Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC*

Siegfried Benkner, Kamran Sanjari, Viera Sipkova, Bob Velkov

Institute for Software Technology and Parallel Systems
University of Vienna
Liechtenstein Strasse 22, A-1090 Vienna, Austria

Abstract. Applications based on highly irregular, dynamically changing data structures cannot be handled efficiently with current High Performance Fortran and available HPF compilers. In order to support such applications, extensions of the language as well as sophisticated analysis and parallelization techniques are required. In this paper we introduce the Vienna HPF+ Compiler (VFC), a new source-to-source parallelization system for HPF+, an optimized version of HPF, which addresses the requirements of advanced irregular applications. In addition to extended data distribution and work distribution mechanisms, VFC supports various features of HPF+ to reduce preprocessing and communication overheads. The system provides powerful runtime parallelization strategies for complex loop nests with indirect array accesses which are common in irregular codes. Performance measurements show that the use of language features for communication schedule reuse results in a performance close to hand-written message passing programs.

1 Introduction

In this paper we describe the Vienna HPF+ Compiler (VFC), with a special focus on language features and parallelization techniques for irregular applications. VFC is a source-to-source transformation system that translates Fortran95/HPF+ programs to Fortran90/MPI message passing programs. HPF+ is an improved variant of HPF for advanced irregular applications which has been developed in the ESPRIT Long Term Research Project "HPF+". HPF+ provides important concepts for the parallelization of advanced industrial applications which are currently not adequately supported by the de-facto standard HPF-2 and commercial HPF compilers.

VFC implements most features of HPF+, including the generalized block and the indirect data distribution formats, distributions to processor subsets, dynamic data redistribution, communication schedule reuse, and halo regions. Unlike most other HPF compilers, VFC provides efficient parallelization strategies for a large class of non-perfectly nested loops with irregular runtime-dependent access patterns which are common in industrial codes. In this context, the concept of communication schedule reuse, allows the user to reduce or amortize the potentially large overhead of the associated runtime compilation strategies. The supply of important modularization features of Fortran such as procedure interfaces, internal procedures, and modules significantly alleviates the parallelization of large programs. Performance analysis and tuning is supported by

* The work described in this paper was partially supported by the ESPRIT IV Long Term Research Project 21033 "HPF+" of the European Commission.

an integrated instrumentation system which enables the user to generate MPI trace-files for selected parts of the program.

The paper is organized as follows. In Section 2 we describe the overall structure of VFC followed by an overview of the supported language features. In Section 3 we present the main parallelization techniques implemented in VFC. Section 4 presents performance result of two irregular HPF+ kernels developed in the "HPF+" project. In the remaining Sections we discuss related work and future extensions of VFC.

2 The VFC System

VFC has been developed within the ESPRIT project "HPF+" at the University of Vienna. The system is based on a Fortran 90 frontend initially developed at GMD Berlin and extended at the University of Vienna. Besides the main implementation languages C and C++, the Cocktail compiler construction toolbox has been used extensively. A high-level approach has been adopted for implementing the various parallelization techniques based on the pattern-matching and tree-transformation tool PUMA.

2.1 System Components

The VFC system consists of six major components: the frontend, the analysis module, the parallelization module, the instrumentation system, the code generator, and the runtime system.

The *frontend* performs scanning and parsing of the input program, and constructs an intermediate representation of the program in the form of an abstract syntax tree. The intermediate representation is formally specified based on the AST tool.

The analysis module constructs different abstractions of the program such as the program dependence graph, the call graph and a flow graph for each program unit. These data structures are the basis for data dependence analysis and for an advanced interprocedural analysis framework [18] which is currently being developed.

The *parallelization module*, which transforms the input program into a parallel message passing SPMD program, consists of several components. The *normalization component* performs various transformations such as array and loop normalization, normalization of data distribution directives, substitution of function calls in expressions that reference distributed objects, normalization of procedure interfaces, and others. The task of the *classification component* is to determine the parallelization strategy that should be applied to each executable construct. A generalized inspector/executor strategy is utilized for parallelizing irregular loop nests. A parallelization technique based on intersection of regular sections is applied to regular loops and explicitly parallel constructs such as array assignments. A default strategy utilizing single-element send/receive/broadcast operations, is applied either to scalar assignments or to non-parallelizable loops. Depending on the availability of analysis information or the presence of user assertions, different variants of these strategies are applied. All parallelization strategies have been implemented based on the PUMA tool.

The *instrumentation component* inserts calls to a measurement library in order to generate MPI trace-files during execution of the program. The instrumentation can be triggered by the user by means of command-line options, and it supports selective program instrumentation, in order to provide the user with detailed information about the execution of the parallelized code.

The *code generator* translates the internal representation of the program into Fortran 90 source code, which is then submitted to the Fortran 90 compiler of the parallel target machine in order to generate the executable parallel program.

The runtime system of VFC consists of two parts: the VHPF and the PARS libraries. The main task of the VHPF library is the management of distributed objects at runtime, including runtime support for dynamic data redistribution. Moreover, the VHPF library provides high level interfaces for communication primitives of the PARS library calls which significantly alleviate code generation. The PARS library provides communication routines utilizing calls to the MPI-1 library [20] and supports all aspects of the inspector/executor runtime parallelization strategy, including the support for schedule reuse and halos.

2.2 Input Language

The input language of VFC is Fortran 95 with HPF+ extensions. Whereas VFC supports full Fortran 95 in the context of non-distributed data objects, the system imposes certain restrictions on the transfer of distributed arguments at procedure boundaries, the input/output of distributed data, and the use of intrinsic functions. In particular, VFC neither supports the reshaping of distributed arrays at procedure boundaries nor the transfer of sections of distributed arrays to procedures.

HPF+ is an improved variant of HPF for advanced irregular applications. HPF+ not only supports the de-facto standard HPF-2 but also many features that have been adopted by the Approved Extensions of HPF-2. In addition HPF+ simplifies the distribution model of HPF without sacrificing required functionality, and introduces new features to provide the compiler with essential information on the applicability of crucial optimizations. In the following we give a brief overview of the HPF+ language features supported by VFC. For a more detailed description of HPF+ the reader is referred to [3].

2.2.1 Processor Arrays Arbitrary multi-dimensional processor arrays may be defined. The program may be parallelized for an arbitrary number of processors by using the intrinsic function `NUMBER_OF_PROCESSORS()` in processor declarations. An explicit relationship between different processor arrays may be established by using *processor views*. This feature enables the user to define named processor array sections or to permute the dimensions of multi-dimensional processor arrays by means of the `RESHAPE` intrinsic function.

2.2.2 Data Distribution Features In addition to the HPF-2 distribution mechanisms such as block, cyclic, block-cyclic distributions, and replication, VFC provides generalized block and indirect distributions. Distributions may be specified with respect to regular subsections of processor arrays. A simplified form of alignments may be used to specify groups of arrays that are to be distributed together. Dummy arguments may either be explicitly distributed using the above mechanisms or may inherit their distribution from the corresponding actual arguments. All distributions mentioned above may be defined for statically allocated as well as *dynamically allocated* objects such as allocatable arrays or pointers. Moreover, VFC supports *dynamic data redistribution* either by means of the REDISTRIBUTE directive or by redistribution as a result of procedure calls.

2.2.3 Independent Loops

The INDEPENDENT directive may be used to assert that the iterations of a subsequent DO loop are independent and, therefore may be executed in any order. In this context VFC supports the NEW clause for defining variables that are private to a loop iteration, the REDUCTION clause for performing reduction operations, and the ON HOME clauses to specify how the loop iterations should be distributed.

2.2.4 Communication Schedule Reuse

The REUSE clause may be specified in the context of an independent loop to indicate to the compiler that the communication schedules are invariant for all distributed arrays accessed in the loop. As a consequence, the schedules have to be computed only once and may be reused in subsequent executions of the loop. The reuse clause may be combined with a scalar logical expression in order to support conditional communication schedule reuse.

2.2.5 Halos

The (super) set of non-local elements of a distributed array that may be accessed by a processor during the execution of a program unit may be specified by means of *halo regions*. A halo region is an irregular, non-contiguous area describing the non-local access pattern for each processor. Halo regions are specified by means of the HALO attribute which may be combined with distribution specifications. The information provided by halo regions can be utilized by the compiler to allocate memory for all non-local accesses, to establish a corresponding address translation scheme, and to compute the required communication schedule. In particular, the use of halo regions may improve the parallelization of irregular independent loops significantly.

2.2.6 Locality Assertions

In order to call procedures which operate exclusively on data private to a processor from within independent loops without introducing any overheads, VFC supports the concept of *purest procedures*. By specifying the PUREST attribute, the user asserts to the compiler that the called procedure is pure (free of side effects) and that its invocation does not result in communication on entry, during execution, and upon exit from the procedure. This information, which is usually difficult to be determined automatically, supports the compiler in generating efficient code for independent loops despite the presence of procedure calls.

Moreover, the RESIDENT clause may be specified in the context of independent loops to indicate that a loop may be parallelized without the need for generating communication.

2.2.7 Extrinsics

VFC supports the extrinsic interfaces of HPF-2, HPF_LOCAL and HPF_SERIAL, by implementing the associated argument transfer mechanisms and the most basic functions from the HPF_LOCAL library such as MY_PROCESSOR, GLOBAL_SIZE, GLOBAL_SHAPE and a few others.

3 Parallelization Techniques

The parallelization strategy of VFC is based on the *SPMD* programming model. VFC translates a source program into an SPMD message passing target program, which is parameterized in such a way that it can be executed on an arbitrary number of processors[2]. At runtime each processor of the target machine executes the same target program operating on its own data.

3.1 Overview of the Parallelization Strategy

The support of advanced concepts, in particular dynamic redistribution and inheritance of distributions by formal arrays, requires a parallelization strategy which is able to deal with *multiple reaching distributions*. In order to handle these features efficiently, VFC provides, besides additional language support and analysis techniques, a sophisticated management scheme for distributed arrays, as well as advanced parallelization strategies and runtime support.

3.1.1 Distribution Analysis and Management of Distribute Arrays

VFC analyzes the distribution specifications and generates calls to runtime routines that initialize the *layout descriptor* of each distributed array. Layout descriptors contain all information concerning the distribution of an array (or a class of arrays) together with global and local shape information and auxiliary information needed, for example, to allocate buffers for storing non-local data. Layout descriptors and the associated runtime routines have been designed in such a way that it is possible to use a single descriptor for arrays that have the same, possibly run-time defined or dynamic, distribution. Moreover, the possibility to update layout descriptors incrementally, significantly simplifies the implementation of distributions in the context of dynamic memory allocation and dynamic data redistribution.

3.1.2 Transformation of Procedure Calls In order to implement the complex transfer mechanisms of distributed arguments at procedure boundaries VFC analyzes all procedure calls and determines whether the distribution of an actual argument and the corresponding dummy argument are different, or whether a distributed actual argument is associated with a non-distributed actual argument or vice versa. This analysis utilizes information from interface blocks (if present) with the goal to avoid implicit redistributions or copy-in/copy-out argument transfers whenever possible.

In cases where actual and formal distribution match or the formal argument inherits the distribution, calls are transformed in such a way that each distributed argument is passed together with its layout descriptor without the need of redistribution or copying.

If, for example, a distributed actual argument is associated with a non-distributed formal argument, or a dummy with a different distribution, an implicit redistribution replicating the actual argument is generated prior to the call, and after the call if it is required to restore the original distribution. Whenever such a redistribution is generated, or arguments have to be copied, the user is informed about this potential runtime overhead by corresponding diagnostic messages.

[2] This is only true if the program has been written in such a way that it is not dependent on a specific number of processors.

3.2 Parallelization of Array Assignments

Before parallelizing array assignments which involve distributed objects, VFC analyzes the code to determine the parallelization strategy that should be applied. Currently three different classes of array assignments are distinguished and handled with varying overheads.

The first class of array assignments can be handled in an optimal way without the need for any transformations. This requires that each dimension of all arrays on the right-hand-side with a non-scalar subscript is distributed in exactly the same way as the associated dimension of the left-hand-side array and that all distributed dimensions of all arrays involved are accessed as a whole.

The second class comprises array assignment statements for which only work distribution and index conversion has to be performed, but which can be processed without introducing inter-processor communication. This strategy is applied if, for all arrays on the right-hand-side, each dimension with a non-scalar subscript is distributed in exactly the same way as the associated dimension of the left-hand-side array. Such statements are transformed by adapting the section-subscripts of distributed dimensions in such a way that each processor only accesses its local data. These local section-subscripts are determined at runtime by intersection of the local index set of an array dimension with the original global section subscripts and converting the resulting section into local indices.

All other array assignment statements are transformed by introducing appropriate calls to runtime library routines that determine the work distribution and generate the required communication of non-local data into temporary arrays.

Unlike most HPF compilers, array statements are not converted into loops, but are transformed to array statements that operate on the local memory on each processor. The main advantage of this strategy is that the parallelism of the original construct is preserved. Therefore, both inter- and intra-processor parallelism can be exploited by the Fortran 90 compiler of the parallel target machine.

3.3 Parallelization of Irregular Loops

Compiling real world codes requires the ability to parallelize complex loop nests. In contrast to most existing HPF compilers, VFC can handle non-perfectly nested independent loops efficiently. Such loop nests may contain references to arrays that are distributed in more than one dimension, conditional statements that access distributed data, reduction statements, and procedure calls. In addition, the information provided by means of the `HALO` directive or the `REUSE` and `RESIDENT` clause is utilized by VFC to perform crucial optimizations.

Previous work to parallelize loop nests accessing arrays distributed in more than one dimension [9] was based on array linearization. Our approach for processing such loop nests is based on a dimensional-wise collection of references and hence does not incorporate such aggressive loop restructuring methods.

In general, irregular loop nests are transformed by VFC into three phases: the *work distribution* phase, the *preprocessing or inspection* phase and the final *execution* phase. Depending on whether distributed arrays appear in the body of conditional-statements, or if they are accessed by means of distributed indirection arrays, VFC applies different variants of this strategy.

3.3.1 Work Distribution In this phase the iterations of a loop nest are assigned to the available processors based on the `ON HOME` clause. Depending on the distribution of the array in the `ON HOME` clause, different work distribution strategies are applied. For regular distributions (i.e. `BLOCK(M)`, `CYCLIC(M)` and `GEN_BLOCK`) the distribution of the loop iterations is determined by intersecting the iteration space of each loop with the local index set of the array in the on-clause using analytical formulas. On the other hand, if the array in the on-clause has an irregular distribution, the local iterations for each processor are determined by a loop that checks the ownership of the array reference in the `ON HOME` clause for all iterations.

3.3.2 Inspector The major task of the inspector is to determine the access patterns of distributed arrays accessed in a loop nest and to construct the corresponding communication schedules for non-local accesses. Schedules are data structures representing the communication patterns for non-local accesses to distributed arrays in a parallel loop. The traditional inspector approach needed to be redesigned and extended, in order to translate non-perfectly independent loops containing conditionals and arrays with multiple distributed dimensions.

In order to generate the inspector code, VFC usually generates a separate loop nest for each dimension of a distributed array to store the global indices on each processor according to the work distribution in a derived type variable `global_box`. The depth of this loop nest depends on the number of loop variables referenced in the subscript expression of the corresponding array dimension. After having processed all dimensions of the distributed array, `global_box` represents all accesses to the array on each processor. `global_box` and the runtime descriptor of the array are then passed to the PARS primitive `dview`. For each dimension `dview` transforms the global indices in `global_box` to the local indices according to the distribution information stored in the array's runtime descriptor and returns it in the derived type variable `local_box`.

The localized references refer either to the local section of the array on each processor, or to the buffer, if they are non-local. The last remaining step is to form the Cartesian product of the localized references, determine the non-local accesses, and generate the corresponding communication schedule.

Multi-level indirections are handled by a multi-phase Inspector/Executor approach. First the schedules are computed for all of the distributed index arrays at the deepest level. They are then used for gathering non-local data for the indirection arrays at the next higher level. This process is repeated until a top-level data array is reached, which is processed as described above.

3.3.3 Executor In the executor phase all non-local data read in the loop nest are gathered according to the computed schedule, and the restructured loop nest is executed. In case non-local data has been written, a final scatter communication is performed.

VFC generates the executor by restructuring the original loop nest in several steps as follows. The iteration space of each independent loop is changed according to the work distribution. Subscript expressions in distributed dimensions of top-level arrays are replaced by the corresponding local indices stored in `local_box`. Finally, depending on whether non-local data is read or written, calls to gather and scatter communication routines are inserted before and after the loop nest, respectively.

3.3.4 Schedule Reuse

Since the inspector phase is usually very time consuming, it has to be avoided whenever possible. Code motion techniques to hoist the inspector code out of time-step loops or even procedures, are too restrictive to be applicable for real codes. Schedule reuse in VFC is currently based on user support. If the `RESUE` clause is specified for an independent loop, the inspector is guarded by a conditional statement, which, in case of conditional schedule reuse, is dependent on the logical expression specified by the user in the `REUSE` clause.

3.3.5 Optimizations Based on Halos

A halo specifies all non-local references to a particular distributed array for the duration of the program execution. At the time the distribution of an array is determined, the information provided by a halo is used to allocate storage for non-local data on each processor and to compute a communication schedule. This schedule is then used for all occurrences of this array in the program. As a consequence, references to such an array do not require an inspector phase. Moreover, redundant communication can be avoided, since the user notifies the compiler with a `UPDATE_HALO` directive, whenever the halo area has to be updated.

3.3.6 Optimizations Based on the `RESIDENT` Clause

If the user asserts that an independent loop can be processed without communication, by specifying the `RESIDENT` clause, the inspector phase can be significantly optimized. In this case calls to `dview` and gather/scatter calls are avoided. Instead, a call to an optimized index conversion routine is generated.

3.4 Translation of distributed-scalar assignments

Scalar assignments to elements of distributed arrays outside independent loops are transformed into conditional statements to enforce the owner computes rule. If non-local data is accessed, send/receive communication statements are inserted to transfer these data to the processor executing the assignment. Assignments to replicated scalars are executed on all processors and may require the insertion of broadcast communication statements.

4 Performance Results

We present performance results on a Meiko CS-2 for two kernels which have been developed in the context of the HPF+ project. The HPF+ kernels have been parallelized with the VFC and the generated message passing programs have been compiled with the SUN F90 compiler. In Figures 1 and 2 the performance of these kernels is compared to hand-written MPI message passing programs.

The first kernel, developed by Guy Lonsdale, represents the basic stress-strain calculation of a crash-simulation code [13] based on 4-node shell elements. The kernel used in our evaluation employed an unstructured mesh with 25760 nodes and 25600 elements. The total size of the kernel is approximately 600 lines of code. It uses an explicit time-marching scheme which is represented by an outer time-step loop performing 1000 iterations. From within this loop, subroutines are called to calculate the shell-element forces and to update the velocities and displacements according to the computed forces. The subroutine performing the calculation of forces contains irregular loop nests which have to be processed with runtime parallelization techniques in order to determine

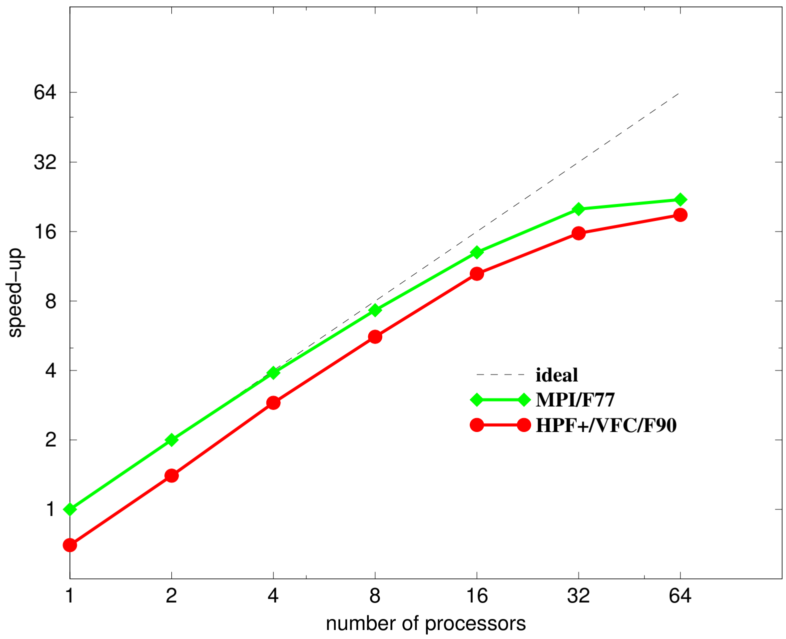

Fig. 1. HPF+ Crash Simulation Kernel. This diagram shows the speed-up of the hand-coded MPI message passing crash kernel and the equivalent HPF+ crash kernel. The MPI kernel was compiled with the SUN F77 compiler, while the VFC used the SUN F90 to compile the generated message passing program. The speed-up of the HPF+ kernel has been computed with respect to the HPF+ program compiled with the SUN F90 compiler and executed on a single node. The scalability obtained with HPF+ and VFC is quite satisfactory and close to the message passing kernel. Nevertheless the elapsed times of the HPF+ kernel are approximately a factor 3.5 slower than those of the MPI kernel. However, this overhead results to a large extent from the use of the Fortran 90 compiler to compile the SPMD program generated by VFC. Experiments indicate that the use of the Fortran 90 compiler amounts to a penalty factor of approximately 2.

the required communication schedules. Once computed, these schedules can be reused for all time-steps. In the code this is expressed by means of the `REUSE` clause.

The second kernel, developed by George Mozdzynski, was extracted from the Integrated Weather Forecasting System (IFS) [2]. The kernel consists of 42 source files with a total of approximately 5000 lines of code and represents the transpositions of the IFS code performed in grid-point space and fourier space. The kernel used in our evaluation employed a grid point space at resolution T63 with a total of 13280 grid points. Grid point space and fourier space cannot be

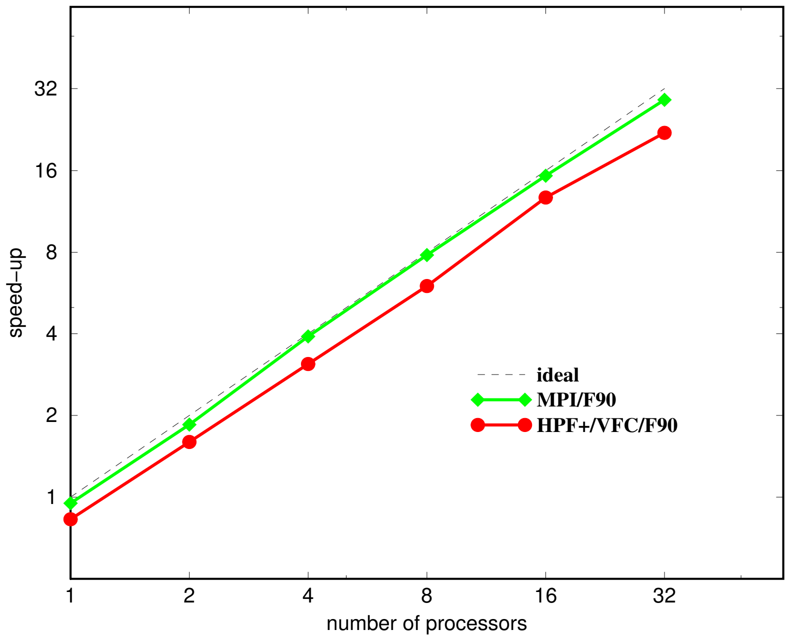

Fig. 2. HPF+ Weather Prediction Kernel. This diagram shows the speed-up of the hand-coded MPI message passing and the equivalent HPF+ kernel with respect to the same serial version of the kernel. Unlike the crash-simulation kernels, the SUN Fortran 90 compiler was used to compile both the MPI message passing program and the code generated by VFC. The HPF+ kernel does not perform quite as well as the message passing kernel, which shows near perfect scaling. The HPF+ code shows a speed-up of 25 on 32 processors which is still respectable. In terms of elapsed time the HPF+ kernel is only a factor 1.2 to 1.3 slower than the MPI kernel.

represented by simple `BLOCK` or `CYCLIC` distributions without the severe overhead of additional interprocessor communication during the algorithmic stages. Instead a combination of `GEN_BLOCK` and `INDIRECT` distributions is required for the grid point space, whereas the distribution of fourier space can be achieved by using a combination of `GEN_BLOCK` and `BLOCK`. The computational routines to perform the Fast Fourier Transforms and to simulate the physics calculations have been integrated based on `HPF_LOCAL` extrinsics. These routines are sequential and operate exclusively on data local to a processor and thus are not modified by the compiler. For the transpositions from grid point space to fourier space nested `INDEPENDENT` loops with indirect array accesses were used. Communication schedules of all irregular loops were computed only once in the first time-step and reused in subsequent time-steps based on the information provided by the `REUSE` clause.

5 Related Work

A number of research compilers for HPF and HPF-like languages have been or are being developed. In the following, some are briefly discussed. The PARADIGM compiler [1] is a source-to-source parallelizing compiler based upon Parafrase-2 which accepts a sequential Fortran 77 or High Performance Fortran (HPF) program and produces an optimized message-passing Fortran 77 parallel program.

The sHPF system [19] transforms a Subset HPF program into a message passing Fortran 90 program with calls to the ADLIB runtime library [10]. Parallel execution in sHPF is obtained from the use of Fortran 90 array syntax.

ADAPTOR [8] is a source to source transformation system for HPF that generates Fortran SPMD programs containing calls to the DALIB runtime system. Besides block and cyclic distributions ADAPTOR supports general block distributions.

HPFC [17] is Fortran 77 based prototype HPF compiler generating Fortran 77 message passing programs with calls to a PVM based runtime library. HPFC provides extensions to HPF for the specification of local code regions. Whereas both ADAPTOR and HPFC provide preliminary support for dynamic redistribution, none of these systems can handle multiple reaching distributions.

The VFCS [5], the predecessor of VFC, is a source-to-source transformation system for Fortran 77 with Vienna Fortran and HPF extensions. VFCS supports besides block and general block distributions restricted forms of indirect distributions. Unlike VFC, VFCS supports only static distributions and requires the number of processor to be fixed at compile-time.

6 Conclusions

In this paper we presented the Vienna HPF+ compiler VFC, a source-to-source parallelization system that transforms Fortran95/HPF+ programs to Fortran 90 message passing programs. VFC provides special support for an efficient handling of advanced codes based on highly irregular, possibly dynamically changing data structures and access patterns. VFC implements most features of HPF+, an optimized version of HPF for irregular codes. This includes block, cyclic, general-block and indirect distributions, processor views, distribution to processor subsets, dynamic data redistribution, and on-clauses. VFC offers powerful compilation strategies for nested loops with indirect accesses to arrays distributed in multiple dimensions, and performs important optimizations that decisively influence a program's performance such as the reuse of runtime generated communication schedules. Performance results with kernels extracted from advanced industrial applications indicate that the HPF paradigm, if suitably extended, may be successfully applied to irregular codes.

Future extensions of the VFC system will remove some of the current restrictions and include additional analysis and optimization techniques. The parallelization strategy for irregular loop nests will be extended to loops with a non-rectilinear iteration space. Furthermore, the support for distributed arrays of derived types, as well as the distribution of components of derived types, will be extended. Task parallel features based on on-directives and task-regions and coordination mechanism as proposed in OPUS [12] are currently being implemented. For an efficient handling of such features and for other optimizations we plan to utilize *one-sided communication* primitives of MPI-2 [20].

References

1. P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, E. Su. The PARADIGM Compiler for Distributed-Memory Multicomputers, IEEE Computer, Vol. 28, Nr. 10, October 1995.
2. S. Barros, D. Dent, L. Isaksen, G. Robinson, G. Mozdzynski, and F. Wollenweber. The IFS model: A parallel production weather code. Parallel Computing 21, 1995.
3. S. Benkner. HPF+: High Performance Fortran for Advanced Industrial Applications. Proceedings HPCN'98, Amsterdam, April 1998.
4. S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima. High-Level Management of Communication Schedules in HPF-Like Languages. NASA Contractor Report 201740, ICASE, Hampton, VA, September 1997.
5. S. Benkner, et.al. Vienna Fortran Compilation System. Version 1.2. User's Guide, Univ. of Vienna, Inst. f. Software Technology and Parallel Systems, February 1996.
6. S. Benkner. Handling Block-Cyclic Distributed Arrays in Vienna Fortran 90. IEEE Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'95) Limassol, Cyprus, June 1995
7. S. Benkner. Vienna Fortran 90 and its Compilation. Ph.D. Thesis. TR 94-8, University of Vienna, Institute for Software Technology and Parallel Systems, 1994.
8. T. Brandes, F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs, In: Programming Environments for Massively Parallel Distributed Systems, Birkhaeuser Verlag, pp. 91-96, April 1994.
9. Brezany,P., Cheron,O., Sanjari,K., van Konijnenburg,E.: Processing Irregular Codes Containing Arrays with Multi-Dimensional Distributions by the PREPARE HPF Compiler. In Proceedings of the International Conference on High Performance Computing and Networking, Milan, May 1995.
10. Carpenter, B. Adlib: A Distributed Array Library to Support HPF Translation, Proc. 5th Workshop on Compilers for Parallel Computers, Malaga, June 1995.
11. B. Chapman, H. Zima, P. Mehrotra. Extending HPF for Advanced Data Parallel Applications. IEEE Parallel and Distributed Technology, Fall 1994.
12. B. Chapman, H. Zima, M. Haines, P. Mehrotra, J. Van Rosendale. OPUS: A Coordination Language for Multidisciplinary Applications. Journal of Scientific Programming, 1995.
13. J. Clinckemaillie, B. Elsner, G. Lonsdale, S. Meliciani, S. Vlachoutsis, F. de Bruyne, M. Holzner. Performance Issues of the Parallel PAM-CRASH Code. Journal of Supercomputing Applications and High-Performance Computing, Vol.11, No.1, 1997.
14. High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 1.1* TR, Rice University, November 10, 1994.
15. High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 2.0* TR, Rice University, January 31, 1997.
16. ISO. Fortran 90 Standard, May 1991, ISO/IEC 1539 :1991 (E)
17. R. Keryell, C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, and P. Jouvelot. PIPS: A Framework for Building Interprocedural Compilers, Parallelizers and Optimizers. Technical Report 289, CRI, Ecole des mines de Paris, April 1996.
18. J. Knoop and E. Mehofer. Interprocedural distribution assignment placement: More than just enhancing intraprocedural placing techniques. IEEE Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'97) San Francisco, CA, November 1997.
19. J. H. Merlin, D. B. Carpenter and A. J. G. Hey. SHPF: a Subset High Performance Fortran compilation system. Fortran Journal, pages 2-6, March/April 1996.
20. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 1.1, June 1995. MPI-2: Extensions to the Message-Passing Interface, July 1997.
21. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald. Vienna Fortran – a language specification. ICASE Internal Report 21, ICASE, Hampton, VA, 1992.