Fortran RED — A Retargetable Environment for Automatic Data Layout*

Ulrich Kremer

Department of Computer Science
Rutgers University
email: uli@cs.rutgers.edu

Abstract. The proliferation of parallel platforms over the last ten years has been dramatic. Parallel platforms come in different flavors, including desk–top multiprocessor PCs and workstations with a few processors, networks of PCs and workstations, and supercomputers with hundreds of processors or more. This diverse collection of parallel platforms provide not only computing cycles, but other important resources for scientific computing as well, such as large amounts of main memory and fast I/O capabilities. As a result of the proliferation of parallel platforms, the "typical profile" of a potential user of such systems has changed considerably. The specialist user who has a good understanding of the complexities of the target parallel system has been replaced by a user who is largely unfamiliar with the underlying system characteristics. While the specialist's main concern is peak performance, the non–specialist user may be willing to trade off performance for ease of programming.

Recent languages such as High Performance Fortran (HPF) and SGI Parallel Fortran are a significant step towards making parallel platforms truly usable for a broadening user community. However, non-trivial user input is required to produce efficient parallel programs. The main challenge for a user is to understand the performance implications of a specified data layout, which requires knowledge about issues such as code generation and analysis strategies of the HPF compiler and its node compiler, and the performance characteristics of the target architecture. This paper discusses our preliminary experiences with the design and implementation of Fortran RED , a tool that supports Fortran as a deterministic, sequential programming model on different parallel target systems. The tool is not part of a compiler. Fortran RED uses HPF as its intermediate program representation since the language is portable across many parallel platforms, and commercial and research HPF compilers are widely available. Fortran RED is able to support different target HPF compilers and target architectures, and allows multi–dimensional distributions in addition to dynamic remapping. This paper focuses on the discussion of the performance prediction component of the tool and reports preliminary results for a single scientific kernel on two target systems, namely PGI's and IBM's HPF compilers with IBM's SP–2 as the target architecture.

* This research was supported by DARPA contract DABT 63-93-C-0064 and experiments were conducted using resources at the Cornell Theory Center.

1 Introduction

Parallel platforms provide not only computing cycles, but also large amounts of main memory[2] and fast I/O capabilities. The increase in availability of parallel computing resources to a larger user community is far from being matched with the availability of software tools that allow easy access and use of these resources. The potential user of a parallel system may not be familiar or willing to deal with the complexities of efficient parallel programming. As a consequence, valuable computing resources may be underutilized or even wasted.

Languages such as High Performance Fortran (HPF) [19] or SGI Parallel Fortran [38] provide a shared name space programming model augmented by directives that specify how the data is mapped onto the individual processors. The user of such languages is able to avoid many complexities of explicit parallel programming. Based on the user supplied data layout specifications, the compiler and/or operating system ensure the correct computation and data placements. However, the user is still faced with the difficult problem of choosing an efficient data layout. An efficient data layout minimizes communication while maximizing exploitable parallelism. The best choice of a data layout depends on many factors including the program's computation characteristics, the number of processors used, the performed compiler optimizations, the target machine characteristics, and the problem size. All these factors and their interactions make it extremely hard for a user to choose an efficient data layout.

In this paper, we discuss and evaluate techniques for automatic data layout in the context of a programming environment that can target different compilers and different parallel platforms. Portability across different target compilers and platforms is necessary to achieve the goal of efficient, platform independent programming. A crucial challenge for our automatic data layout approach is the design of a performance predictor that is portable across different target systems and is able to rank correctly a given set of data layout alternatives, while being as simple and efficient as possible. This paper describes the design of a compositional performance predictor for an automatic data layout tool.

A prototype tool based on our framework for automatic data layout has been implemented on top of the D System compiler infrastructure developed at Rice University [1]. The Fortran RED (R*etargetable* E*nvironment for* D*atalayout*) tool takes sequential Fortran programs as input and generates HPF programs as output. The input Fortran programs solve regular problems, i.e., use dense arrays as their main data structures. Regular problems allow computation and communication requirements of the application to be derived at compile time. The overall structure of the Fortran RED tool is shown in Figure 1.

Fortran RED builds and examines search spaces of candidate data layouts. A candidate layout is an efficient layout for some part of the program. These program parts are called *program phases* and are automatically determined by

[2] In fact, some researchers argue that machines with a large main memory should always have multiple processors in order to make cost–effective use of the memory's capacity and bandwidth [41].

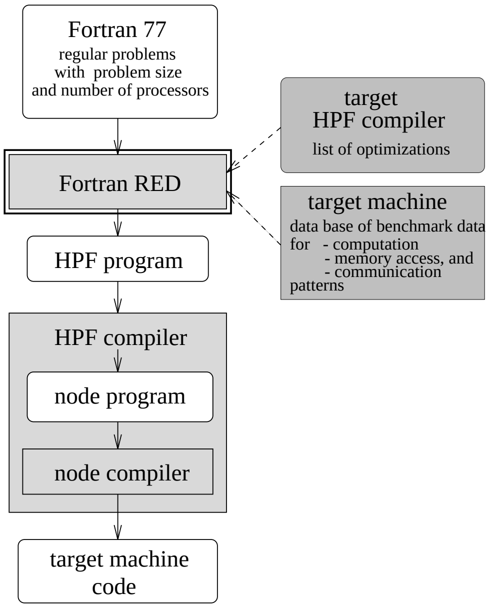
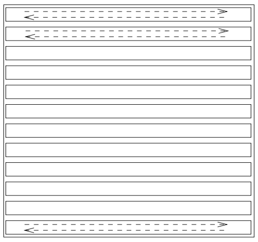


Fig. 1. Overview of Fortran RED

the tool. After the generation of candidate layout search spaces for each phase, a single candidate layout is selected from each search space, resulting in a data layout for the entire program. Note that the optimal selection may consist of candidate data layouts that are each suboptimal for their phases, or may require the remapping of arrays between program phases. A compositional performance estimator based on a compiler, execution, and a machine model is used to predict the execution time of each candidate layout and the costs of possible remappings between candidate data layouts. The compiler model is parameterized with respect to the optimizations performed by the target HPF compiler, including its node compiler. The execution model classifies program parts according to their communication and synchronization patterns in order to identify the critical path for its execution. The machine model uses a set of HPF benchmark kernels to measure the performance of basic computation, memory access, and communication patterns. The benchmark kernels are compiled with the target compiler and executed on the target machine once at tool installation time. The resulting measurements are stored in a performance data base and accessed by the tool during program analysis. A detailed discussion of our general framework for automatic data layout can be found in [22].

The paper is organized as follows. Section 2 presents the scientific program kernel used in our preliminary experiments. The example program that shows the need for a tool such as Fortran RED . Section 3 discusses our compositional performance prediction approach in detail. The paper concludes with a discussion

```
REAL c(N, N), a(N, N), b(N, N)

// loop nest performing READ (c, a, b) operation

DO   iter = 1, max
   // Forward and backward sweeps along rows

   DO j = 2, N
      DO i = 1, N
         c(i, j) = c(i, j) - c(i, j - 1) * a(i, j) / b(i, j - 1)
         b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j - 1)
      ENDDO
   ENDDO

   DO i = 1, N
      c(i, N) = c(i, N) / b(i, N)
   ENDDO

   DO j = N - 1, 1, -1
      DO i = 2, N
         c(i, j) = ( c(i, j) - a(i, j + 1) * c(i, j + 1) ) / b(i, j)
      ENDDO
   ENDDO
```



```
   // Downward and upward sweeps along columns

   DO i = 2, N
      DO j = 1, N
         c(i, j) = c(i, j) - c(i - 1, j) * a(i, j) / b(i - 1, j)
         b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i - 1, j)
      ENDDO
   ENDDO

   DO j = 1, N
      c(N, j) = c(N, j) / b(N, j)
   ENDDO

   DO i = N - 1, 1, -1
      DO j = 1, N
         c(i, j) = ( c(i, j) - a(i + 1, j) * c(i + 1, j) ) / b(i, j)
      ENDDO
   ENDDO

ENDDO

// loop nest performing WRITE (c, b) operation
```

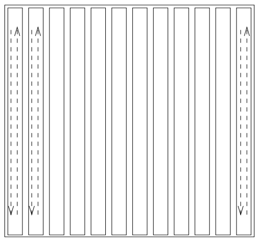


Fig. 2. ADI integration kernel with computation illustration

of experimental results in Section 4, related work in Section 5, and our overall conclusions and directions for future work in Section 6.

2 Example Scientific Program

ADI integration is a technique frequently used to solve partial differential equations (PDEs). Figure 2 shows the code of an ADI kernel computation. The execution of the ADI integration kernel consists of a repeated sequence of forward and backward sweeps along rows, followed by downward and upward sweeps along columns. For the sweeps along the rows, a row layout has the best performance. The same holds for a column layout for the column sweeps. Transposing the arrays between the all row and all column sweeps eliminates communication within the sweeps. In contrast, choosing the same data layout for both, row and column sweeps will avoid communication between the sweeps but will make communication necessary either in the row or column sweeps.

The best data layout choice will depend on the speed of the communication hardware and software of the parallel target architecture, and the ability of the compiler to exploit the available parallelism efficiently. In addition to the compiler dependence, the performance characteristics of the underlying I/O system may be considered in the data layout choice since the program performs read and write disk accesses. Finally, the actual size N of the arrays and the number of available processors may influence the data layout choice. The prototype implementation of Fortran RED generates a data layout for ADI for a given target system, number of available processors, and problem size. The current implementation does not consider the characteristics of the underlying I/O subsystem of the target system.

3 Performance Model

Any optimizing compiler relies on some form of performance prediction to guide its decision process between optimization alternatives. The precision of the performance prediction has to allow the correct ranking of the optimization alternatives, i.e., the relative ranking should match the measured performance improvements of the optimization alternatives. The performance model in Fortran RED has to be able to rank correctly the different candidate data layouts for the entire program or single program segments in most cases. This is a non-trivial challenge since the performance of a candidate layout is compiler, problem size, machine architecture and machine size dependent. Clearly, there is a tradeoff between prediction cost and prediction quality. The performance model has to be portable across different target systems. In fact, the key to the portability of Fortran RED is the portability of its underlying performance prediction model.

The performance model used in Fortran RED is compositional in the sense that it first models the optimizations performed by the compiler, followed by a model for the execution of the generated, message–passing node programs. In turn, the execution model itself relies on a machine model that produces performance numbers for basic computation, communication, and memory access patterns. Based on the machine model predictions, the problem size, and the number of processors, the execution model aggregates the costs along the critical path of the execution.

The compiler model is parameterized with respect to a set of optimization transformations, in particular communication optimizations such as message vectorization and message coalescing. The execution model is target compiler and target architecture independent, but uses the problem size and number of processors to look–up the corresponding performance predictions in the machine model data base. The machine model consists of a collection of kernel programs written in HPF. These programs are compiled with the target compiler, and executed on the target system for ranges of different parameter values. These parameters include message length, unit or non-unit stride memory accesses, and number of processors. Retargeting the machine model to a new target system

```
Do i = 1, N
   do j = 1, N
      b(i,j) = a(i-1,1)
```

Fig. 3. Candidate layout dependent communication patterns

consists of rebuilding the performance data base by compiling and executing all HPF kernel patterns on the new system. This has only to be done once at tool installation time. It is important to note that the same HPF kernel patterns are used for all target systems, allowing true portability. The following sections discuss the compiler, execution, and machine models in more detail.

3.1 Compiler Model

The compiler simulation module analyzes RHS and LHS of assignment statements for each program phase and candidate data layout. In the first step, local iteration sets are computed that represent local loop iterations that have to be executed by each node processor. Based on these iterations, a data structure is built for each phase that maps phase loop levels to a sets of regular section descriptors (RSDs [7, 17]) that describe the generated communication patterns. Currently supported communication patterns are `shift`, `send_receive`, `broadcast`, and `reduction`. Compiler flags are used to enable communication optimizations such as message vectorization, message coalescing, and message aggregation [39, 15, 4]. These optimizations are simulated using array subscript dependence analysis to determine *when* the communication occurs and RSDs information to describe *what* data elements have to be communicated. A communication pattern that is marked as loop carried at a loop level will lead to a pipelined execution at the entire phase loop nest, where a single execution of the loop at the communication level and the levels below is a single stage of the pipeline.

RSDs are constructed for each distributed dimension of an array separately and stored at the appropriate phase nesting level. The subsequent execution model interprets the overall interactions of the communication patterns. A single array reference, such as $a(i-1,1)$ shown in Figure 3, may lead to different communication patterns for different candidate layouts. It is important to note that the goal of the compiler model is not to determine the best communication optimizations or compilation strategy possible, but to be able to simulate the transformation and optimization process performed by a set of target compilers. Clearly, the compiler model has to make tradeoffs between simulation cost and precision.

Assuming that all candidate layouts have the perfect alignment (`Align a With b`) for the program segment shown in Figure 3, the `(*,block)` distribution will require the first column of array a to be broadcast to all processors. In the case of a `(block,*)` distribution, a single element, nearest neighbor shift communication is sufficient. A `(block,block)` distribution may result in a multicast and shift communication. The placement of the communication within the loop is compiler dependent.

3.2 Execution Model

Once locations and types of compiler generated communications are known for a candidate layout and its phase, an *execution model* is used to estimate the performance effects of synchronizations induced by the communications. Communication inside a phase may lead to a pipelined execution of the loop. Communication outside of the phase may result in a loosely synchronous execution scheme [12]. In addition, special communication patterns may be recognized that represent global operations such as reductions. Based on the synchronization schemes and the costs for simple communication patterns and basic computations, the execution model determines the overall cost estimate for a candidate layout and its phase.

In a loosely synchronous phase, all processors operate in a loose lockstep, consisting of alternating steps of parallel asynchronous computation and synchronous communication. The performance of the overall program can be estimated by predicting the cost of the "representative" node program that operates on the largest local data segment. This performance model assumes that the largest segment has the highest computation and communication costs and therefore represents the critical path in a loosely synchronous execution of the entire program. The computation and communication costs for the largest local data segment are just added up to determine the overall phase performance.

Performance prediction of a pipelined phase execution is more complicated than in the loosely synchronous case due to the structure of the underlying critical execution path. Execution models that can estimate pipelines of different granularity have been discussed in the literature, for instance in [26, 32]. For a pipelined phase, Fortran RED uses the innermost level that carries a true dependence to determine the granularity of the pipeline. This level is referred to as the pipeline level. The performance estimate for a single pipeline stage is the predicted computation and memory access cost for a single iteration at the pipeline level. The number of stages executed by each processor is the number of iterations of the outer loops within the phase that enclose the pipeline level. The execution time for the entire pipeline is determined by the time needed by the last processor to finish all its stages and by its starting delay relative to the first processor of the pipelined execution. The pipeline model implemented in Fortran RED is similar to the model described by Mellor-Crummey, Adve, and Koelbel [26], and allows pipelined execution across different distributed dimensions, i.e., multi–dimensional pipelines. For the special case of a reduction phase, Fortran RED determines the kind of reduction operation and its data type, for instance `min`, `max`, or `sum` of type double_precision. The costs of the global reductions are derived from corresponding HPF reduction benchmark kernels.

PATTERN	NAME	HPF CODE†	PREDICTION
		COMMUNICATION	

shift
unit stride
message size=N
#processors=P
	$shift_{unit}(N,P)$	`Dimension x(N,P), y(N,P)`
		`Align x With y`
		`Processors t(P)`
		`Distrib (*,block) x Onto t`
		`do j=2,P`
		`do i=1,N`
		` x(i,j) = y(i,j-1)`
			$exec\_time$

broadcast
unit stride
message size=N
#processors=P
	$broadcast_{unit}(N,P)$	`Dimension x(N,P), y(N)`
		`Align y(i) With x(i,1)`
		`Processors t(P)`
		`Distrib (*,block) x Onto t`
		`do j=1,P`
		`do i=1,N`
		` x(i,j) = y(i)`
			$exec\_time$

		MEMORY ACCESS	

write array ref.
unit stride
	$cache\_hit_{write}$	`do j=1,N`
		`do i=1,N`
		` x(i,j)=1.0`
			$exec\_time/N^2$

read array ref.
unit stride
	$cache\_hit_{read}$	`do j=1,N`
		`do i=1,N`
		` x(i,j)=y(i,j)`
			$(exec\_time/N^2) - cache\_hit_{write}$

write array ref.
nonunit stride
	$cache\_miss_{write}$	`do i=1,N`‡
		`do j=1,N`
		` x(i,j)=1.0`
			$exec\_time/N^2$

read array ref.
nonunit stride
	$cache\_miss_{read}$	`do i=1,N`‡
		`do j=1,N`
		` x(i,j)=y(i,j)`
			$(exec\_time/N^2) - cache\_miss_{write}$

		COMPUTATION	

addition	$cost_+$	`do j=1,N`
		`do i=1,N`
		` x(i,j)=x(i,j)+y(i,j)`
			$(exec\_time/N^2) - (cache\_hit_{read} + cache\_hit_{write})$

multiplication	$cost_*$	`do j=1,N`
		`do i=1,N`
		` x(i,j)=x(i,j)*y(i,j)`
			$(exec\_time/N^2) - (cache\_hit_{read} + cache\_hit_{write})$

† Unless stated otherwise, arrays are declared of size `(N,N)` and stored in column–major order.
‡ The value of `N` is big enough to generate cache misses for each innermost `j` iteration.

Table 1. Example communication, memory access, and computation benchmark kernels

3.3 Machine Model

The machine model consists of a set of HPF programs that measure the execution times of basic computations, memory access, and communication patterns. A single pattern may generate a set of different performance numbers. For example, each global communication pattern is executed for different message sizes and numbers of processors.

Communication Model For each communication pattern, the actual size of the messages and the number of processors involved have to be determined. The current model does not consider contention between groups of processors. If two groups of processors have logically independent communication operations, each operation is predicted separately for the number of processors in each group. Table 1 shows two benchmark kernels, for nearest neighbor *shift* communication and for global *broadcast* communication.

Computation Model Two computation benchmark kernels are listed in Table 1. Based on the timings, the computation costs of basic operations are determined by subtracting out the memory access costs. The measured computation costs may be interpreted differently for different target architectures. For example, if the target architecture has two floating point units, the benchmark kernel may represent the ideal situation where computations are independent. Additional analysis may be needed to model computation that is not independent, i.e., may lead to stalls in the different floating point units. The current prototype of Fortran RED does not contain an instruction scheduling simulator, but adds up the individual costs of of operations.

Memory Hierarchy Model The memory model tries to capture the cost for the address computation and the cost resulting from possible cache misses. This is done by assigning *each* array reference a unit stride array reference cost ($cache\_hit_{read/write}$), and in the case of a cache miss, add an additional cache miss penalty ($cache\_miss_{read/write}$). The cache model to determine the overall cache miss penalty is based on the work by McKinley, Carr and Tseng [27]. Our model is simpler in the sense that it relies on basic pattern matching to identify spatial and temporal reuse. Spatial or temporal reuse across iterations of the innermost loop, or cache conflict misses [18] are not considered in the current model. However, a model to predict the number of capacity misses is used similar to the model introduced by Ferrante, Sarkar, and Thrash [11].

All array references in the loop are partitioned into RefGroups, where each RefGroup contributes a single cache miss penalty to the overall cache miss penalty for an innermost loop iteration. The RefGroup partitioning avoids the overcounting of cache misses since only the first reference in the group will result in a cache miss, followed by cache hits for the remaining references in the group. RefGroups that contain only references that are invariant within the innermost loop, or have only unit stride accesses do not contribute to the cache miss penalty. The remaining groups contain non–unit stride accesses and are classified as either

```
Do i = 2, N      // e.g., parameter (N = 256)
   Do j = 1, N
      c(i,j) = c(i,j) - c(i-1,j) * a(i,j) / b(i-1,j)
      b(i,j) = b(i,j) - a(i,j) * a(i,j) / b(i-1,j)
```

Fig. 4. Example loop with non–unit stride accesses

read or write. A write group contains at least one array reference that occurs on the left–hand–side of an assignment statement.

A simple model for capacity misses determines whether cache misses occur for non-unit stride accesses. Basically, the model determines the number of distinct cache lines touched by an innermost iteration. If this number exceeds the available number of lines in the cache, a cache miss is predicted.

Assuming column-major order allocation of multi-dimensional arrays, the example loop of our ADI kernel in Figure 4 has three RefGroups, namely {`c(i,j),c(i-1,j)`}, {`b(i,j), b(i-1,j)`}, and {`a(i,j)`}. The first two groups are write groups, and the last is a read group. All three groups are non-unit stride. For each group, the number of cache misses is determined based on the cache architecture of the target machine and the sizes of the data segments allocated on each processor. The current Fortran RED implementation uses the following simple model to determine cache capacity misses of a reference group in an innermost loop with n iteration:

$$\#\text{ cache misses} = \begin{cases} n & \text{if } s \geq l \text{ and } n > S \\ \frac{n*s}{l} & \text{if } s < l \text{ and } \frac{n*s}{l} > S \\ 0 & \text{otherwise} \end{cases}$$

where s is the access stride, l is the cache line size, and S is the number of cache sets available for the reference group. In the current implementation, the number of available sets is the total number of sets in the cache divided by the number of arrays referenced in the loop. In the example of Figure 4, a perfect alignment and column distribution of all three arrays onto two processors[3] will result in local innermost iterations of size $n = 128$ and a stride $s = 256$. Assuming a cache with 256 sets and a line size $l = 32$, the available number of sets for each array is set to $S = \lfloor 256/3 \rfloor = 85$. As a result, each reference group contributes one cache misses to each innermost iteration. The final memory access cost, including the cache miss penalty for capacity misses is

$$2 * cache\_hit_{write} + 8 * cache\_hit_{read} +$$
$$2 * (cache\_miss_{write} - cache\_hit_{write}) + cache\_miss_{read} - cache\_hit_{read}$$

Each of the 10 references in the loop is assigned a cache hit cost, with additional penalties for three cache misses, two due to write accesses and one due to read accesses.

[3] `Processor procs(2); Align a With b,c ; Distribute (*,block) a Onto procs`

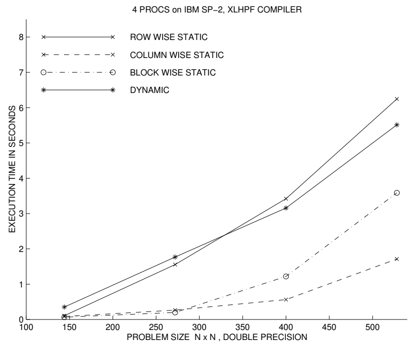
	PGI pghpf/xlf-1.1	IBM xlhpf/xlf-1.3
message vectorization	$\surd$	$\surd$
message coalescing	$\surd$	$\surd$
message aggregation		$\surd$
communication routine calls	$\surd$	
explicit send/receive		$\surd$
software pipeling	$\surd$	$\surd$

Table 2. Characteristics of the HPF target compilers

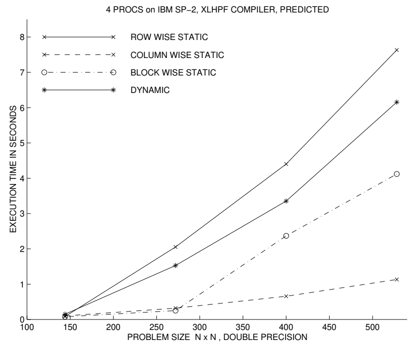
4 Experiments

The alternating direction implicit integration kernel (Adi) as shown in Figure 2 was the basis for our preliminary experiments. There were two target systems for Fortran RED , PGI's HPF compiler with IBM's xlf version 1.1 compiler as its node compiler, and IBM's HPF compilers with IBM's xlf version 1.3 node compiler. In both cases, the IBM SP–2 distributed memory multiprocessor was the target architecture. Experiments were performed for 4, 8, and 16 thin nodes (PowerPC2). Both target compilers were used at the highest compiler optimization level (-O3). The two commercially available HPF compilers (IBM's xlhpf/xlf-1.3 and PGI's pghpf/xlf-1.1) use different communication generation strategies. For a nearest neighbor *shift* communication pattern, PGI's pghpf compiler generates a single call to a global communication routine that implements the pattern. In contrast, IBM's xlhpf compiler generates explicit calls to lower level send and receive communication routines for the pattern. Using a single global communication routine allows runtime communication optimization since the decisions regarding execution order and type of lower level send and receive routines may be postponed until the global routine is actually called. Inserting send and receive communications directly into the program allows compile time optimizations such as message latency hiding by moving sends ahead of receives across program segments. However, runtime optimizations are more restricted since the placement and type of communication routines are determined at compile time. Both HPF compilers use message vectorization and message coalescing, but only IBM's compiler performs message aggregation [39]. Both compilation systems perform software pipeling at the node program level.

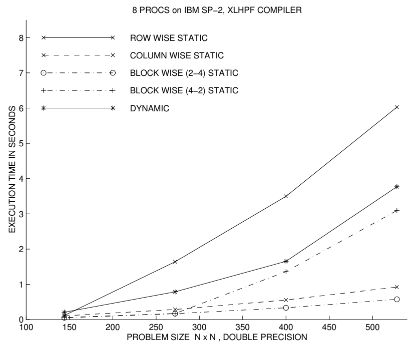
A summary of the HPF compiler characteristics is given in Table 2. These characteristics were provided to Fortran RED as the description of the target compiler. For the experiments, our prototype system simulated message vectorization and coalescing. In addition to the results of the benchmark kernels in the performance data base, Fortran RED exploited knowledge about the structure of the PowerPC2 node architecture such as the cache design of the IBM SP–2. The data cache size is 126Kbytes, with 256bytes for each cache line (512 sets).



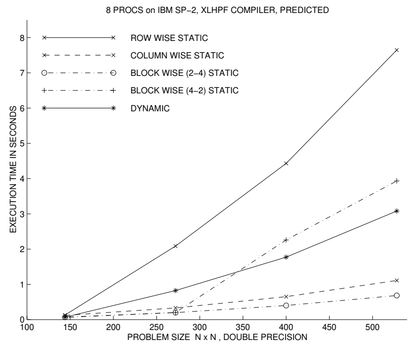
Measured execution times on 4 procs.



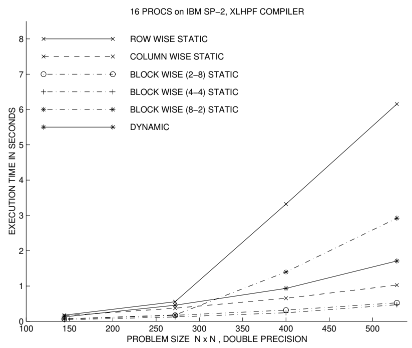
Predicted execution times on 4 procs.



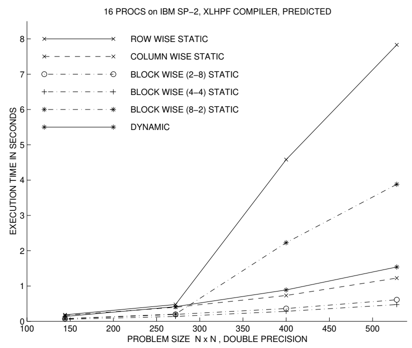
Measured execution times on 8 procs.



Predicted execution times on 8 procs.



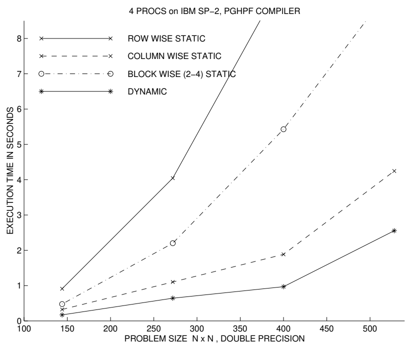
Measured execution times on 16 procs.



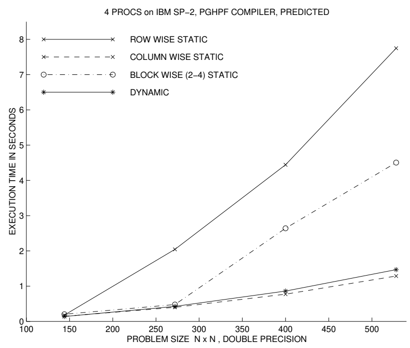
Predicted execution times on 16 procs.

Fig. 5. IBM's xlhpf/xlf-1.3 compiler and 4, 8, and 16 processors of IBM SP-2.

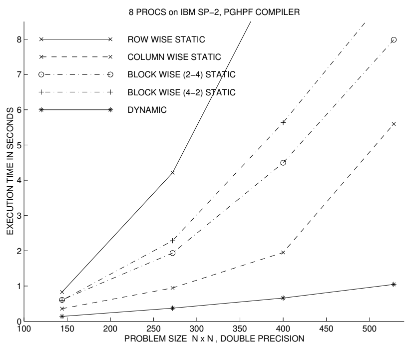
For the Adi code, Fortran RED generated candidate layouts consisting of perfect alignments of the two dimensional arrays (`Align a With b, c`) and all possible combinations of one and two dimensional block distributions for 4, 8, and 16 processors. In order to validate the precision of the performance prediction component of the tool, we hand coded the corresponding static global layouts in HPF and measured their execution times on the target systems. In addition, a



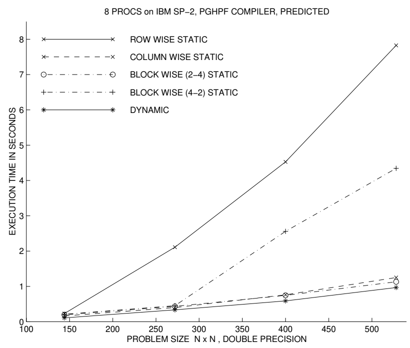
Measured execution times on 4 procs.



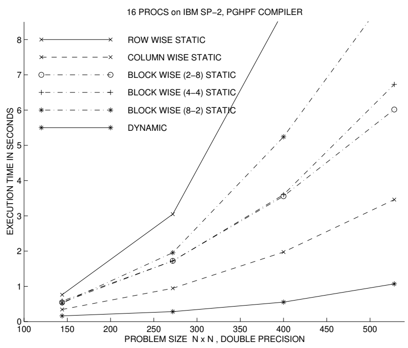
Predicted execution times on 4 procs.



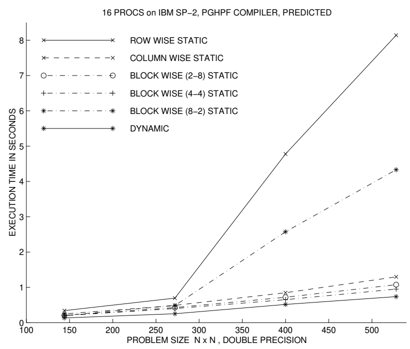
Measured execution times on 8 procs.



Predicted execution times on 8 procs.



Measured execution times on 16 procs.



Predicted execution times on 16 procs.

Fig. 6. PGI's pghpf/xlf-1.1 compiler and 4, 8, and 16 processors of IBM SP-2.

dynamic layout was considered that performed array transposes between the row and column sweeps of the computation. Figures 5 and 6 show the measured and predicted execution times of the different global data layouts for Adi for the two target system on 4, 8 and 16 processors. Each figure shows four data points: problem sizes (144x144), (272x272), (400x400), and (528x528) of double precision.

For the xlhpf/xlf-1.3 target compiler, the best data layout varied with the number of processors and problem sizes. In contrast, PGI's pghpf/xlf-1.1 compiler always performed best using a dynamic data layout. Fortran RED was able to predict the performance of all measured data layouts with high accuracy. This is a remarkable result, considering the complexity of the target compiler and target architecture. In particular, the tool always selected the best data layout. For the pghpf/xlf-1.1 target compiler, Fortran RED failed to predict the best layout in four cases (see Figure 6, 4 processors), which resulted in an execution time increase of up to 89% due to the suboptimal selection. In all other cases, the best data layout was selected. Although the predictions for most data layouts were much lower than the actual measurements, this underestimations was not significant enough to lead to the wrong data layout selection (with the exception of the four cases mentioned above). In addition, the ranking of some suboptimal data layouts were not preserved in the prediction. As it turned out, the 1.1 version of the xlf compiler generated redundant code within software pipelined and unrolled loops[4]. This problem has been fixed in version 1.3. Unfortunately, we were not able to rerun the experiments with the PGI compiler and the updated version of the xlf compiler due to the closure of the Cornell Theory Center. However, it is interesting to note that we were able to find this bug because of our performance prediction results.

5 Related Work

The problem of automatic data layout has been addressed by many researchers [3, 9, 16, 20, 23, 24, 25, 34, 6, 21]. The presented solutions differ significantly in the assumptions that are made about the input language, the possible set of data layouts, the compilation system, and the target machine architecture.

Our work is similar in nature to the recent work done by Anderson and Lam at Stanford University [3, 2], Chatterjee, Gilbert, Schreiber, Sheffler, and Pugh at RIACS, Xerox Parc, and the University of Maryland [37, 8], Palermo and Banerjee at the University of Illinois at Urbana Champaign [31, 30], Ayguadé, Garcia, Girones, Labarta, Torres and Valero at the University of Catalunya in Barcelona, [14, 13], and Ning, Van Dongen, and Gao at CRIM and McGill University [28]. In contrast to previous work, Fortran RED is design to target different compilation systems and parallel architectures, allowing program portability across a range of parallel platforms. A crucial component within Fortran RED is performance prediction. Our work on performance prediction in the context of Fortran RED has been based on our previous work on "training sets" [5]. This work is very similar to the *micro-benchmarking* approach developed by Saavedra and Smith [36, 35]. The importance of performance prediction for optimizing compilers has been recognized by many researchers and several proposals have been published in the literature, such as [40, 33, 10, 29]. However, accurate and cost efficient performance prediction, in particular for superscalar architectures, is still an open research problem.

[4] Source: Henry Zongaro, XL Fortran/XL HPF Compiler Development, IBM Toronto

6 Conclusion

This paper discussed preliminary experiences with Fortran RED , a tool designed to support Fortran on a variety of different parallel target platforms. The tool uses HPF as an intermediate program representation. The portability requires that Fortran RED is able to predict the quality of data layouts across different HPF compilers and target architectures. The performance prediction component within the tool uses benchmarks of communication, computation, and memory access patterns to determine the expected performance of a given data layout. These benchmark patterns are written in HPF to allow portability. Experiments were performed for two different target systems consisting of PGI's and IBM's HPF compilers for the IBM SP–2 multiprocessor, and a single scientific application. For 20 out of 24 test cases, Fortran RED determined the best data layout, where a test case is a particular problem size and number of processors used. For most test cases, the data layout choices were non-trivial, and a wrong data layout choice would result in a dramatic reduction in program performance. For the IBM-xlhpf/SP–2 target system, the performance predictions had very high accuracy. However, in particular for the PGI-pghpf/SP–2 target system, the tool failed to rank subsets of suboptimal data layout alternatives correctly. We believe that this problem resulted from a previously unknown compiler performance bug in the IBM xlf-1.1 compiler and should disappear by using the updated version xlf-1.3.

Clearly, more experimental results are needed to validate our overall approach. The preliminary experiments showed that one of the main challenges in designing a tool such as Fortran RED is accurate performance prediction. A compositional performance prediction approach is needed to model the different components of a target system such as the HPF and node compilers, and the cache. We are currently investigating performance prediction at the node compiler level that takes superscalar optimizations and processor characteristics such as multiple functional units into account. In addition, models to predict cache conflict misses will be included.

References

1. V. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, C-W. Tseng, and S. Warren. Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, 2(3):48–58, 1994.
2. J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors*. PhD thesis, Stanford University, March 1997.
3. J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, 1993.
4. V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, pages 1160–1170, 1990.

5. V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 1991.
6. D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.
7. D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.
8. S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. Sheffler. Array distribution in data-parallel programs. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.
9. S. Chatterjee, J.R. Gilbert, R. Schreiber, and S-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, pages 16–28, 1993.
10. T. Fahringer and H.P. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.
11. J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing – LNCS, Springer Verlag*, 589:328–343, 1991.
12. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Prentice-Hall, Englewood Cliffs, NJ, 1988.
13. J. Garcia. *Automatic Data Distribution for Massively Parallel Processors*. PhD thesis, Universitat Politècnica de Catalunya, Barcelona, April 1997.
14. J. Garcia, E. Ayguadé, and J. Labarta. Dynamic data distribution with control flow analysis. In *Proceedings of Supercomputing '96*, 1996.
15. M. Gerndt. Updating distributed variables in local computations. *Concurrency—Practice & Experience*, 2(3):171–193, September 1990.
16. M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, September 1992.
17. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
18. J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach (2nd edition)*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.
19. High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.
20. D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pages 187–200, 1990.
21. W. Kelly and W. Pugh. Minimizing communication while preserving parallelism. In *Proceedings of the 1996 ACM International Conference on Supercomputing*, pages 52–60, Philadelphia, PA, May 1996.
22. K. Kennedy and U. Kremer. Automatic data layout for distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, July 1998.
23. K. Knobe, J.D. Lukas, and W.J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel*

Computers, Vienna, Austria, July 1992.

24. P. Lee and T-B. Tsai. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages II161–II165, St. Charles, IL, August 1993.
25. J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.
26. J. Mellor-Crummey, V. Adve, and C. Koelbel. The Compiler's Role in Analysis and Tuning of Data-Parallel Programs. In *Proceedings of The Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 211–220, 1994.
27. K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
28. Q. Ning, V. V. Dongen, and G. R. Gao. Automatic data and computation decomposition for distributed memory machines. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, January 1995.
29. D.B. Noonburg and J.P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, CA, December 1994.
30. D. Palermo. *Compiler Techniques for Optimizing Communication and Data Distribution for Distributed–Memory Multicomputers.* PhD thesis, University of Illinois at Urbana–Champaign, June 1996. Available as CRHC-96-09.
31. D. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. Technical Report CRHC-95-09, University of Illinois at Urbana-Champaign, April 1995.
32. D. Palermo, E. Su, J. A. Chandy, and P. Banerjee. Communication optimizations used in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the 1994 International Conference on Parallel Processing*, 1994.
33. M. Parashar, S. Hariri, H. Haupt, and G. Fox. Interpreting the performance of HPF/Fortran90D. In *Proceedings of Supercomputing '94*, 1994.
34. J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, pages 637–646, Reno, NV, November 1989.
35. R.H. Saavedra and A.J. Smith. Performance characterization of optimizing compilers. *IEEE Transactions on Software Engineering*, 21(7):615–628, July 1995.
36. R.H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking.* PhD thesis, U.C. Berkeley, February 1992. UCB/CSD-92-684.
37. T. Sheffler, R. Schreiber, W. Pugh, J.R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. *International Journal of Parallel Programming*, 24(6):599–620, December 1996.
38. Silicon Graphics Inc. *F77 User's Manual: Chapter 6 – Parallel Programming on Origin2000*, 1997.
39. C-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993. Rice COMP TR93-199.
40. K-Y. Wang. Precise compile-time performance prediction for superscalar-based computers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.
41. D. Wood and M. Hill. Cost–effective parallel computing. *IEEE Computer*, 1995.