PPPE Deliverable D4.3.a

Comparison of HPF-like Systems

Deliverable Ref. No.	:	*D4.3.a*
Responsible Partner	:	*University of Southampton*
Contributing Partners	:	*GMD, University of Vienna*
Version	:	*1.1*
Date	:	*November 22, 1993*
Authors	:	*V. Getov[1], T. Brandes[2], B. Chapman[3], A. Dunlop[1], T. Hey[1], D. Pritchard[1]*
Availability	:	*Public*
Controlled Status	:	*Uncontrolled*
Review Status	:	*Pending Review*

[1]University of Southampton
[2]GMD
[3]University of Vienna

Executive summary

- This report describes our experience with some of the existing prototype HPF systems which were available to the members of the PPPE project. The comparison and evaluation of these early HPF compilers was equally difficult and important in order to gain some feedback to the compiler designers but also to accumulate some initial skills in writing efficient HPF programs.
- The available HPF-like systems were applied to the existing GENESIS benchmarks that have already been rewritten in Fortran 90 with HPF data distribution directives. First experiences have shown that HPF/Fortran 90 itself requires very good compilers as Fortran 90 programs are very often slower than their Fortran 77 counterpart when compiled with the existing early Fortran 90 compilers.
- The subset HPF compiler systems under consideration have been having different level of development during the evaluation exercise. Therefore the extent of evaluation results is different for different systems. Similarly, different subset HPF systems have been having different level of availability. For example, APR Inc. have been willing to perform an indirect evaluation of the xHPF90 compiler by themselves. They decided, however, not to disclose the time measurements and therefore we were not able to present benchmark results for xHPF90 in this report.
- It is absolutely necessary to give the user feedback about the generated parallel program from his code. He should be informed which statements will require communication and whether this communication is efficient or not. As HPF systems will not implement all possible optimizations in an early stage, the documentation should demonstrate how to implement efficient programs.
- Optimization strategies for data parallel program were and still are a topic of research. None of the early HPF systems has any of the number of optimization proposals implemented. Indeed, the benchmark results in section 5 showed performance losses within the range from 2 to more than 100 times in comparison to the corresponding hand-written PARMACS versions of benchmarks.
- The HPF parallelisation tools are still in their infancy. They still need some more time to develop in the areas of generality and performance before being used for commercial code parallelisation.

Contents

1 Introduction

The specification of High Performance Fortran (HPF) language [15] represents an extension to Fortran 90 [18] which addresses the problem of writing data parallel programs within the Single Program Multiple Data (SPMD) model of parallel computations. The main extension streams of the language are data distribution features (new directives ALIGN, DISTRIBUTE, TEMPLATE, PROCESSORS, etc.), concurrent execution features (FORALL construct and several new intrinsics) and EXTRINSIC procedures. In order to allow early compiler availability, subset HPF has been defined within the HPF language specification document. It is intended that subset HPF will be capable of being implemented more rapidly by vendors, which will ensure that subset HPF codes will be able to run at an early date on a variety of different machines. In general, subset HPF includes data distribution features, Fortran 90 array language, dynamic storage allocation, FORALL statement and long names. It does not include dynamic data distribution, FORALL construct, EXTRINSIC procedures, generic functions and the Fortran 90 free source form.

There are already some HPF-like systems available that make an automatic partitioning of data parallel Fortran 90 programs. These include:

- ADAPTOR is a tool developed at the GMD, Germany;
- The Fortran 90D compiler has been developed at the University of Syracuse;
- FORGE-90/xHPF90 is an automatic paralleliser for HPF developed by Applied Parallel Research, Inc.;
- The Vienna Fortran compilation system developed at the University of Vienna.

These tools were applied to existing applications (GENESIS Benchmark Suite) that have already been rewritten in Fortran 90 with HPF data distribution directives. The goal of the task is to accumulate some early experience, which would allow us to find out how efficient such automatic partitioning tools are, for which kind of applications they are useful and for which applications they fail.

This report gives a detailed description of the experiences, possible improvements for the tools, hints how to write efficient applications that can be translated with these tools, and a qualification whose applications are not suitable for an efficient translation. The report also presents a comparison of capabilities for the prototype systems listed above and recommendations for a competitive HPF mapper [17]. The evaluation results are meant as input into NAS's work on HPF (WP 4).

2 Description of prototype HPF-like systems

2.1 Adaptor

Adaptor (Automatic Data Parallelism Translator) is an HPF system developed at the GMD for translating data parallel Fortran programs into equivalent Fortran 77 message passing programs [6, 7]. Most features of Connection Machine Fortran (CMF) and many features of HPF are supported.

The central idea of the automatic translation is to distribute the large data structures like arrays among the available processors. This should be performed in such a way that most operations could be done locally without need of communication. The corresponding message passing statements are inserted automatically where global operations are necessary. Therefore the data parallel program with global data references is translated together with a user specified or implicitly defined data distribution, into a program with local and non-local references, where the latter are satisfied by automatically inserted message-passing statements.

Although the user will need to understand some issues of parallelism and has to know for efficiency reasons where message passing will be generated, the effectiveness of Adaptor is based on the fact that the user does not have to know any message passing commands, neither does he have to manage the control of the data partitioning. The programmer can change types of variables (e.g. single to double precision) and data distributions without rewriting any other statements in the program. There is no need to write two versions of code (host and node program) and many global array operations are translated to the most efficient code for the underlying architecture.

The parallel program can be written in such a way that it could be developed on a serial machine and is also suitable for vector machines or parallel machines with shared memory. Many features supported by Adaptor result also in good execution times for these architectures. In this way, it helps to design programs that run efficiently on nearly all architectures.

Adaptor only takes advantage of the parallelism in the array operations and of the parallel loops. It has no features for automatic parallelisation.

The source language of Adaptor can be defined briefly in the following way:

- Fortran 77
- the array extensions of Fortran 90 (inclusive dynamic arrays and intrinsic functions for arrays),
- layout directives to specify the data distribution,
- features of Connection Machine Fortran and High Performance Fortran (`FORALL` statement, new intrinsics, timing, random numbers).

The user can define host arrays, replicated arrays and distributed arrays. Similar directives as in CMF or HPF are also used for the specification of data layouts in Adaptor. As well as that, the parallel `FORALL` statement supported by Adaptor has the same syntax and semantics as proposed in these data parallel languages.

Many features of Fortran 90 and HPF cannot be used with Adaptor. The most serious restrictions are that Adaptor supports no modules, no pointers, no array-valued functions and no assumed-shaped arrays. Adaptor only supports block distributions along one dimension. More distributions will be supported in future releases.

Explicit alignment can be used in CMF and HPF to reduce communication. For Adaptor this feature has not been supported until now, but of course there is an implicit alignment of arrays that are declared and distributed in the same way.

The user can select between the following three programming models:

- If the HOST-NODE programming model is selected, Adaptor will generate a host program and a node program. The node program runs on all available nodes of the parallel machine, while the host program contains all I/O operations that will be executed on the front end system.
- In the HOSTLESS programming model, only one program will be generated, running on all available nodes. There is no host program. The first node takes care of all I/O operations.
- A program that runs only on a single node is generated when using the UNIPROC programming model. The program has no communication and therefore it ought to be faster than

the previous one running on a single node. By choosing this model, programs with array operations, not available in Fortran 77 can be translated into sequential Fortran 77 programs.

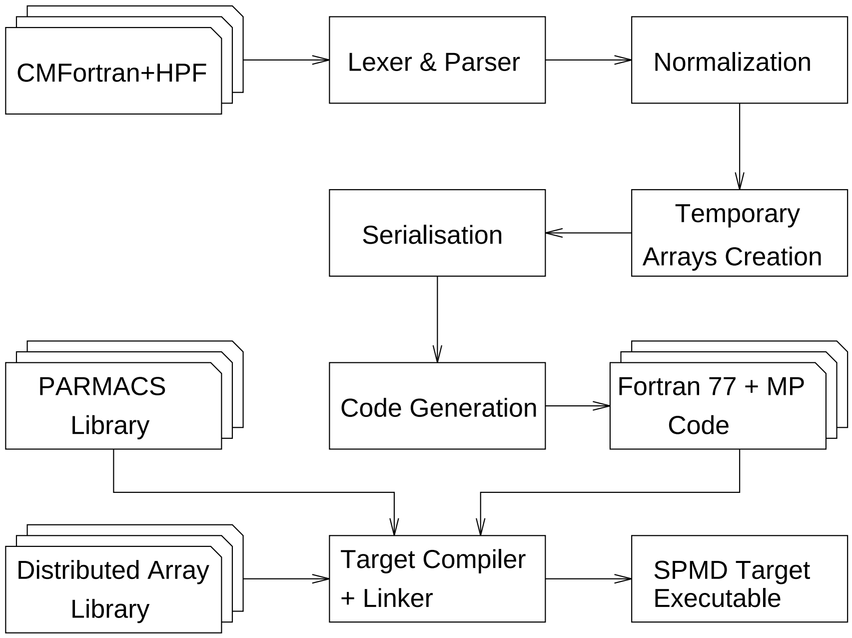


Figure 1: Overview of Adaptor

The following steps are done during the source to source transformation of Adaptor (see Figure 1):

1. The source program is parsed and an abstract syntax tree is generated. Symbol tables are created and used for a semantic analysis.
2. The program (abstract syntax tree) is normalized as far as possible to reduce the complexity of the translation system.
3. The program statements are splitted up into local and non-local operations. Temporary variables are created where necessary.
4. The array operations are translated into `FORALL` statements in the serialisation phase and the parallel loops are analysed.
5. The parallel loops are restricted to the local part owned by every processor in the final translation phase and communication statements for exchanging non-local data are generated. The new internal abstract syntax tree is unparsed back to source text.

Adaptor generates also a `Makefile`. Compiling and linking the generated message passing programs is performed by the Fortran compiler available on the target parallel machine. In order to perform the communication needed for global operations on distributed arrays, many library functions that build the distributed array library (DALIB) are implemented. This library contains:

- low level communication (send, receive, wait, etc.);
- high level communication (broadcast, reduction, barrier, etc.);

- data movements based on regular and irregular communication patterns;
- timing functions and tracing facilities;
- a parallel random number generator.

DALIB is implemented in C and can be considered as the runtime system of the whole compilation system. Most of the routines in this library are portable between different parallel platforms. Only the low level message passing commands, the timing functions and the random number generator have to be adapted to the target hardware architecture.

The following parallel machines are currently supported: CM-5, iPSC/860, Meiko CS1 and CS2, Parsytec GCel, KSR 1, SGI Multiprocessor Systems, and Alliant FX/2800. One version of DALIB is also implemented upon the public domain software PVM [11]. It guarantees the portability of the generated parallel programs to all machines where PVM is available.

The functionality and stability of Adaptor has improved dramatically if compared with version 0.1 from October 1992. Now the tool can also be used to translate a data parallel program to a serial Fortran 77 program. The source files of the current version 1.1 (September 1993) of Adaptor, the documentation files in PostScript and a number of example programs are available via 'anonymous ftp'. The address of the ftp-server is `ftp.gmd.de` (129.26.8.90), the files are in the directory `gmd/adaptor`.

Adaptor is a prototype compilation system for High Performance Fortran. Although the current version of Adaptor has many restrictions, the system gave useful insights in how to design efficient HPF compilers and to develop and integrate useful optimization strategies. Many users could take advantage of the Adaptor tool to develop future HPF applications.

2.2 Fortran 90D compiler

Fortran 90D is a Fortran 90 version of the Fortran D language [10]. Since Fortran D essentially adds to Fortran 90 extensions for data partitioning very similar to HPF, the authors from Syracuse University call their compilation system the Fortran 90D/HPF compiler. This compiler exploits only the parallelism in the data parallel constructs. It does not attempt to parallelise other constructs, such as do-loops and while-loops, since they are used only as naturally sequential control constructs in this language. The foundation of the design lies in recognizing commonly occurring computation and communication patterns. These patterns are then replaced by calls to optimized run-time support system routines. The run-time support system includes parallel intrinsic functions, data distribution functions, communication primitives and several others miscellaneous routines.

Figure 2 shows the basic components of the Fortran 90D compiler. The first step of the compilation is to check the syntax correctness of the source code. Given syntactically correct Fortran 90D/HPF program, the second step of the compilation is to generate a parse tree. The front-end module for the compiler, which incorporates both lexer and parser for Fortran 90 programs has been obtained from ParaSoft Corporation. In this module, the compiler also transforms each array assignment statement and where statement into equivalent FORALL statement with no loss of information. In this way the subsequent steps need only deal with FORALL statements.

The partitioning module processes the subset HPF data distribution directives; namely, decomposition distribute and align. Using these directives, it partitions data and computation (only FORALL statements) among processors. After partitioning, the parallel constructs in the node program are sequentialised since it will be executed on a single processor. The Fortran 90D compiler has to sequentialise Fortran 90 parallel constructs into sequential loops as there are few Fortran 90 compilers available for the nodes of MIMD distributed memory computers. This is performed by

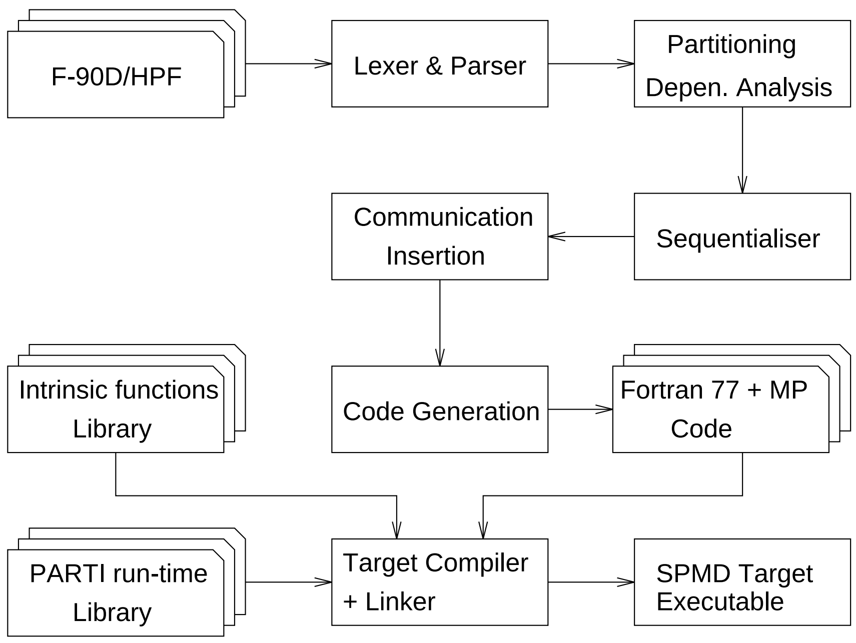


Figure 2: Steps in creating a SPMD program using the Fortran 90D/HPF compiler

the sequentialisation module. Array operations and FORALL statements in the original program are transferred into loops or nested loops.

The communication module detects communication requirements and inserts appropriate communication primitives. The Fortran 90D compiler produces calls to collective communication routines instead of generating individual processor send and receive calls inside the compiled code. There are three main reasons for using collective communication to support interprocessor communication in the compiler:

- Improved performance of HPF programs - this can be achieved by developing a separate library of interprocessor routines where each routine can be optimized;
- Increased portability of the Fortran 90D compiler - by separating the communication library from the basic compiler design only the machine specific low-level communication calls need to be changed when porting the compiler to a new platform;
- Improved performance estimation of communication costs - the costs of collective communication routines can be determined more precisely, thereby enabling the compiler to generate better distributions.

Finally, the code generator produces loosely synchronous SPMD code. The generated code is structured as alternating phase of local computation and global communication. Local computations consist of operations carried out by each processor on the data in its own memory. Global communication includes any transfer of data among processors, possibly with arithmetic or logical computation on the data as it is transferred (e.g. reduction functions). In these circumstances, processes do not need to synchronize during local computation. But, if two or more nodes interact, they will be implicitly synchronized by global communication.

The run-time support system of the Fortran 90D compiler consists of functions which can be called from the node programs of a distributed memory machine. Intrinsic functions support many of the basic data parallel operations in Fortran 90. Not only do they provide a concise means of expressing operations on arrays, but they also identify parallel computation patterns that may be difficult to detect automatically. The intrinsic functions that may induce communications can be divided into five categories as follows:

- structured communications - CSHIFT, EOSHIFT
- reduction - DOTPRODUCT, ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM, MAXLOC, MINLOC
- multicasting - SPREAD
- unstructured communications - PACK, UNPACK, RESHAPE, TRANSPOSE
- special routines - MATMUL

The first category requires data to be transferred using less overhead structured shift communication operations. The second category of intrinsic functions requires computations based on local data, followed by the use of a reduction tree on the processors involved in the execution of the intrinsic function. The third category uses multiple broadcast trees to spread data. The fourth category is implemented using unstructured communication patterns. The fifth category is implemented using existing research on parallel matrix algorithms. The current implementation of Fortran 90D (β release, August 1993) has more than 500 parallel run-time support routines divided into two libraries - INTRIN (Intrinsic) and PARTI (Parallel Automatic Runtime Toolkit from ICASE) built on the top of Express message passing primitives [19]. Hence, in order to switch to any other message passing system, it will only be necessary to replace the calls to the communication primitives in the two libraries.

2.3 The FORGE 90 toolset

The FORGE 90 package consists of four modules that are currently available for user-guided or automatic parallelisation of Fortran programs. The baseline system is an analysis tool for Fortran 77 programs and is required in order to use any other modules. The remaining modules include a distributed memory parallelisation tool (DMP), and a subset HPF translator (xHPF90). These modules are discussed below.

2.3.1 FORGE 90 baseline

The baseline system is an extensive interprocedural analysis tool that can be seen to consist of four parts:

1. *Parser.* FORGE 90 contains a complete Fortran 77 syntax checker that is able to recognize most Fortran extensions for specific systems. If an error is detected it can be corrected interactively.
2. *Code Reformatter.* The code reformatter can significantly tidy old codes. In addition to the standard 'pretty printing' features, it includes more serious transformations such as declaring all variables, removal of unused labels, replacement of arithmetic IF's with 'IF ... THEN ... ELSE ... ENDIF' constructs, and the conversion of 'IF ... GOTO ...', to block IF. The extent of reformatting is under user control.

3. *Interprocedural Analysis.* On parsing the Fortran code, FORGE creates a database which can then be interrogated for extensive interprocedural analysis. Some interprocedural analysis features provided are listed below:

   - A static call chain can be shown.
   - Variable usage can be traced through the program, including through the use of common blocks and through equivalence variables.
   - All constant declarations can be displayed.
   - A common block grid showing the use of all common blocks in the program can be displayed.
   - Variables can be displayed by subroutine and side-effects shown.

   In addition to providing the above analysis features directly, FORGE allows the user to define templates to identify specific aspects of the code under analysis.

4. *Instrumenter/Profiler.* The instrumenter is an interactive profiler that inserts timing calls in the user code. Following a run, the statistics can be viewed showing the time spent in each loop and the average and maximum loop sizes. Following an instrumented run of the code, the interprocedural analysis tool is able to provide a dynamic call chain.

FORGE 90 also provides interactive facilities, which are integrated around the following databases:

- symbolic database - a digest of the complete program, recording the use of all variables, COMMON blocks, statements and subprograms;
- control flow database - a map of the control flow within each routine of the program, including conditional and unconditional transfers, and `DO` loops;
- data flow database - a record of the variable usage-definition chains between the basic code blocks of the program;
- parallel directives database - a schematic of the directives for data communication, loop distribution, and process synchronization.

2.3.2 Distributed memory paralleliser (DMP)

The DMP module assists the user in developing a SPMD parallel program from sequential Fortran 77 (see Figure 3). Parallelisation is achieved through user-guided data partitioning and loop distribution. Parallelisation normally follows an instrumented run of the sequential code using baseline FORGE 90. This highlights the loops that require the largest percentage of processing time. Using the baseline system, the arrays used in the loops can then be traced throughout the code and their use examined.

Through the use of the baseline system, the user is required to identify the data partitioning required and which loops to parallelise. The DMP does not provide any information as to which data should be partitioned or the dimension for decomposition. It is up to the user to decide which data to partition and whether it is to be a block or cyclic distribution. The data can only be partitioned along a single dimension, although this is due for change in the September release. Furthermore, multiple arrays can only be aligned if they are conformable.

The data distribution itself does not describe the parallelisation. The user is required to specify all loops that are to be parallelised. If loops are not marked as parallel they are replicated on all processors.

The data partitioning commands and loop distribution commands are stored separately from the sequential code. There is a preprocessor called 'pref77' which is used to create the SPMD program from the user's sequential code and the commands stored by the DMP. This has the advantage that the user is able to maintain the sequential code and execute the same parallelisation commands to result in a new parallel program.

The parallel program produced by 'pref77' contains calls to FORGE's message passing library. This code must then be compiled by the target system compiler and linked to a specific communication syntax. Currently FORGE have communication libraries for native INTEL, native nCUBE, PVM, EXPRESS and P4.

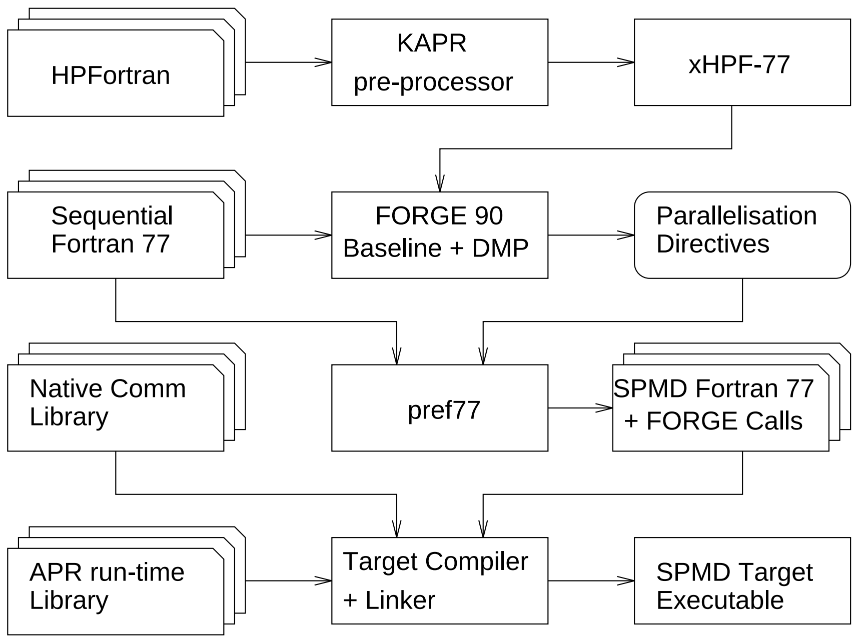


Figure 3: Steps in creating a SPMD program using the FORGE 90 system

2.3.3 xHPF90 translator

The xHPF90 Distributed Memory Paralleliser is the most recent part of the FORGE 90 parallel programming environment [4]. It performs automatic parallelisation of subset HPF programs. The xHPF90 system is identical to the xHPF77 system, with the addition that the Fortran 90 syntax is initially translated into Fortran 77. The xHPF90 system is limited to the Fortran 90 features identified by the HPFF.

Fortran 90 programs must first be converted into Fortran 77 and then the Fortran 77 code examined. This is performed by KAPR, a special modification of the KAP pre-processor from Kuck and Associates Inc. (KAI), which acts as a front-end for xHPF90 (see Figure 3).

The xHPF77 is a new module offered by APR which is available as an addition to the DMP. The xHPF77 system accepts Fortran 77 with HPF data distribution directives and translates this to Fortran 77 with message passing using the DMP. The xHPF77 module does not implement any additional features for parallelising code over the DMP. Accordingly data distribution in the xHPF77 system is restricted to one dimension.

The HPF directives specify the data partitions only, while the DMP requires both data distributions and loops for parallelisation to be specified. The approach xHPF77 follows is that all loops that contain the distributed data are parallelised. As this translator is built upon the DMP, all the limitations of the DMP still apply. Hence no code restructuring takes place resulting in possible inefficiencies in the parallelised code.

Calls to the subset HPF intrinsic functions and subroutines that are not already part of Fortran 77 are linked to routines provided by the APR's run-time library.

2.4 Vienna Fortran Compiler

The Vienna Fortran Compilation System (VFCS) is a research tool under continuing development at the University of Vienna. It is both a compiler for Vienna Fortran (VF) and an environment for developing parallel applications. In addition to the features required for proper parallelisation, the interactive mode permits the user to gain a significant amount of information on his source program and to invoke a profile tool which derives several kinds of profile data on the program to help in parallelisation [8].

This profile tool [9] has features very similar to the Code Reformatter, Interprocedural Analysis and Instrumenter/Profiler in FORGE 90, and similarly provides information and runtime feedback to the user. In this section we do not further describe the analysis and profiling capabilities of the system, but concentrate on its parallelising functionality. The description reflects the current status of the system.

Parallelization under VFCS is based on the SPMD model of parallelisation. Arrays in the source program are distributed to the processors of the target machine by the user, who provides this information to the compilation system in an appropriate way (see below). In contrast to Forge 90, the user does not need to help the system to determine which loops should be parallelised: this is fully automatic. The output code is generated according to the host-node model, whereby the host does not perform any computation. The compilation approach is described in detail in [21].

The system can be used as a command-line compiler (batch mode) or in interactive mode. Both of these have the same input languages. The following describes the capabilities of Version 1.0; the next version will have a much extended functionality. Currently, VFCS is based upon Fortran 77 and strictly speaking is not a HPF compiler. It accepts the following input languages and features:

- Fortran 77 together with a specification of the data distribution in an ad hoc partitioning language specific to this system. The language constructs are powerful, and roughly equivalent to VF. The advantage of this approach is that the user does not need to modify the original source code.
- Vienna Fortran [20]: Vienna Fortran constructs are provided for specifying data distributions, including irregular distributions, generalized block distributions and a special syntax for workspace (list-based) distributions. The latter permits reshaping of linear arrays to give them a multi-dimensional distribution. Distribution to subsets of processor arrays and reshaping of processor arrays are implemented. Vienna Fortran FORALL is implemented with REDUCE statement. DYNAMIC arrays and DISTRIBUTE statement are implemented, as is the IDT query and RANGE attribute.
- High Performance Fortran: HPF data alignment and distribution syntax is supported in Fortran 77 format. The INDEPENDENT directive is roughly equivalent to the VF FORALL (which also includes the NEW feature). The CYCLIC distribution is not yet fully supported. It does not include the new intrinsic procedures and other features of HPF closely related to Fortran 90 in the current version, although a Fortran 77 variant of the reduction operations is implemented in VF FORALL loops which are automatically recognized elsewhere.

- Fortran 90: this is currently under development. The alignment and distribution syntax will be extended along with this option which will also include the HPF FORALL statement.

VFCS thus implements a much wider range of decompositions than HPF. Furthermore, it has some support for irregular codes. There are no restrictions on the dimensionality of distributions, or on the number of dimensions partially replicated. VFCS can also create a vector code for the nodes of the target machine. However, it does not include many popular Fortran extensions. It does handle COMMON blocks, whose arrays may be distributed if they conform to the language requirements, and permits sequence and storage association across procedure boundaries.

VFCS has several libraries for performing communication, in particular for handling special features such as reduction operations. Some communication routines are customized for an application. It has special routines for handling the communication that arises in irregular computations. These include the PARTI subroutine library.

The current version supports the following platforms and message passing environments: iPSC/860, Paragon, PARMACS.

The command-line compiler applies a number of transformations to the input program and will process additional directives from the user (providing information on arguments to subroutines) if present. It has been used as an experimental system for developing compilation strategies and is being worked on and extended considerably in many directions. The interactive version is under full user control and is a multipurpose system. Currently, this version is being partially adapted to provide a stable system. At the moment the command-line implementation is more reliable, but cannot always produce results of the same quality. Currently VFCS cannot handle large problems.

2.5 Compiler features of prototype systems

2.5.1 Supported features of Fortran 90

Table 1 gives a general overview of the current implementation status of the Fortran 90 features as defined in the subset HPF definition [15] within the compiler tools under evaluation. The information is based on the existing documentation, papers, etc. and also on the experiments with the GENESIS benchmarks and some other small examples where necessary.

The information in Table 1 shows that all Fortran 90 features in subset HPF have been implemented in the xHPF90 compiler. Most entries for VFCS are negative, as the current version of this system still supports only Fortran 77. Array-valued functions, allocatable arrays, interface blocks and optional arguments are not implemented in Adaptor and Fortran 90D yet. Both compiler tools support most of Fortran 90 array intrinsic functions (see sec. 2.1 and 2.2) with some exceptions regarding data types, construction functions, number of dimensions, etc.

Some Fortran 90 features (free source form, modules, pointers, derived types, recursion, new control constructs, etc.) are not in the subset. In most cases the HPF systems under consideration do not support these features as they have committed themselves to the subset HPF language.

2.5.2 Supported HPF features

Table 2 shows which subset HPF directives are currently recognized and processed by the compiler tools under evaluation.

The `PROCESSORS` directive is vital for both Fortran 90D and VFCS, whereas it is accepted but not implemented by Adaptor and xHPF90. The latter case is the basis for generating an output code which is independent of the number of processors (see Table 4). One of the most important

Table 1: Implementation of Fortran 90 features in subset HPF

Features	Adaptor	Fortran 90D	xHPF90	VFCS
MIL-STD-1753 features	yes	yes[a]	yes	no
Array syntax	yes	yes[b]	yes	no
WHERE construct	yes	yes	yes	no
Array valued functions	no	no	yes	no
Automatic arrays	yes	no	yes	no
Allocatable arrays	no	no	yes	yes
Intrinsic functions	not all	not all	yes	some
New type declarations	yes	yes	yes	no
Interface blocks	ignored[c]	no	yes	no
Optional arguments	no	no	yes	no
Syntax improvements	yes	yes[d]	yes[e]	yes

[a]There are problems with the implementation of the IMPLICIT NONE statement (see section 4.2).
[b]Currently only the WHERE statement is implemented
[c]Feature is parsed and recognized but not really supported.
[d]The '!' form of comments is not implemented for the HPF directives.
[e]HPF directives may not appear within continued statements.

Table 2: Implementation of subset HPF features

Features	Adaptor	Fortran 90D	xHPF90	VFCS
Processors directive	ignored[a]	yes	ignored[a]	yes
Template directive	yes	yes	yes	no
Distribution dimensions	1 dim	2 dim	1 dim	7 dim
Decomposition attributes	BLOCK	BLOCK, CYCLIC	BLOCK,CYCLIC[b]	BLOCK [c]
Simple alignment	yes	yes	yes	yes
Full alignment ($a * I + b$)	no	no	no	yes
FORALL statement	canonical[d]	yes	yes	VF only

[a]Feature is parsed and recognized but not really supported.
[b]xHPF90 provides also several other attributes to specify the memory allocation for distributed arrays.
[c]Also several other block-based and general block, including lists of distributions and reshapes.
[d]The indices must be simple functions of the loop variables of the FORALL statement.

parameters of currently available HPF-like compiler tools is the number of distribution dimensions. These are one of the strongest restrictions being only one or two at the moment (VFCS makes the only exception). The HPF `DISTRIBUTE` directive provides `BLOCK` and `CYCLIC` decomposition attributes in most of the compiler tools (Adaptor is the only exception). Both xHPF and VFCS provide some extra explicit (non-standard) decomposition attributes to widen the variety of possible mappings of data objects to abstract processors.

We consider the case when all data objects are aligned to the template using simple constant expressions, as a simple alignment. This alignment simplifies data decomposition and does not imply the need of generating complex alignment trees, as the data objects are directly aligned to the template. In this case at least one template directive must exist in the program. The full (in terms of subset HPF) alignment lifts these restrictions but does not accept alignment subscripts more complicated than first order with constant offset ($a * I + b$).

Although very rarely, the HPF-like systems under consideration already support some of the features of full HPF. Table 3 contains the current information on this matter.

Table 3: Implementation of full HPF features not in subset HPF

Features	Adaptor	Fortran 90D	xHPF90	VFCS
Redistributions	no	no	no	yes
Inherited dist.	no	no	yes	yes
FORALL construct	no	no	yes	no
HPF library	only scatter	no	yes	no
Extrinsic Procedures	possible	no	yes	soon
PURE attribute	yes	no	no	no

Table 4: Some important compilation characteristics

Features	Adaptor	Fortran 90D	xHPF90	VFCS
Compiler driver	batch	batch	batch	driver (Intel)
Makefile	yes	no	no	yes
Separate compilation	no	no	yes	no
Number of processors	dynamic	static	dynamic	static
Overlap areas	restricted	no	no	yes
Redistribution at subroutine boundaries	no	no	no	no

2.5.3 Compilation characteristics

It is important for the user to know how to deal with a translation tool. Table 4 gives a summary of the compilation characteristics for the four HPF-like systems under consideration. It can be possible that the tool has its own driver program that translates, compiles and links the program with the run-time system. One would also expect the tool to generate a `Makefile`. Some tools can be used in a batch manner and/or interactively.

When considering larger programs the HPF-like systems should be able to perform separate compilation. This means that the whole program can be split up in several files which can be compiled independently.

The executable of the generated SPMD program will be loaded on the nodes of the parallel machine. Sometimes, the code runs only on the number of processors that has been specified for the translation with the tool (static case). It would be more convenient for the users if the executable is not dependent on the number of processors the parallel program will be running on(dynamic case).

In the context of message passing programming overlap areas are used for distributed arrays. The overlap area contains data that is not owned but only used by the processor. Data in overlap areas is replicated data and must therefore be the same among all the processors. If a compiler for HPF does not insert overlap areas in the generated message passing program, many additional temporary arrays are usually required. This will not increase the communication traffic, but more memory space and more local memory transfer is necessary. Sometimes it might be possible that the user specifies the overlap area by its own (restricted case), but the automatic insertion of overlap areas is obviously more convenient.

HPF is a language where actual and dummy arguments might have different distributions. Therefore implicit redistributions at the entry and after the end of the subroutine might be required. Subset HPF compilers have to support this feature, otherwise the user must take care of the redistributions.

3 Description of test programs

3.1 The HPF version of the GENESIS benchmark suite

Most of the GENESIS benchmarks were initially developed within ESPRIT Project P2702 – GENESIS [14] by a number of project partners including Liverpool and Southampton Universities, PALLAS GmbH, the European Centre for Medium Range Weather Forecasting and the Universitat Politechnica de Catalunya. The current release (2.2) of the GENESIS benchmark suite is based on subset HPF, whilst future releases will be able to take advantage of dynamic data distribution, data parallel programming (FORALL construct) and code tuning by EXTRINSIC procedures. A subset HPF conversion strategy, which assumes that the conversion process is divided into two phases (see Fig. 4), has been adopted. During the first phase the work is concentrated on the development of Fortran 90 version of the benchmarks. A main task in this phase is to introduce array constructs into the codes. The second phase addresses the data distribution, resulting in the subset HPF version of benchmarks.

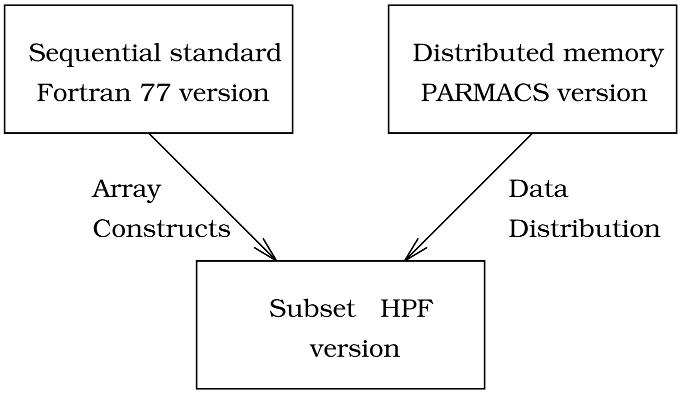


Figure 4: Subset HPF conversion strategy

Release 2.2 of the GENESIS benchmarks comprises sixteen codes that vary from synthetic code fragments measuring basic machine parameters, through important application kernels, to compact research applications [1]. This hierarchical structure allows information derived from the simpler codes to be used in explaining the performance characteristics of the more complicated codes. Thus the benchmark suite can be used to evaluate performance on a range of levels from simple machine parameters to full applications where effects due to non-parallelisable sections of code, and memory, communication or I/O bottlenecks may become important. The following codes have been used in the evaluation of HPF-like systems:

TICK1 Measures resolution of system clock.

Versions F-77, Subset HPF

TICK2 Measures accuracy of system wall-clock time.

Versions F-77, Subset HPF

TRANS1 This benchmark distributes a matrix into square submatrices for parallel transposition using naive algorithm. During the execution pairs of processors communicate with each other simultaneously exchanging very long messages (depending on the problem size and the number of processors).

Versions F-77, PARMACS, Subset HPF

FFT1 One-dimensional FFT. Popular test with relatively high level of parallelism, but low level of vectorisation and a hypercube type communication.

Versions F-77, PARMACS, Subset HPF

PDE1 Three-dimensional Poisson solver using red-black relaxation. Only nearest neighbour interactions are required and the number of floating point operations per grid point is very small when compared to other more complex PDEs.

Versions F-77, PARMACS, Subset HPF

PDE2 Two-dimensional multi-grid Poisson solver. This kernel requires highly structured long distance communication. During the solution process messages and vectors ranging from a single word upwards are used. Efficient execution of this problem therefore requires good short vector performance for each processing node together with a reasonably low start-up time for message passing.

Versions F-77, PARMACS, Subset HPF

MD1 Molecular dynamics code. This benchmark uses techniques from molecular dynamics to solve the Newtonian equations of motion for large number of interacting particles. The solution is used to calculate thermodynamic properties of the system. The benchmark uses a Lennard-Jones potential with a linked-list algorithm for solution of the equations of motion.

Versions F-77, PARMACS, Subset HPF

The GENESIS benchmarks are publicly available by anonymous `ftp` from `ecs.soton.ac.uk`, directory `pub/benchmarks`.

3.2 Utilized subset HPF features

The current set of GENESIS benchmarks includes regular problems in one, two, three and four dimensions. From this point of view all benchmarks conform to the SPMD programming model and allow HPF transformations. It is essential to know which subset HPF features are utilized in the GENESIS benchmark codes. The two tables below contain this information. 'Stack allocate' in Table 5 means that a DEALLOCATE statement deallocates both the specified array and any others allocated after it. There are no codes which use features of Fortran 90 or HPF, that are not in the subset. In the case of `MD1` one could take advantage of the HPF library by using `COPY_SCATTER`.

Some additional comments:

- `INHERIT` directive is used in FFT1 and PDE1, but the corresponding subroutines will always be executed with the same distribution and no redistribution is required.
- `MD1` has one- and three-dimensional distributions.
- `FFT1` and `MD1` use indirect addressing.

Concerning the importance of the different codes for the HPF compilers the following can be pointed out:

- TICK1 and TICK2 are only used for testing timer functions.
- TRANS1 and PDE1 contain very important features and should run efficiently. PDE1 would take advantage from overlap areas.

Table 5: Utilized Fortran 90 subset features

Features	TRANS1	FFT1	PDE1	PDE2	MD1
MIL-STD-1753 features	yes	yes	yes	yes	yes
array syntax	yes	yes	yes	yes	yes
intrinsic functions	TRANSPOSE	-	MAXVAL	MAXVAL	RESHAPE CSHIFT, SPREAD
automatic arrays	no	yes	no	no	yes(stack allocate)
full allocatable arrays	no	no	no	no	no
array valued functions	no	no	no	no	no
new type declarations	yes	yes	yes	yes	yes
optional arguments	no	no	no	no	no
interface blocks	no	no	no	no	no
syntax improvement	yes	yes	yes	no	no

Table 6: Utilized subset HPF features

Features	TRANS1	FFT1	PDE1	PDE2	MD1
processors directive	yes	yes	yes	no	no
template directive	no	no	no	no	no
distribute (BLOCK)	no	yes	no	no	yes
distribute (BLOCK,BLOCK)	yes	no	no	yes	no
distribute (BLOCK,BLOCK,BLOCK)	no	no	yes	no	yes
simple alignment	yes	yes	yes	yes	yes
full alignment ($a * I + b$)	no	no	no	no	no
FORALL statement	no	no	no	no	yes

- PDE2 should run efficiently and proves how well the compiler can deal with structured communications and array syntax. It uses parameter functions.
- FFT1 tests indirect addressing and array movements, but the current version does not seem to be so efficient.
- MD1 will test many HPF features, but the HPF version is very slow, compared to the serial program.

4 Evaluation results

4.1 Introductory comments

The subset HPF compiler systems under consideration have been having different level of development during the evaluation exercise. Therefore the extent of evaluation results is different for different systems. For example, VFCS has not been evaluated by the GENESIS test benchmarks, as it does not support Fortran 90 features yet. Similarly, the different subset HPF systems have been having different level of availability. The worst case in this respect is the xHPF90 compiler from APR Inc. The only option available for us has been an indirect evaluation through the company. APR have been willing to perform the whole evaluation by themselves which they finally did. They decided, however, not to disclose the measurement results for now and therefore we were not able to present benchmark measurements for xHPF90 in section 5.

The HPF versions of the GENESIS benchmarks have also been a subject of development and alterations during the evaluation procedure. The changes to the benchmarks can be classified in three categories as follows:

- First set of changes. A few syntax errors have been discovered as the subset HPF GENESIS benchmarks were produced with the NAG's Fortran 90 compiler and had not been tested with a real HPF system before.
- Second set of changes. The input language of different HPF tools conforms only to a subset of subset HPF. The data partitioning has to be modified quite often so that the HPF product could parallelise the code. The main restriction here was the limited number of dimensions for data decomposition. This peculiarity of the current HPF-like compilers has been a source for a number of editions in order to achieve successful compilation. These editions and the corresponding limitations of every HPF tool under evaluation are described at the beginning of the following subsections.
- Many specific changes have been attempted in order to help the compilers to overcome particular problems. Most of them are discussed in detail in the specific evaluation subsections below.

4.2 Adaptor

Adaptor is able to translate the HPF codes to a SPMD version running on all nodes (HOSTLESS model) or a single node version without any communication (UNIPROC model). For comparison both versions should be generated to measure the overhead due to the parallel execution. The SPMD version is not dependent on the number of processors, which means that the same executable could be used for different processor numbers without recompiling.

The following limitations of the current release of Adaptor were identified:

- Adaptor supports only block distribution along one dimension;
- many intrinsic functions are not supported (matmul, reshape);
- prescriptive layout directives must be replaced with descriptive ones;
- not all Fortran 90 features, as defined in subset HPF, are implemented;
- parameter functions cannot be used;
- in some situations, complex array expressions must be resolved to some simpler array expressions by the user itself;
- in a FORALL statement the index expressions of arrays must be of the form `I + c` if the index is dependent on a FORALL index `I`.

It should be mentioned that Adaptor has many restrictions, for example it may happen that it fails to translate complex array operations. In most cases of GENESIS test programs, however, it was possible to make some editions by hand that does not change the semantic of the code but would allow the translation with the Adaptor tool. Most changes are due to the preliminary nature of Adaptor and will not be necessary in a future version. Adaptor supports only one-dimensional block distribution. Some codes will therefore not scale well (e.g. PDE1, TRANS1).

The following general changes to the data distribution directives have been made:

- the PROCESSORS directive has been removed;
- a TEMPLATE was introduced as arrays cannot be aligned to arrays by Adaptor;
- the INHERIT directives have been removed.

TICK1 Adaptor has its own timer function. When calling the timer function, all nodes will be synchronized. This synchronization needs some additional time. Therefore the CLOCK TICK is dependent on the number of processors. For 4 processors the CLOCK TICK is less or equal to 5.800128E-04 sec on the iPSC/860.

TICK2 The code could be translated with Adaptor. The run on the iPSC/860 verified that the percentage error was less than one percent.

TRANS1 In order to make the translation with the current version of Adaptor possible the array DATA has been renamed to DDATA, as the name DATA is being used by Adaptor internally. There were no other essential problems with this code.

FFT1 The following changes have been made to make a translation with the current version of Adaptor possible:

- The descriptive distribution and alignment directives have been changed to prescriptive ones (but redistributions are not necessary),
- A temporary array was necessary for some assignments in subroutine FFTCTA. For example[1],

```
<           FR(J:JMAX)  = FR(K:J-1) - G(J:JMAX,0)
---
>           TMP(J:JMAX) = FR(K:J-1)
>           FR(J:JMAX)  = TMP(J:JMAX) - G(J:JMAX,0)
```

- The indirect addressing in the subroutine SHUFFL have also been simplified.

```
<     DO I=0,NPROC-1
<       T(0:NTOTIN-1) = FR(I*NTOTIN+A(0:NTOTIN-1))
<       FR(I*NTOTIN:I*NTOTIN+NTOTIN-1) = T(0:NTOTIN-1)
<     ENDDO
---
>     DO I=0,NPROC-1
>       A1(0:NTOTIN-1) = A(0:NTOTIN-1) + I * NTOTIN + 1
>       T = FR (A1-1)      ! indirect addressing
>       FR(I*NTOTIN:I*NTOTIN+NTOTIN-1) = T(0:NTOTIN-1)
>     ENDDO
```

- Similar editions were necessary to the subroutine TFGEN.

```
<       AR(OMIND) = DCOS(ANGLE)
<       AI(OMIND) = DSIN(ANGLE)
---
>       ANGLE1    = DCOS(ANGLE)
>       ANGLE     = DSIN(ANGLE)
>       INDEX = OMIND + 1
>       AR(INDEX-1) = ANGLE1
>       AI(INDEX-1) = ANGLE
```

[1]For the examples in this section we are using the style of the output from diff command.

After these initial changes, the code caused problems due to the fact that it contains some arrays for which no layout directives are given. Adaptor chooses a block distribution by default, contrary to the intended replicated distribution. Adaptor has a flag to select the default distribution and his flag must be enabled when the default distribution is to be replicated.

Another problem is that the code has some automatic arrays. Therefore the user of Adaptor has to specify manually as a command option the maximal size of the arrays for the translation.

PDE1 There were no essential problems with this benchmark. It should be mentioned, however, that only a one-dimensional distribution is possible instead of the preferred three-dimensional one.

In detail, the following changes were necessary:

- `FUNCTION RES` became a `SUBROUTINE`,
- an additional array `U1` with the same size and shape as `U` was necessary, otherwise the serialization of some array operations would result in problems,
- the Adaptor timing routine was used.

In the current version of Adaptor, it is possible to use overlap areas. But the overlap area must be explicitly defined by the user. With the following declaration it is specified that the last distributed dimension has an overlapping of 1 on both sides.

```
DOUBLE PRECISION U1(NX,NY,NZ[1:1])
```

If the array `U1` has an overlap area, the body of the 'red-black' relaxation will require no communication. If no overlap area is specified, the 'red-black' array operation will require two additional temporary arrays.

```
U_TMP1(2:NX-1,2:NY-1,2:NZ-1) = U1(2:NX-1,2:NY-1,1:NZ-2)
U_TMP2(2:NX-1,2:NY-1,2:NZ-1) = U1(2:NX-1,2:NY-1,3:NZ)
```

In this code the assignments to the temporary arrays require communication. When using an overlap area, only the assignment to `U1` requires communication.

```
U1 = U     ! overlap is also updated
```

PDE2 This benchmark has the advantage of using parameter functions, but Adaptor does not support this feature and therefore the code needed some essential changes. Afterwards, the HPF program could be translated to a serial F77 program, but the current version of Adaptor does not support array assignments with non-unit strides that will require communication.

```
A(2:N:2) = B(1:N/2)
```

It was therefore not possible for Adaptor to successfully translate the code to produce an acceptable SPMD version. The features for unstructured communication could be used which, however, will introduce a substantial amount of communication overhead.

MD1 This code needed similar editions as the other ones, but also some more specific changes as follows.

- Scalar elements of distributed arrays cannot be arguments of subroutines in the current Adaptor version. For example,

```
<             CALL XSRAND (IS, VX (I))
---
>        DOUBLE PRECISION VHELP
...
>             CALL XSRAND (IS, VHELP)
>             VX(I) = VHELP
```

- Reduction operations with a `MASK` argument are still not supported and the relevant code had to be sequentialised by hand.
- The subroutine `MKVECT` uses indirect addressing within a `WHERE` statement. In the case of Adaptor an own intrinsic subroutine (`GLOBAL_SEND`) had to be used instead that requires a logical mask. Fortunately, this solution makes the use of the intrinsic function `RESHAPE` unnecessary, as this intrinsic function is not supported within Adaptor yet.

```
<       NATMCV = RESHAPE (NATMCL, (/ NCELLS /))
<       DO I = 1, MAXCL
<         IDV = RESHAPE (IDL (I, :, :, :), (/ NCELLS /))
<         FXV = RESHAPE (FXL (I, :, :, :), (/ NCELLS /))
<         FYV = RESHAPE (FYL (I, :, :, :), (/ NCELLS /))
<         FZV = RESHAPE (FZL (I, :, :, :), (/ NCELLS /))
<         WHERE (I .LE. NATMCV)
<           FX (IDV) = FXV
<           FY (IDV) = FYV
<           FZ (IDV) = FZV
<         ENDWHERE
<       ENDDO
---
>       FORALL (I=1:LMAX)
>          MASK(I,:,:,:) = I .LE. NATMCL(:,:,:)
>       END FORALL
>       CALL GLOBAL_SEND (FX, IDL, FXL, MASK)
>       CALL GLOBAL_SEND (FY, IDL, FYL, MASK)
>       CALL GLOBAL_SEND (FZ, IDL, FZL, MASK)
```

- Similar changes were necessary to subroutine `MKLIST` using the intrinsic subroutine `GLOBAL_GET`.
- The loop for computing the array `IDL` within the subroutine `MKLIST` has not been replaced with the attempted HPF solution.

After all these changes the code runs in the sequential and parallel version and produces the same results.

4.3 Fortran 90D compiler

The alpha release of the Fortran 90D compiler has been obtained from the University of Syracuse to carry out this evaluation. It is believed that this compiler implements a Subset HPF-like language on the basis of Fortran 90 with Fortran-D data distribution directives. The same compiler has been used for the production of HPF benchmarks for the HPF Forum at the University of Syracuse.

The initial experiments with the alpha release of the compiler showed that the existing version can be ported only to the Express communication libraries on iPSC/860. It was found practically impossible to gain access to such installation in the U.K., but fortunately a limited access was granted to a small (4 nodes) platform in Leuven, Belgium. The Fortran 90D compiler was installed and tested there. The short initial experience showed the following problems:

- the error handling by the compiler is not sufficiently developed - in many different situations it suddenly terminates with a segmentation fault;
- the automatic decomposition is possible in not more than 2 dimensions;
- there are certain syntactical differences in comparison with the HPF language specification.
- many intrinsic functions (for example TRANSPOSE, MATMUL, etc.) do not have double precision versions;

The new beta release of the Fortran 90D compiler (obtained at the end of August 1993) represents significant improvements over the first and the last of the above problems. Nevertheless there are still a number significant restrictions of general importance. These include:

- The array indices in the PROCESSORS directive must be integer constants (not variables, not parameters);
- The TEMPLATE directive clashes with the IMPLICIT NONE statement;
- The compiler does not recognize the keyword ONTO within the DISTRIBUTE directive;
- The ALIGN directive has to be after the DISTRIBUTE directive;
- The array indices in the TEMPLATE directive must be integer constants (not variables, not parameters);
- The INHERIT directive is not implemented;
- The HPF directives must start with `CHPF$` and not with `!HPF$`;
- The arrays subject of data distribution have to be declared with indices in the form of integer constants or parameters, but only with a simple assignment of the exact value.

TICK1, TICK2 There were no essential problems with these sequential codes apart from the fact that the Fortran 90D compiler does not accept the standard nonelemental intrinsic subroutine DATA_AND_TIME which has been used for time measurements in the Fortran 90 versions of the GENESIS benchmarks. The Express library call KXTIME was used instead, which proved convenient for taking wall-clock time measurements.

TRANS1 The first parallelisation phase of the compilation was successful after rewriting the data distribution directives in a suitable for the Fortran 90D compiler style (see the above general comments). The second on-site phase however could not link the executable as the intrinsic function TRANSPOSE is not implemented for DOUBLE PRECISION data types in the INTRIN library. Three experiments were attempted in order to overcome this problem:

- Converting the benchmark from DOUBLE PRECISION to pure REAL data types. This option worked, but it was recognized as a not very convenient solution for comparative benchmark measurements.
- Including a new DOUBLE PRECISION version of the TRANSPOSE intrinsic function into the INTRIN library. The compilation and linking of the new subroutine and the benchmark code was successful but the execution was terminated because of a run-time error. No further attempts have been taken to overcome this problem.
- Replacing the TRANSPOSE intrinsic call by a FORALL statement. This attempt was also successful and the time measurements in section 5 have been produced with this version of the benchmark.

FFT1 The following evaluation steps were performed after editing the code in accordance with the general comments above:

- It was found that the subroutine boundaries appear to be a significant problem for the compiler. Therefore the evaluation was firstly performed on every subroutine as a separate file.
- The first attempts to parallelise pieces of the FFT1 benchmark showed that the compiler can not handle data decomposition correctly if the arrays to be distributed are declared like `A(0:N-1)`, but the whole benchmark is written in this stile. The code was rewritten where necessary in order to meet this restriction.
- The sequentialiser had difficulties when replacing some more complex array expressions by FORALL statements. In both INGENB and TFGEN subroutines was necessary to use Fortran 77 replacement code in order to help the sequentialisation of the benchmark. Subroutine TFGEN.

```
<       ANGLE     = TEMP*(OFFSET+INDEX)
<       AR(OMIND) = DCOS(ANGLE)
<       AI(OMIND) = DSIN(ANGLE)
---
>       DO 100 I=0,LENGTH - 1
>          J=OFFSET+I
>          ANGLE=TEMP*DBLE(J)
>          AR(OMIND(I))=DCOS(ANGLE)
>          AI(OMIND(I))=DSIN(ANGLE)
>100    CONTINUE
```

  Subroutine INGENB.

```
<          B(STRIDE:2*STRIDE-1)=B(0:STRIDE-1)+OFFSET
---
>          DO 100 I=0,STRIDE-1
>             B(I+STRIDE)=B(I)+OFFSET
>100       CONTINUE
```

- The whole benchmark was restructured performing an 'in-lining' by hand in order to overcome the problem of Fortran 90D compiler with the subroutines. Unfortunately the compiler was not able to parallelise the code although it had successfully parallelised all its pieces separately.
- The code was divided into two halves, which the compiler could parallelise but the on-site iPSC/860 Fortran compiler reported two missing library subroutines at the linkage phase - `tshift_d` and `pack_d`.
- Another attempt to run the benchmark was taken after converting the code from `DOUBLE PRECISION` to `REAL` data type, but the iPSC/860 Fortran compiler terminated with segmentation fault at compilation time.

Obviously the FFT1 code is too large for the current release of the Fortran 90D compiler, which was not able to generate correct output code with explicit message passing directives.

PDE1 It was necessary at the beginning to rewrite the data distribution directives in a suitable for the Fortran 90D compiler style and to declare decomposition in two dimensions rather then in three dimensions as it had been in the original code.

- function RES

  Nested intrinsic functions terminate the parallelisation phase with segmentation violation. The following line caused problems for the compiler:

```
RES = MAXVAL(ABS(F(2:NX-1,2:NY-1,2:NZ-1)+
...
```

Also the code generator produced a call to the library routine MAXVAL_3d which does not exist yet. This is obviously a mistake because the compiler does not recognize that one of the dimensions has collapsed and therefore it should generate a call to the library routine MAXVAL_2d. In order to overcome these problems the Fortran 90 code was replaced by the standard Fortran 77 code of the RES function.

- subroutine RELAX

  Unfortunately the compiler did not accept the WHERE array construct in the subroutine RELAX. This is because only the WHERE statement is implemented by the Fortran 90D compiler. The WHERE construct was replaced by two WHERE statements with mask but, although the parallelisation phase was successful, the compiler did not generate a correct output code.

  – input HPF code fragment:

```
      WHERE(RED(2:NX-1,2:NY-1,2:NZ-1))
     &  U(2:NX-1,2:NY-1,2:NZ-1) = FACTOR * ...
      ....
```

  – output code fragment:

```
      DO Ti1 = 2, NX - 1
      DO Ti2 = Tllb2, Tlub2
      DO Ti3 = Tllb1, Tlub1
      IF (RED ((2 + Ti1, (2 + Ti2, (2 + Ti3)) THEN
C        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ - unbalanced parentheses
      U (Ti1, Ti2, Ti3) = FACTOR * ...

      END IF
      END DO
      END DO
      END DO
```

  A new version of this subroutine based on the FORALL statement with mask was not successful as well.

  – input HPF code fragment:

```
      FORALL(I=2:NX-1,J=2:NY-1,K=2:NZ-1, RED(I,J,K)) U(I,J,K)=FACTOR
        ....
```

  – output code fragment:

```
      DO K = Tllb2, Tlub2
      DO J = Tllb1, Tlub1
      DO I = 2, NX - 1
      IF (, RED (I, J, K)) THEN
C        ^^ - syntax error
      U (I, J, K) = FACTOR * ...

      END IF
      END DO
      END DO
      END DO
```

  Finally the compiler produced an output code with the sequential Fortran 77 version of this subroutine. However the if77 compiler in the second on-site phase found an error, which obviously comes from the first parallelisation phase. An example from the error code is given below:

```
 IF ((coord(1) .EQ. ((J - 1) / 8)) .AND. (coord(2) .EQ. ((K - 1)
&/))) U(I,J,K) = FACTOR * (HSQ * TT14temp + (TT15temp + TT16temp
```

```
C     ^^ - error
    & + TT17temp + TT18temp + TT19temp + TT20temp))
```

It is difficult to investigate further the source of this error. The same 'red-black' relaxation subroutine but in two dimensions was one of the first examples for the Fortran D compiler [10]. In our three dimensional version, however, the Fortran 90D compiler was not able to parallelise the code.

PDE2, MD1 These codes were too large for the current release of the Fortran 90D compiler and therefore it was not able to produce correct output SPMD programs.

4.4 FORGE 90

4.4.1 General comments on the DMP

The DMP should not be seen as a 'quick-fix' to parallelise Fortran 77 codes. In order to parallelise a program using the DMP, the user must have an intimate knowledge of the algorithm and the code. In this sense the DMP is only inserting the message passing syntax for the user, thus reducing the time required to do the parallelisation and the possibility of user error.

To use the DMP effectively the user should know how to parallelise the code by hand, and then instruct the DMP to insert the communication where required. The user must be aware of the consequences of the commands issued to the DMP as it does not provide any user feedback on parallelisation decisions. In addition, the DMP does not provide any code restructuring facilities to improve performance. Consider the following example:

```
DO J = ...
  DO I = ...
    A(I,J) = f(A(I,J-1))
  ENDDO
ENDDO
```

First, the user must decide how to partition the data. In the above example it is obvious to partition matrix A along the first dimension, I. Parallelising the loop directly along I is however inefficient as it requires processor synchronization and communication of J, each time loop I is entered. The code should be restructured first by exchanging the I and J loops and then parallelised. FORGE does not provide this restructuring, nor does it inform the user of this problem. The user must identify this and restructure the code by hand.

In addition to the DMP requirement that the user be familiar with the sequential code being parallelised, there are a number of limitations of the DMP system. One specific limitation concerns the way data is partitioned and allocated. First, data can only be partitioned along a single dimension. This is not a natural partitioning scheme for many grid based problems and may therefore result in a non-optimal code parallelisation. In specifying the data to be partitioned, the user is further required to specify whether the data allocated on each processor is 'full size' or 'shrunk'. If array $A(n)$ is allocated 'full size', each processor contains the complete array $A(n)$. In contrast, the 'shrunk' allocation scheme allocates only $A(n/p)$ data elements to each processor, where p is the number of processors. In general we consider data partitioning to be the latter definition. The DMP allows the user to specify 'full size' due to the possible inefficiency in handling 'shrunk' arrays. This is illustrated in the following example:

```
REAL A(100), B(100)
DO I = 2, 99
```

```
  A(I) = f(B(I), B(I-1), B(I+1)
ENDDO
```

Consider the case where $A(100)$ and $B(100)$ are partitioned block on 4 nodes. Node 1 owns $A(26:50)$ and $B(26:50)$, but requires $B(25:51)$. The DMP identifies that communication is necessary to retrieve $B(25)$ from node 0 and $B(51)$ from node 2. The difficulty lies in where to store these values. If $B(100)$ is allocated 'full size', then the data retrieved from node 0 and node 1 is placed in the data array B. If $B(100)$ is allocated 'shrunk' however, node 1 only contains memory for $B(26:50)$. When the DMP identifies that an array is accessed out of its bounds, a dynamic array is created from the minimum (actual lower bound, accessed lower bound) to the maximum (actual upper bound, accessed upper bound). In the case of node 1, a new array, $tempB$ is created with the bounds $tempB(25:51)$. $B(26:50)$ is copied into $tempB$ and the data received from nodes 0 and 2 is added to $tempB$. All references to B are offset to refer to $tempB$. $tempB$ is deallocated after the loop.

A significant overhead could therefore be introduced with the specification of 'shrunk' arrays. The access of a single data element from a neighbouring processor requires at least two times the memory needed to store the distributed array. This could clearly be problematic for applications that require a massive distributed memory.

A further limitation of the distributed memory paralleliser is that arrays of variable size cannot be partitioned. It is also worth mentioning that the DMP imposes a heavy execution penalty in accessing an array that is equivalenced to a distributed array.

The above limitations are seen as some of the most severe restrictions imposed on the user of the DMP. These restrictions are compounded by FORGE's 'all or nothing approach' - The output from pref77 is not meant to be in a readable form. Therefore, it is not possible for the user to use the DMP on certain parts of the code, while reserving the right to parallelise other aspects by hand.

4.4.2 xHPF90 evaluation

Unlike the interprocedural analysis performed by baseline FORGE, the DMP is attempting to solve a large complex problem that is not well understood. While APR have made significant steps in this area, the DMP is not viewed as a mature product capable of industrial use. The current restrictions exclude a large class of programs requiring parallelisation, while the performance of FORGE parallelised codes has not yet been proven.

The xHPF90 transformation system offered by APR provides a way of driving FORGE's DMP from the user subset HPF source code. As this system is functionally equivalent to the DMP the same limitations apply. In this regard APR are using FORGE xHPF90 to parallelise the GENESIS benchmarks and will make them available for evaluation. Although APR completed the evaluation by the end of October 1993, they are not willing to disclose the benchmarking results yet. Therefore we list below some limited indirect information about the xHPF90 evaluation which has reached us by now.

TICK1, TICK2 The time measurement benchmarks have discovered a bug in the KAPR converter from Fortran 90 into Fortran 77. This is the error message.

```
timer.f

  ###       CALL DATE_AND_TIME(VALUES=VAL)
  ### in line 15 procedure TIMER of file timer.f ###
  ### error 1526-649 At least two operands in the expression lack
      shape conformance. ###
```

```
1 error in file timer.f
 KAP -- Syntax Errors Detected
    **** 'timer.f' error code 2
Failure in Fortran 90 pass
```

TRANS1, FFT1 Both benchmarks are running and no essential problems have been reported from APR.

PDE1, PDE2 A problem with the timings to the allocation and deallocation of temporary arrays around the WHERE constructs has been identified. Given the granularity of the actual work, the overhead of continuously allocating and deallocaing the temporary arrays is very large.

MD1 The MD1 benchmark is working, however, the conversion from Fortran 90 to Fortran 77 is very inefficient.

5 Benchmark measurements

5.1 Introductory comments

The main emphasis of the benchmarking measurements has been focussed on evaluating where possible the HPF-like systems under consideration on iPSC/860 machines. The iPSC/860 is the best parallel computer to perform this evaluation, as it is the only common platform for all the three HPF-like translators, which would allow comparative performance results. Another interesting and useful comparison is between the time measurements when compiling with the prototype HPF systems and the time measurements from the hand-written code with explicit message passing primitives (PARMACS macros [13] in our particular case).

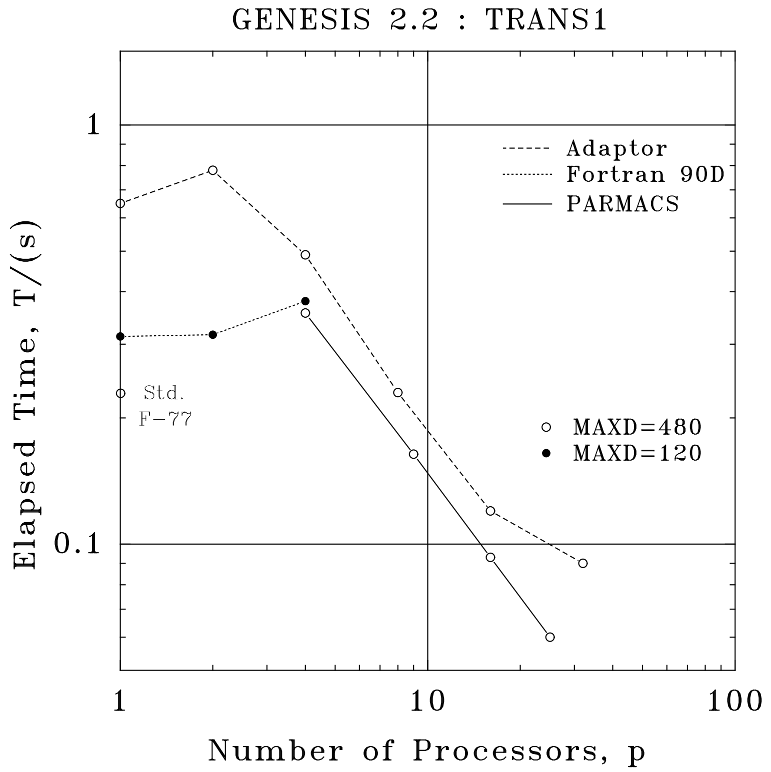
The benchmark measurements were taken on three different iPSC/860 machines - the 64-node configuration in Daresbury (U.K.), the 32-node hypercube in Jülich (Germany) and the small development platform at K.U. Leuven (Belgium). Although the convenient and easy access has always been important, the main reason for using so many different platforms has been the availability of all necessary pieces of system software and programming environments. The evaluation details of the three hypercubes are given in Table 7.

Table 7: Evaluation details of the iPSC/860 parallel platforms

Benchmarking Characteristics	ADAPTOR iPSC/860 at Jülich	Fortran 90D iPSC/860 at Leuven	PARMACS
No. of nodes	32	4	64
Local memory	16 Mbyte	8 Mbyte	16 Mbyte
iPSC System s/w	Release 3.3.2	Release 3.3.2	Release 3.3.2
Fortran compiler	if77/NXSun4 Rel4.0	if77/NXSRM Rel4.0	if77/NX SRM Rel4.0
Compiler switches	-Mvect -O2	-Mvect -O4	-Mvect -O4
M/P environment	PARMACS 5.1	Express 3.2.1	PARMACS 5.1

5.2 TRANS1 benchmark

Figure 5 shows time measurements for the matrix transpose benchmark on the iPSC/860 and plots the elapsed time against number of processors for different HPF-like systems and the hand-written code for the PARMACS message passing environment. The execution time of the standard Fortran



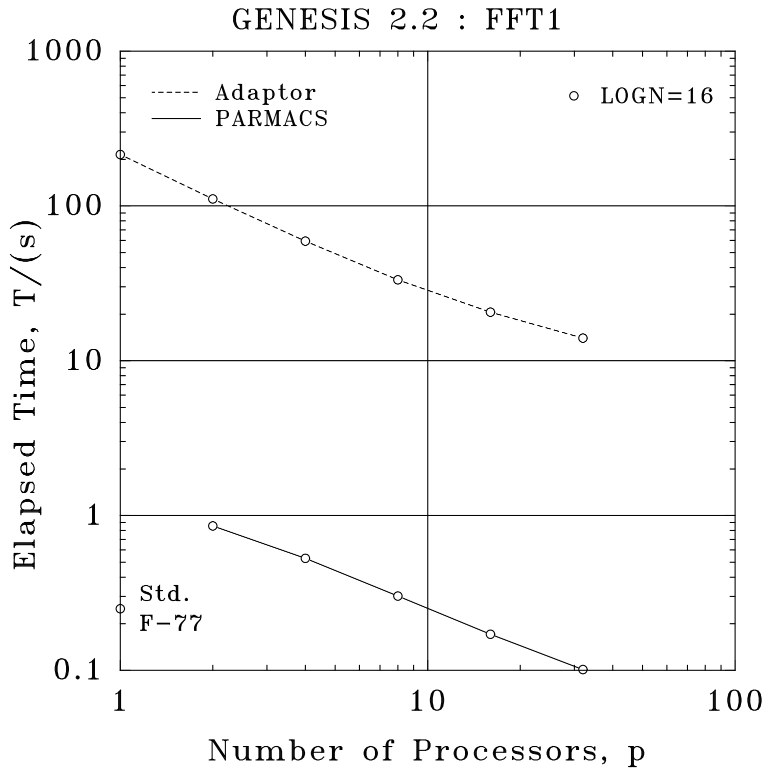
HPF system / MP env.	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 9$	$p = 16$	$p = 25$	$p = 32$
ADAPTOR	0.65	0.78	0.49	0.23	–	0.12	–	0.09
Fortran 90D(MAXD=120)	0.31	0.32	0.38	–	–	–	–	–
PARMACS	–	–	0.356	–	0.164	0.093	0.060	–

Figure 5: Elapsed time in seconds for the execution of TRANS1 on iPSC/860, compiled with Adaptor, Fortran 90D and PARMACS. The problem size is defined by MAXD = 480.

77 single-processor version (0.214 sec.) is also shown for comparison. The problem size for the TRANS1 measurements was defined as MAXD = 480, which means a 480 × 480 square matrix. It was not possible, however, to run the Fortran 90D measurements with this problem size and the benchmarking results for this particular case were taken for a smaller problem size (MAXD = 120). We see at least two reasons for that:

- The compiler generates too many temporary arrays and other extra code;
- The local memories on the hypercube at Leuven are smaller then the local memories on the other platforms.

The hand-written PARMACS version of TRANS1 can only be run on square configurations of processors, as shown in the table. It has also been noted that Adaptor performs only one-dimensional distribution of the arrays. The communication patterns are therefore different from the patterns of the Fortran 90D and PARMACS versions of this benchmark.



HPF system / MP env.	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
ADAPTOR	215.0	110.9	59.3	33.3	20.6	14.0
PARMACS	–	0.857	0.529	0.302	0.171	0.101

Figure 6: Elapsed time in seconds for the execution of FFT1 on iPSC/860, compiled with Adaptor and PARMACS. The problem size is defined by LOGN = 16.

5.3 FFT1 benchmark

Figure 6 shows time measurements for the 1-dimensional FFT benchmark on the iPSC/860 and plots the elapsed time against number of processors for the automatically produced code by Adaptor and the hand-written code for the PARMACS message passing environment. The execution time of the standard Fortran 77 single-processor version (1.403 sec.) is also shown for comparison. The problem size for the FFT1 measurements was defined as LOGN = 16, which means a transformation of 65536 complex data points.

The results have shown that the current ADAPTOR implementation of the FFT1 butterfly is very inefficient for parallel execution. Although this code scales well, the HPF version is more than 100 times slower than the PARMACS version. In order to further investigate the reason for this enormous delay a trace file has been generated and the timing diagram for the execution of the butterfly phase produced using ParaGraph visualisation tool (see Figure 7). The diagram shows that most of the time overhead has been spent in calculation operations. It is believed that this significant delay is not a result of the temporary arrays generated by Adaptor, but only a further investigation can give a detailed description of this problem.

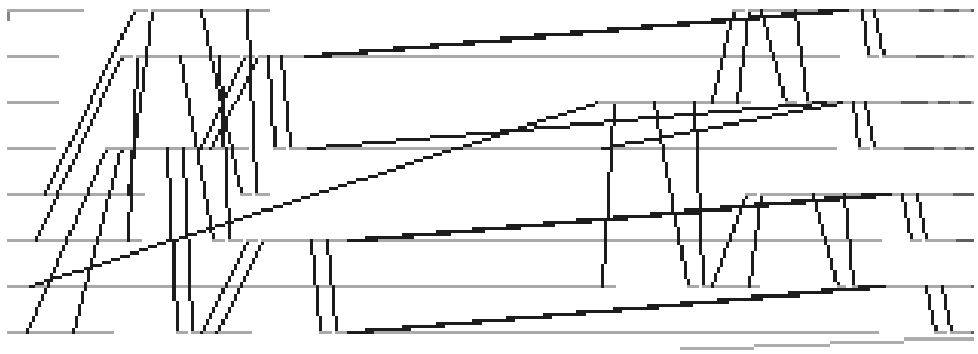


Figure 7: Timing diagram for the execution of the FFT1 butterfly phase on 8 processors, compiled with Adaptor

Usually the communication structure of the butterfly phase of 1-dimensional hand-written FFTs employs only one type of bidirectional communication between pairs of nodes. The two messages associated with every pair of nodes are independent and therefore they can be exchanged simultaneously in order to decrease the communication time [12]. The diagram, however, shows that the communication exchange has been sequentialized by Adaptor. It is also clear from the diagram that Adaptor generates twice as many messages in comparison to the hand-written message passing version. This is because the tool can not recognize the possibility of packing two messages (one for the real and one for the imaginary parts of the transform) in one and to decrease twice the overall message start-up time.

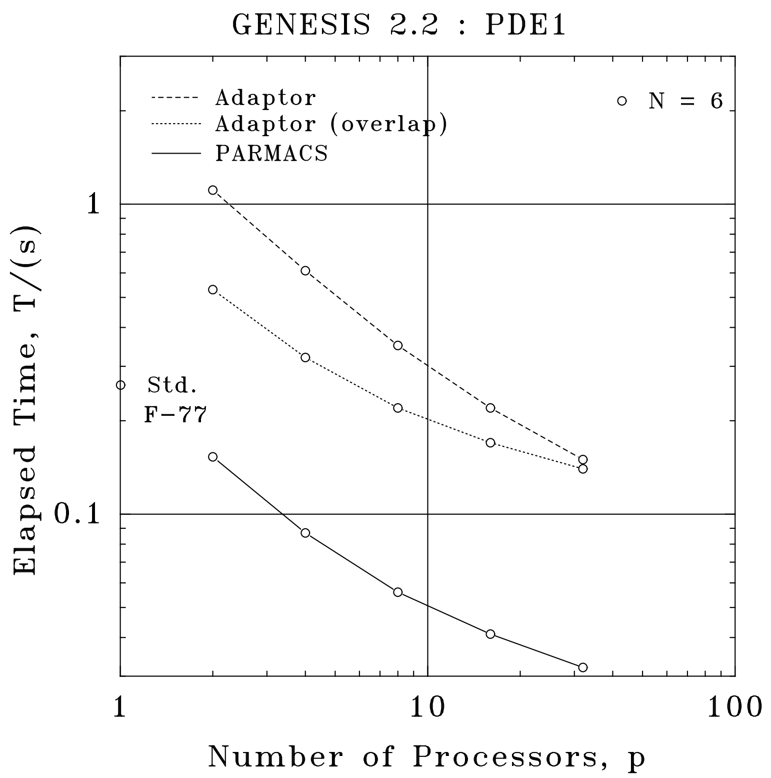
5.4 PDE1 benchmark

Figure 8 shows time measurements for the 3-dimensional 'red-black' relaxation benchmark on the iPSC/860 and plots the elapsed time against number of processors for the automatically produced code by Adaptor and the hand-written code for the PARMACS message passing environment. The execution time of the standard Fortran 77 single-processor version (0.250 sec.) is also shown for comparison. The problem size for the PDE1 measurements was defined as N = 6, which means $64 \times 64 \times 64$ data points.

The use of an overlap area in the `PDE1` code does not save any communication, but it saves memory space and data movements to local temporary arrays. In order to investigate the efficiency of this solution two series of time measurements have been taken - with and without an overlap area. The results show that the version with an overlap area is significantly faster for smaller number of processors. For larger number of processors the effect of the overlap area is negligible.

5.5 MD1 benchmark

The efficiency of the code is extremely poor. This is due to the fact that the HPF program itself is very inefficient if compared with the sequential Fortran 77 program. Therefore no time measurements for this benchmark are represented here.



HPF system / MP env.	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
ADAPTOR	[a]	1.11	0.61	0.35	0.22	0.15
ADAPTOR(overlap)	[a]	0.53	0.32	0.22	0.17	0.14
PARMACS	–	0.153	0.087	0.056	0.041	0.032

[a]No results, because the problem size is too large to fit into the local memories.

Figure 8: Elapsed time in seconds for the execution of PDE1 on iPSC/860, compiled with Adaptor and PARMACS. The problem size is defined by N = 6.

6 Recommendations to HPF systems developers

The HPF parallelisation tools are still in their infancy. They should be given time to develop in the areas of generality and performance before being used for commercial code parallelisation.

Feedback for the user

It is absolutely necessary to give the user feedback about the generated parallel program from his code. He should be informed which statements will require communication and whether this communication is efficient or not.

Structured and unstructured communication

There are many situations where the user will expect efficient translation of his code. HPF mappers should concentrate on optimizing communication patterns that can be computed at run-time.

Documentation

As HPF systems will not implement all possible optimizations in an early stage, the documentation should demonstrate how to implement efficient programs.

Compiler vs. run-time system

All of the early HPF systems have their own run-time system that performs operations on distributed arrays. The more complex the operations of the run-time system are, the less will be the potential of optimizations for future versions.

```
      integer IND(N)
      real A(N), B(N)
!HPF$ distributed (BLOCK) :: A, B, IND
      ...
      A =  B(IND)
      ...
```

If the above assignment with indirect addressing is translated in one call, it will not be possible to reuse the communication pattern. A better solution is to compute a schedule for the implied communication pattern and to use it for sending and receiving values. This will result in three calls, and the schedule can be reused. When more communications patterns are used, the run-time system should support the combination of more schedules to one schedule to minimize start-up times.

7 Conclusions

7.1 Portability issues

Portability of a program is a property that is most important for users. A first requirement is that the code runs on all machines of interest. But especially in the context of parallel machines it is also important that the application runs efficiently on all target machines. It may be necessary to tune the code where appropriate in order to achieve that.

While the portability of sequential Fortran 77 programs is no longer a real problem on sequential machines and also on vector machines, the portability of High Performance Fortran programs will still be a problem for some years. There are many reasons for that, but the most important from our point of view are listed below:

- HPF is based on Fortran 90. Therefore recoding of existing Fortran 77 applications is required.
- First experiences have shown that Fortran 90 itself requires very good compilers as Fortran 90 programs are very often slower than their Fortran 77 counterpart when compiled with the existing early Fortran 90 compilers.

- HPF is a new language and the early HPF systems do not support all features of subset HPF and hence require some canonical representations. For example, there are several different sorts of syntax for the distribution directives which are not fully implemented in the early HPF systems under evaluation.

- The task of an HPF compiler is also to translate implicit communication to explicit communication. Minimization of communication is a central issue for optimizations. However, different systems have different strategies for generating message passing directives that makes efficiency and portability very difficult issues.

7.2 The Fortran 90 problem

HPF is intended to use the inherent parallelism of array operations and of parallel loops. In most cases, this requires recoding of existing applications. Users will be reluctant to recoding in Fortran 90 even if Fortran 90 is as efficient as Fortran 77.

The following example of `MD1` shows the typical problem of a Fortran 90 compiler:

```
      NATMCN (:, :, :) =
     1         CSHIFT (CSHIFT (CSHIFT (NATMCL (:, :, :),
     2                 SX, DIM = 1), SY, DIM = 2), SZ, DIM = 3)
```

When using serial Fortran 77 this would be coded in such a way that one READ and one WRITE for every element of the arrays is required. But straightforward Fortran 90 compilers will implement this array operation by using temporary arrays:

```
TMP1   (:, :, :) = CSHIFT (NATMCL (:, :, :), SX, DIM = 1)
TMP1   (:, :, :) = CSHIFT (TMP1   (:, :, :), SY, DIM = 2)
NATMCN (:, :, :) = CSHIFT (TMP1   (:, :, :), SZ, DIM = 3)
```

This needs at least three READ and three WRITE operations.

Another problem of the array syntax is that due to the blocking of operations the cache of a processor may not be utilized well. Therefore it is very important for an HPF system to generate efficient serial code and to avoid temporary arrays and local data movements. All optimization issues of Fortran 90 compilers are issues of HPF compilers.

7.3 Optimization issues

Optimization strategies for data parallel program were and are still a topic of research. None of the early HPF systems has any of the number of optimization proposals implemented. Indeed, the benchmark results in section 5 showed performance losses within the range from 2 to more than 100 times in comparison to the corresponding hand-written PARMACS versions of benchmarks.

Optimization is very important due to the fact that in many cases the user has no possibility to write the code in such a way that the compiler can generate the most efficient output code. For example:

```
DO I = 1, MAXCL
  IDV = RESHAPE (IDL (I, :, :, :), (/ NCELLS /))
  FXV = RESHAPE (FXL (I, :, :, :), (/ NCELLS /))
  FYV = RESHAPE (FYL (I, :, :, :), (/ NCELLS /))
  FZV = RESHAPE (FZL (I, :, :, :), (/ NCELLS /))
```

```
    WHERE (I .LE. NATMCV)
      FX (IDV) = FXV
      FY (IDV) = FYV
      FZ (IDV) = FZV
    ENDWHERE
  END DO
```

This example from `MD1` shows a code fragment where indirect addressing is used. As always the same index vector `IDV` is used, the communication can be optimized by combining the values of `FXV`, `FYV` and `FZV` in one message. But none of the existing HPF-like systems can implement this optimization. On the other hand, the user has no possibility to write the code in a more efficient way.

Acknowledgments

Amongst those who have helped by discussion and criticism with the preparation of this report we would like to mention Dave Watson (NA Software) and John Merlin (University of Southampton) for their useful and detailed comments. John Levesque, the President of Applied Parallel Research. Inc. has provided us with the information about their xHPF90 compiler and also ran the GENESIS HPF benchmarks with it. The Fortran 90D compiler was obtained from the University of Syracuse with the help of Zeki Bozkus. We have also received very generous assistance from Dirk Roose (Leuven University) with the access to their iPSC/860 installation and the Express communication libraries.

References

[1] C.A. Addison, V.S. Getov, A.J.G. Hey, R.W. Hockney, I.C. Wolton. The GENESIS Distributed-Memory Benchmarks. In: J. Dongarra and W. Gentzsch (Eds.), *Computer Benchmarks*, North-Holland 1993, pp. 257-271.

[2] I. Ahmad, R. Bordawekar, Z. Bozkus, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, R. Thakur. *Implementation and Scalability of Fortran 90D Intrinsic Functions on Distributed Memory Machines.* Technical Report SCCS-256, NPAC, University of Syracuse, 1993.

[3] Applied Parallel Research Inc. *APR's FORGE 90 Parallelization Tools for High Performance Fortran (HPF).* June 1993.

[4] Applied Parallel Research Inc. *FORGE 90: xHPF 1.0 Automatic Parallelizer for High Performance Fortran on Distributed Memory Systems.* User's Guide, April 1993.

[5] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, M.-Y. Wu. *Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers.* Technical Report SCCS-444, NPAC, University of Syracuse, 1993.

[6] T. Brandes. Adaptor: A Compilation System for Data Parallel Fortran Programs. *Proc. of AP'93.* Saarbrücken, March 1993.

[7] T. Brandes. Compiling Data Parallel Programs to Message Passing Programs for Massively Parallel MIMD Systems. *Proc. of Working Conference on Massively Parallel Programming Models.* Berlin, September 1993.

[8] B. Chapman, et al. *VIENNA FORTRAN Compilation System. Version 1.0. User's Guide.* Department of Statistics and Computer Science, University of Vienna, January 1993.

[9] T. Fahringer. The Weight Finder - An advanced Profiler for Fortran Programs. *Proc. of AP'93.* Saarbrücken, March 1993.

[10] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, M. Wu. *Fortran D Language Specification.* Technical reprot COMP TR90-141, Rice University, April 1991.

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. *PVM 3 User's Guide and Reference Manual.* Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[12] V. Getov. 1-Dimensional Parallel FFT Benchmark on SUPRENUM. In: D. Etiemble and J.-C. Syre (Eds.) PARLE'92, Parallel Architectures and Languages Europe. *Lecture Notes in Computer Science*, **605**, 1992, pp 163-174.

[13] R. Hempel. *The ANL/GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model.* User's Guide and Reference Manual, Version 5.1. GMD, November 1991.

[14] A. J. G. Hey. The GENESIS Distributed-Memory Benchmarks. *Parallel Computing*, **17**(10-11), 1991, pp 1275-1283.

[15] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0.* Technical report CRPC-TR-92225, Rice University, Houston, May 1993.

[16] J. Levesque. *FORGE 90 and High Performance Fortran (HPF).* Applied Parallel Research, Inc., 1993.

[17] R. Lovely, A. Marshall, D. Watson. *Design Specification of HPF Mapper.* PPPE Deliverable D4.1.a, February 1993.

[18] M. Metcalf and J. Reid. *Fortran 90 Explained.* Oxford Science Publications/OUP, Oxford and New York, 1990.

[19] ParaSoft Corporation. *Express Fortran Reference Guide.* Version 3.0, 1990.

[20] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald. *Vienna Fortran – a language specification.* ICASE Internal Report 21, ICASE, Hampton, VA, 1992.

[21] H. Zima, B. Chapman. Compiling for Distributed-Memory Systems. *Proc. of the IEEE.* Special Section on Languages and Compilers for Parallel Machines, February 1993.