A Static Approach for Compiling Communications in Parallel Scientific Programs

DAMIEN GAUTIER DE LAHAUT AND CÉCILE GERMAIN

LRI CNRS-Université Paris-Sud, LRI Bât 490, 91405 ORSAY CEDEX, Paris, France; e-mail: {gautier, cecile}@lri.fr

ABSTRACT

On most massively parallel architectures, the actual communication performance remains much less than the hardware capabilities. The main reason for this difference lies in the dynamic routing, because the software mechanisms for managing the routing represent a large overhead. This article presents experimental studies on benchmark programs concerning scientific computing; the results show that most communication patterns in application programs are predictable at compile-time. An execution model is proposed that utilizes this knowledge such that predictable communications are directly compiled and dynamic communications are emulated by scheduling an appropriate set of compiled communications. The performance of the model is evaluated, showing that performance is better in static cases and gracefully degrades with the growing complexity and dynamic aspect of the communication patterns.

1 INTRODUCTION

Parallel architectures suffer from a recurrent problem, which is the large gap between peak and actual performance. Despite the progress in hardware and software, most recent experimental studies [1, 6, 24] show that the actual performance usually remains below the peak. One major cause of this sobering fact is the data transfer and especially the interconnection network. For instance, recent studies [6, 11] show that the best performance figures are achieved by programs that have the lowest remote data access to floating-point operations ratio.

Although communication seems to be the bottleneck for parallel architectures, not much is known about the characteristics of the communications used by parallel programs. The first objective of this article is to give some experimental results about the statistical distribution of the communication patterns. The communications that are known at compile-time will be called *static* and those that can only be determined at run-time will be called *dynamic*. To obtain satisfactory statistics, a significant benchmark set has been studied; this set amounts to around 25,000 lines of code written in various dialects of parallel Fortran. The set is composed of two parts: The first is a set of scientific parallel codes, partially handwritten and partially generated by automatic parallelization; the second is a subset of library

Received September 1994
Revised February 1995

Scientific Programming, Vol. 4, pp. 291–305 (1995)
CCC 1058-9244/95/040291-15

routines from LAPACK. The dynamic (run-time) occurrences of both static and dynamic communication schemes have been gathered. The main result is that static communications are nearly exclusive in parallelized codes and dominant in user programs, whereas the situation is much more complex in library routines.

We are interested in this taxonomy (static/dynamic) not for classification purposes but because a considerable speedup in parallel computations can be achieved by a careful exploitation of the compile-time information about static communications. In fact, a parallel execution model where the communications are computed at compile-time can achieve the hardware's raw performance for the most frequently used static communication schemes. This contrasts with the actual communication performance of most parallel architectures, which is dominated by the communication protocol overhead. However, the overall speedup must take into account the contributions of all communication types, both static and dynamic (Amdahl's law). The task is then to assess the penalty of compiling the dynamic communications. This is very difficult, because many factors are involved, and it is almost impossible to quantify their respective impacts and interactions. Nevertheless, meaningful results can be derived by evaluating, for broad classes of communication schemes, the speedup achieved on each class by the static execution model. As a testbed, we compare the CM-5 communication figures with the expected performance of the static model. The speedup is significant, even in the dynamic case.

The rest of this article is organized as follows. The first section discusses dynamic routing, the basic communication mechanism of almost all parallel architectures, and the background of compiled communications. The second section presents a classification of communication schemes. The third section is devoted to the experiments, methodology, and results. Finally, we assess the cost of emulating dynamic communications in the static model and present the expected performance.

2 BACKGROUND

2.1 Dynamic Routing

Almost all massively parallel architectures use asynchronous dynamic routing, which means that the routing circuits in each network node determine the path of each message at run-time. This requires extra hardware (the routing circuits) and network bandwidth (the address header carried by each message). The routing is asynchronous in the sense that the latency of the messages depends on the network load, thus is unknown; a processor/network interface is necessary to synchronize the message and the computing threads. The overhead of this interface is large: For instance, it costs more than 90% of the latency of the Paragon machine [13], and it is from 3 to 90 μs for the CM-5 [20, 23].

One could expect that, for large data transfers, this overhead would ultimately vanish. In fact, a significant part of the effort in practical parallel programming is careful data organization in order to pack the data such that the transfers are of the appropriate size; a lot of research is devoted to sophisticated compilation techniques, such as message vectorization, with the same goal [28]. However, the startup penalty is so high that effective use of the network is extremely difficult. For instance, to use half of the peak bandwidth of the network, the message size must be more than 1 kilobyte for the CM-5; to reach full use of the bandwidth, the message size must be more than 8 kilobyte [6].

Moreover, parallel scientific programs are highly synchronous, because communications come from parallel array statements: in general, consecutive communications must proceed only in lockstep fashion. Thus, the major opportunity to enlarge the message size comes from *virtualization*. In a data-parallel language, the parallelism is not limited: For instance, the FORALL instruction has the semantics of evaluating first the righthand side of an assignment, then performing the assignment. However, the available parallelism on a particular computer is clearly limited by the number of processors. To take into account the limitation of the actual parallel computer, the unlimited parallelism of the source code is folded on the limited parallel computer by automatic or user-defined distributions such as cyclic, block, or block–cyclic. This is virtualization. For instance, consider the parallel assignment *Forall* $(i = 0{:}\ 14)\ a(i) = b(i + 1)$ on a four-processor machine. Each processor has to iterate sequentially over its own piece of arrays a and b to exchange data and compute. In particular, each processor sends to another one from three to four array elements; sending one piece of data by message is highly inefficient; aggregating data to be sent to one processor in one message is known as message vectorization [16]. However, message vectorization is limited by the virtualization ratio (roughly speaking, the ratio between the size of a

FORALL index set and the machine size). A high startup penalty limits the efficiency of massively parallel architectures on huge problems. This overhead can be greatly reduced if analyzing the communications at compile-time provides some knowledge of the communication behavior at run-time. The hardware design and software tools that provide efficient means to use this knowledge have been developed in the PTAH project. They are beyond the scope of this article: the architecture is described in [4] and the principles of the compiler in [10].

The results presented in this article indicate that, at least in scientific programs, a large part of the communications can be determined from analysis of source code. Moreover, almost all other programs provide information that can be used to limit the communications overhead. In fact, the idea that a lot of communication patterns in scientific programs can be determined at compile-time is the cornerstone of vectorizers and automatic parallelizers. In the following sections, we consider a number of parallel programs, and quantify this idea.

2.2 Compiled Communications

In the static execution model, all the parameters of the communications are computed at compile-time. This model has been exemplified in the IBM GF11 [17], in the iWarp ConSet [25], and by the Communication Compiler of TMC CM-2 [7]. The model assumes an off-line routed network. Off-line means that the message paths are computed in the back-end compiler, by a "communication generator" that is an equivalent for communication of the code generator for computation. All the physical parameters of a communication are then computed at compile-time. At run-time, the switch settings are simply scheduled under program control. This is opposite to the on-line routing model, where the message paths are determined at run-time, the network routing circuits acting on the addresses as an interpreter. The compilation problem is to embed the communication graph into the physical network.

Off-line routing improves the network throughput, by removing the overhead of address headers encapsulated within each message. As no more routing decisions have to be made, the latency can ultimately be reduced to the hardware propagation delay. Finally, shifting the routing task from run-time to compile-time allows more complex routing algorithms, resulting in better resource (links and buffers) utilization. Theoretical studies [15, 21, 22] show that, for some interconnection networks, off-line routing is feasible in the sense that the off-line routing algorithm has acceptable complexity, and may be asymptotically optimal [19]. The practical experiments on the CM-2 [7] show that a one order of magnitude speedup can be achieved by off-line routing on the hypercube, without any additional hardware: the simulated annealing algorithm provides global optimization of the link allocation.

Off-line routing supposes that the communication generator may be fed with the communication graph, which has been constructed by the compiler. This issue is beyond the scope of this article; however, recent research in the message-passing framework [14, 28], and in the static framework [10] provides techniques to tackle this issue. Moreover, these techniques remove the potential drawback of the first experiments on the CM-2, which was the long compilation time: As a formal description of the graph can be exhibited, the complexity of the off-line routing process can be simplified in many cases.

3 COMMUNICATION PATTERNS

As our benchmarks are written in data parallel Fortran (CM Fortran, Fortran 90, high-performance Fortran [HPF]), the following discussion uses an HPF syntax. However, this only exemplifies the main data-parallel communication feature: The communications are implicit, derived from operations on parallel data structures (arrays in Fortran). In HPF, parallel data operations come from, either FORALL loops or array notations, or Intrinsics that summarize multiple parallel data operations. As each of these structures involves parallel array references, our taxonomy begins with a classification of parallel references.

3.1 Parallel References

A typical parallel construct is a nest of FORALL loops as illustrated next:

```
Forall (i1 = a1 : b1 : c1)
   Forall (i2 = a2 : b2 : c2)
      ...
      Forall (in = an : bn : cn)
         A[e1, e2, ..., en] = F(B[f1, f2,
         ..., fn], ...)
      endforall
   endforall
endforall
```

where a_k and b_k may depend on i_l for $l < k$. For short, it can be summarized in the following pseudosyntax:

```
Forall I in 𝒯
   A[f(I)] = F(B[g(I)], ...)
endforall
```

where I is the vector of parallel indices $(i_1, i_2, \ldots, i_n)$. $\mathcal{T}$ is the convex polyhedron (see example below) defining the loop bounds, A and B are two arrays, and finally, $A[f(I)]$ and $B[g(I)]$ are two parallel references.

A typical parallel reference is a reference to an m-dimensional array A, in a nest of n FORALL loops: $A[e_1, e_2, \ldots, e_m]$, where e_i are functions of the FORALL subscripts (another syntax is the parallel array reference $A[a_1 : b_1 : c_1, \ldots, a_m : b_m : c_m]$, which can be expressed with a FORALL syntax). Analytical analysis can be performed at compile-time only if the e_i are affine in the FORALL subscripts, with integer coefficients, i.e.,

$$e_i = \sum_{j=1}^{n} a_{ij} I_j + b_i.$$

An affine reference can be written $A[MI + U]$, where M is a $m \times n$ integer matrix and U a vector in Z^m. We give an example from Jacobi's method for the Laplace solver:

```
Forall (i=2:9, j=2:9)
   A(i,j)=(A(i-1,j)+A(i+1,j)+A(i,j-1)
                            +A(i,j+1))*0.25
endforall
```

Here, there are five parallel references to A (1 store and 4 fetches): the first one $(A(i - 1, j))$ may be expressed with:

$$M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \; U = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \text{ and}$$

$$\mathcal{T} = \left\{ |(i,j)| \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \leq \begin{pmatrix} 2 \\ 9 \\ 2 \\ 9 \end{pmatrix} \right\}.$$

Affine references where M and U only include numerical constants are called static and nonstatic affine references are called parametric. For example the parallel reference $A(i - 1, j)$ is static, whereas a reference such as $A(i + k, j)$ will be parametric if k is a variable which is not a FORALL index, as in the following assignment:

```
do k ...
   Forall i ...
        ...=A(i + k)
```

This scheme is dominant in LAPACK routines.

In fact, a finer classification would be possible: If the vector U is a scalar variable, the reference can occasionally be determined at compile-time; for instance, if U linearly depends on sequential loop subscripts, as in the previous example. However, using this information in the static execution model would require the unrolling of the sequential loop to compute the communication patterns. As the sequential index set is almost always too large to allow this optimization, there is no point in using a finer classification.

In our benchmarks, nonlinear references were represented by gather and scatter operations, where the array subscripts are themselves array elements; the generic form being $A[L[I]]$.

3.2 FORALL Communications

In the typical parallel instruction

```
Forall I in 𝒯
   A[f(I)] = B[g(I)] + ...
endforall
```

the assignment creates communication patterns where, for each I, the source is the processor owning the reference $B[g(I)]$, and the destination is the processor owning the reference $A[f(I)]$. The patterns depend on the computation location rule and on the mapping. We consider the Owner Computes Rule, which is used by most existing parallel compilers and assumed by many researchers in this field; it means that the computing processor is the destination processor. The mapping between arrays is created by the ALIGN directives. If an array is compressed along one dimension, the corresponding FORALL subscript must not be considered for classification because it is not a parallel dimension. For instance, if A is of dimension 2 and compressed along its second dimension, then $A(i, j)$ is located on the same processor as $A(i, 0)$. With these assumptions, a communication occurs for each array in the righthand side of the parallel assignment, if combining the mapping and the Owner Computes Rule does not result in an intraprocessor assignment. The communication is labeled by the worse case of the two

references, e.g., left and right member affine static will result in a static communication, but a one-member nonaffine will result in a nonaffine communication and so on.

A typical use of the FORALL notation is to describe partial permutations of the index set. Although the FORALL syntax does not preclude more complex schemes, efficient programming would encapsulate such patterns in intrinsics to take advantage of the global communication features of the target architecture.

3.3 Intrinsic Communications

In data-parallel Fortran languages, complex data transfers can be described by special functions that are part of intrinsics. The most important communication intrinsics implement multireduction (multiple many-to-one communication), multibroadcast (multiple one-to-many), special permutations, and gather/scatter operations.

The reduction intrinsics are SUM, ALL, ANY, MAXVAL, MINVAL, MAXLOC, MINLOC. They compute the result of applying an associative operator to all the elements of their array argument. The respective operators are sum, logical and, logical or, max, min; MAXLOC (resp. MINLOC) returns the location of the maximal (resp. minimal) value. The reduction intrinsics have three parameters; for instance, SUM(ARRAY, DIM, MASK) adds the elements of ARRAY along the dimension DIM, selecting the elements described by MASK. We considered that a reduction intrinsic is static as soon as the ARRAY parameter is a static reference and the DIM parameter is a constant; The unit element of the operator (e.g., 0 or 0.0 for a SUM, or IEEE $-\infty$ for a floating-point MINVAL) can replace the masked references, and this local test can be done at run-time.

The intrinsic SPREAD allows broadcasts and segmented broadcasts: An n-dimensional array is replicated to create an $n + 1$ dimensional one. The syntax is SPREAD(SOURCE, DIM, NCOPIES); to compute the communication scheme at compile-time, the SOURCE parameter must be a static reference and DIM must be a constant (in this case, the pattern is considered as static). In the following, we call broadcast a one-to-many pattern, multibroadcast a segmented broadcast, reduction a reduction that results in a scalar, and multireduction a segmented reduction.

Examples of special permutations intrinsics are the cyclic and noncyclic SHIFTS and TRANSPOSE. All these intrinsics summarize a FORALL permutation and require the same analysis. More complex intrinsics, such as MATMUL and DOTPRODUCT, are intended to allow an optimal implementation of basic linear algebra operators. These intrinsics will be considered as static if their parameters are static or scalar constants.

4 EXPERIMENTAL RESULTS

4.1 The Benchmark Set

Three benchmark sets have been analyzed (Table 1). The first, called NPAC in the following, is the applications benchmark set for Fortran D and HPF of the Northeast Parallel Architecture Center at Syracuse University [25]. It includes complete applications and mathematical packages for dense linear algebra. Some applications have two different versions: the Cluster Spin and Revised Simplex have been redesigned for parallelism, whereas the Conventional Spin and Simplex are the straightforward parallel versions of the well-known sequential benchmarks. The second set, called PRE, is composed of outputs of the automatic parallelizer VAST 90 of Pacific Sierra Research with some handcoded parts. PRE has been assembled by J. K. Prentice from Quetzal Computational Associates [26]. The third is a benchmark from Institut Français du Pétrole (IFP). We have rewritten it as an HPF version and validated by IFP. The classification of the benchmarks in three categories (kernel, application, and algorithms) follows the approach used in [2].

Apart from the limitations of any benchmark set compared with real applications, this benchmark set may be considered as representative of dense computations. No sparse code is included for the following reason: Although the present state of the art in algorithms for sparse computations indeed favors dynamic data structures and communications, the situation is quickly evolving. Recent work [3] focuses on the dynamic to static transformation; hence statistics in this field may not be significant at the present time.

4.2 Methodology

The tool used for analysis is a parser built from the Tiny tool set [29]; it consists of an intraprocedural constant propagation package and a program for automatic reference analysis based on the abstract syntactic representation that we developed. The output of these tools is a characterization of

Table 1. The Analyzed Benchmarks

Benchmark Set	Name	Size in lines	Automatic Parallelization	Language	Category
	PHYSICS Conventional Spin	933	No	CM Fortran	Application
	PHYSICS Cluster Spin	456	No	CM Fortran	Application
	Weather climate	1783	No	CM Fortran	Application
	LAPACK Block-QR	1380	No	CM Fortran	Algorithm
	LAPACK Block-Cholesky	516	No	CM Fortran	Algorithm
NPAC	LAPACK Block-LU	2529	No	CM Fortran	Algorithm
	2D-FFT	201	No	CM Fortran	Algorithm
	Laplace Solver	267	No	CM Fortran	Application
	Gaussian Elimination	90	No	CM Fortran	Algorithm
	Nbody	149	No	CM Fortran	Application
	Simplex	623	No	CM Fortran	Application
	Revised Simplex	556	No	CM Fortran	Application
	Livermore Fortran Kernel	6124	Yes	Fortran 90	Kernel
PRE	Gas Dynamics	2307	Yes	Fortran 90	Application
	Kepler	276	No	Fortran 90	Application
IFP	IFP	547	No	HPF	Application

each reference and intrinsic in the source code, following the classification of Section 2. Next we evaluated the dynamic (run-time) frequencies of each communication type by manual examination of the code.

4.3 Results

Tables 2 to 5 present the statistics. Tables 2 and 4 give the formal expression as a function of the parameters, respectively, for static and dynamic communication patterns; Tables 3 and 5 give the numerical percentage of the total communication patterns. The first column is the benchmark name. The column labeled "Loop Parameters" in Tables 2 and 4 is the name of the program parameters that are used as sequential loops subscripts. For instance, Cluster Spin shows three nested sequential loops: the indices are M, the number of measures, and I and J, which are internal to the algorithm. The numbers in parentheses are the parameter values used for Tables 3 and 5, if necessary; most of them were indicated by the benchmark. The following columns give the total number of occurrences of each communication scheme, for a complete execution of the benchmark: the column labeled "Affine and Cyclic" describes affine communications (all these com-

Table 2. Formal Expression of Static Communications

Benchmark	Loop Parameters	Affine and Cyclic	Broadcast	Reduction	Special
Cluster Spin	M (100) I (10), J (200)	M(2 + 3I + 2IJ)		M(1 + 2I + IJ)	
Conventional Spin	M (100), I (10)	2M(4I + 1)		2M	
Weather Climate	I (5)	626I + 300	800I + 200	403I + 100	9I
LAPACK block-QR	N (1000), NB (64)	4N/NB + N	2N		2N/NB + N
LAPACK block-Cholesky	N (1000), NB (64)				
LAPACK block-LU	N (1000), NB (64)				
2D-FFT	N (512)				
Laplace Solver	I (1000)	4I		I	
Gaussian Elimination	N (255)	2N			
Nbody	I (1000)	15I + 16			
Simplex	I (1000)		I	I + 2	
Revised Simplex	I (1000)	2I	2I	5I + 1	
Livermore Fortran Kernel	I (21)	24I		9I	2I
Gas Dynamics	I (10000)	16I		5I	
Kepler	T (365000)				6T
IFP	N (4000)	59N		2	

Table 3. Static Communications as a Percentage of the Total Communications

Benchmark	Affine and Cyclic	Broadcast	Reduction	Special	Total static
Cluster Spin	33.4	0.0	16.8	0.0	50.2
Conventional Spin	97.6	0.0	2.4	0.0	100.0
Weather Climate	24.2	29.6	14.9	0.3	69.0
LAPACK block-QR	0.0	17.8	0.0	18.7	36.5
LAPACK block-Cholesky	0.0	0.0	0.0	0.0	0.0
LAPACK block-LU	0.0	0.0	0.0	0.0	0.0
2D-FFT	0.0	0.0	0.0	0.0	0.0
Laplace Solver	80.0	0.0	20.0	0.0	100.0
Gaussian Elimination	50.0	0.0	0.0	0.0	50.0
Nbody	100.0	0.0	0.0	0.0	100.0
Simplex	0.0	11.1	11.2	0.0	22.3
Revised Simplex	12.5	12.5	31.3	0.0	56.3
Livermore Fortran Kernel	52.2	0.0	19.6	4.3	76.1
Gas Dynamics	64.0	20.0	0.0	0.0	84.0
Kepler	0.0	0.0	0.0	100.0	100.0
IFP	100.0	0.0	0.0	0.0	100.0

munications are translations, apart of LAPACK block-QR where the scheme is a matrix transpose); the "Broadcast" and "Reduction" columns are, in general, multibroadcasts and multireductions; the column "Special" gathers all the instances of the intrinsics MATMUL and DOTPRODUCT and, for the Weather Climate benchmark, calls to the fast Fourier transfom (FFT) library routine. The column "Total" in Tables 3 and 5 is the partial total of each broad class, static and dynamic.

Most of the application benchmarks have a high percentage of static communications, the exceptions being Cluster Spin and Simplex. However, these benchmarks are particular implementations of an application and have another version (Conventional Spin and Revised Simplex), which is much better for the static model. The IFP benchmark is especially interesting: From the sequential version, it was possible and even easy to write a fully static HPF version of the benchmark, without any change in the initial algorithm.

The category Algorithms presents much more diverse results: 50% static communications for the No-Block Gaussian Elimination, but 0% for LAPACK block-LU. The reason is that in the LAPACK subset, the applications are matrix decomposition, but the implementations are block algo-

Table 4. Formal Expression of Dynamic Communications

		Parametric				
Benchmark	Loop Parameters	Affine and Cyclic	Broadcast	Reduction	Special	Gather-Scatter
Cluster Spin	M (100) I (10), J (200)					3MIJ
Conventional Spin	M (100) I (10)					
Weather Climate	I (5)	2200	2200			
LAPACK block-QR	N (1000) NB (64)	8N/NB + 4N			3N	
LAPACK block-Cholesky	id	N/NB	4N		2N/NB	
LAPACK block-LU	id	4N + 2N/NB	N^2/NB	N^2/NB	2N + N/NB	
2D-FFT	N (512)	$\log_2 N + 1$				
Laplace Solver	I (1000)					
Gaussian Elimination	N (255)	N		N		1
Nbody	I (1000)					
Simplex	I (1000)	3I	I	2I		I + 1
Revised Simplex	I (1000)	3I	2I	I		I + 1
Livermore Fortran Kernel	I (21)	5I	2I	2I		2I
Gas Dynamics	I (10000)					4I
Kepler	T (365000)					
IFP	N (4000)					

Table 5. Dynamic Communications as a Percentage of the Total Communications

Benchmark	Parametric				Gather Scatter	Total
	Affine and Cyclic	Broadcast	Reduction	Special		
Cluster Spin	0.0	0.0	0.0	0.0	49.8	49.8
Conventional Spin	0.0	0.0	0.0	0.0	0.0	0.0
Weather Climate	15.5	15.5	0.0	0.0	0.0	31.0
LAPACK block-QR	36.8	0.0	0.0	26.7	0.0	63.5
LAPACK block-Cholesky	0.4	98.8	0.0	0.8	0.0	100.0
LAPACK block-LU	10.8	41.9	41.9	5.4	0.0	100.0
2D-FFT	100.0	0.0	0.0	0.0	0.0	100.0
Laplace Solver	0.0	0.0	0.0	0.0	0.0	0.0
Gaussian Elimination	25.0	0.0	25.0	0.0	0.1	50.1
Nbody	0.0	0.0	0.0	0.0	0.0	0.0
Simplex	33.3	11.1	22.2	0.0	11.1	77.7
Revised Simplex	18.8	12.5	6.2	0.0	6.2	43.7
Livermore Fortran Kernel	10.9	4.3	4.3	0.0	4.3	23.8
Gas Dynamics	0.0	0.0	0.0	0.0	16.0	16.0
Kepler	0.0	0.0	0.0	0.0	0.0	0.0
IFP	0.0	0.0	0.0	0.0	0.0	0.0

rithms. As stated in [25], the target architectures were multiple instruction multiple data (MIMD) shared memory, and blocking increases performance in this case by reducing memory traffic. The No-Block version of the LU decomposition (the routine SGETF2) is fully parametric, but with a much lower communication count: 2N parametric MATMUL and N parametric translations. However, the applications are inherently dynamic, because they are sequential in either the rows or the columns of the basic matrix. A typical communication is

```
... =
  MATMUL (A(J:N, 1:J-1), A(1:J-1, J)),
```

where J is a sequential index. As J ranges over the matrix linear size, no loop unrolling may be considered. On the other hand, although the 2D FFT seems fully parametric, this is mostly an implementation artefact: The communication patterns of a FFT are the folding onto the processor set of the well-known butterflies, and are known at compile-time, at least if the array argument of the FFT is static.

5 PERFORMANCE EVALUATIONS

The previous results indicate that the static communications are frequent enough to deserve specific optimizations, such as the static execution model. However, Amdahl's law requires a comparison with the speedup expected from these optimizations, and the penalty when executing dynamic communications. This evaluation needs to take into account details of the hardware and software underlying the static execution model. The basic assumptions are the following:

1. The overall architecture is distributed memory MIMD, with P processors.
2. The network is strictly synchronous and controlled in a lockstep fashion. In some sense, this is the single-program multiple data (SPMD) execution model, but as an assumption at the hardware level.
3. For each communication, the data incoming from each processor has fixed size.
4. The routing is off-line, which means that the routing switches do not process at all. They only orientate the messages according to a configuration given by the processors before sending the whole data set. The configuration of the switches for one data set is called a communication pattern. All the useful patterns (that the network can use in a run) are compiled.
5. The network can realize any permutation in constant time. This time is the basic unit of the network operations, and is called an elementary step in the following.

Among general-purpose commercial parallel machines, none has an interconnection network with these properties. However, such a network has

been successfully built for the GF11, a research prototype of IBM. The iWarp network may be used in this manner, although the fact that it is primarily intended for message passing raises the cost in time of its static use; many research studies, especially in the field of optical interconnection networks, consider off-line routed networks [27]. For an in-depth presentation of such networks, see [5, 9, 17].

We must stress that, as the network cannot do any on-line routing, dynamic patterns have to be emulated by a sequence of static (i.e., compile-time computed) patterns. The size of such a sequence is the emulation cost of dynamic communications.

In the following, we assume that the shape of the processor set matches exactly the shape of the arrays, and that each processor owns only one datum, which has the prescribed size. The issues of generating code for cyclically or block-cyclically distributed arrays have been successfully treated in the PTAH compiler and are not described here. The impact of virtualization on performance will be outlined in a later section.

5.1 Permutations

We first consider the simplest parameter permutations (shifts, cyclic shifts, transpositions) and study the case of gather/scatter operations later.

Parametric Shifts

A one-dimensional parametric shift may be defined by three parameters: the domain bounds and the value of the shift. The following example shows a parametric shift where the domain is limited by s and f and the shift value is k.

```
Forall (i = s:f) A(i) = B(i+k)
```

To cope with the domain parameters, the communication pattern is extended to all processors (using a temporary array) and the final store is conditioned by the membership to the domain. Without virtualization, the previous code becomes:

```
Forall (i = 0:P-1) Temp(i)=B(i+k)
Forall (i = 0:P-1)
   Where(s <= i and i <= f)
         A(i)=Temp(i)
   endwhere
endforall
```

Now, parametric shifts depend only on one parameter, the value of the shift. It is possible to define all the communication patterns corresponding to all the shifts inside the processor set, and to use k (or k mod P in the case of virtualization) to select at run-time the appropriate communication pattern. However, each pattern has a significant storage cost; for instance $O(P \log P)$ bits for a Benes network, leading to $O(P^2 \log P)$ for the P possible shifts (log means $\log_2$). A reasonable solution is to use only power of two shifts, and to emulate the k-shift by the following procedure:

```
PARAMETRIC_SHIFT(V, a, s, f)
   do i=1:P
      if ((a.AND. i) = 1)
         SHIFT(V, i, s, f)
      i = i*2
   endo
```

where V is the array to be shifted, P the number of processors, s and f the limits of the domain of V, a is the value of the shift, and *AND* is bitwise. In this case, the actual value of a will be k, or k mod P if virtualization occurs. Thus, the emulation cost, which is the number of patterns to be scheduled, is $\log P$.

For multidimensional shifts like $A(i, j) = B(i + k_1, j + k_2)$ where A and B are matrices, the same method holds, except that we have to define the input parameter a as a vector. Assuming that the n-dimensional processor geometry (two-dimensional in this example) is linearly mapped to a numbering of the processor set, in row (or column) major order, the (a_1, a_2) vector shift ultimately produces a shift with value $pa_1 + a_2$, where p is the extent of the processor geometry in the first dimension.

Parametric cyclic shifts are split into two shifts, the modulo part and the nonmodulo part. A priori, 2 log P steps are needed but as we can interleave the two patterns, the number is only log P steps.

Parametric Transpositions

The general form is

```
Forall (i=s1:f1, j=s2:f2)
     A(i,j)=B(j,i)
endforall
```

The only parameter required is the domain of the transposition. One solution is first to do a parametric shift of B so that $B(s1, s2)$ goes to

(0, 0). This can be done in log P steps. The result of this first shift is stored in a temporary array. Then the transposition of the temporary array takes only one step. Finally, the result is stored in A with a parametric shift. The whole operation takes $2logP + 1$ steps.

Gather and Scatter Operations

These are the most difficult communications for the static paradigm. The data referenced are in an array dynamically computed. The scatter operation sorts an array B according to indices L:

```
Forall i...
      A(L(i))=B(i)
```

And the gather operation is:

```
Forall i...
      A(i)=B(L(i))
```

A parallel gather operation makes sense only if the mapping of the index set onto itself is a one-to-one operation. Let array K be defined by $K(L(i)) = i$; the gather operation may be written as the scatter operation: $A(K(i)) = B(i)$. Building K at run-time requires one gather operation. From this, a gather operation is amenable to two scatter operations.

Usually the gather operation is used to pack an array into a smaller one, whereas the scatter operation expands an array. We assume first that the arrays have the same size and that there is no conflict while reading or storing elements. We study later array size differences and conflicts.

To emulate dynamic routing, the key idea [18] is to sort the destination addresses of the data to be routed. The sorting algorithm uses the principle of the odd–even merge sorting network. Figure 1 shows this principle where the list L is to be sorted; if the message follows the number of the receiver, the network realizes the scatter operation communication $A(L(i)) = B(i)$. At each stage of the sorting network, crossing links symbolize comparison of two values and perhaps their exchange.

As the switches do not have any logic, the network cannot perform the comparisons. We simulate each stage of the odd–even network by a crossing of our network and a comparison inside the processors. As the links between the stages are static, it is possible to compile each corresponding permutation. The number of patterns to schedule is log $P(\log P + 1)/2$, i.e., $O(\log^2 P)$.

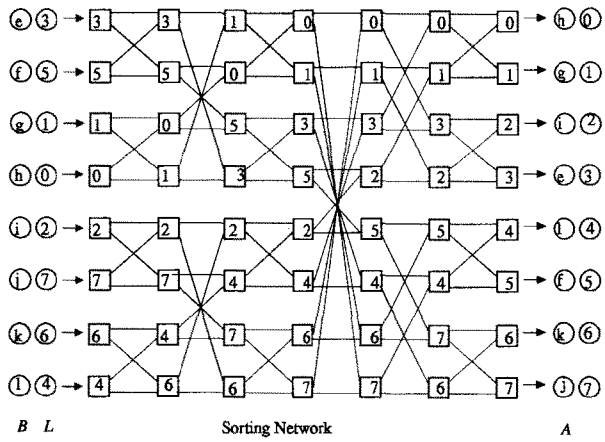

FIGURE 1 Using an odd–even merge sorting network to realize a scatter operation communication.

Consider the case where A is larger than B. In the example, let L be equal to 3, 5, 1, 0, 4, 7, 6 and assume that the network sorts the values into the sorted list 0, 1, 3, 4, 5, 6, 7, but the values are not all located at their destinations. However, sending them to their destination is a monotone routing problem. Monotone means that the source-to-destination map is a monotone function. We can realize monotone routing using the greedy routing algorithm on the butterfly network. Monotone routing of a sorted list on hypercubic networks is conflict free [18]. Figure 2 presents the example of monotone routing in the butterfly network. On stage k of the butterfly, the network transmits the data according to bit k of the destination address.

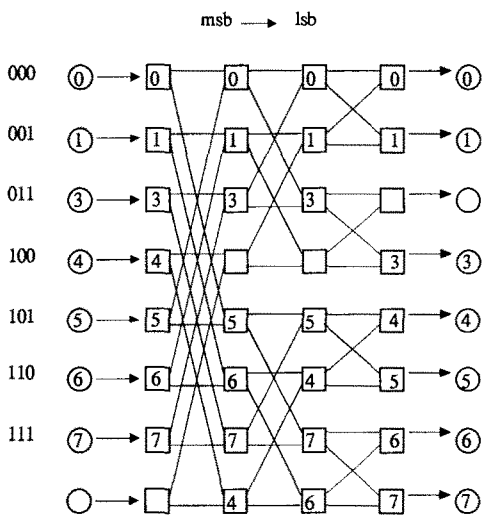

FIGURE 2 Monotone routing on a Butterfly network.

Each stage of the butterfly is emulated by one permutation in our network and by the test of bit k (for stage k) by the processors. The number of permutations scheduled is $O(\log P)$. As monotone routing is conflict free, the routing process remains very simple for the processing elements (no buffering or priority managing).

Storing conflicts are prohibited for a scatter operation, but reading conflicts are possible for a gather operation. In this case, the communications must be partially sequentialized. First, the odd–even sorting network sorts the destinations that can be realized without conflict. The sorted list shows repetitions at contiguous stages. These repetitions lead to conflicts while executing the monotone routing. If two identical references are located on the same processor, it stores one of them in a temporary buffer and carries on with the routing, then a second stage is started for the buffered messages. After that, a second scatter operation takes place. This procedure is expensive; however, the most complex case is where a multicast is hidden in the gather operation, and thus will also be expensive with any routing mechanism.

5.2 Broadcasts

Broadcasts and multibroadcasts have two possible origins: one-to-many gather operations and the SPREAD intrinsic. Assume the network is a Benes network [18]. Benes networks are rearrangeable: Any permutation may be routed without conflict. Hence, an elementary step is one network crossing in this particular case. However, the results may be extended, up to a constant factor, to any network emulating the well-known butterfly network in a finite number of steps, because a Benes network may be considered as two back-to-back butterfly networks [18]. In particular, Omega and Inverse Omega networks are topologically equivalent to the butterfly network.

Consider simple broadcasts; any static broadcast can be completed in one step and any parametric broadcast in $\log P + 1$ steps. If the broadcast source is a program scalar, the broadcast costs nothing, because all processors own the data (by parallel execution of the scalar code or any other way). Thus, we need only consider the case of broadcasting an element of a parallel array. Any input of the Benes network is the root of a P-leaf complete binary tree. Thus, the static broadcast costs one step.

A parametric broadcast cannot use the same technique. Even though the broadcasting tree does exist, the exact setting of the switches is not known at compile-time because the position of the root is a program variable. The simplest means to perform a parametric broadcast is to shift the source to a fixed position (e.g., processor 0) and to use a static broadcast. Shifting data is a parametric point-to-point communication, and has the same cost as a parametric translation.

Significant results have been obtained about the implementation of the most general multibroadcast patterns on butterfly and other hypercubic networks [18]. However, their implementation in the static execution model incurs extremely high costs because they involve irregular segmented prefix operations. Thus, the problem of compiling multibroadcast patterns must be carefully stated.

Consider the following legal HPF code:

```
Forall I
    A(I) = B(L(I))
```

With L non one-to-one, there are only two ways to compile such patterns: serializing the FORALL loop, as shown previously, or using Leighton's general algorithm [18]. However, these gather-based multibroadcasts are extremely rare in our benchmarks. The reason is perhaps that a clever user will avoid that programming style: Recognizing the hidden broadcast may be quite difficult for a compiler, whatever the execution model. Many architectures do offer special spreading or scanning hardware, and optimal exploitation of these features requires the broadcast to be expressed as a SPREAD, if possible. Thus, we consider the implementation of a SPREAD intrinsic.

Using the SPREAD intrinsic, a static multispread can be completed in one step, whereas if parametric, it requires $2 \log P + 1$ steps.

We only outline the proof. To avoid a lot of subscripts, we consider the generic example $B = SPREAD\ (A(k, a{:}\ b), DIM = 1, NCOPIES = n)$. The result is a two-dimensional array B, with $B(i, j) = A(k, j)$ for all i and j, $1 \le i \le n$ and $a \le j \le b$.

Consider the following data distribution: Each processor set has a virtual bidimensional $p \times q$ geometry, with p and q integer powers of 2, $p.q = P$ and $\log p = r$. Each processor has two coordinates (s_1, s_2) with $0 \le s_1 \le p - 1$ and $0 \le s_2 \le q - 1$ and each reference $A(i, j)$ is located on processor number $(i - 1, j - 1)$. When a processor is consid-

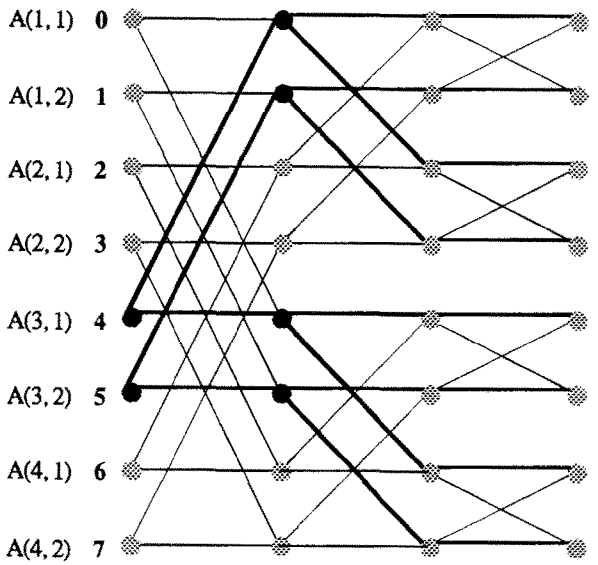

FIGURE 3 *SPREAD* (*A*(3, 1:2), 1, 4). Each dark node forwards its input to its two outputs: dark lines show the paths.

ered as a network input, its identification number is $p.s_1 + s_2$. When k is a constant, the paradigmatic spread is static. The principle of the multibroadcast is to use the butterfly network where stages 0 to $r - 1$ are broadcasting and stages r to $\log P$ realize a direct transmission of their values. Figure 3 gives an example, with $p = 4$ and $q = 2$. With DIM equal to 2, we would have to consider the reverse butterfly. More general dimensions come under the same analysis, as it depends only on the division of a processor address into $\log p$ bits for the fixed dimensions plus $\log q$ bits for the parallel dimensions.

If the dimension of an array is not a power of 2, we embed the array in an array of power of 2 size, execute the multispread on the temporary array, and conditionally store the result according to the real size.

As the Benes network includes two back-to-back butterfly networks, it can emulate this action in one step so that the multibroadcast using the SPREAD intrinsic takes one step.

In the parametric case, the $\log P$ factor comes from a parametric translation, with vector $-k$ (recall that we compile SPREAD (A(k, a: b), DIM=1, NCOPIES=n)). Thus, row k of A will be copied onto the first row of B and a static spread can take place in one step. Finally, we have to move the result to the correct position with another parametric translation requiring $\log P$ steps. As a remark, if DIM is a variable, we can compile the static spread for each dimension because the number of dimensions is generally low. Moreover, if the domain of the multispread is variable, again a global multispread can be performed on a temporary array and conditionally store the data according to the real domain.

These figures may seem quite high; however, all the available parallelism is exploited. Moreover, for static multibroadcasts, the solution is optimal in the sense that there is only one step. This contrasts for instance with the CM-5 broadcasting capabilities, which are limited to one processor at a time.

5.3 Multireduction

The (multi-)reduction differs from the (multi-)diffusion in the sense that the network has to combine values. Combining values means that the network switches can forward a unique result computed from its inputs by an associative operator (sum, max). We can realize the static (multi-)reduction by combining butterfly with our network: Each stage of the butterfly is executed by a crossing of our network and the combining operation is realized on the processors. Thus, the number of routing steps is equal to the number of stages in the butterfly, i.e., $\log P$.

In the case of parametric (multi-)reduction, again we process a parametric shift to move the data to a fixed position (for instance beginning at processor 0); then we apply the static (multi-) reduction with a conditional store and process a parametric shift to move the result to the correct position. Thus, it takes 3 $\log P$ steps.

5.4 Special Intrinsics and Functions

We have already shown that the FFT with a static argument may be transformed into a fully static routine. Systolic algorithms provide fully static implementations of the linear algebra intrinsics. For instance, the following algorithm realizes *MATMUL*(*MATRIX-A*, *MATRIX-B*):

```
C matrix conditioning
    Forall (i = 1:n)
         CSHIFT(MATRIX-A, DIM=2, i-1)
    Forall (i = 1:m)
         CSHIFT(MATRIX-B, DIM=1, i-1)
    R=0
C iterative computation
     do k = 1, n
          R = R + MATRIX-A * MATRIX-B
          CSHIFT(MATRIX-A, DIM=2, 1)
          CSHIFT(MATRIX-B, DIM=1, 1)
     end do
C the *product is a pointwise product
```

This algorithm was first designed for reasons that are similar to our objective, i.e., to get the best performance from a grid network and to avoid general communications. The grid network may, in turn, be emulated under the general assumptions stated at the beginning of this section, with one step for each of the grid NEWS (North East West South) directions.

5.5 Comparison

Comparisons between theoretical studies and actual machines are both presumptuous and unrealistic. Thus, the following results are not intended to compare what would be the execution of any program on the CM-5 and on a possible static machine. We consider the figures from the CM-5 network only as a testbed, i.e., giving the orders of magnitude for the performance of a recent dynamic routing network.

Two parameters characterize the performance of a network: Let r_m be the maximal network bandwidth per node and s the time to transmit a zero-sized message. To an approximation, r_m depends on the network bandwidth and on the source and destination memory bandwidth. With pipelined communications, the latency of a data transfer is

$$T = s + L/r_m. \tag{1}$$

where L is the data transfer size. With careful optimization, in the infinite data-size limit, the performance will be limited only by the processor's performance if the communication-to-computation ratio is lower than 1, and by the asymptotic network performance (r_m) if this ratio is larger than 1. In fact, assuming equal bandwidth performance, being better on "little" problems is the only advantage that one model has over the other.

We consider two characteristic figures for this comparison: T and $L_{1/2}$, the size for which the network reaches half of its maximal bandwidth. $L_{1/2}$ is the communication analog of the so-called $n_{1/2}$ for vector computations [12]. T measures the performance for programs where significant data transfer pipelining is not possible. The reason may be a very low virtualization ratio or the peculiar characteristics of the algorithm. For instance, a blocked algorithm with block data distribution will provide few communications: if the communications are not overlapped with the computations, T^{-1} will give the actual performance in most practical cases. On the other hand, $L_{1/2}$ gives one estimate of what would be an effective size for a problem if the communications dominate the computations, but can be arranged to exploit fully the network bandwidth in the asymptotic limit.

Many different values of the CM-5's performance have been reported. We consider the experimental values in [23] with the vendor message-passing library CMMD 1.3.1, and the values associated with the Active Message model [8]. It should be noted that CMMD 1.3.1 is the lowest level general-purpose communication library and may be considered as assembly-level programming. The results are based on permutation communications.

For the static network, we wanted to assess two speedups separately. The first comes from the static execution model, assuming off-the-shelf technology for the network design. The second comes from the fact that a network intended for this model can be designed with a more aggressive technology than a message-passing network, because its functionalities are simpler. Hence, we consider two cases: equal bandwidth performance and the network we are currently designing [4] (fast network in the following). For the equal bandwidth network, we have to assess raw hardware latency for a 512-processor machine, for which the figures of the CM-5 cannot be used because they involve the routing delay. We consider a 600 ns latency: this figure was reached by the GF11 using 1985 technology [17]. Table 6 shows the estimates for the translations patterns using formula 1. For the CM-5, the results do not depend on the distinction static or parametric. For the static network, we use the results of Section 5.1: thus, the parametric value for T is nine times its value for the static case (using $\log 512 = 9$): this comes from the fact that the consecutive translations must proceed in a lockstep fashion. Both implementations of the static model outperform the CM-5 network with the vendor message-passing library by one to two orders of magnitudes. With active messages, both static networks are better for the static translations, but only the fast network remains better for the parametric ones.

As no data concerning broadcasts and reductions were available to the authors, we had to limit our numerical comparisons to the translation case. Nevertheless, we must stress the following: for the CM-5, broadcasts and reductions use the control network: as it is a usual binary tree [20], no multioperations are allowed. Thus, even if multioperations incur high penalization in our model, this may be lower than pure sequentialization.

Table 6. Performance for Static and Dynamic Routing

		CM-5		Static	
		CMMD	Active Messages	Equal Bandwidth	Fast Network
Network parameters	r_m (MByte/s)	10	10	10	128
	s (μs)	97	3.3	0.6	0.3
Static translation	T (s)	98	4.1	1.4	0.4
	L1/2 (Byte)	970	33	6	39
Parametric translation	T	98	4.1	12.6	3.6
	L1/2	970	33	6	39

6 CONCLUSION

The key idea of the static model is to adapt the RISC principle to communications, i.e., to be optimal on the most frequent cases and correct on the others. Both the experimental results and the gross performance evaluations developed in this article show that the static model provides a significant speedup over dynamic routing. However, these figures isolate the network behavior, whereas the static model has consequences in other parts of a parallel architecture. With synchronous communications, all the processors have to be synchronized at each network cycle. This synchronization may be realized either by synchronization barriers or by a dedicated processor architecture. Synchronization barriers are the simplest solution, but may create overhead, because they preclude efficient network pipelining. For the second solution, the superscalar design and complex memory hierarchy of recent microprocessor architectures create many pipeline hazards. As adjusting the instruction threads by the compiler may be impossible, a VLIW-style architecture is recommended.

More generally, the current situation in parallel architectures is unbalanced. Many detailed studies are available about the performance of the processor's different parts (functional units, caches, . . .). However, experimental data about communications are sparse, and, except in a very few cases, mainly concern simple and synthetic situations. Our future research in this area will gather other experimental data about applications: in particular, the development of HPF to provide richer semantics than previous parallel Fortran and better communication statistics. In addition, we want to investigate the possible softening of the static model, e.g., using synchronous on-line routing in multistage networks would allow the direct execution of a set of dynamic communications.

ACKNOWLEDGMENTS

The authors thank F. Cappello, F. Delaplace, and D. Etiemble for many fruitful discussions. The detailed comments of the anonymous referees were of great help in making this article more readable.

REFERENCES

[1] D. Bailey, et al., "NAS parallel benchmarks results," in *Supercomputing 92*. New York: IEEE Computer Society Press, 1992, pp. 386–393.

[2] M. Berry, G. Cybenko, and J. Larson, "Scientific benchmark characterizations," *Parallel Comput.*, Vol. 17, pp. 1173–1194, 1991.

[3] A. Bik and H. Wijksoff, "Compilation techniques for sparse matrix computations," in *Proc. of International Conference on Supercomputing*, 1993, p. 416.

[4] F. Cappello, et al., "Balanced distributed memory parallel computers," in *22nd International Conference on Parallel Processing*, 1993.

[5] F. Cappello and C. Germain, "Towards high communication performance through compiled communications on a circuit-switched interconnection network," in *1st IEEE Symposium on High Performance Computer Architecture*, 1995, p. 44.

[6] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina, "Architectural requirements of parallel scientific applications with explicit communication," in *20th International Symposium on Computer Architecture*, 1993, p. 2.

[7] E. Dahl, "Mapping and compiled communication on the Connection Machine system," in *5th Distributed Memory Computing Conference*, 1990.

[8] V. Eicken, et al., "Active messages: A mechanism for integrated communication and computation," in *19th International Symposium on Computer Architecture*, 1992, p. 256.

[9] A. Feldmann, T. M. Stricker, and T. E. Warfel, "Supporting sets of arbitrary connections on

iWarp through communication context switches," in *5th ACM Symposium on Algorithms and Architectures*, 1993, p. 203.

[10] C. Germain, F. Delaplace, and R. Carlier, "A static execution model for data parallelism," *Parallel Processing Lett.*, vol. 4, pp. 367–378, Dec. 1994.

[11] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2*, IOP, 1988.

[12] R. Hockney, "Performance parameters and benchmarking of supercomputers," *Parallel Comput.*, vol. 17, pp. 1111–1130, 1991.

[13] INTEL Scientific Computers, *Paragon XP/S Product Overview*, Intel, 1991.

[14] F. Irigoin, et al., "A linear algebra framework for static HPF code distribution," in *4th International Workshop on Compilers for Parallel Computers*, 1993, p. 117.

[15] K. Iwama, E. Miyano, and Y. Kambayashi, "Routing problems on the mesh of buses," in *3rd ISAAC*, 1992, p. 155.

[16] C. Koelbel, "Compile time generation of regular communication patterns," in *Supercomuting '91*, 1991, p. 101.

[17] M. Kumar, "Unique design concepts in GF11 and their impact on performance," *IBM J. Res. Dev.*, vol. 36, pp. 990–999, 1992.

[18] F. Leighton, *Parallel Algorithms and Architectures*, Morgan Kaufmann, 1992.

[19] C. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. 34, pp. 892–901, Oct. 1985.

[20] C. E. Leiserson, et al., "The network architecture of the Connection Machine CM-5," in *SPAA '92*, 1992, p. 272.

[21] J. Lenfant, "A versatile mechanism to move data in an array processor," *IEEE Trans. Comput.*, vol. 34, pp. 506–522, June 1985.

[22] G. Lev, N. Pippenger, and L. Valiant, "A fast parallel algorithm for routing in permutation networks," *IEEE Trans. Comput.*, vol. 30, pp. 93–100, Feb. 1981.

[23] M. Lin, et al., "Performance evaluations of the CM-5 interconnection network," in *COMPCON 93*, New York: IEEE Computer Society Press, 1993, pp. 189–198.

[24] O. Lubeck and M. Simmons, "The performance realities of massively parallel processors: A case study," in *Supercomputing 92*, New York: IEEE Computer Society Press, 1992, pp. 551–560.

[25] A. G. Mohamed, et al., "Applications benchmark set for Fortran-D and high performance fortran," Northeast Parallel Architecture Center, Syracuse University, Tech. Rep. TR-SCCS 327.

[26] J. K. Prentice, "A performance benchmark study of Fortran 90 compilers," *Fortran J.*, vol. 5, 1993.

[27] C. Qiao and R. Melhem, "Reconfiguration with time-division multiplexed MINs for multiprocessor communications," *IEEE Trans. Parallel Distrib. Systems*, vol. 5, pp. 337–352, 1994.

[28] C. W. Tseng, "An optimizing Fortran D compiler for MIMD distributed memory machines," PhD Thesis, Rice University, 1993.

[29] M. Wolfe, "Tiny: A loop restructuring research tool," Oregon Graduate Institute of Science and Technology, Tech. Rep. 1992.