PARADIGM (version 2.0): A New HPF Compilation System*

Pramod G. Joisha
Northwestern University
ECE Department
2145 Sheridan Road, Evanston IL, USA
pjoisha@ece.nwu.edu

Prithviraj Banerjee
Northwestern University
ECE Department
2145 Sheridan Road, Evanston IL, USA
banerjee@ece.nwu.edu

Abstract

In this paper,[a] we present sample performance figures for a new linear algebra-based compilation framework implemented in a research HPF compiler called PARADIGM. The metrics considered include compilation times, execution times, and communication costs. We compare all of these metrics against commercial, industrial strength compilers such as `pghpf` *(v 2.2) and* `xlhpf` *(v 1.01) and show the superior benefits of PARADIGM (v 2.0) in all of the metrics used. We also demonstrate how robustly our framework performs in the presence of arbitrary alignments and distributions. The framework's symbolic manipulation capability is derived from an off-the-shelf commercial symbolic analysis software called Mathematica.[b] Measured metrics for a few popular benchmarks such as Automatic Differentiation and Integration (ADI), Euler Fluxes, TOMCATV and 2-D Explicit Hydrodynamics (EXPL) have been presented.*

1 Introduction

The High Performance Fortran Language (HPF) [9] had been proposed several years ago in response to the problem of making it easier for parallel application developers to write parallel code in a portable manner across different classes of parallel machines. The HPF language has very complicated and general semantics associated with data alignments (syntactically restricted linear functions) and data distributions (such as the general `CYCLIC`(B) distribution). In addition, input programs could contain loops having non-trivial loop bound expressions and array references bearing arbitrary subscript expressions. The compilation problem for such a complex scenario is indeed challenging and has been attempted by numerous research projects such as the Fortran D work from Rice University [8], the Vienna Fortran Project [6], the Fortran 90D Compiler Project from Syracuse University [5], and PARADIGM [3]. That apart, numerous commercial compilers from the Portland Group, Inc., (PGI), Digital Equipment Corporation (DEC), Applied Parallel Research (APR) and the International Business Machines (IBM) have been announced as well.

Different compiler research groups have taken different approaches to solve the general problem of compilation. Some groups, such as the original Fortran D research team from Rice University, and the IBM PTRAN Project [7], provided support for only `BLOCK` distributions—others, such as PARADIGM, could handle `BLOCK` and purely `CYCLIC` distributions. Some commercial compilers such as `pghpf` handle arbitrary alignments and distributions but they do so by incurring considerable communication overheads.

Thus, the questions that naturally arise are whether a compiler can handle the most general compilation cases, how effective is the code generated in terms of execution times and communication costs, and how intricate are the complexities of the overall compilation process, and to what extent does the latter affect compilation times.

In this study, we report on the latest version of the PARADIGM compiler, whose newest salient features include:

- Ability to support arbitrary alignments and distributions.
- Generate very efficient code in terms of execution times and communication costs (measured as the average number of bytes sent between processor pairs).
- Compilation times comparable to that of commercial compilers such as `pghpf` and `xlhpf`.

*This research was partially supported by the National Science Foundation under Grant NSF CCR–9526325, and in part by DARPA under Contract F30602–98–2–0144.

[a] A longer version of this paper has been submitted to the *IEEE Transactions on Software Engineering* (Special Issue on Parallel Computing).

[b] *Mathematica*, *MathLink* are registered trademarks of Wolfram Research, Inc.

1.1 Motivation

The crux of the HPF compilation problem involves finding a suitable means for capturing the information implied by the alignment and distribution directives. Originally, this was done in PARADIGM using a data structure known as the *Processor Tagged Descriptor* [13] or PTD for short. Though PTDs are attractive from the standpoint of supporting symbolic array sizes, variable number of processors and multidimensional distributions, their utility is restricted to `BLOCK` distributions alone. In fact, the original version of PARADIGM provided support for only `BLOCK` distributions and then subsequent work resulted in limited assistance for the `CYCLIC`(B) case through the Fourier-Motzkin Elimination (FME) technique [12]. The challenge arises in attempting to devise a uniform scheme that provides simultaneous support for the general `CYCLIC`(B) distribution.

1.2 Contributions

The main contributions of this work are threefold. First, we have shown how a linear algebra framework can be used to efficiently generate *efficient* SPMD code in the global address space, for inputs containing array accesses bearing complex affine subscript expressions and which occur within loops having affine bound expressions. In addition, the arrays are assumed to have the most general data alignments and data distributions. Ancourt et al. have shown in [2] how systems of inequalities could be used to conveniently express alignment and distribution information, in addition to synthesizing distributed code for loops qualified by the `INDEPENDENT` directive. We have applied their representations in our framework, extending the scheme to handle general `DO` loops.

Second, we have shown how to integrate a commercial symbolic analysis package such as *Mathematica* so as to achieve a fast compilation system in such a framework. Furthermore, through our implementation in *Mathematica*, we have shown how an environment that favors easy creation, debugging and maintenance of symbolic compilation algorithms can be readily designed.

Third, the performance of our techniques, both in terms of compilation times, run times and communication costs are compared against those used in PGI's `pghpf` and in IBM's `xlhpf`. We show that in all of these metrics, our schemes are very competitive.

1.3 Critique of our approach

Some critics of our approach to using *Mathematica* to perform symbolic analysis [12] believe that having an expensive communication conduit in the form of *MathLink* is an infeasible approach for realizing a fast compiler architecture. Agreeably in the past, the *MathLink* channel has been the hindering factor with regard to compilation times, since this constitutes the only way an external program can interact with the *Mathematica* kernel. Initially, our framework was implemented using release 0 of version 3 of *Mathematica*. The average compilation time exhibited in this case was within a factor of 2.50 of that of `pghpf`. However, in the latest release of *Mathematica* (release 1 of version 3), the *MathLink* interface has been significantly optimized for the Unix environment, and this makes it possible for our implementation to display an average compilation time that is *8% lesser* than that taken by PGI's `pghpf`! And for nearly all input program samples considered, the code generated by our infrastructure was also superior in execution times and communication costs than that generated by `pghpf`. Considering the fact that the compilation times were comparable on an average, even while having separate phases for the source-to-source conversion (which was done by PARADIGM) and the source-to-executable conversion (which was done using `mpxlf`), these results prove that the *Mathematica* engine is a justifiable choice and a strong contender for the realization of such frameworks.

2 An Overview of the Linear Algebra Framework

All information pertaining to arrays, alignments, distributions and sets in general are expressed through the solutions of systems of inequalities in our framework. Consider the following code excerpt in HPF:

```
...
      REAL A(31, 100)
...

!HPF$ TEMPLATE T(1:65, 2:150)
!HPF$ PROCESSORS P(2, 2)
!HPF$ ALIGN A(i, j) WITH T(2*i+3, j+3)
!HPF$ DISTRIBUTE T(BLOCK(35), CYCLIC(9)) ONTO P
```

Following the representations seen in [2], we can compactly express the above alignment and distribution information in terms of a collection of equalities and inequalities:

$$\hat{\mathcal{R}}\vec{t} = \hat{\mathcal{A}}\vec{a} + \vec{s_0} - \hat{\mathcal{R}}\vec{l_T} \tag{1}$$

$$\hat{\pi}\vec{t} = \hat{C}\hat{P}\vec{c} + \hat{C}\vec{p} + \vec{l} \tag{2}$$

$$\hat{\lambda}\vec{c} = \vec{0} \tag{3}$$

where

$$\vec{l_A} \leq \vec{a} \leq \vec{u_A} \tag{4}$$

$$\vec{0} \leq \vec{t} \leq \vec{u_T} - \vec{l_T} \tag{5}$$

$$\vec{0} \leq \vec{p} < \hat{P}\vec{1} \tag{6}$$

$$\vec{0} \leq \vec{l} < \hat{C}\vec{1} \tag{7}$$

For the given example, the corresponding matrices and vectors are:

$$\hat{\mathcal{R}} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \hat{\mathcal{A}} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}, \vec{s_0} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

and,

$$l_A^{\vec{}} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, u_A^{\vec{}} = \begin{pmatrix} 31 \\ 100 \end{pmatrix}$$

$$l_T^{\vec{}} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, u_T^{\vec{}} = \begin{pmatrix} 65 \\ 150 \end{pmatrix}$$

Also,

$$\hat{\pi} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \hat{C} = \begin{pmatrix} 35 & 0 \\ 0 & 9 \end{pmatrix}, \hat{\lambda} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

$$\hat{P} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

The solution set for $\vec{a}$ in the above system directly gives us the *ownership set*, which is a function of the processor identity vector $\vec{p}$, amongst other parameters such as the template cell vector $\vec{t}$.

2.1 Loop Partitioning

Due to the owner-computes rule, iterations associated with a sequential loop get distributed across a collection of processors in the SPMD equivalent version of the code. The *compute set* is the set of all those iterations—denoted by $\vec{\alpha}$—which cause $\vec{p}$ to execute the particular assignment statement by the owner-computes rule. Our definition for the compute set rests upon $\hat{L}$ and $\vec{b_0}$ which describe the bound expressions of the loop nest under consideration. For a loop nest having unit strides, $\hat{L}$ and $\vec{b_0}$ are so chosen such that the inequality $\hat{L}\vec{\alpha} \leq \vec{b_0}$ always holds.

If $\Delta_p(X)$ describes the ownership set for an array X with respect to a processor $\vec{p}$, then the compute set for an assignment statement S' whose left-hand side array reference is $X(\hat{S}_x\vec{\alpha} + a_{0_x}^{\vec{}})$, becomes

$$\wp_p(S') = \{\vec{\alpha} | \hat{S}_x\vec{\alpha} + a_{0_x}^{\vec{}} \in \Delta_p(X) \ \wedge \ \hat{L}\vec{\alpha} \leq \vec{b_0}\} \qquad (8)$$

2.2 Communication Generation

Defining the *view set* $\nabla_p(S', Y(\hat{S}_y\vec{\alpha} + a_{0_y}^{\vec{}}))$ for a right-hand side array reference $Y(\hat{S}_y\vec{\alpha} + a_{0_y}^{\vec{}})$ contained in an assignment statement S' as the set of elements of the array Y required for execution by a processor $\vec{p}$ on account of its compute work, we have, in set-theoretic notation

$$\begin{array}{l} \nabla_p(S', Y(\hat{S}_y\vec{\alpha} + a_{0_y}^{\vec{}})) = \\ \{\vec{a} | \vec{\alpha} \in \wp_p(S') \ \wedge \ \vec{a} = \hat{S}_y\vec{\alpha} + a_{0_y}^{\vec{}}\} \end{array} \qquad (9)$$

Assuming that a dependence stretches from a reference to the array Y occurring on the left-hand side of another assignment statement S'', to $Y(\hat{S}_y\vec{\alpha} + a_{0_y}^{\vec{}})$ occurring on the right-hand side of S', we find the *send set* ($\overline{\mathcal{S}}$) to be

$$\begin{array}{l} \overline{\mathcal{S}}(S'', S', Y(\hat{S}_y\vec{\alpha} + a_{0_y}^{\vec{}}), \vec{p}, \vec{q}) = \\ \Delta_p(Y) \cap \nabla_q(S', Y(\hat{S}_y\vec{\alpha} + a_{0_y}^{\vec{}})) \end{array} \qquad (10)$$

The *receive set* ($\overline{\mathcal{R}}$) is defined similarly.

If the right-hand side array reference of an assignment statement is *identically mapped* onto the same processor which owns the left-hand side array reference, one could completely avoid generating communication code for the pair. By being identically mapped onto the same processor, we imply that for *all possible legal values* that the subscript expressions may assume by freely varying the loop iteration vector, the two array references still get ultimately distributed onto the same processor.

3 Execution Environment

Compilation times for our framework have been evaluated by considering the source-to-source transformation effected by PARADIGM (version 2.0), *as well as* the source-to-executable compilation done using `mpxlf` (version 3.02). The source-to-source compilation times for PARADIGM were measured on an HP Visualize C180 with a 180MHz HP PA-8000 CPU, running HP-UX 10.20 and having 128MB of RAM. Compilation times for `pghpf` (version 2.2), `xlhpf` (version 1.01) as well as `mpxlf` were measured on an IBM E30 running AIX 4.1 and having a 133MHz PowerPC 604 processor and 96MB of main memory. The parallel codes were executed on a 16-node IBM SP2 multicomputer, running AIX 4.1 and having the high performance SP switch and adapter, and in which each processor was a 62.5MHz POWER node having 128MB of RAM. Results for the codes generated by PARADIGM were also obtained on an IBM J30* shared-memory 8-node multiprocessor running AIX 4.1 and having 1024MB of main memory and in which each processor was the 166MHz PowerPC 604e processor. To gauge performance on a distributed-shared-memory machine, results for codes compiled by PARADIGM were also taken on an 8-node SGI Origin 2000 having the 195MHz MIPS R10000 CPU along with 1024MB of main memory, and running IRIX 6.4.

The *MathLink* protocol used was based on the Unix IPC Pipe primitive. The communication costs were measured using IBM's Visualization Tools (version 2.0). In those tables that tabulate the execution times, the RS6000 column refers to the sequential execution times obtained on the IBM E30. The R10000 column similarly refers to the sequential execution times obtained on the SGI Origin 2000.

3.1 About the `pghpf` (version 2.2) Measurements

For all compilations done using `pghpf` on the original sequential input programs, the `-Mautopar` option was always used. According to [11], the `-Mautopar` option causes the compiler to generate `FORALL` statements and calls to reduction intrinsics when "parallel" `DO` loops which

operate on distributed arrays are discovered. The manual also states that under this option, the compiler may also perform loop interchange and distribution operations with the objective of increasing the number of parallel loops.

Codes were also generated with `pghpf` by replacing the `DO` loops in the various input samples with equivalent `FORALL` constructs where possible, and by inserting the HPF `INDEPENDENT` directive appropriately. Note that the `FORALL` constructs and the `INDEPENDENT` directives were not mixed in any of the input samples. In fact, to obtain good performance with codes generated using `pghpf`, it is normally recommended to manually replace the `DO` loops in the original source by equivalent `FORALL` forms where possible, and to introduce the `INDEPENDENT` directive when appropriate.

In general, the `pghpf` compiler was found to generate efficient code for the `BLOCK` distribution case, in the presence of the `FORALL` construct. The tabulated execution times show this to be true for most input samples across the various benchmarks. For the `CYCLIC`(B) distribution case, we hypothesize that the large communication costs exhibited by the codes compiled using `pghpf` are in part due to the exchange of whole arrays between processors. In fact, since `pghpf` produces code in the local address space, it appears that the generated code dynamically allocates and deallocates significantly large temporary arrays, which apart from affecting the run times, causes the compiled code to often complain of a memory allocation error at run-time. Thus, in those tables that compare the execution times of `pghpf`-generated code with that of PARADIGM and `xlhpf`, we have chosen to tabulate only the best run times for the `pghpf`-compiled code; the readings are suffixed by either of the letters A (indicating the use of the `-Mautopar` option), F (indicating the use of the `FORALL` statement) or I (indicating the use of the `INDEPENDENT` directive).

3.2 Data Alignments and Data Distributions

For all benchmarks, the most suitable data alignments and data distributions were chosen through the `ALIGN` and `DISTRIBUTE` directives respectively. In addition, for each of the input samples, we varied the data distributions through a set of patterns, maintaining the same data alignments with the templates involved. Only the `PROCESSORS` and the `DISTRIBUTE` directives were changed in every benchmark's input sample. For example, Figure 1 displays the particular data alignment that was chosen for all of the ADI benchmark input samples. The data distributions were *arbitrarily* chosen, the idea being to demonstrate the ability of our framework to handle any given data alignment and data distribution combination.

4 Measurements

In this section, we present timing and communication cost results that reflect the quality of the code generated by our framework, in addition to illustrating the efficiency of our compilation schemes. Apart from showing how well the framework performs in terms of the various input cases that it is capable of handling, these results demonstrate that such gains are possible without losing out on compilation times. For all input samples, we compare our infrastructure's performance against those of commercially available compilers. Compilations done by PARADIGM on the sequential input samples were in no way user-aided.

4.1 ADI

The only flow dependencies that the ADI benchmark (from the Livermore Kernel 8 [10]) exhibited were three loop-independent ones, and all of the source-sink statement pairs were contained in the innermost loop. However, the right-hand side references on which these dependencies terminate could be aligned and distributed onto the same processor onto which the corresponding left-hand side references were ultimately mapped. This resulted in partitioned loops which did not contain any communication code.

```
!HPF$ TEMPLATE T(4, 1024, 2)
!HPF$ ALIGN DU1(i) WITH T(*, i, *)
!HPF$ ALIGN DU2(i) WITH T(*, i, *)
!HPF$ ALIGN DU3(i) WITH T(*, i, *)
!HPF$ ALIGN AU1(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN AU2(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN AU3(i, j, k) WITH T(i, j, k)
```

Figure 1. Alignment directives used for all the ADI benchmark input samples.

As may be gleaned from the information presented in the table of compilation times, we see that for this benchmark, our system with *Mathematica* 3.0.1 required an av-

[c] Arithmetic means for communication costs were taken by considering a sample size of at least 6 readings across various processor array sizes and distributions.

[d] Averages for compilation times were taken by considering a sample size of at least 12 readings across various processor array sizes and distributions.

[e] Code compiled using `pghpf` complains of "invalid alignment."

[f] `pghpf` compilation abnormally terminated due to a signal.

[g] `xlhpf` does not permit a `CYCLIC` blocking factor greater than unity.

[h] Code generated by `xlhpf` unable to finish; terminated even when job time was extended to a full hour.

[i] `xlhpf` produces code that abnormally exits due to a segmentation fault.

[j] The IBM J30* and the SGI Origin 2000 consist of only 8 processors.

erage source-to-executable compilation time that was 0.94 times that required by pghpf.

In the ADI benchmark, the innermost loop maybe directly rewritten using the FORALL construct. However, as far as pghpf was concerned, this did not make any difference to the execution times and communication costs of the compiled codes. In fact, in the case of this benchmark, on analyzing the intermediate Fortran 77 codes generated by pghpf for the original input samples, it appeared that with the -Mautopar option, the compiler was intelligent enough to detect that the innermost loop could be replaced by a FORALL equivalent. Even when the INDEPENDENT directive was inserted into the original input samples for this benchmark, the execution times as well as the communication costs displayed no overall improvements, but in fact on an average increased.

4.2 Euler Fluxes

The Euler Fluxes benchmark (from FLO52 in the Perfect Club Suite [4]) demonstrates our framework's ability to handle scalar assignments. Two communication constructs (each consisting of an MPI_SEND and MPI_RECV pair) were generated corresponding to a pair of array references on which loop-independent flow dependencies terminated. Other right-hand side array references on which flow dependencies ended were identically mapped onto the same processor that owned the left-hand side array references of the same assignment statements.

```
!HPF$ TEMPLATE T(0:5001, 34, 4)
!HPF$ ALIGN FS(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN DW(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN W(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN X(i, j, k) WITH T(i, j, k)
!HPF$ ALIGN P(i, j) WITH T(i, j, *)
```

Figure 2. Alignment directives used for all the Euler Fluxes benchmark input samples.

From Table 1, we see that the average compilation time required by our system using *Mathematica* 3.0.1, was about half of that required by pghpf.

Amongst the five loop nests contained in this benchmark, two could be directly converted to equivalent FORALL forms, while all the five loop nests were fully qualified to be declared independent. Converting the two loop nests into the corresponding FORALL forms did not improve the performance of the input samples when compiled using pghpf. In fact, the communication volume increased on only two occasions, though the execution times remained more or less the same. Insertion of the INDEPENDENT directive further worsened the situation of the pghpf-compiled codes, since the executables for only a few input samples were able to successfully execute. Surprisingly, those that were able to execute displayed appreciably reduced execution times with respect to the corresponding FORALL cases, though the communication volumes were significantly larger. Codes compiled by PARADIGM however, were on an average over twice as fast as the input samples that were rewritten with the FORALL construct and which were compiled using pghpf.

4.3 TOMCATV

The TOMCATV benchmark [1] comprises of four doubly nested loops and for the chosen alignment and distribution directives, the only communication constructs that arise are due to a loop-carried flow dependence and a simple flow dependence.

```
!HPF$ TEMPLATE T(0:1025, 0:1025)
!HPF$ ALIGN AA(i, j) WITH T(i, j+1)
!HPF$ ALIGN DD(i, j) WITH T(i, j)
!HPF$ ALIGN D(i, j) WITH T(i, j)
!HPF$ ALIGN X(i, j) WITH T(i, j)
!HPF$ ALIGN Y(i, j) WITH T(i, j)
!HPF$ ALIGN RX(i, j) WITH T(i, j)
!HPF$ ALIGN RY(i, j) WITH T(i, j)
```

Figure 3. Alignment directives used for all the TOMCATV benchmark input samples.

From the table of compilation times, we see that for this benchmark, our framework's average compilation time using *Mathematica* 3.0.1 was about 0.78 times that of pghpf. While the average communication cost for codes compiled using PARADIGM was 745 (measured as the average number of bytes sent per processor pair), the same figures for the codes compiled using pghpf were 1101027—both with the -Mautopar option as well as with the FORALL construct—and 3043537 with the INDEPENDENT directive inserted. The execution times for the codes produced using PARADIGM were better than that of codes produced by pghpf, except when the INDEPENDENT directive was used.

4.4 2-D Explicit Hydrodynamics

The EXPL benchmark (from the Livermore Kernel 18 [10]) provides opportunities for performing message coalescing and aggregation. However, currently for the general case, this feature is yet to be implemented in our infrastructure. As a result, for this benchmark, code generated by PARADIGM exhibits communication costs that are larger than necessary; in fact, the communication volumes are consistently more than that seen in codes compiled with pghpf with the -Mautopar option. The same

was also true when the input samples were rewritten using the FORALL construct and compiled by pghpf. However, with the INDEPENDENT directive inserted, the communication cost exhibited by the pghpf-compiled codes was once again much larger than that of the codes generated using PARADIGM.

```
!HPF$ TEMPLATE U(0:1049, 0:1049)
!HPF$ ALIGN ZA(i, j) WITH U(i, j)
!HPF$ ALIGN ZB(i, j) WITH U(i, j)
!HPF$ ALIGN ZM(i, j) WITH U(i-1, j+1)
!HPF$ ALIGN ZP(i, j) WITH U(i+1, j+1)
!HPF$ ALIGN ZQ(i, j) WITH U(i-1, j+1)
!HPF$ ALIGN ZR(i, j) WITH U(i, j)
!HPF$ ALIGN ZU(i, j) WITH U(i, j)
!HPF$ ALIGN ZV(i, j) WITH U(i, j)
!HPF$ ALIGN ZZ(i, j) WITH U(i, j)
```

Figure 4. Alignment directives used for all the EXPL benchmark input samples.

Table 1 reveals that for this benchmark, our framework's average compilation time using *Mathematica* 3.0.1 was about 3.12 times that of pghpf. Also, the amount of communication code that was generated by our system was the largest for this benchmark. Though the communication volume displayed by codes generated by PARADIGM was larger than the pghpf equivalents, yet in terms of execution times, the codes produced by PARADIGM were the fastest, when compared to both the FORALL and the INDEPENDENT versions that were compiled using pghpf, as well as the parallel codes directly produced by pghpf with the -Mautopar option.

5 Conclusions

The preceding results show the promising nature of our new linear algebra-based HPF compilation framework. The system's design was motivated by a need to efficiently handle the general problem of generating parallel code in the presence of arbitrary alignment and distribution directives as well as complex affine expressions. By harnessing the advanced symbolic manipulation capabilities of *Mathematica*, we were able to realize efficient implementations of our algorithms.

In terms of communication costs, the parallel codes generated by our framework were better in all instances except for the EXPL benchmark input samples, and this was due to an absence of a coalescing and message aggregation phase in our system. The incorporation of such a pass, capable of handling the general case is a future direction of our work.

On numerous occasions, huge differences in communication volumes between codes produced by PARADIGM and those compiled by pghpf were seen. Moreover, the execution times of the parallel codes generated by PARADIGM were in general superior compared to that of code compiled by pghpf with the -Mautopar option. This was often true even when DO loops were replaced by the equivalent FORALL forms where possible, and when the INDEPENDENT directive was inserted. And yet, the remarkable feat was that these achievements were possible without adversely increasing compilation times—the compilation times of our framework were still comparable to that of pghpf.

Thus, all of the above facts taken together strongly justify the choice of using *Mathematica* for advanced symbolic analysis and prove rigorously the practicality and feasibility of our approach.

References

[1] The SPEC CPU92 Benchmark Suite. At http://www.specbench.org/osg/cpu92.

[2] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A Linear Algebra Framework for Static HPF Code Distribution. Technical Report A–278–CRI, Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris, 35, rue Saint-Honoroé, F–77305 Fontainebleau cedex, France, Nov. 1995.

[3] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Y.-H. Su. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, Oct. 1995.

[4] M. Berry et al. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, Fall 1989.

[5] Z. Bozkus, A. Choudhary, G. C. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for Distributed-Memory MIMD Computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, Apr. 1994.

[6] B. M. Chapman, P. Mehrotra, and H. P. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Aug. 1992.

[7] M. Gupta, S. P. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo. An HPF Compiler for the IBM SP2. In *Supercomputing '95*, San Diego, CA, USA, Dec. 1995.

[8] S. Hiranandani, K. Kennedy, C. H. Koelbel, U. Kremer, and C.-W. Tseng. An Overview of the Fortran D Programming System. In *4th Workshop on Languages and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science*, pages 18–34, Santa Clara, CA, USA, Aug. 1991. Springer Verlag.

[9] C. H. Koelbel, , D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA 02142, USA, 1994.

[10] F. M. McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. Technical Report UCRL–55745, Lawrence Livermore National Laboratory, 7000 East Avenue, Livemore, CA, USA, Dec. 1986.

[11] The Portland Group, Inc., 9150 SW Pioneer Court, Wilsonville, OR 97070, USA. *pghpf User's Guide*, May 1996.

[12] E. Y.-H. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-Memory Multicomputers. In *9th ACM International Conference on Supercomputing*, pages 424–433, Barcelona, Spain, July 1995.

[13] E. Y.-H. Su, D. J. Palermo, and P. Banerjee. Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers. In *1994 International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Montréal, Canada, Aug. 1994.

Table 1. Compilation times in seconds.

Benchmark	Processor Array Size	Distribution	pdm (v 2.0) (with *Mathematica* 3.0.0)	pdm (v 2.0) (with *Mathematica* 3.0.1)	mpxlf (v 3.02)	pghpf (v 2.2)	xlhpf (v 1.01)
ADI	2×2	(BLOCK, BLOCK, *)	37	07	01	09	10
	1×8	(BLOCK, CYCLIC(200), *)	47	08	01	09	-[g]
	1×16	(BLOCK, CYCLIC(200), *)	40	08	01	09	-[g]
Average compilation times-[d]			40.42	8.08	1.00	9.67	9.00
Euler Fluxes	4×1	(BLOCK, CYCLIC(8), *)	33	15	13	41	-[g]
	2×4	(BLOCK, *, CYCLIC)	33	27	07	54	28
	16×1	(*, BLOCK, CYCLIC)	42	14	07	50	07
Average compilation times			37.12	16.19	9.62	44.56	13.33
TOMCATV	2×2	(BLOCK, BLOCK)	46	10	11	37	22
	4×2	(BLOCK, CYCLIC(260))	56	14	32	63	-[g]
	4×4	(CYCLIC(256), CYCLIC(300))	65	21	43	61	-[g]
Average compilation times			55.33	14.83	28.08	55.00	23.75
EXPL	2×2	(BLOCK, BLOCK)	47	11	23	9	11
	4×2	(BLOCK, CYCLIC(250))	60	16	78	32	-[g]
	4×4	(CYCLIC(290), CYCLIC(256))	80	31	102	38	-[g]
Average compilation times			61.75	17.25	67.42	27.17	14.50

Table 2. Execution times in seconds for different processor congurations and distributions.

Benchmark	Processor Array Size	Distribution	IBM AIX 4.1					SGI IRIX 6.4	
			RS6000	SP2			J30*	R10000	Origin 2000
			xlf (v 3.02)	pdm (v 2.0)	pghpf (v 2.2)	xlhpf (v 1.01)	pdm (v 2.0)	f90 (v 7.2)	pdm (v 2.0)
ADI	2×2	(BLOCK, BLOCK, *)	5.03	0.71	50.45A	0.48	2.30	3.74	1.11
	1×8	(BLOCK, CYCLIC(200), *)	4.65	6.14	5.98F	-[g]	4.00	3.74	0.72
	1×16	(BLOCK, CYCLIC(200), *)	4.07	5.01	6.11F	-[g]	*[j]	3.74	*[j]
Euler Fluxes	4×1	(BLOCK, CYCLIC(8), *)	70.32	34.31	94.40I	-[g]	50.07	18.24	3.48
	2×4	(BLOCK, *, CYCLIC)	70.33	6.50	16.06F	-[h]	17.57	18.25	3.71
	16×1	(*, BLOCK, CYCLIC)	70.19	3.94	6.83F	6.27	*[j]	18.25	*[j]
TOMCATV	2×2	(BLOCK, BLOCK)	10.43	3.09	1.43A	-[i]	4.37	5.94	5.71
	4×2	(BLOCK, CYCLIC(260))	10.08	3.41	2.23I	-[g]	4.32	5.95	4.22
	4×4	(CYCLIC(256), CYCLIC(300))	10.33	4.09	4.43I	-[g]	*[j]	5.95	*[j]
EXPL	2×2	(BLOCK, BLOCK)	20.59	3.33	2.59A	2.53	7.04	5.38	1.53
	4×2	(BLOCK, CYCLIC(250))	20.61	1.86	2.24A	-[g]	4.51	5.38	0.96
	4×4	(CYCLIC(290), CYCLIC(256))	20.76	1.02	1.37F	-[g]	*[j]	5.38	*[j]

Table 3. Communication costs (measured as the average number of bytes sent per processor pair) obtained using pghpf with the -Mautopar option, the FORALL construct and the INDEPENDENT directive respectively, and compared against that obtained using pdm and xlhpf.

Benchmark	Processor Array Size	Distribution	IBM AIX 4.1 (SP2)				
			pdm (v 2.0)	pghpf (v 2.2)			xlhpf (v 1.01)
				-Mautopar	FORALL	INDEPENDENT	
ADI	2×2	(BLOCK, BLOCK, *)	0	18868302	18868302	75399246	4622
	1×4	(BLOCK, CYCLIC(200), *)	0	30798	30798	30798	-[g]
	1×8	(BLOCK, CYCLIC(200), *)	0	7725	7725	7725	-[g]
Arithmetic means-[c]			0	5514045	5514045	15134272	1936
Euler Fluxes	4×1	(BLOCK, CYCLIC(8), *)	9600	10458	10458	116379503	-[g]
	4×1	(*, BLOCK, CYCLIC)	1499400	1560063	1560063	-[f]	1510660
	2×4	(BLOCK, *, CYCLIC)	2400	604491	604491	-[f]	-[h]
Arithmetic means			362531	584903	694986	116379503	790009
TOMCATV	2×2	(BLOCK, BLOCK)	511	1573461	1573461	-[e]	-[i]
	2×2	(BLOCK, CYCLIC(260))	1533	1849380	1849380	4774200	-[g]
	4×2	(CYCLIC(256), CYCLIC(300))	383	464919	464919	1315597	-[g]
Arithmetic means			745	1101027	1101027	3043537	-
EXPL	2×2	(BLOCK, BLOCK)	36505	29223	29223	160303	34449
	4×2	(BLOCK, CYCLIC(250))	31941	29738	29738	13151045	-[g]
	4×2	(CYCLIC(290), CYCLIC(256))	31941	28130	28130	12961068	-[g]
Arithmetic means			56278	49820	49820	19396911	25772