocessing of critical code segments.

A description of the debugger, the performance monitor, and the analyzer as well as more details on Kemari can be found in [13–17]. In this article, we first provide a global view of our system. We describe the compilation process of well-structured and unstructured applications in more detail. Then we outline how the compiler is integrated into the tool environment and demonstrate the compiler's efficiency using a suite of benchmark programs compiled for an NEC Cenju-3 DMPP.

2 GLOBAL SYSTEM VIEW

Kemari consists of two parts, an HPF compiler built by NEC and the parallelization support tool (PST), a compiler built by CSCS-ETH. The NEC compiler accepts subset HPF plus language extensions for comprehensive computation mapping; PST extends HPF for the support of unstructured computations. The two systems currently exist as two separate binaries and are combined by a common shell script driver which also accepts C and Fortran message-passing sources, and which controls the back-end compilation and the linking processes. While all regular HPF code is translated by the HPF compiler, PST compiles only subroutines declared `EXTRINSIC (PST_LOCAL)`. As shown in Figure 1, both the HPF compiler and the PST share the same parser and first intermediate language (IL).

The first-level IL consists of a parse tree of the input program. It is a single-threaded image of the input source, with a one-to-one representation of parallel language constructs. A first analysis phase generates the information necessary for the translation of the single-threaded representation into a multithreaded program image. Such information includes a data-dependence graph and results of the program flow analysis. Several optimization phases are then applied to the multithreaded representation. For instance, temporary variables are allocated and message aggregation is performed. A final phase generates Fortran 77 code with calls to the HPF compiler's run-time library.

In PST, first an abstract syntax tree is generated from the first-level IL, then the control-flow graphs of the `EXTRINSIC (PST_LOCAL)` subroutines are generated. These graphs also store information about the nesting of control-flow constructs and about the implicit and explicit control-flow dependencies of accesses to nonlocal array elements. Several optimization phases precede the code-generation phase. These optimizations mainly exploit loop-invariant index clauses to avoid expensive run-time global-to-local index translations and owner computations. PST outputs C code with `m4` [18] postprocessor macros and calls to the PST run-time library. The message-passing platform for both compilers is MPI.

2.1 Integration of HPF and PST Sources

Both the HPF compiler and PST maintain information about distributed arrays in array descriptions, which are passed to subroutines as additional arguments. Because the run-time systems of the two compilers have different requirements, different descriptor data structures are used. HPF subroutines can call PST subroutines and vice versa. If necessary, descriptors are automatically converted and data movement is performed.

*Kemari is a traditional Japanese ball game.

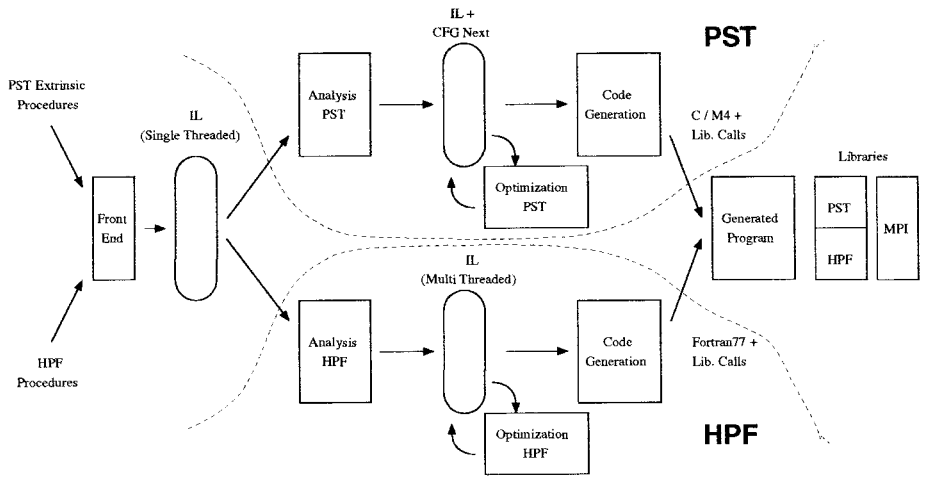

FIGURE 1 Kemari consists of an HPF compiler and PST, a tool which extends HPF for the support of unstructured computations. Both compilers share the same front end and the same intermediate language (IL).

2.2 Integration of Extended HPF and MPI Sources

We believe that it is important to enable the integration of message-passing code into high-level extended HPF sources for two reasons. First, with current technology, the generation of efficient parallel programs from HPF sources is based on a large number of heuristics, and one should not expect any HPF compiler to generate optimal code for any input source. For tuning selected program regions, the user should have access to the underlying message-passing platform. Second, it should be possible to integrate calls to standard libraries within HPF sources. Several parallel numerical libraries are currently being developed to ease scientific computing on DMPPs. One prominent example is ScaLAPACK [19] which is based on MPI.

For the integration of library subroutines and manually tuned message-passing sources into HPF programs we support the following extrinsic interface:

```
INTERFACE
   EXTRINSIC (NONE) SUBROUTINE
   Foo(argument-list)
   END SUBROUTINE
END INTERFACE
```

When an HPF procedure is called, all processors have to execute the call statement and must enter the called procedure, because it may contain global communications. Therefore, under the current HPF specification, parallel loops or statement blocks which should be executed by a specific processor must not include procedure calls, except for calls to PURE functions.

An `EXTRINSIC (NONE)` procedure can be called inside parallel loops. Such a procedure might be implemented in a foreign language and cause side effects. Side effects can be specified with an `INTENT` clause extended to include an access range. The compiler generates necessary communications before and after the procedure call.

We explicitly define that sequential code is replicated both by the HPF compiler and by PST. This allows the user to apply MPI collective communication primitives to replicated data. Both the HPF compiler and PST use dedicated MPI communicators in their run-time libraries such that library communications do not interfere with user-defined communication primitives.

3 COMPILATION OF WELL-STRUCTURED PROBLEMS

We provide a set of HPF extensions for explicit computation mapping and communication optimization. By default, we map computation via the owner-computes rule [20].

For cases where this strategy leads to inefficient code [21], additional mapping control—especially for loop iterations—can largely reduce communication overhead. We introduce iteration templates to specify iteration mappings using the framework of data mappings. An iteration template is a virtual array corresponding to the iteration space of a loop nest. An iteration template can be aligned with another iteration template or with a data object. It can also be

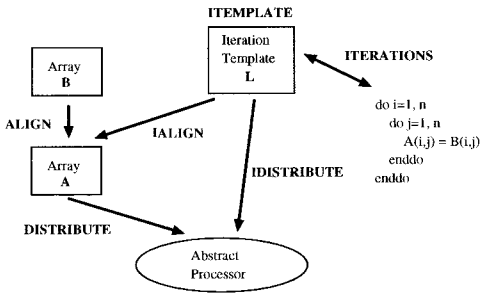

FIGURE 2 Iterations of nested loops can be mapped through alignments and distributions.

distributed onto an abstract processor. Figure 2 shows the relation between data and iteration mappings.

In contrast to the one-dimensional *ON clause* implemented with Vienna Fortran [10], with our directives, also multiply nested loops can be mapped. An iteration template can be distributed even when there are no data objects with which to align. The compiler framework for dealing with iteration and HPF data mappings is the same. The compiler ensures that data are remapped when data and iteration mappings are different.

Note that sequential loops can also be mapped. In addition we provide directives to map statement blocks.

3.1 Language Extensions

Iteration Templates, Distributions, and Alignments

```
!DIR$ ITEMPLATE template-decl-list
!DIR$ ITERATIONS template-name
(index1, index2, ...)
!DIR$ IDISTRIBUTE distributee
dist-directive-stuff
!DIR$ IALIGN alignee align-directive-
stuff
```

The ITEMPLATE directive declares the shape of an array which represents the iteration space of a loop. The ITERATIONS directive associates an iteration template with the loop nest defined by the loop indices (index1, index2, . . .).

The IDISTRIBUTE and IALIGN directives define the distribution of an iteration template. Their syntax is identical to the HPF DISTRIBUTE and ALIGN directives. The target of a loop alignment can be an HPF template, an array, or an iteration template.

As an example, in Figure 3, we show how a two-dimensional triangular loop nest (a) is distributed with an iteration template on the one hand (b) and a Vienna Fortran ON clause on the other hand (c). Iteration templates achieve better load balance. Note that the loops could also be distributed equally using a CYCLIC distribution. However, in many cases (for instance stencil-based computations) this could induce additional communication.

OVERLAP **Directive**

The OVERLAP directive is placed after a DISTRIBUTE or IDISTRIBUTE directive, and it specifies the overlapping of iteration or data distributions. The overlapping depth for the i-th dimension is specified in Fortran 90 notation as also shown in Figure 4.

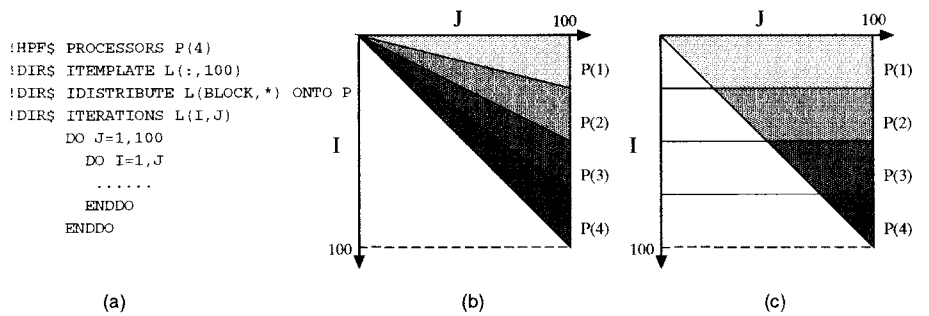

FIGURE 3 Parallelization of an example loop nest (a) with iteration templates (b) and the ON clause (c).

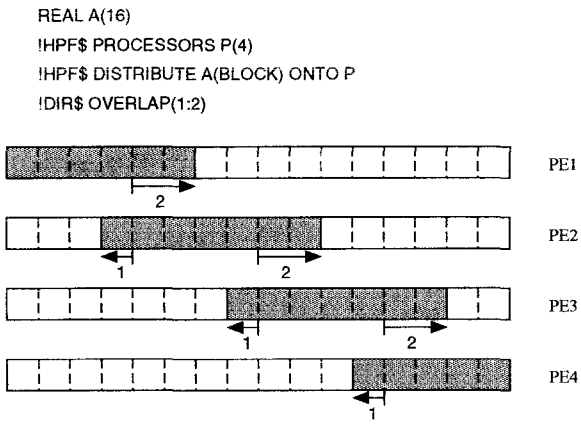

FIGURE 4 The array region owned by each processor is extended with the OVERLAP directive.

SINGLE/OWNER

The program fragment enclosed in the following two directives

```
!DIR$ SINGLE [processor-name(int-expr-
list)]
!DIR$ END SINGLE
```

is executed by a single given processor. EXTRINSIC (NONE) procedures can be called in the fragment.

```
!DIR$ OWNER object-name[(int-expr-list)]
!DIR$ END OWNER
```

The OWNER directive is similar to the SINGLE directive except that multiple processors which own a specified array element execute the computation.

PARALLEL/SERIAL

```
!DIR$ PARALLEL
!DIR$ SERIAL
```

These directives can be placed in front of a DO/FORALL loop, and specify whether it should be executed sequentially or in parallel. The SERIAL directive is useful to inhibit automatic parallelization of loops which would cause too much run-time overhead.

NO_PRE_MOVE/NO_POST_MOVE

The following directives

```
!DIR$ NO_PRE_MOVE data-object-list
!DIR$ NO_POST_MOVE data-object-list
```

control communication. For remote accesses to objects included in the data-object-list the communication is suppressed. With these directives a programmer can assist the compiler in reducing communication overhead.

INTENT Attribute with Access Range

The INTENT clause is extended so that the access range of an array can be specified to define a more precise interface of a procedure.

```
INTENT(IN | OUT | INOUT) ::
var(access-range-specifier)
```

Section 6.1 contains an example code segment which illustrates the use of our INTENT directive.

3.2 Communication Generation

Kemari generates required communications for regular loop computations based on the alignment mechanism. In order to generate a communication for a distributed array, first, the data alignment which requires no remote accesses for the array is computed. We call this alignment optimal data distribution. A library routine call is generated to realign the original with the optimal data distribution.

When an array mapping is only known at run-time, general remapping library routine calls are generated. For data mappings known at compile-time, the compiler selects a faster specialized routine. Communications are generated in three phases: preanalysis, mapping generation, and communication optimization.

Preanalysis

In this phase, all the information which is required for mapping generation and communication optimiza-

tion is prepared. The information includes data dependencies, default data and loop iteration mappings, array access regions in loops, and a control-flow graph annotated with variable kill information.

The default mappings are determined by the following rules:

1. A scalar variable is replicated.
2. An array which is not defined as a formal parameter is replicated.
3. An array passed as argument inherits its distribution from the calling subroutine.
4. A loop iteration is mapped so that most frequently accessed arrays of the same data mapping do not require communication.
5. Sequential program fragments are replicated.

The array access range is represented by a regular section descriptor (RSD) [22]. Multiple accesses to the same array are reduced to a single RSD representation for message coalescing [20, 23]. For example, two accesses to `A(1:N)` and `A(2:N+1)` are replaced by an access to `A(1:N+1)`. Kill information can sometimes be used to eliminate unnecessary communication.

Mapping Generation

The communication analysis for an access to array `A` is carried out as follows:

1. If data distribution information for `A` is not available at compile-time and the data access region can be represented by an RSD, the compiler computes the optimal data distribution for `A` using the RSD and the loop alignment. An `LREALIGN` library call (described in Section 3.4) for the region described by the RSD is generated.
2. If the access to `A` appears in the lefthand side, and the corresponding RSD is not available, a `POST_WRITE` is generated after the update of `A`. In addition, a SWEEP is generated after the loop to receive the data sent by POST_WRITE.
3. If the access to `A` appears in the righthand side, and the corresponding RSD is not available, the whole region of `A` is copied to all the abstract processors using a `REPLICATE_ALL` before the loop.
4. When data distribution information is available at compile-time, communication optimizations described in the following section are applied.

Figure 5 shows an example of the computation of an optimal data distribution. Array `B` and iteration template `L` are aligned differently with array `A`. In the preanalysis phase, the access region of `B` in the loop is set to `[1:15:2]`. Correspondingly, the iteration alignment with array `A` is represented as `[2:16:2]`. Using the two triplets, an aligned triplet for the optimal data distribution is computed. If array `A` is distributed either block-wise or block-cyclic, a `SHIFT` communication for the array section `B(1:15:2)` is generated.

Communication Optimization

The compiler recognizes four regular communication patterns:

SHIFT. If the distributions of the original and the target mapping are the same, and the strides of both align triplets are the same, a SHIFT communication is generated. The generation is made for each dimension in which the upper or lower boundaries of the align triplets differ.

REPLICATE, GATHER, and SCATTER. REPLICATE is used for generating communications for `REALIGN` directives which require data replication. GATHER and SCATTER are used when distributed data are accessed in a sequential execution part such as a `SINGLE` or `OWNER` region. GATHER collects the data to the executing processor, while SCATTER writes the data back to the owner processors.

3.3 Reduction of Memory Requirements

The current HPF compiler is implemented under the all-replicate strategy, where the whole global array space is allocated for each distributed array. This strategy has advantages: Address translation from global into local is not needed and memory management is not required to remap arrays at subroutine boundaries or at `REALIGN/REDISTRIBUTE` directives. On the other hand, a program using large distributed arrays can possibly not be executed. With the next version of the compiler we plan to implement a more sophisticated allocation strategy.

3.4 Run-Time Library

We have designed a high-level run-time communication library with communications optimized inside the library. The compiler can be ported to other target systems just by porting (and optimizing) this library.

Table 1 describes the communication library routines. At the moment, two versions of the library are implemented, namely, an MPI version and a version using native Cenju-3 communication primitives. All routines are implemented using block data transfers

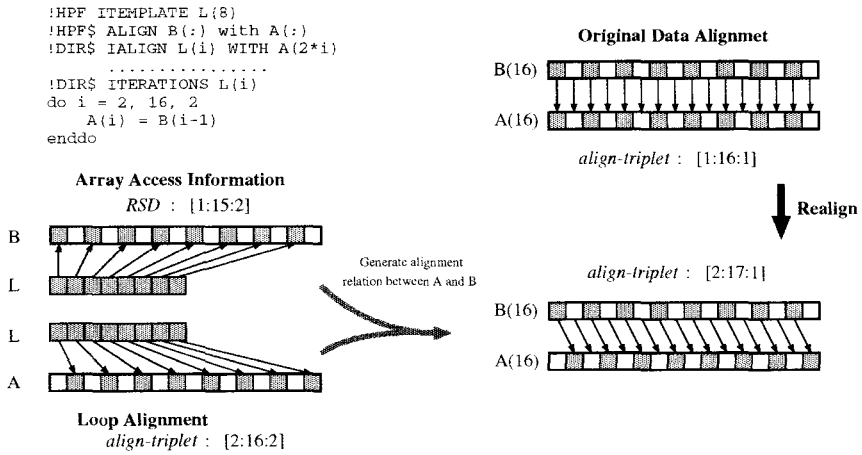

FIGURE 5 An optimal data distribution is computed using array access and loop alignment information.

in order to minimize the overhead of communication latency.

The compiler uses the LREALIGN routine mainly to generate communications for remote memory accesses inside parallelized loops. Its functionality is almost the same as that of REALIGN, except that the array part can be aligned and the realignment of an array does not affect the mapping of other data objects which are ultimately aligned to the array.

4 COMPILATION OF UNSTRUCTURED PROBLEMS

PST supplements the compilation mechanisms described in Section 3 by providing an extended input language, a different compilation technique, and a different underlying programming paradigm. The paradigm is similar to a shared memory model: By default, data are private, i.e., the compiler does not enforce consistency of nondistributed data across processors; only distributed data are part of the global name space which is supported and maintained by run-time analysis. Also code is replicated, i.e., by default, all statements are 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 (i.e., by using the single-program multiple-data paradigm).

4.1 Language Extensions

Subroutine Specifiers

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

Routines to be compiled by PST instead of the HPF

Table 1. List of Communication Library Routines: The Compiler System can Generate any Communication Pattern Using These Routines

Routine Name	Function
REALIGN	Realign a data object following a `REALIGN` directive.
REDIST	Redistribute a data object following a `REDISTRIBUTE` directive.
LREALIGN	Temporarily realign a data object for the remote access required in nested loops.
LREDIST	Temporarily redistribute a data object for the remote access required in nested loops.
PRE_SHIFT	Shift a set of data objects along one dimension for remote data reference.
POST_SHIFT	Shift a set of data objects along one dimension for remote data update.
REPLICATE	Replicate a data object along the specified dimensions.
GATHER	Gather a data object to the specified logical processors from the owners.
SCATTER	Scatter a data object to the owners from specified logical processors.
MULTICAST	Multicast a data object to other logical processors.
POST_WRITE	Transfer a data element to the specified logical processor. Used for unstructured communications.
SWEEP	Sweep data elements sent by POST_WRITE.

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 | PUBLIC }
!PST$ SAVECOM(key)
```

For PUBLIC routines, PST supports a global name space at run-time by generating a run-time preprocessing phase (inspector) which precedes the actual computation in the executor. In LOCAL routines (the default), the compiler does not permit accesses to nonlocal array elements.

With the SAVECOM directive, an EXTRINSIC(PST_LOCAL) subroutine can be declared to be start-time schedulable. As a consequence, 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 subroutine.

Dynamic 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 array var is partitioned into contiguous blocks of possibly different sizes. In contrast to a similar distribution described by Chapman et al. [24], 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. 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 multidimensional arrays are mapped (or "linearized") into a single integer by using reverse lexicographic ordering (for instance column-major 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.

Nested Fortran DO loops can be parallelized by aligning them with a distributed variable of the same dimensions as the loop nest. Loops aligned with dynamically distributed arrays are transformed into loops over the local index range, and the first statement in their body computes the respective global indices using the local-to-global index mapping.

4.2 Communication Generation

Depending on the nesting of distributed-array accesses, the inspector consists of one or more slices of an EXTRINSIC(PST_LOCAL) procedure. Such a preprocessing strategy was first implemented and described with Oxygen [25], and has also been investigated recently by other research groups [26, 27].

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. 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. If required, users can explicitly set the serial time with a directive. This feature can be used to parallelize loops with data dependencies.

The inspector preprocesses code segments to determine where nonlocal 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 paralellism in the form of aligned DO loops), but with all references to nonlocal elements of distributed arrays replaced by references to dynamically allocated buffers. These buffers store data communicated in previous checkpoints (for remote data fetches), or updates of nonlocal data to be communicated in future checkpoints.

For the computation of nonlocal data dependencies, the data flow in the program must be analyzed. This can be done at run-time by adding tags (or guards) to each data element which store the number of inspector iterations required for the complete computation of a respective envelope. Alternatively, a less accurate data-flow analysis can be performed at compile-time. Inspectors 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.

PST supports both inspector mechanisms and allows the user to choose (with a command line flag) between the more precise (guard-based) dynamic inspector and the faster and less memory-consuming static inspector, generated with compile-time data-flow analysis. For the generation of static inspectors, PST uses the same algorithms as Oxygen; for a detailed explanation, see [9]. The generation of dynamic inspectors has been described in [17].

4.3 Reduction of Memory Requirements

We call the buffers mentioned above communication caches because of their faint structural resemblance to hardware caches. Also the problematic nature is similar: A compromise has to be made between memory consumption and run-time overhead. Figure 6 depicts the four alternatives. In the simplest organization (0), we completely replicate the allocation of distributed data on all processors. With organization (1), the global index of a distributed array element is hashed by first computing the element's owner (i.e., the processor which maintains a consistent copy of the element), and then by allocating a complete copy of that processor's part of the distributed array. Organization (2) differs from organization (1) in the fact that a processor's local array segment is allocated in blocks of fixed size (b) rather than as a whole. The index of an array element in such a block is equal to the element's local index, modulo b. When remote data are accessed, organization (3) inserts remote values of a distributed array in a sorted buffer. The buffer includes both data values and global array indices.

Note that the main concept behind our communication caches is not new and a similar mechanism with simpler buffer organizations was already described in 1991 by J. Saltz and his colleagues [28]. In [29] we summarized the impact the choice between either of the four cache organizations can have on performance and memory consumption of a full application.

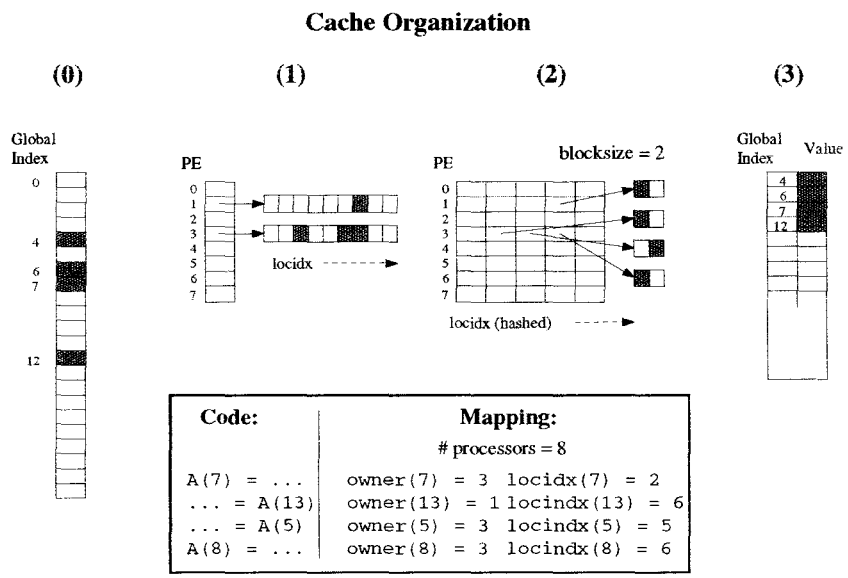

FIGURE 6 Different communication cache organizations implemented in PST.

4.4 Run-Time Library

The PST run-time library consists of two parts: A set of `m4` [18] preprocessor macros and the actual library routines written in either C or C++. We use `m4` macros whenever calling a procedure would be too expensive, for instance when creating symbol-table entries, or for the compilation of complicated control-flow constructs. The library routines manage a great deal of the run-time preprocessing mechanism, like the management of multiple communication patterns for the same `PUBLIC EXTRINSIC(PST_LOCAL)` subroutine.

5 INTEGRATION INTO THE TOOL ENVIRONMENT ANNAI

The software described in this report has been integrated into the multicomputer programming environment Annai [12]. Since Kemari 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. Annai also includes a parallel debugging tool (PDT), a performance monitor and analyzer (PMA), and a common graphical user interface (UI). The main design objectives of Annai can be summarized as the

1. Design and implementation of tools for the development of parallel programs in a high-level data-parallel multiple-instruction multiple-data (MIMD) language and also with low-level message passing.
2. Parallelization, monitoring, and debugging support for applications considered today difficult to parallelize on DMPPs.
3. 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.

Kemari supports Annai's parallel debugger and performance monitor and analyzer in various aspects. First of all, both PDT and PMA allow the instrumentation of the loaded binary with breakpoints and instrumentation points, respectively. A comfortable instrumentation is best performed close to the high-level source code, and therefore Kemari must maintain correct line number information when translating from extended HPF to C and Fortran with message passing. The compiler also changes some of the original symbols, a mechanism which is kept transparent to the user of PDT and PMA. Annai's UI includes a program structure browser (PSB), a display which allows depiction of performance statistics close to related program structures such as subroutines and `DO` loops for example. For PSB, Kemari generates tables with information about the program structure. These tables are stored in the generated object, and can be read by both PDT and PMA at run-time. PDT allows graphical display of one and two-dimensional arrays and their HPF/PST distributions using the distributed data visualizer (DDV, see also "The NAS MG Kernel" in Section 6). For DDV, the PST run-time library includes procedures which return information about array descriptors and which are used to dump array values to the file system (in parallel). Finally, PMA features a variety of displays which can depict run-time overheads both in terms of execution time and memory for compiler-generated parallel programs. We are also currently developing a display with which the user can relate communication costs back to the movement of array segments necessary for the parallelization of a given loop nest. Such PMA displays are supported by Kemari through instrumentation of the generated code and of its run-time libraries.

6 SYSTEM APPLICATION

The NEC Cenju-3 can be configured with up to 256 processing nodes, each comprising a 75-MHz VR4400SC RISC processor, 32 Kbyte primary on-chip chache, 1 Mbyte of secondary cache, and 64 Mbyte main dynamic memory. Processors communicate via a packet-switched multistage interconnection network. For more details see [30]. For the benchmarks presented in Section 6.1, we use the NEC `cjf77` compiler version r1.0 as back-end for our HPF translator (with options `-O -Kmips2 -Kfullasopt`). For the benchmarks presented in Section 6.2, we use the GNU `gcc` compiler version 2.5.7 (with option `-O3`) as back-end for PST.

6.1 Application of Static Compilation Techniques

Table 2 lists HPF benchmark programs and their characteristics. The "Distribution" column describes the way the main data for each program are distributed onto abstract processors. The corresponding entries are checked when the respective directives were used.

Table 3 shows the measurement results of HPF benchmark programs on several Cenju-3 configurations. The figures inside parentheses show the speedup ratio, where the sequential execution performance of the corresponding original program is 1.00. The shapes of the abstract processors onto which PDE1's main

Table 2. These Benchmark Programs are all Well Structured: The Shape of the Main Data, Applied Data Distribution, and Applied Extended Directives for Them are Described

Benchmark Name	From	Computation Model	Shape of Main Data	Distribution	Remark	EXTRINSIC (NONE)	OWNER SINGLE	IALIGN IDIST.	NO_PRE/ POST_MOVE
EP	NAS	Particle code	Scalar			X	X	X	
Scalgam	U. Calif. (APR)	Particle code	Scalar		NPART=2*10⁶	X	X	X	X
Shallow	NCAR	Weather prediction (FDM)	(513,513)	(*,block)			X		X
Baro	APR	General circulation model	(302,152)	(*,block)		X	X		X
Grid	APR	Simple relaxation	(502,502,2)	(*,block,*)			X		X
X+2	APR	Fourth-order differencing	(512,512)	(*,block)			X		X
Tomcatv	SPEC	Mesh generation with Thompson solver	(513,513)	(*,block)			X	X	X
PDE1	GENESIS	3-D red black poisson solver	(128,128,128)	(block,block block)	NITER=10		X	X	X
SCG	NEC	3-D SCG poisson solver	(64,64,64)	(*,*,block)			X		X

three-dimensional data are distributed are: 1PE-(1,1,1), 2PE-(1,1,2), 4PE-(1,2,2), 8PE-(2,2,2), 16PE-(2,2,4), 32PE-(2,4,4), and 64PE-(2,4,8). Array RED is initialized so that the value of RED(i,j,k) is "true" when $i + j + k$ is even.

Extrinsic Interface

Non-HPF procedures with side effects can be invoked in parallel loops using the EXTRINSIC(NONE) interface. With this feature, programmers can specify procedure-level parallelism. The parallelization of Baro, Scalgam, and EP can take advantage of that capability. For example, in the case of Baro, there is a subroutine CMSLOW which processes a column of two-dimensional arrays. The main loop (DO 100) in subroutine COMP can be parallelized without inlining by specifying an interface block with INTENT directives as follows:

```
      SUBROUTINE COMP(...)
          ..........
CHPF$ TEMPLATE TWOD(302,152)
CHPF$ ALIGN WITH TWOD::PVORT,YGRADS,VTSP,UTSP
CHPF$ DISTRIBUTE TWOD(*,BLOCK)
          ..........
       INTERFACE
        EXTRINSIC(NONE) SUBROUTINE
CMSLOW(J,YGRADS,UTSP,VTSP)
         COMMON/BLK/ ..., PVORT(302,152), ...
          ..........
         DIMENSION
YGRADS(302,152),UTSP(302,152),VTSP(302,152)
         INTENT (IN)    :: YGRADS(1:IM,J:J)
         INTENT (INOUT) :: UTSP(1:IM+2,J+1:J+1)
         INTENT (INOUT) :: VTSP(1:IM+2,J+1:J+1)
         INTENT (IN)    :: PVORT(1:IM+2,J+2:J+2)
        END SUBROUTINE
       END INTERFACE
          ..........
      DO 100 J=1,JM
          ..........
       CALL CMSLOW(J,YGRADS,UTSP,VTSP)
100 CONTINUE
```

Table 3. Execution Time (in Seconds) and Speedup Ratio (in Parentheses) of Selected HPF Benchmarks on several Cenju-3 Configurations

	Sequential*	1 PE	2 PEs	4PEs	8PEs	16 PEs	32 PEs	64 PEs
EP	89.56	89.93	44.98	22.50	11.25	5.66	2.83	1.42
	(1.00)	(1.00)	(1.99)	(3.98)	(7.96)	(15.82)	(31.65)	(63.07)
Scalgam	718.80	710.67	357.54	181.35	91.70	47.22	22.52	12.16
	(1.00)	(1.01)	(2.01)	(3.96)	(7.84)	(15.22)	(31.92)	(59.11)
Baro	8.90	8.88	4.36	2.14	1.17	0.73	0.48	0.41
	(1.00)	(1.00)	(2.04)	(4.16)	(7.61)	(12.19)	(18.54)	(21.70)
Shallow	50.50	50.65	25.60	12.95	6.50	3.50	2.00	1.20
	(1.00)	(1.00)	(1.97)	(3.90)	(7.77)	(14.43)	(25.25)	(42.08)
Grid	57.92	57.82	28.97	14.66	7.38	3.81	1.96	1.05
	(1.00)	(1.00)	(2.00)	(3.95)	(7.85)	(15.20)	(29.55)	(55.16)
Tomcatv	209.44	130.07	65.62	33.48	17.42	9.38	5.35	3.45
	(−)	(1.00)	(1.98)	(3.89)	(7.47)	(13.87)	(24.31)	(37.72)
X42	17.85	21.16	10.70	5.45	2.79	1.46	0.75	0.36
	(1.00)	(0.84)	(1.67)	(3.28)	(6.40)	(12.23)	(23.80)	(49.58)
SCG	274.23	293.84	146.12	77.75	40.26	21.66	12.71	8.40
	(1.00)	(0.93)	(1.88)	(3.53)	(6.81)	(12.66)	(21.58)	(32.65)
PDE1	6.22	6.29	3.18	1.61	0.85	0.43	0.22	0.11
	(1.00)	(0.99)	(1.95)	(3.87)	(7.31)	(14.60)	(28.92)	(58.66)

* Shown are execution times of the original sequential program without any parallelization overhead.

Scalgam is a typical Monte-Carlo particle code. In each iteration of the main loop, a particle generated by random numbers is traced, and contributions of the particle are pushed to the global data. The trace computation is very complicated and it contains several subroutine calls (e.g., SOURCE, TRACK, and PUSH). However, the program can be easily parallelized by introducing a new subroutine with an EXTRINSIC(NONE) attribute which describes the trace operation for a particle. By defining the interface of the newly defined subroutine, all the detailed computation can be encapsulated inside the subroutine as follows.

```
 INTERFACE
  EXTRINSIC(NONE) SUBROUTINE
EXEC(NPS,KUTOF,KILLED,DEAD,
&
MXSTAK,MXBANK,XC,XPP,XPE,INDEX,WINDEX,KTAL,MAT,ISTACK)
   COMMON /RANCHO/ NRAND,NHRAN
   INTENT(INOUT) :: KUTOF,KILLED,DEAD,MXBANK,MXSTAK,
&                   NRAND,NHRAN
  END SUBROUTINE
END INTERFACE
  .......
 DO NPS = 1,NPTOT
    CALL EXEC(NPS,KUTOF,KILLED,DEAD,MXSTAK,MXBANK,XC,
&         XPP,XPE,INDEX,WINDEX,KTAL,MAT,ISTACK)
    (update global variables
       i.e. GLOBAL_DEAD= GLOBAL_DEAD + DEAD)
 END DO
 (local variables (- global variables
    i.e. DEAD = GLOBAL_DEAD)
```

Table 4. Execution Time (in Seconds) and Speedup Ratio (in Parentheses) for PDE with and without *SHIFT* Optimization

	Sequential	1 PE	2 PEs	4 PEs	8 PEs	16 PEs	32 PEs	64 PEs
PDE1	6.22	6.29	3.18	1.61	0.85	0.43	0.22	0.11
(with shift optimization)	(1.00)	(0.99)	(1.95)	(3.87)	(7.31)	(14.60)	(28.92)	(58.66)
PDE1	6.22	6.29	3.19	1.61	1.10	0.54	0.28	0.14
(without shift optimization)	(1.00)	(0.99)	(1.95)	(3.86)	(5.65)	(11.47)	(22.45)	(46.06)

NOTE: Optimizations performed during the compilation of PDE include stride aggregation.

A new subroutine EXEC is introduced, which performs the particle trace. For each variable which is used for the reduction operations (e.g., MAX and SUM), global variables are defined to push local results. After the parallel loop, the value of the local variables is replaced by the value of corresponding global variables, such that the following computations do not have to be changed. The loop can be parallelized using the `EXTRINSIC(NONE)` interface with little modifications to the loop body. Moreover, the subroutines called from EXEC require no change. These subroutines and EXEC can be compiled just with a backend Fortran compiler.

Iteration Mapping

PDE1 is a three-dimensional Poisson solver using red-black relaxation. The best performance can be achieved using the (BLOCK,BLOCK,BLOCK) distribution. For the following nested loop, the iteration mapping can be specified using an iteration template.

```
!DIR$ ITERATIONS IT(I,J,K)
      DO 2 K=2,NZ-1
        DO 2 J=2,NY-1
          DO 2 I=2,NX-1
           IF (RED(I,J,K)) THEN
            U(I,J,K) = FACTOR*(HSQ*F(I,J,K) +
     *                        (U(I-1,J,K) + U(I+1,J,K) +
     *                         U(I,J-1,K) + U(I,J+1,K) +
     *                         U(I,J,K-1) + U(I,J,K+1)))
           ENDIF
    2 CONTINUE
```

Stride Aggregation

Table 4 shows the effect of the communication optimization. In the above example loop, a SHIFT communication along with all the three dimensions is required. This type of communication can be optimized using stride aggregation as described in Section 3.

6.2 Application of Dynamic Compilation Techniques

We collected performance results of two PST-compiled parallel programs: the package of iterative linear solvers (PILS) [31] and the NAS MultiGrid benchmark kernel (MG). The two examples show different applications of user-defined dynamic data distributions: In PILS the distribution of the main data structures is defined via a mapping array. MG uses mapping functions for the distribution of a work array, employed in an old-fashioned way in the original kernel Fortran 77 source instead of dynamic allocation.

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 (vector-vector and matrix-vector operations) and which we parallelized with Kemari. The paralellization strategy was similar to that chosen in an earlier project [32].

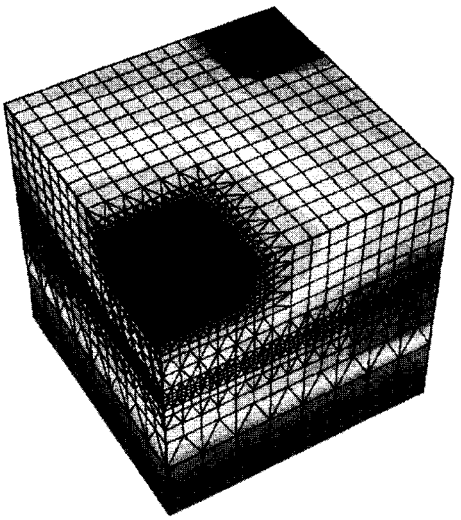

FIGURE 7 Finite-element discretization grid for the three-dimensional model of a bipolar transistor (BIPOL).

The parallelized library is applied to solve equation systems originating from the three-dimensional finite-element simulation of two semiconductor devices. The sparse matrix of the first system (BIPOL, see Fig. 7) stems from a bipolar transistor and has 20.412 rows, 263.920 nonzeros and, on the average, 13 nonzeros per row. The matrix of the second system (DRAM) stems from the coupled solution of a submicron DRAM cell with 46.692 matrix rows, 986.042 nonzeros, and on the average 21 nonzeros per matrix row. We use BiCGSTAB as the iterative solution method, preconditioned by a D-ILU preconditioner in split position.

All vector-vector operations are parallelized in local PST routines. 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.

Figure 8 summarizes our measurements on the Cenju-3 when solving the two equation systems BIPOL and DRAM. For the PILS measurements we chose communication cache organization (1).

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 of two to three.

The inspector costs (broken down into a computation-phase "analyzer" and a communication-phase "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.

The main performance bottlenecks when running PILS on large DMPPs are the executor checkpoints.

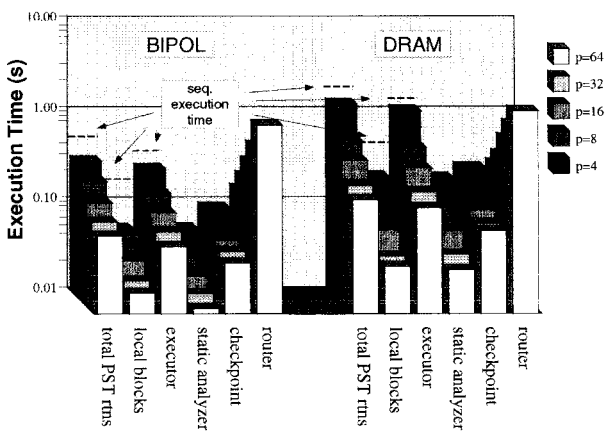

FIGURE 8 Execution times on several Cenju-3 configurations of different PILS components during one preconditioned BiCGSTAB iteration of the solution of BIPOL (left) and DRAM (right). The leftmost columns show total time spent in PST routines in the iteration's steady state. This is broken down into time spent in PST local routines and in time spent in the executor of public routines. In the first iteration, the inspector (broken down into "analyzer" and "router") accounts for additional execution time over-head 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.

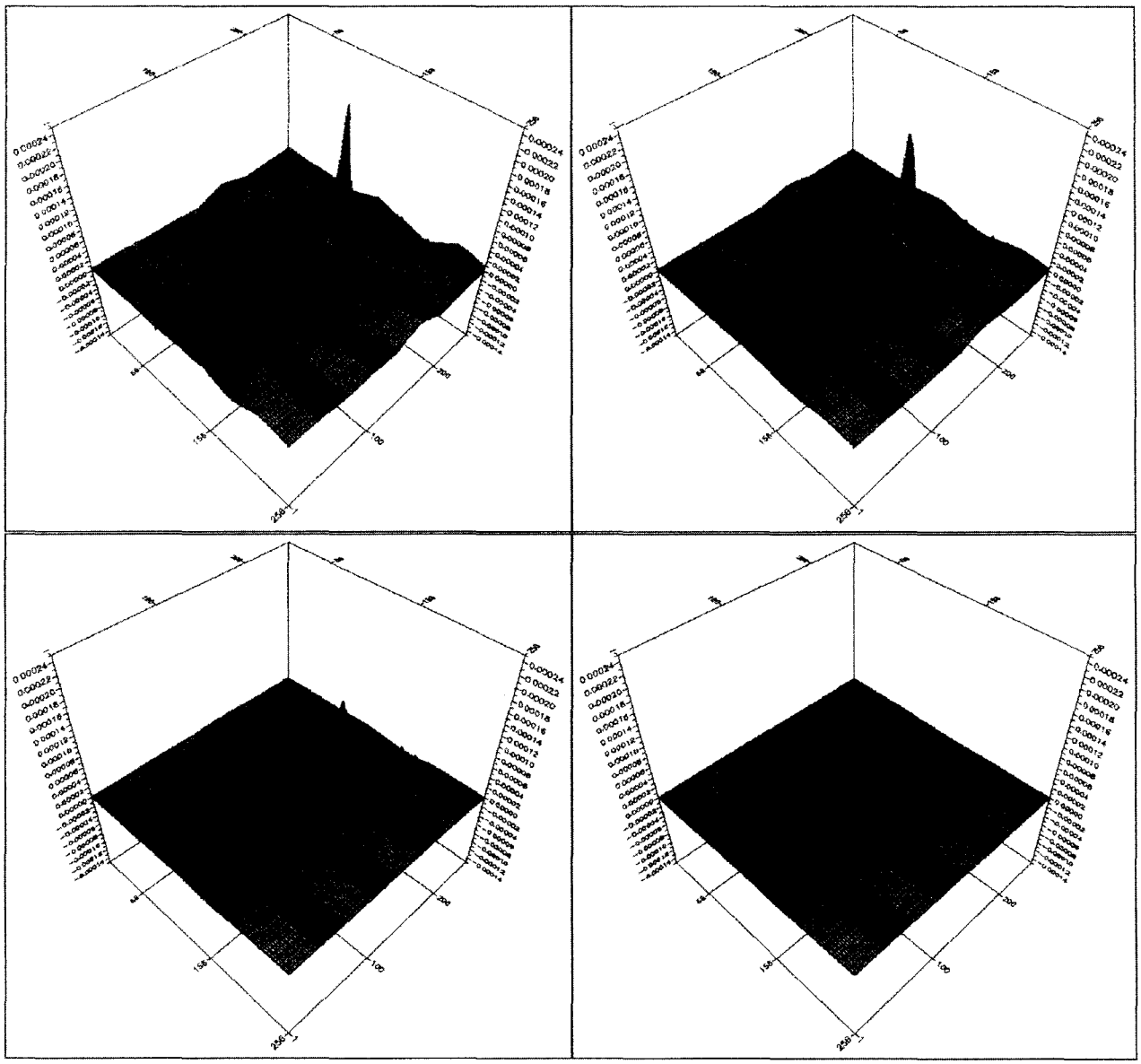

FIGURE 9 Four PDT distributed data views show the value of the residual of the CLASS_B problem ($256 \times 256 \times 256$) for a given x-coordinate after 1, 2, 4, and 8 V-cycles of the MG iteration. The high-frequency error components (i.e., the discontinuities) are efficiently removed in a few V-cycles.

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 on average, 34 data items are communicated in each checkpoint.

The NAS MG Kernel

The NAS MG kernel [33] calculates an approximate solution to the discrete Poisson problem using four iterations of the V-cycle multigrid algorithm on an $n \times n \times n$ grid with periodic boundary conditions. MG is known to be a difficult problem for data-parallel

Table 5. Performance of MG when Parallelized with PST on 64 and 128 Processor Cenju-3 Configurations

No. PEs	n	HPF/PST $T(i = 1)$	HPF/PST $T(i = 2)$	HPF/PST MFlop/s	C/MPI (MFlop/s)
64	64	1.72	0.20	159	405
	128	3.78	1.00	249	718
	256	14.95	6.45	311	856
128	64	3.06	0.15	208	586
	128	4.92	0.66	377	1197
	256	12.81	4.11	488	1568

NOTE: Execution times of the first and second V-cycle are shown and MFlops for the second cycle (when the iteration enters steady state, and PST-generated communication patterns are reused) are compared to what is achieved with a manually parallized C code.

compilers, most of which cannot accommodate the V-cycle which goes from large data sets to small and back [34].

We parallelized the original Fortran source using HPF/PST directives and compared performance with an optimized C/MPI version of the benchmark. For both versions, the cubic grid is partitioned into "matchsticks": All grid points in one dimension are on one processor and distributed block-wise in the other two dimensions, in other words an HPF `(*,BLOCK,BLOCK)` distribution.

In the PST version, user-defined mapping functions are used to distribute one long "work-array" which keeps all levels of the grid, and the amount of data for each subsequent grid level decreases exponentially. Each grid level in the work array is distributed regularly in the matchstock fashion. Loop distribution directives are applied to all of the $n \times n \times n$ loops according to the data distribution.

As an example for the integration of PST with PDT, and for the PDT distributed data visualization capabilities, Figure 9 shows different PDT representations of a slice (with a given x coordinate) of the MG residual at four different iterations. Both data values and data distribution of the two dimensional array representing the slice are depicted. The three-dimensional graphs in the picture show data values of the slice after 1, 2, 4, and 8 V-cycles of the MG iteration, and depict how the algorithm converges.

The performance of the C/MPI and HPF/PST versions is compared in Table 5. The C/MPI version is faster because all communication is concentrated into one MPI message exchange, whereas in the HPF/PST version communication is dispersed, reflecting the structure of the sequential code. For the HPF/PST version we chose communication cache organization (2) (as described in Section 4.3).

7 CONCLUSIONS AND FUTURE WORK

Although the HPF standard (version 1.0) was finalized more than 2 years ago, only a few HPF and subset HPF compilers are commercially available. These compilers have not been accepted too well by scientific programmers, mostly because the compilers lack robustness and perform unpredictably on real-world applications. One of the reasons for this situation is the sophisticated compilation technology required for a language as complex as HPF. The compiler is responsible for both the computation mapping and the communication generation and optimization. Another reason is that some of the major research institutions which contributed to the definition of HPF and which have the required technology and expertise continue to develop systems know-how and keep on working on systems adapted to similar but different languages such as Fortran D and Vienna Fortran.

We believe that part of this acceptance problem can be overcome by relieving the compiler from the computational mapping task, at least for performance-critical code segments to be specified by the user. For increased acceptance we integrated our compiler in a tool environment featuring debugging and performance monitoring support, and we added directives for the support of unstructured problems. We have built a compiler prototype which is currently being used by pilot users. Their feedback and the performance results shown in Section 6 confirm that we are on the right way to develop an acceptable system which generates efficient code and is indeed easier to use than plain message passing.

Beside steadily increasing the robustness of our system, future work will include the consideration of other target platforms, in particular machines with shared memory and a larger number of vector processors. When compiling well-structured problems, the memory allocation strategy of our compiler is rather simple: Distributed arrays are simply replicated. We hope to change this allocation strategy to a more efficient buffer management in the future.

ACKNOWLEDGMENTS

We thank K. M. Decker, H. Katayama, R. Nakazaki, and T. Nakata for their valuable advice. S. Sakon, Y. Suzuki, Y. Hayashi, K. Kusano, Y. Shiroto, and Y. Watanabe helped to develop the HPF compiler. They also contributed to the performance evaluation presented in Section 6.1. The development of the Annai tool environment was a joint effort with C. Clémençon, A. Endo, J. Fritscher, and B. Wylie. Other members of the Joint CSCS/NEC Collaboration in

Parallel Processing, namely N. Masuda, W. Sawyer, V. Deshpande and F. Zimmerman, have continuously evaluated Kemari prototypes and given numerous suggestions which influenced the definition of our HPF extensions.

REFERENCES

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

[2] MPIF (Message Passing Interface Forum), "Document for a standard message-passing interface," Oak Ridge National Laboratory, Tech. Rep., TN, Apr. 1994.

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

[4] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka, "Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results," in *Proc. Supercomputing '93*, 1993.

[5] S. Hirandani, K. Kennedy, and C. W. Tseng, "Compiler optimizations for Fortran D on MIMD distributed-memory machines," in *Proc. Supercomputing 91*, 1991.

[6] C. Koelbel, P. Mehrotra, and J. van Rosendale, "Supporting shared memory data structures on distributed memory architectures," in *Second Symp. on Principles and Practice of Parallel Programming*, 1990, p. 177.

[7] M. Gupta and P. Banerjee, "Paradigm: A compiler for automatic data distribution on multicompilers," in *Proc. ACM Int. Conf. on Supercomputing*, 1993.

[8] R. Ponnusamy, Y.-S. Hwang, J. Saltz, A. Choudhary, and G. Fox, "Supporting irregular distributions in Fortran 90D/HPF compilers," University of Maryland, Department of Computer Science, Tech. Rep. CS-TR-3268, 1994.

[9] R. Rühl, *A Parallelizing Compiler for Distributed-Memory Parallel Processors* (PhD thesis, ETH-Zürich). Konstanz, Germany: Hartung-Gorre Verlag, 1992.

[10] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, "Vienna Fortran—A language specification," ICASE, Hampton, VA, Tech. Rep. 21, 1992.

[11] 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. Zimmerman, "Application-driven development of an integrated tool environment for distributed memory parallel processors," in *Proceedings of the First International Workshop on Parallel Processing, Bangalore, India, December 27–30, 1994*, V. K. Prasanna, V. P. Bhatkar, L. M. Patnaik, and S. K. Tripathi, Eds. New Delhi, India: Tata McGraw-Hill, 1994, pp. 110–116.

[12] C. Clémençon, A. Endo, J. Fritscher, A. Müller, R. Rühl, and B. J. N. Wylie, "The Annai environment for portable distributed parallel programming," in *Proceedings of the 28th Hawaii International Conference on System Sciences*, Vol. II, Maui, Hawaii, *3–6 January, 1995*, H. El-Rewini and B. D. Shriver, Eds. New York: IEEE Computer Society Press, 1995, pp. 242–251.

[13] C. Clémençon, J. Fritscher, and R. Rühl, "Visualization, execution control and replay of massively parallel programs within Annai's debugging tool," in *Proceedings of the High Performance Computing Symposium '95, Montréal, Canada*, V. Van Dongen, Ed. Montreal: Centre de recherche informatique de Montréal, 1995, pp. 393–404.

[14] A. Endo and B. J. N. Wylie, "The Annai/PMA performance monitor and analyzer," in *Proc. of MASCOTS'96*, 1996. (To appear as part of the Tools Fair.)

[15] T. Kamachi, K. Kusano, K. Suehiro, Y. Seo, M. Tamura, and S. Sakon, "Design and implementation of an HPF compiler and its performance results on Cenju-3 (in Japanese)," in *Proc. of Joint Symp. on Parallel Processing*, 1995, p. 361.

[16] K. Suehiro, K. Kusano, T. Kamachi, Y. Seo, M. Tamura, and S. Sakon, "Computation mapping in an HPF compiler (in Japanese)," in *Proc. of the Joint Symp. on Parallel Processing*, 1995, p. 369.

[17] A. Müller and R. Rühl, "Extending high performance Fortran for the support of unstructured computations," in *Proceedings of the 9th International Conference on Supercomputing '95 (Barcelona, Spain)*, ACM, 1995, pp. 127–136.

[18] B. Kernighan and D. Ritchie, *The M4 Macro Processor*, Murray Hill, NJ: Bell Laboratories, 1977.

[19] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "A user's guide to BLACS v1.0," World-Wide Web documentation ⟨http://www.netlib.org/scalapack/index.html⟩, 1994.

[20] H. Gerndt, "Automatic parallelization for distributed-memory multiprocessing systems," PhD thesis, University of Bonn, Tech. Rep. ACPC/TR 90-1, Austrian Center for Parallel Computation, 1989.

[21] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S. Teng, "Optimal evaluation of array expressions on massively parallel machines," *ACM Trans. Prog. Languages Systems*, Vol. 17, pp. 123–155, Jan. 1995.

[22] P. Havlak and K. Kennedy, "Experience with interprocedural analysis of array side effects," in *Proc. Supercomputing '91*, 1991.

[23] C.-W. Tseng, "An optimizing Fortran D compiler for MIMD distributed-memory machines," PhD thesis, Rice University, Jan. 1993.

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

[25] 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*, 1991.

[26] R. Das, J. Saltz, and R. Hanxleden, "Slicing analysis and indirect access to distributed arrays," in *Proc. of the 6th Workshop on Languages and Compilers for Parallel Computing*, 1993.

[27] C. Chase and A. Reeves, "Data remapping for distributed-memory multicomputers," in *Scalable High Performance Computing Conf.*, 1992, p. 137.

[28] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman, "Performance of hashed cache data migration schemes on multicomputers," *J. Parallel Distrib. Comput.*, Vol. 12, pp. 415–422, 1991.

[29] Müller and R. Rühl, "Communication-buffer organisation and performance of data-parallel unstructured computations," in *Proceedings of the 3rd Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, Troy, New York, May 1995*, B. K. Szymansky and B. Sinharoy, Eds. Kluwer Academic, 1995, pp. 295–298.

[30] 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. Zimmerman, "Architecture and programmability of the NEC Cenju-3," *Speedup J.*, vol. 8, 1994.

[31] C. Pommerell, *Solution of Large Unsymmetric Systems of Linear Equations* (PhD thesis, ETH-Zürich). Konstanz, Germany: Hartung-Gorre Verlag, 1992.

[32] C. Pommerell and R. Rühl, "Migration of vectorized iterative solvers to distributed memory architectures," *SIAM J. Sci. Comput.*, vol. 17, Jan. 1996 (in press).

[33] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon, "NAS parallel benchmarks results," NASA Ames Research Center, CA, Tech. Rep. RNR-92-002, Dec. 1992.

[34] B. Chapman, H. Zima, and P. Mehrotra, "Extending HPF for advanced data-parallel applications," *IEEE Parallel Distrib. Technol.*, vol. 2, pp. 59–70, Fall 1994.