Automated Scalability Prediction via Data Parallel Compiler Support

Celso L. Mendes[*]
Image Processing Division
Institute of Space Research
São José dos Campos, SP 12201
E-mail: celso@dpi.inpe.br

Daniel A. Reed[†]
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
E-mail: reed@cs.uiuc.edu

ABSTRACT

Despite the performance potential of multicomputers, several factors have limited their widespread adoption. Of these, performance variability is among the most significant. Execution of some programs may yield only a small fraction of peak system performance, whereas others approach the system's theoretical performance peak. Moreover, the observed performance may change substantially as application program parameters vary. Data parallel languages, which facilitate the programming of multicomputers, increase the semantic distance between the program's source code and its observable performance, thus aggravating the performance problem.

In this paper, we propose a new methodology to automatically predict the performance scalability of data parallel applications on multicomputers. Our technique represents the execution time of a program as a symbolic expression that is a function of the number of processors (P), problem size (N), and other system-dependent parameters. This methodology is based on information collected at compile time. By extending an existing data parallel compiler (Fortran D95), we derive, during compilation, a symbolic model that represents the cost of each high-level program section and, inductively, of the complete program. These symbolic expressions may be simplified externally with current symbolic tools. Predicting performance of the program for a given pair (P, N) requires simply the evaluation of its corresponding cost expression. We validate our implementation by predicting scalability of a variety of loop nests, with distinct computation and communication patterns.

[*]Supported by the Brazilian Institute of Space Research (INPE) and by a scholarship from CNPq/Brazil.

[†]Supported in part by the Advanced Research Projects Agency under ARPA contract numbers DAVT63-91-C-0029 and DABT63-93-C-0040, by the National Science Foundation under grants NSF IRI 92-12976 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA contract numbers NAG-1-613 and USRA 5555-22, and by a collaborative research agreement with the Intel Supercomputer Systems Division.

1 Introduction

Multicomputers have been successfully used on a variety of scientific applications recently. Machines have been built with over a thousand processors, and there are no insurmountable technological obstacles that would prevent multicomputers from scaling to sizes with multiple teraflop performance. However, some factors have limited their widespread adoption. Among these, performance variability is one of the most important. Execution of some programs may yield only a small fraction of peak system performance, whereas others approach the system's theoretical performance peak. Moreover, the observed performance may change substantially as application program parameters vary.

In this paper, we present a methodology to automatically predict the performance scalability of data parallel applications on multicomputers. Our technique represents the execution time of a program as a symbolic expression that is a function of the number of processors, problem size, and other system-dependent parameters. We derive these expressions using support provided by the compiler and by a symbolic manipulator. By integrating compilation, performance analysis and symbolic manipulation tools, we show that it is possible to correctly predict the variations in behavior of a data parallel program written in a high-level language.

1.1 Motivation

Performance prediction can be a valuable tool in parallel program development and tuning. By offering estimates of the potential performance achievable by a program on a given system, it can help programmers assess the performance impact of selected constructs in their source codes even before the complete program is executed in a real system.

However, to maximize its potential benefits, the prediction mechanism must be closely integrated with the other programming tools available to the user. Tightly coupling the prediction, compilation and performance analysis components has the following advantages:

- Predictions can help in the code generation process;
- The compiler can evaluate the performance implications of a given data distribution;
- Bottlenecks in the program can be automatically identified and tracked.

A key feature of this integrated approach is automated performance prediction. With automation, the prediction module may become an integral part of the compilation mechanism. It can evaluate the code synthesized by other modules in the compiler, and provide the results of this evaluation as feedback information to those modules. This could possibly lead to the decision of synthesizing a new code, if other options exist. Thus, the data parallel compiler, extended with prediction capabilities, might be able to improve its code generation decisions, guided by the expected performance from each possible option.

Bottleneck identification is one of the most important phases in program optimization, as it allows one to concentrate optimization efforts on those program sections for which there might be the best potential gains in performance. We define a *bottleneck*, in a generic form, as the fragment with the greatest execution cost among the fragments comprising a certain program. Given the automatically generated costs of each program fragment as a function of the number of processors (P) and the problem size[1] (N), the compiler can easily determine the bottleneck associated with a particular system with a configuration (P_0, N_0) by evaluating the individual cost functions; the function with the greatest cost corresponds to the bottleneck fragment for that (P_0, N_0). For different values of N and P, the new bottlenecks can be identified in a similar form.

1.2 Paper Organization

The remaining of this paper is organized as follows. §2 reviews related work in the area, and puts our work in this context. We then describe, in §3, the compilation infrastructure required to extract symbolic information for derivation of cost models for each program fragment. §4 illustrates the construction of these cost models. In §5, we show the generality of our approach, by applying the automated process to predict performance of a wide variety of loops with distinct computation and communication patterns. §6 concludes our presentation.

2 Related Work

Performance scalability prediction on parallel systems has been attracting great interest recently. Many researchers have proposed techniques in the form of tools that predict the scalability of an application/system combination. Most of the proposed tools, however, still require considerable user intervention.

Clement and Quinn [5] presented an analytical modeling technique to predict the speedup of applications written in Dataparallel C [8], a SIMD model of parallel programming with explicit parallel extensions to the C language. They decomposed the execution time of an application into a sequential component, a parallelizable component, and some overhead due to communication. Their compiler generated an expression for the program speedup as a function of the number of processors, under a fixed problem size, by associating costs to each of those three components, regarding execution on a certain parallel system. However, some of their assumptions (e.g. that communication time is independent of message length, or that the number of cached and uncached memory accesses can be obtained from source code) seem too optimistic. Their technique was also limited to the case where parallelism is already explicit in the source program, and they did not originally conduct any study of scalability under variations in problem size. More recently, they extended this work [6] to study scalability of both the problem size and the number of processors, and build a symbolic model that represents the predicted execution time as a function of those parameters. The derivation of this model, however, required statistical methods and several experimental runs of the program with different problem sizes and numbers of processors.

Fahringer [7] designed a performance prediction tool named PPPT (Parameter-based Performance Prediction Tool), which analyzes a set of parameters that characterize the behavior of a parallel program, including work distribution, amount of communication and data locality. The tool correlates statically computed information with actual performance measurements to provide the program's performance estimates both to a compilation system and to general users. Such estimates, however, are presented in terms of predicted values for those selected parameters, instead of the execution time of the program or its component sections.

[1]In general, the problem size may include multiple dimensions. For simplicity, we will represent it in a single dimension.

[1]In general, the problem size may include multiple dimensions. For simplicity, we will represent it in a single dimension.

Adve *et al* [1] provided an overview of the various challenges involved in creating an environment for efficient programming in data parallel languages. They proposed an environment containing the compiler that we used in our work; in that environment, however, the analysis guiding the compilation process was to be provided by a separate tool named "Data-Mapping Assistant". This tool would enable user interaction, and would evaluate candidate distributions using an integer-programming framework [4]; the expected performance from each candidate distribution would be derived using the previously observed computation and communication behavior of *training sets* [2], consisting of small meta-benchmarks with the various constructs that are common in data parallel programs.

Some of the existing methods provide extremely accurate predictions, but at the cost of intensive analytical and experimental preparation. Given the increasing adoption of higher-level programming models for parallel systems, performance prediction methods relying on user intervention become even more undesirable. The increased distance between the structures in the source code of the application and the object code executed in the system places one more degree of complexity in the development of performance models. It seems clear that automation is the key factor to make this process feasible. Some of the more recent studies have been conducted in this direction, trying to integrate automatic prediction and compilation tools. However, that effort has not yet reached an ideal stage. Some methods derive a prediction for a specific combination of number of processors (P) and problem size (N), like in [1] and [7]. Others provide a symbolic model that can be evaluated at desired combinations of N and P, but either have a very limited application domain, as in [12], or require several executions of the program for model calibration, as in [6].

There has been no proposed method, so far, that provides a first-order, easily derivable model of the application's execution time (and of the execution times for internal code sections) as a function of the number of processors and problem size. Our symbolic scalability prediction method targets precisely this area. Because it is automated, it can be integrated in other compilation and analysis tools. Instead of aiming at extremely accurate predictions, we derive models that reliably bound the expected performance. The derivation of such models does not depend on instrumented executions; when available, these executions provide information to improve the predictions. Thus, the models can be obtained quickly, by properly leveraging on the capabilities of a data parallel compiler and of a symbolic manipulator.

3 Compilation Infrastructure

To derive scalability models automatically, we used the infrastructure of the new Fortran D95 compilation system [10]. This system was designed to support research on data parallel programming in High Performance Fortran (HPF) and to explore extensions that would broaden HPF's applicability or enhance performance.

Unlike Fortran D, which only extended Fortran with directives for data alignment and distribution, the Fortran D95 language contains nearly all the features of HPF, including syntax for array operations and support for parallel loops using the "FORALL" construct. Figure 1 shows the general organization of the Fortran D95 compiler; for more details about its internal structure, see [10].

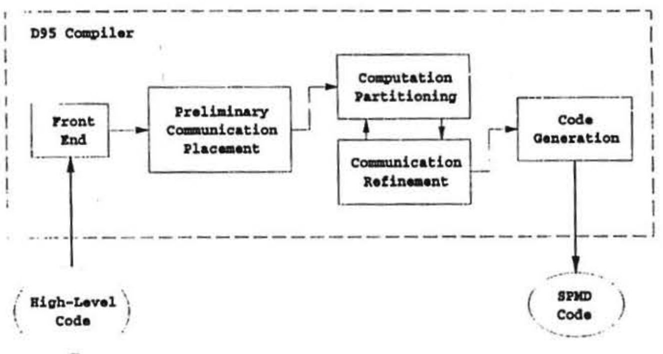


Figure 1: Fortran D95 compiler organization.

```
C  High-level code                     C  Synthesized SPMD code
         real a(1024),b(1024)                 real a(64), b(64), r(64)
CHPF$    processors proc(16)                  if ( MyNodeID >= 8 ) then
CHPF$    template t(1024)                       --<send b to node MyNodeID-P/2>--
CHPF$    align a(i) with t(i)                 endif
CHPF$    align b(i) with t(i)                 if ( MyNodeID <= 7 ) then
CHPF$    distribute t(block) onto proc          --<receive r from MyNodeID+P/2>--
                                                do i = 1 , 64
         do i = 1 , 512                            a(i) = 5 * b(i) + r(i)
            a(i) = 5 * b(i) + b(i+512)          enddo
         enddo                                endif
```

Figure 2: Example of SPMD code synthesization by the Fortran D95 compiler.

3.1 Code Translation Model

After selecting a specific computation partitioning, the D95 compiler generates message passing calls to communicate the nonlocal references in a given statement. To understand how this code translation process occurs, consider the D95 code fragment in Figure 2, with the corresponding pseudocode for its SPMD equivalent.

Because both arrays are distributed in a blocked form, the D95 compiler recognizes that there are two candidate computation partitions for this loop: the first partition would assign loop iteration i_0 to the processor that owns $\mathtt{a}(i_0)$ and $\mathtt{b}(i_0)$, which would require a remote access to read $\mathtt{b}(i_0 + 512)$; the second partition would assign loop iteration i_0 to the processor that owns $\mathtt{b}(i_0 + 512)$, thus requiring remote acesses to read $\mathtt{b}(i_0)$ and to write $\mathtt{a}(i_0)$. The D95 compiler applies a simple decision rule that minimizes the number of remote accesses; hence, it selects the first partition, and implements the appropriate message passing calls to access $\mathtt{b}(i_0 + 512)$ remotely, as indicated in the SPMD code. Notice that, in this particular case, this has the same effect of applying the "owner-computes" rule. Also, in the communication refinement process, the compiler optimizes communication by hoisting it out of the loop, grouping the remote accesses from all iterations into a single message.

In general, the creation of SPMD code follows a similar approach: the compiler translates the computation

High-Level Loop Body	Pattern
`a(i) = b(i-1)`	shift
`a(i) = b(1)`	broadcast
`do j=1,N {a(i) = a(i) + b(j)}`	all-to-all
`a(i) = b(N+1-i)`	unknown, regular

Table 1: Communication patterns implemented by the Fortran D95 compiler.

segment in each loop of the high-level code, then inserts the required communication code (before or after the computation, as appropriate), optimizing it if possible. Sections with conditional execution in the SPMD code are protected by *if* statements that specify which nodes execute that section.

For this specific loop, the required communication pattern synthesized by the D95 compiler is a *shift*, where node k receives data from node $k+P/2$. This compiler is capable of implementing a few distinct communication patterns, as indicated in Table 1, according to the body of the loop in the high-level source code.

We can associate a specific cost model with each particular communication pattern. In the shift, that cost is one message send and one message receive. For a broadcast, there are $P-1$ message sends, and one message receive. In the all-to-all pattern all P processors communicate with each other; each processor executes $P-1$ message sends and $P-1$ message receives. Finally, the unknown pattern corresponds to the situations where the precise type of communication is not known until runtime. In this case, we assume a cost model with a range of values varying from a minimum of one message send and one message receive to a maximum of $P-1$ sends and $P-1$ receives.

3.2 Compiler Extensions for Scalability Prediction

We extended the Fortran D95 compiler with the appropriate functionality to extract information regarding the execution cost, in symbolic form, of all the loops and message passing activity in the translated program. Specifically, we capture, during the compilation process, the following pieces of information:

- Loop limits for every loop in the program;
- Number of arithmetic operations in the right-hand side of an assignment;
- Type of communication pattern for every remote access resulting in a send/receive message pair in the translated code;
- Length of every message in the program, if known at compile time;
- Number of processors declared in a `processors` directive;
- Problem size, represented by the template extent declared in a `template` directive.

Figure 3 shows the organization of the new Fortran D95 compiler, extended with the features to support performance prediction. The Parameter Extraction module converts loop limits and message lengths in the synthesized code to their symbolic equivalents. The Cost Model Construction module receives these symbolic

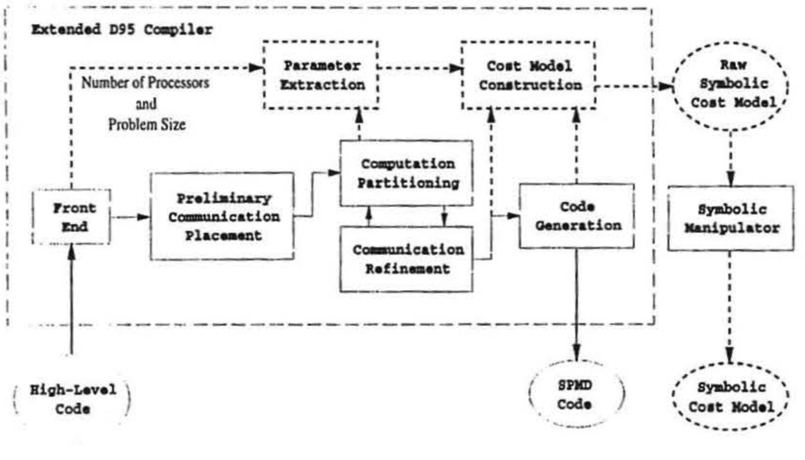


Figure 3: Extended Fortran D95 compiler organization.

terms, as well as the type of communication pattern implemented by the compiler after a computation partition is selected; then, for each segment in the generated SPMD code, this module creates a corresponding symbolic expression, representing the execution cost of that segment. Some of these raw cost expressions may be combined and reduced symbolically by a symbolic manipulator, to obtain simpler expressions representing larger parts of the code. The set of all resulting expressions constitutes the final cost model for the program. To predict the program's performance, one simply has to evaluate this cost model for specific values of N and P.

4 Derivation of Symbolic Cost Models

Given the extensions to the D95 compiler presented in the previous section, we now describe how to apply them for the derivation of cost models that we use to estimate the execution time of a program. Our presentation is based on a small example that illustrates all the relevant aspects involved in the automated construction of symbolic cost models.

4.1 Code Fragment Costs

Consider again the code fragment of Figure 2. Using our extended version of the D95 compiler to compile this fragment, the Parameter Extraction module produces the following information:

- Loop iteration space for each processor, in symbolic form:
  - Loop index variable: i
  - Symbolic lower limit: $\frac{N}{P}$ MyNodeID $+ 1$
  - Symbolic upper limit: $\frac{N}{P}$ MyNodeID $+ \frac{N}{P}$
  - Symbolic stride: 1
- Information regarding the body of the loop:

- One assignment, two arithmetic operations;
- Information about communication due to remote access in the loop:
  - Type of remote access: nonlocal read of `b(i + 512)`
  - Communication pattern synthesized: shift
  - Message length in symbolic form: $\frac{N}{P}$ floating-point elements

In these expressions, `MyNodeID` represents the processor number of each node. The information above is passed to the Cost Model Construction module, which builds a cost expression for each fragment of the generated SPMD code, where each term containing a loop limit or a message length is represented by its symbolic equivalent. These symbolic expressions are stored in an external file, such that they can be handled by a symbolic manipulation package (we are currently using Mathematica, although several other similar packages could have been used). For the program in our example, the predicted execution cost becomes

$$Cost(P,N) = S\left(\frac{N}{P}\right) + R\left(\frac{N}{P}\right) + \sum_{i=I_L}^{I_U} (K_a + 2K_r) \qquad (1)$$

where:

- $S\left(\frac{N}{P}\right)$ and $R\left(\frac{N}{P}\right)$ are functions representing the time to send and to receive, respectively, a message with $\frac{N}{P}$ floating-point elements;
- I_L is $\frac{N}{P}$ `MyNodeID` $+ 1$, and I_U is $\frac{N}{P}$ `MyNodeID` $+ \frac{N}{P}$;
- K_a is the time of an assignment;
- K_r is the time of an arithmetic operation.

Notice that functions S, R, and parameters K_a and K_r are system-dependent; we will show, in §4.3, how to determine their values for a particular system. The terms in (1) constitute what we call the *raw* symbolic cost model; they can be symbolically reduced to the form

$$Cost(P,N) = S\left(\frac{N}{P}\right) + R\left(\frac{N}{P}\right) + \frac{N}{P}(K_a + 2K_r) \qquad (2)$$

4.2 Bounds on Predicted Costs

For a program that has its cost represented by an expression similar to (2), predicting the execution time for a pair (P_1, N_1) would simply require the evaluation of $Cost(P_1, N_1)$. However, two problems may prevent the derivation of accurate predictions. The first problem is that some of the constants may change as we scale the problem size. As an example, the data access time is strongly dependent on whether the data item is cached or uncached. The second problem is that some of the terms in the cost model may not be determinable at compile time. The length of a given message, for example, might be unknown until the program is executed.

Although these factors occur in many real applications, it is generally possible to determine *minimal* and *maximal* values for the parameters that are unknown at compile time. If we use their minimal values in the cost expression, we can obtain a *lower bound* on the expected execution time for the program. By the same reason, using their maximal values results in an *upper bound* estimate for the execution time.

Hence, instead of building a single cost model for a given code fragment, we build two models: one for the lower bound and one for the upper bound estimate of the execution time. For each of these two models, we derive an appropriate set of parameters, corresponding to the characteristics of the program and of the underlying system.

4.3 Determination of System-Dependent Parameters

In our model, we represent the cost of a statement involving arithmetic operations, T_a, by

$$T_a = K_a + mK_r$$

where K_a is the time for an assignment, K_r is the time for an arithmetic operation and m is the number of arithmetic operations in the expression on the right-hand side of the statement. For communication, we keep using a model that represents the send and receive times as linear functions of the message length. For a specific system, we can estimate the lower and upper bound values for computation and communication parameters, as follows.

4.3.1 Computation Parameters

We determine estimates for the computation parameters, K_a and K_r, using meta-benchmarks where we measure the time for extreme cases of the corresponding operations. To obtain the lower and upper bound estimates for K_a, we measure the time for assignments with cached and non-cached operands, respectively. We estimate the lower bound on K_r as the time for the fastest arithmetic operation between two scalar operands, and its upper bound as the maximum time for any arithmetic operation between two multidimensional array elements.

There is another computation parameter in our model, K_f, that is used to represent the overhead associated to runtime functions invoked by the D95 compiler, for tasks like message buffer allocation and deallocation, etc. We represent such overhead simply as the cost of a dummy function call with the same number of arguments as the original call.[2]

We conducted experiments to determine the computation parameters on two systems, an Intel Paragon XP/S and an IBM SP/2. Table 2 shows the values that we obtained.

4.3.2 Communication Parameters

With repeated executions of communication benchmarks, one can take the smallest and the greatest values found for each parameter as estimates of the lower and upper bounds, respectively, for that particular parameter. From tests like these on the Intel Paragon and on the IBM SP/2, we obtained the values on Table 3. K_{Slat} and K_{Sbw} are the latency and per-byte costs, respectively, for a send, while K_{Rlat} and K_{Rbw} are the corresponding values for a receive.

[2]Because the runtime support of the Fortran D95 compiler is currently available only for the Intel Paragon, we replace these runtime functions by dummy subroutines, so that we can execute the tests on other platforms. Since our test programs are data independent, this approach does not change the computation or communication behavior in other parts of these programs.

	Intel Paragon XP/S		IBM SP/2	
Parameter	Lower Bound	Upper Bound	Lower Bound	Upper Bound
K_a	3.04×10^{-8}	6.91×10^{-7}	1.50×10^{-8}	3.68×10^{-8}
K_r	5.06×10^{-8}	6.73×10^{-7}	8.90×10^{-9}	2.20×10^{-6}
K_f	3.17×10^{-7}	3.94×10^{-7}	1.12×10^{-7}	1.61×10^{-7}

Table 2: Values (in seconds) of computation parameters for an Intel Paragon XP/S and for an IBM SP/2.

	Intel Paragon XP/S		IBM SP/2	
Parameter	Lower Bound	Upper Bound	Lower Bound	Upper Bound
K_{Slat}	3.65×10^{-5}	5.74×10^{-5}	5.01×10^{-5}	5.32×10^{-5}
K_{Sbw}	1.43×10^{-8}	1.46×10^{-8}	3.67×10^{-8}	5.19×10^{-7}
K_{Rlat}	5.54×10^{-5}	8.44×10^{-5}	1.23×10^{-5}	1.94×10^{-5}
K_{Rbw}	1.48×10^{-8}	1.53×10^{-8}	1.48×10^{-8}	1.60×10^{-8}

Table 3: Values (in seconds) of communication parameters.

5 Generality of the Prediction Method

Earlier in this paper, we had shown the automated derivation of the prediction model for the case of a simple loop. We now show that this technique works for a much wider range of cases. We take a large collection of loops, with many different computation and communication patterns, and show that our methodology produces symbolic scalability expressions for all of them; also, in almost all cases, such expressions correctly predict performance under varying values of N and P on an existing parallel system.

5.1 Collection of Loops

We consider the collection of loops prepared by Levine *et al* [9]. That collection consists of a variety of loop nests that represent different constructs intended to test the analysis capabilities of a vectorizing compiler. It comprises distinct types of computations that occur frequently on scientific applications.

Some of the loops in the collection could not be compiled with the original Fortran D95 compiler, due to limitations in the current compiler version. For those loops that were compiled correctly, we selected a representative subset, such that no computation pattern is repeated; this subset included twenty-two loops. In our first tests, we used a block distribution for the various arrays in the loops. After that, we repeated the tests for a few loops, this time using a cyclic distribution. The specific data distribution determines the required communication between processors, based on the data dependences existing on a given loop.

The subset of loops that we used in our tests presented a reasonable diversity of features, including the following:

- **Loop nesting level:** The loops in the subset were either singly or doubly nested loops; some of the doubly nested loops were perfectly[3] nested, while others were imperfectly nested. Also, some of the imperfectly nested loops had multiple inner loops inside the outer loop.

- **Type of iteration space**: For the double loops, the iteration space was either rectangular or triangular; most of the loops had unit stride, with a few exceptions where the stride was a constant greater than one.
- **Number of arithmetic operations**: Some of the loops contained a large number of operations, involving many different elements, while others contained only one operation.
- **Type of data dependences**: There were all types of data dependences between iterations in the various loops: flow, anti and output dependences [3]; some of the loops presented more than one type of dependence. For most loops, the distance vector of the dependence was constant, but for a few of them it was variable.

By selecting a specific data distribution for the arrays in each loop, we constrain the potential parallelism existing on a given loop nest. For a selected data distribution, the particular data items accessed in each iteration, plus the data dependences across iterations, determine the valid translations of the original high-level code into low-level SPMD code with explicit message passing. Thus, given our selected data distributions for the various loops in our subset, we obtained representatives for the following features:

- **Available parallelism**: Some of the loops had no dependence across iterations, and all iterations could proceed in parallel. In some other loops, because of dependences across iterations executed by different processors, the execution became partially or even completely serialized.
- **Number of remote references**: For a selected data distribution, the criteria used by the compiler to partition the computation may affect the number of local and nonlocal references on a given statement. Because the D95 compiler currently seeks partitions that minimize the number of remote accesses, our loops presented either no remote references or, in the majority of cases, small numbers of such references.
- **Type of communication pattern**: For most loops, some form of communication was required. All those communication patterns in Table 1 were used, in at least one of the loop nests, by the D95 compiler.
- **Grain size of the parallelism**: There was a large diversity in the resulting computation to communication ratio among the loops. Some cases required just one message passing transfer at the very beginning, and the rest of the execution was purely computational. For some of the doubly nested loops, however, each iteration of the inner loop required a message exchange, making the computation grain size extremely small.

5.2 Scalability of Individual Loops

After compiling each loop nest in the subset with our extended Fortran D95 compiler, we obtained the corresponding symbolic cost expressions. Although the loop nests are simple, their execution costs vary widely, mainly because of the different communication patterns imposed by the data dependences in each of them.

[3]A perfectly nested loop is such that, except for the innermost loop, each loop body contains one loop and no other statements.

```
C  Loop nest s113:
      do i = 2 , N
         a(i) = a(1) + b(i)
      enddo
C  Loop nest s242:
      do i = 2 , N
         a(i) = a(i-1) + s1 + s2 + b(i) + c(i) + d(i)
      enddo
```

Figure 4: High-level source code for some of the loops in the subset.

To illustrate this process of cost model generation, and to serve as a basis for the discussion that will follow, we present the details for two of our loop nests, s113 and s242. Their high-level source codes are in Figure 4. All the arrays in these loop nests had a block distribution.

The cost model derived for loop nest s113 is

$$C_{s113}(P,N) = \left(\frac{4N}{P}-1\right)K_a + \left(\frac{2N}{P}-1\right)K_r + \left(\frac{N}{P}\right)K_f + R(1) + (P-1)S(1) \tag{3}$$

where R represents the cost to receive a message with one element ($a(1)$) and $S(1)$ represents the cost for the first processor to send a message with that element to each of the remaining $P-1$ processors.

For loop nest s242, the cost model created by the extended D95 compiler is

$$C_{s242}(P,N) = PR_1(1) + P\left(\frac{N}{P}-1\right)(3K_a + 5K_r + K_f + R_2(1) + S_2(1)) + PS_1(1) \tag{4}$$

where R_1, S_1, R_2 and S_2 represent the costs of message passing functions inserted by the D95 compiler for the remote access to $a(i-1)$. Notice that this loop nest presents a flow dependence across iterations, where array element $a(i_0)$ is written in iteration $i = i_0$ and read in iteration $i' = i_0 + 1$. Thus, processor k must wait until processor $k-1$ completes all of its iterations, making the loop execution completely serialized; hence, the terms in the execution cost become proportional to the number of processors P.

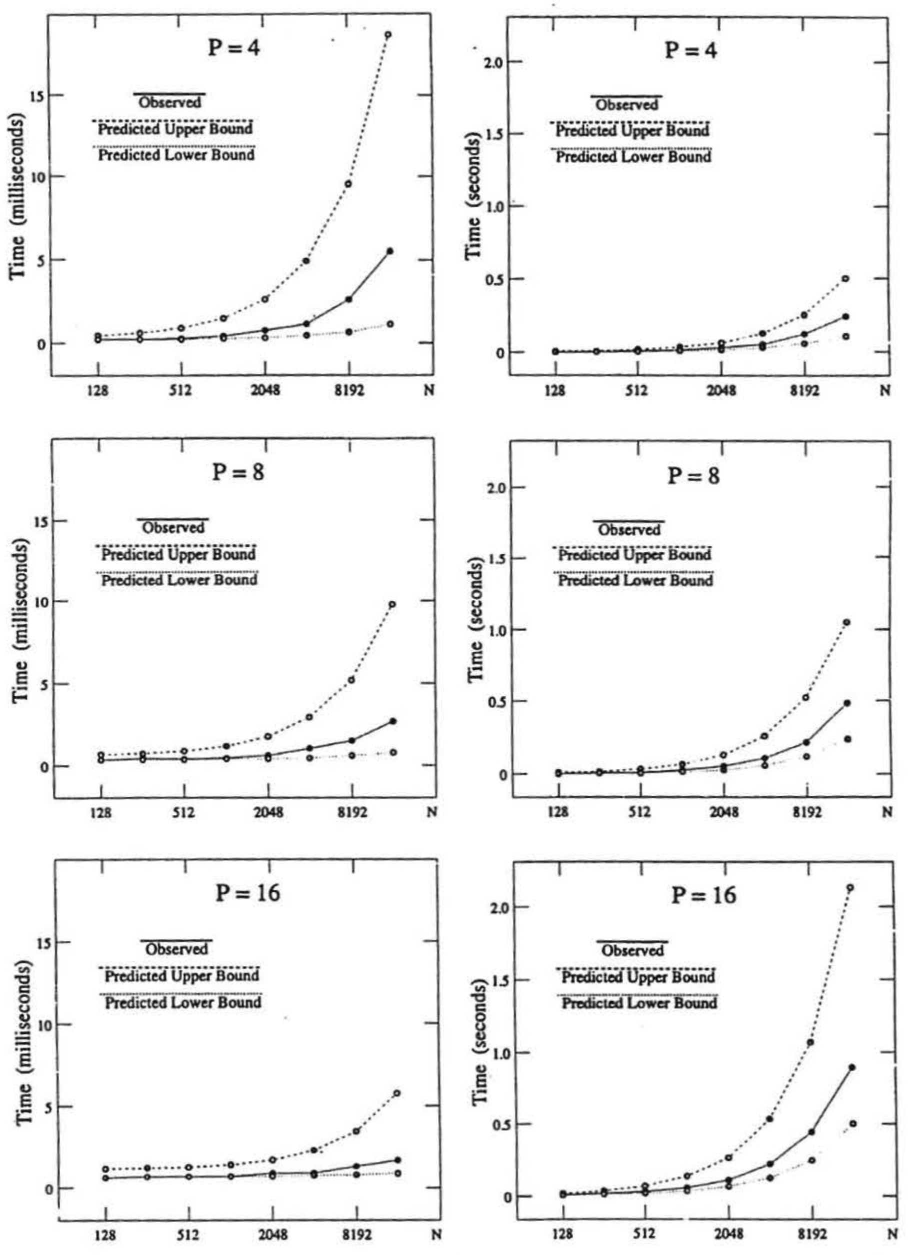
5.3 Prediction Experiments

After obtaining the scalability expressions for each loop nest, we used the constants described in §4.3 to compute lower and upper bound estimates of their execution times for selected values of P and N.

Assuming the computation and communication constants for the Intel Paragon XP/S from Tables 2 and 3, we computed predictions for each loop nest, varying the number of processors P such that $P \in \{4,8,16\}$ and the problem size N such that $N \in \{128,256,512,1024,2048,4096,8192,16384\}$ for the single loops, or $N \in \{32,64,128,256,512,1024,2048\}$ for the double loops. Figure 5 shows some of our prediction results, in comparison to the observed execution times on the Intel Paragon; the results for some of the other loop nests can be found in [11]. All the results in this section reflect predicted and observed behavior for the node with the *maximum* execution time.

To quantitatively evaluate our predictions, we introduce the error functions

$$RatioLB(P,N) \triangleq \frac{LowerBound_{predicted}(P,N)}{T_{observed}(P,N)} \quad , \quad RatioUB(P,N) \triangleq \frac{UpperBound_{predicted}(P,N)}{T_{observed}(P,N)}$$



(a) Loop s113 (b) Loop s242

Figure 5: Predicted and observed execution times for some of the loop nests.

Loop Nest	RatioLB	RatioUB	Loop Nest	RatioLB	RatioUB
s111	0.112	1.893	s221	0.554	2.341
s112	0.340	1.463	s233	0.168	2.752
s113	0.629	2.755	s235	0.101	1.562
s115	0.629	3.127	s242	0.562	2.370
s119	0.497	1.334	s254	0.444	2.911
s121	0.370	2.165	s256	0.433	1.777
s122	0.345	5.777	s311	0.384	2.855
s131	0.334	1.436	s3112	0.382	2.858
s132	0.860	2.351	s322	0.393	1.686
s2102	0.041	0.927	s323	0.554	2.343
s211	0.375	2.135	s112-cyclic	0.329	5.799
s2111	0.347	1.253	s3112-cyclic	0.265	2.047

Table 4: Mean values of ratios between predicted and oserved times on the Paragon.

and compute the geometric means of their values across all the range of variation for P and N used in the experiments. Table 4 contains the results of our predictions for the Intel Paragon XP/S.

5.4 Analysis of Prediction Results

In general, the numbers in Table 4 show that our predictions correctly estimate the intervals bounding the observed execution times for nearly all cases. This is particularly relevant if we consider that the observed execution times for these loops vary by several orders of magnitude (e.g. a few microseconds for loop nest s113 and more than twenty seconds for loop nest s233).

As the problem size grows, the observed behavior for the doubly nested loops tends toward the upper bound predictions faster than the single loops. This is expected; for the same problem size N, most arrays in the double loop cases have size N^2, and thus are more susceptible to cache misses than the corresponding unidimensional arrays in the single loops. Our lower bound computation constants implicitly assume no cache misses for data access.

Some of the loops (e.g. s113) scale well with increasing the number of processors. Other loops, however, do not present the same scalability, or even show a decrease in performance with more processors, as in the case of loop s242 (note in Figure 5(b) that both the predicted and observed execution times increase as the number of processors increases, for any problem size). These loops contain a flow dependence along the same dimension in which the arrays are distributed, and thus their execution is completely serialized. Nevertheless, our predictions correctly capture this effect, and provide bounds that clearly expose this behavior, as one can see in Figure 5(b).

In general, we can analyze the loop nests, regarding their scalability with the number of processors, by considering the data dependences and the distribution of the arrays involved in those dependences. We can classify the loop nests in two groups. The first group contains those loop nests with one of these properties:

- *No dependences between iterations* (like in loop nest s113): There may be a preliminary phase to access remotely stored data, but then all the iterations can be executed concurrently.
- *Only anti-dependences between iterations*: If the dependence is between iterations executed on the same processor, the processors can execute independently of each other. If there is dependence between iterations executed on different processors, there might be a preliminary communication phase, as in the previous case; the rest of the processing is purely computational. The processors can execute this computational phase independently of each other.
- *Flow dependence along a direction that is not distributed*: The dependence is between iterations executed in the same processor, and the processors can execute in parallel.

In all the cases of this first group, there is potential for nearly full parallel execution; thus, there is good scalability with more processors.

The second group of loop nests includes those cases that present a flow dependence along the same direction in which the arrays are distributed. For this group, the scalability will depend on the nesting level of the loop carrying the dependence, as follows:

- *Dependence carried by the outer loop* (like in loop nest s242): The computation grain size is maximal, and the execution becomes completely serialized. Because there is no overlap between computation on different nodes, adding more processors does not reduce the total computation time, and only increases the total communication time; thus, performance degrades with more processors.
- *Dependence not_carried by the outer loop*: In this case, the computation grain size is smaller, and the execution is pipelined across the processors. There is potential for overlap between distinct iterations of the outer loop on different processors. The scalability depends, basically, on the constant terms associated to the computation and communication costs, and on the grain size.

By deriving cost models in the extended D95 compiler, our predictions tend to match the observed behavior for the loops in the first group, because, in general, we assume that all processors execute all statements (total parallelism). We also detect, with the compiler, those cases of loops in the second group where the dependence direction is the same as that of the outer loop, and adjust the cost model to reflect the cost of their serialized execution (notice the factor P applied to all terms of equation (4)). For loops in the second group where there is partial parallelism, our models would predict execution times slightly smaller than observed in practice; however, the "deviation" from full parallelism in these cases is proportional to $(P-1)/N$ (the delay from the first to the last processor in the pipeline, relative to the total work), and that error becomes insignificant when $P \ll N$.

6 Conclusion

With automation of the scalability cost model derivation via the data parallel compiler, we have a powerful mechanism that represents the expected execution cost of code fragments as functions of N and P. We can find the system-dependent constants in these models using specific benchmarks on the system of interest. Thus, predictions for new systems can be easily derived once the corresponding constants are available.

The tight connection between the prediction and compilation mechanisms opens a new set of opportunities for code optimization. A data parallel compiler, extended with predictive capability, can make code generation decisions guided by the specific computation and communication characteristics of the underlying system. This approach can potentially lead to more flexible data parallel languages, where the programmer would be relieved from the (sometimes difficult) task of specifying the distribution of data across the processors of a given system. Under this new scenario, programs would become even more portable, as the compilers would automatically find the best data distribution for each parallel system.

Our extended data parallel compiler derives predictions for total execution time as a simple concatenation of predictions for individual code sections. Predictions of total execution time can help in the decision of porting the code to different systems. Predictions for individual code sections are useful to identify scaling bottlenecks in the program. We have shown how to obtain these predictions, and successfully applied the methodology to a variety of code fragments with different computation and communication patterns.

References

[1] Adve, V., Carle, A., Granston, E., Hiranandani, S., Kennedy, K., Koelbel, C., Mellor-Crummey, J., and Warren, S. Requirements for data parallel programming environments. *IEEE Parallel & Distributed Technology 2*, 3 (Fall 1994), 48–58.

[2] Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. A static performance estimator in the Fortran D programming system. In *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[3] Banerjee, U. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, Massachusetts, 1993.

[4] Bixby, R., Kennedy, K., and Kremer, U. Automatic data layout using 0-1 integer programming. Tech. Rep. CRPC-TR93349-S, CRPC/Rice University, 1993.

[5] Clement, M. J., and Quinn, M. J. Analytical performance prediction on multicomputers. In *Proceedings of Supercomputing'93* (Portland, November 1993), pp. 886–894.

[6] Clement, M. J., and Quinn, M. J. Symbolic performance prediction of scalable parallel programs. In *Proceedings of the* 9th *International Parallel Processing Symposium* (April 1995).

[7] Fahringer, T. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Norwell, Massachusetts, 1996.

[8] Hatcher, P. J., and Quinn, M. J. *Data Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, Massachusetts, 1991.

[9] Levine, D., Callahan, D., and Dongarra, J. *Test Suite for Vectorizing Compilers*. 1991.

[10] Mellor-Crummey, J., and Adve, V. *Fortran D95 Compiler Overview*. Available from http://www.cs.rice.edu/ mpal/SC95, 1996.

[11] Mendes, C. L. *Performance Scalability Prediction on Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997.

[12] van Gemund, A. J. C. Compile-time performance prediction of parallel systems. In *Proceedings of the* 8th *International Conference on Modeling Techniques and Tools for Computer Performance Evaluation* (Heidelberg, September 1995), pp. 299–313.