Compile-Time Estimation of Communication Costs for Data Parallel Programs

Thomas Fahringer
Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090 Vienna, Austria
e-mail: tf@par.univie.ac.at

published in: Journal of Parallel and Distributed Computing 39(1), 46-65, Nov. 1996

Abstract

Most of the current compiler projects for distributed memory architectures leave the critical and time-consuming problem of finding performance-efficient data distributions and profitable program transformations for a given parallel program almost entirely to the programmer. Performance estimators provide critical performance information to both programmers and parallelizing compilers, the most crucial part of which involves determining the communication overhead induced by a program. In this paper, we present a very practical approach to the problem of compile-time estimation of communication costs for regular codes that includes analytical methods to model the number of messages exchanged, data volume transferred, transfer time, and network contention. In order to achieve high estimation accuracy, our estimator aggressively exploits compiler analysis and optimization information. It is assumed that machine parameters and problem size are known at compile time. We conducted a variety of experiments to validate the estimation accuracy and the ability to support both the programmer and compiler in the effort of performance tuning of parallel programs. We believe that our approach can be automatically applied to a large class of regular codes.

1 Introduction

Distributed memory multiprocessor systems (DMMPs) are increasingly being expected to deliver very high performance for scientific applications. A major obstacle to the widespread use of these machines is the fact that they are hard to program. Much recent research has thus centered around efforts to develop parallelizing compilers [2, 3, 4, 16, 17] in order to reduce the cost of developing new code or porting existing programs. A major problem of most existing compilers is that they require the programmer to interactively invoke optimizing program transformations and explicitly specify data distribution strategies. Unfortunately, the programmer frequently is not able to foresee the performance effects and trade-offs of applied code changes. As a result, many programmers compile, execute, and evaluate their code on the real machine every time they select a data distribution strategy or apply an important code transformation.

A key measure of the performance of a distributed memory parallel program is the communication overhead. On most current parallel systems sending data from a local to a remote processor still takes one or two orders of magnitude longer than the time to access data on a local processor. The behavior of the communication overhead of a parallel program can be strongly influenced by program transformations and data distribution strategies. For instance, loop distribution may break a data dependence, and consequently eliminate communication. Loop interchange may allow pulling communication statements out to a higher loop level. Inherently, it is the data distribution scheme which determines the degree of parallelism of a program, and when interprocessor communication is required to carry out a computation. Therefore, providing both programmer and compiler with an automatic communication cost function can considerably alleviate performance tuning of parallel codes.

This paper describes an analytical approach to communication cost estimation of distributed memory regular parallel codes. We will show how to estimate the number of transfers, amount of data transferred, transfer time, and network contention. These parameters can be selectively obtained for statements, loops, procedures, and the entire program; furthermore, their effect with respect to individual processors can be examined. Our method is based on modeling loop iteration spaces, array access patterns, and data distributions by intersection and volume operations on n-dimensional polytopes. Additional architecture specific factors are incorporated for transfer times and network contention. It is assumed that problem size and machine parameters are known at compile time which is a common assumption made for many state-of-the-art performance estimators. Program unknowns (e.g. statement execution counts) are derived by a single profile run based on the original sequential program. Large portions of the profile data can be automatically adapted for many important program changes without redoing the profile run.

Clearly, improving other performance parameters such as load balance, cache performance, computation overhead, execution time, etc. can be as important as an efficient communication behavior. The importance of different parameters with respect to the performance of a program is strongly architecture dependent. In previous work [11, 10, 9] we have described other performance parameters such as work distribution, number of cache misses and computation time, and validated their accuracy and performance impact. The principal contribution of this paper is an in-depth evaluation of communication parameters.

All communication cost functions as described in this paper have been fully implemented under P^3T, a performance estimator for parallel programs. This tool has been developed in the context of the Vienna Fortran Compilation System (VFCS - [4]), which is a parallelization system for DMMPs.

The rest of this paper is organized as follows. In Section 2 related work is presented. Section 3 describes the compilation and programming model on which the performance estimation is based on, and introduces some terms that we shall use throughout the paper. Section 4 briefly outlines the functionality of P^3T. In the following four sections we describe the cost models for each of the four major communication overhead sources: number of transfers, data volume exchanged, communication time, and network contention. Section 9 presents experimental results to validate the estimation accuracy of our model and the ability to successfully support parallelizing and optimizing small sized regular codes under *VFCS*.

Finally, conclusions along with a discussion on future work are presented in Section 10.

2 Related Work

The area of performance prediction for parallel programs has seen considerable research activity during the last few years:

U. Kremer [18] implemented an automatic data distribution tool for HPF style programs which uses performance prediction to examine different data layouts. A compiler model simulates compilation phases without generating a parallel program. A machine model pre-measures the execution time of a set of communication and computation kernel routines for different data layouts, processor numbers, and array sizes. Our method is similar to Kremer's approach with respect to separating machine independent from machine dependent performance characteristics. A main advantage of Kremer's approach is that compiler phases are simulated without creating a complete program. We believe that our estimates are inherently more accurate, as they are based on an explicitly parallel program with all communication completely specified. Furthermore, we use sophisticated control flow analysis to accurately model complex loop bounds and array reference patterns, whereas Kremer's approach is based on simplifying assumptions (e.g. fixed loop bounds) at the cost of estimation accuracy.

M. Gupta and P. Banerjee [15] describe a model which has a similar goal as this paper: to drive automatic data distribution for distributed memory architectures by communication cost estimation. It is assumed that each loop index used as a subscript in an array reference varies so as to sweep over the entire range of array elements along that dimension. Our analysis detects the array portions accessed inside of loop nests with high accuracy based on modeling loop iteration spaces in combination with array subscript expressions. Gupta's and Banerjee's approach also considers cyclic and block-cyclic distributions and goes beyond performance prediction by also determining what type of communication takes place and at what point in the program.

V. Sarkar [22] estimates the runtime of parallel programs at compile time incorporating profiling and pre-measured kernel codes. This approach targets single assignment languages which simplifies performance analysis substantially.

K.Y. Wang [25] characterizes parallel programs by a parameterized performance model which aims at symbolic runtime information. His approach also includes superscalar architectures. Our communication cost function is more accurate due to our fine-grain modeling of loop iteration spaces and array access patterns.

A. Agarwal [1] describes the impact of network contention for multithreaded processors. His model is reasonably accurate, however, assuming uniform traffic rates from all the nodes and uniformly distributed and independent message destinations considerably alleviates performance analysis.

3 Compilation and Programming Model

The communication cost functions which are described in this paper are based on the underlying compilation and programming model of *VFCS*, which is explained as follows:

The parallelization strategy of *VFCS* focuses on domain decomposition in conjunction with the Single-Program-

Multiple-Data (SPMD) programming model. This model implies that each processor is executing the same program based on a different data domain. The input to *VFCS* are Vienna Fortran programs [26]. Vienna Fortran is a machine-independent language extension to Fortran77, which provides annotations for the specification of data distributions. Vienna Fortran allows the user to declare processor arrays by means of the *PROCESSORS* statement. For instance, *PROCESSORS P(4,8)* declares a two-dimensional processor array with 4x8 processors. Distribution annotations may be appended to array declarations to specify distributions of arrays to processors. For instance, *REAL A(N,N) DIST(BLOCK,BLOCK) TO P* distributes array A two dimensional block-wise to the processor array P. The output of *VFCS* is a parallel Fortran program with explicit message passing.

Let r_1, r_2 be two array references, then r_1 is *data dependent* on r_2 if r_1 and r_2 read or write a common memory location in a way that requires their execution order to be preserved. A *true dependence* occurs when r_1 is written before r_2 is used.

Let P denote the set of processors available for the execution of the parallel program. The specification of the mapping of the array elements to the set of processors is called the *data distribution* of a program. A processor is then thought of as *owning* the data assigned to it; these data elements are stored in its local memory. P^A refers to the set of all processors to which A is distributed. For each $p \in P^A$ and each A, $LS(A,p)$ is called the *local segment* of A associated with p. The analysis described in this paper can be applied to general block distributions and replication of arrays. General block distribution enables mapping of arbitrary sized array dimension segments to processors which goes beyond what has been defined by conventional block distributions [17].

The *work distribution* of a parallel program is determined – based on the data distribution – according to the *owner computes rule* which means that the processor that owns a datum will perform the computations that make an assignment to this datum.

Non-local data referenced by a processor are buffered in so-called *overlap areas* that extend the memory allocated for the local segment. The updating of the overlap areas is automatically organized by the system via explicit message passing. Updating overlap areas is the result of a series of optimization steps that determine the communication pattern, extract single element messages from a loop and combine to vectors (communication vectorization), remove redundant communication (communication fusion), and aggregate different communication statements (communication aggregation). These optimizations are applied after the first step in the parallelization of the program has been performed, which is described as follows: for each potentially non-local access to a variable, a communication statement *EXSR* (EXchange Send Receive) is automatically inserted. An *EXSR* statement [4] is syntactically described as $EXSR\ A(I_1,...,I_n)\ [l_1/u_1,...,l_n/u_n]$, where $v = A(I_1,...,I_n)$ is the array element inducing communication and $[l_1/u_1,...,l_n/u_n]$ is the overlap area for array A. For each i, l_i and u_i respectively specify the left and right extension of dimension i in the local segment. The dynamic behavior of an *EXSR* statement can be described as follows:

```
IF (executing processor p owns v) THEN
        send v to all processors p', such that
        (1) p' reads v, and
        (2) p' does not own v
ELSE IF (v is in the overlap area of p) THEN
         receive v
     ENDIF
ENDIF
```

In the remainder of this paper we refer to EXSR statements as communication statements.

In general, the estimation approach as described in this paper models communication based on overlap areas. Note that the concept of overlap areas is not restricted to neighboring communications. If the overlap area extends the local segment of a processor beyond the segments owned by neighboring processors then also non-neighboring communication is implied. In extreme cases, total overlap across an entire array dimension may induce communication with all processors onto which this array dimension is distributed. For such cases buffer communication [23] should be used – instead of communication based on overlaps – which is most useful when non-local data is bounded in size but not near the local segment. Our performance model provides only restricted support for buffer communication. If the underlying compiler can statically describe the non-local data areas that need to be communicated and their ownership, then our analysis is applicable to buffer communication as well. In reality, however, buffer communication often depends on runtime resolution which prevents static performance prediction.

Every communication statement is implied by a unique array reference. $\mathcal{C}_L$ denotes the set of communication statements in a loop nest L. All communication statements which immediately enclose a loop nest L belong to $\mathcal{C}_L$.

For our performance model we separate the communication statements of a parallel program into two *communication statement classes*: $\mathcal{C}_i$ is the set of *independent* communication statements whose associated overlap areas do not depend on any loop variable. Independent communication is induced by non-local array references that are not involved in true dependences. It can occur within or outside of loops, and all processors can communicate in concert. All other statements are part of $\mathcal{C}_d$, the set of *dependent* communication statements, whose overlap area depends on loop variables. Dependent communication statements are caused by true dependences and always appear inside of loop nests. Partial loop vectorization may be possible for a dependent communication statement C, however, this is restricted by the deepest loop level that imposes a true dependence on the array reference associated with C. Dependent communication statements can be further characterized by the loop nest level at which they appear. If $C \in \mathcal{C}_d$ cannot be vectorized at all because of a true dependence inside of the loop directly enclosing C, then C is referred to as *innermost communication*, otherwise *nested communication*. $\mathcal{C}_{in}$ and $\mathcal{C}_{ne}$ respectively define the set of innermost and nested communication statements in $\mathcal{C}_d$. Note that $\mathcal{C}_d = \mathcal{C}_{in} \cup \mathcal{C}_{ne}$, $\mathcal{C}_i \cap \mathcal{C}_d = \phi$, and $\mathcal{C}_{in} \cap \mathcal{C}_{ne} = \phi$, where ϕ is the empty set. In the remainder of this paper, we always assume that communication vectorization, aggregation and fusion has been applied by the underlying compiler except where indicated otherwise.

The described communication cost functions are applicable to arbitrarily nested loops with communication statements at all loop nest levels. Loop bounds and array subscript functions are linear functions of loop variables of all enclosing loop nests. Codes containing irregular problems (e.g. indirect array accesses) cannot be processed.

4 P^{3}T

The communication cost models as described in this paper have been fully implemented and integrated under P^3T which is a performance estimator for data parallel programs based on DMMPs. The goal of P^3T is to support the programmer and compiler in performance tuning of programs which includes the selection of data distributions and program transformations.

The underlying approach of P^3T is organized around two major components: *profiling* and *performance parameters*. The original input program is profiled once – using the Weight Finder [8] which is a highly optimized profiler – to determine characteristic values for branching probabilities, statement execution counts and unknown parameters in loop bounds. In [8] we have shown that large portions of the profile data can be automatically adapted for most important program transformations without redoing the profile run. However, for changing the problem size, we currently have to repeat the profile run. We are investigating scalable methods in order to automatically scale profile data based on a small problem size – to minimize the profile time – for larger ones. We believe that profiling is a reasonable choice to handle program unknowns, in particular, if we consider the fact that most performance estimators use guessing or unrealistic distribution functions to model program unknowns.

P^3T computes a set of parallel program parameters which includes: work distribution, number of cache misses, number of transfers, amount of data transferred, network contention, transfer and computation time. We plan to extend P^3T by an execution time parameter in the future. Note that the analysis to obtain the parallel program parameters is done primarily at the machine independent program level. Therefore, even though these parameters inherently refer to machine specific behavior, they are despite denoted program parameters as opposed to machine parameters.

The parallel program parameters are designed to be as machine independent as possible. However, in order to build a highly efficient performance estimator we added some of the most important machine specific factors including cache line size, overall number of cache lines, data type sizes, routing policy, startup times, message transfer time per byte, etc. These machine parameters as well as problem size and number of processors are supposed to be known at compile time. Large portions of the machine parameters can be easily adapted for different architectures.

In order to effectively use the parallel program parameters for a specific architecture, we need to determine parameter priorities for all architectures of interest. In order to derive these priorities, we apply a training phase of P^3T – using a large set of kernel codes and application programs – for each different target architecture. For instance, on the iPSC/860 hypercube, transfer time (tt) and the number of transfers (nt) turned out to be most critical. Computation time, work distribution, and number of cache misses are in most cases less critical than the previous two parameters. Network contention usually has the lowest priority. We have validated the parameter priorities through many experiments [11, 10]. For detailed information about P^3T, the reader may refer to [11, 10, 13].

5 Number of Transfers

One of the most significant contributors to the communication overhead of a parallel program on most distributed memory multicomputers is the number of messages exchanged, which reflects the high message startup times. In the remainder of this section we refer to a single message exchanged between a pair of processors as a *transfer*.

Since the number of transfers has a major impact on performance, the compiler tries to reduce the effect of message startup times by applying communication vectorization, fusion and aggregation [4]. In this section, we describe how to estimate the number of transfers implied by a parallel program. We demonstrate that carefully modeling a program's loop iteration spaces, array access patterns, and data distribution strategies, on the one hand, and the compiler communication model, on the other hand, yields a high estimation accuracy of this parameter. Our model for the number of transfers ports easily to other distributed memory architectures as it does not depend on machine specific factors.

In the following we describe for each different communication statement class (see Section 3) how to estimate the number of transfers.

5.1 Independent Communication

Independent communication statements reflect the so-called loosely synchronous programming model [14]. It is assumed that all processors communicate simultaneously when executing an *independent* communication statement. The following code shows an excerpt of a pointwise relaxation using red-black checkerboard algorithm [24]:

```
    DOUBLE PRECISION VAL(N,N)
    ...
    DO K = 1, TIME
      ...
      DO J=1,(N-1)/2
        DO I=1,(N-1)/2
          VAL(1+2*I,1+2*J)=A*(VAL(1+2*I,2*J)+
*             VAL(2*I,1+2*J)+VAL(1+2*I,2+2*J)+
*             VAL(2+2*I,1+2*J))+B*VAL(1+2*I,1+2*J)
        ENDDO
      ENDDO
      ...
    ENDDO
```

Parallelizing this code by 2Dblock-wise distributing all arrays yields the following parallel code under *VFCS*:

```
      PROCESSORS P(4,4)
                 DOUBLE    PRECISION    VAL(N,N)
  DIST(BLOCK,BLOCK)
      ...
  L1:  DO K = 1, TIME
        ...
  C1:    EXSR VAL(:,:) [0/1,0/0]
  C2:    EXSR VAL(:,:) [0/0,0/1]
  C3:    EXSR VAL(:,:) [1/0,0/0]
  C4:    EXSR VAL(:,:) [0/0,1/0]
         mask→ DO J=1,(N-1) / 2
             mask→ DO I=1,(N-1) / 2
               VAL(1+2*I,1+2*J)=A*(VAL(1+2*I,2*J)+
  *                VAL(2*I,1+2*J)+VAL(1+2*I,2+2*J)+
  *                VAL(2+2*I,1+2*J))+B*VAL(1+2*I,1+2*J)
           ENDDO
         ENDDO
        ...
      ENDDO
```

Note that this code is an intermediate representation as constructed by the underlying compiler to make the code more readable for the user. The loops in this code are masked, e.g. "mask → DO J=1,(N-1)/2" means that a processor p only executes those loop iterations that induce an assignment to an array element owned by p. The

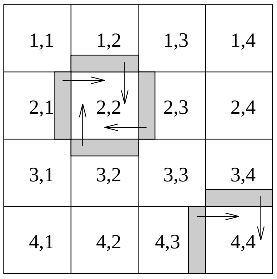


Figure 1: Block-wise distribution of array VAL

parallel red-black relaxation code contains four independent communication statements as their overlap areas do not depend on any loop variables. Although all communication is done within an outer loop L1, all processors can communicate simultaneously within L1. Note that C1, ..., C4 cannot be hoisted out of L1 due to true-dependences caused by other code portions not shown in this code. The communication pattern of this code implies a message exchange for each processor with its upper, lower, left and right neighboring processor for every iteration of L1. Figure 1 illustrates the distribution of array VAL to a 4x4 processor array P. For instance, processor P(2,2) receives four data messages from its neighbors during every iteration of L1.

The non-local array segments of VAL that are required by P(2,2) for its local computations are referred to as the *non-local segments* of P(2,2).

We now present a precise definition for a non-local segment. Let C be a communication statement which implies a set of data transfers due to a reference to a distributed array A, and $OA(A,p)$ the overlap area of a processor $p \in P^A$ with respect to A, then the set of non-local segments of p with respect to C is defined as follows:

$$\mathcal{S}(C,p) \;=\; \{K | K = OA(A,p) \cap LS(A,q) \text{ for some } q \in P^A \text{ and } p \neq q; K \neq \phi\}$$

By definition of the local segment and the overlap area for a processor p on the basis of a general block distribution (see Section 3), non-local segments are always rectilinear contiguous areas. $OA(A,p)$ is defined as the smallest rectilinear contiguous area around the local segment of a processor p, containing all non-local elements accessed.

As processor p receives for each non-local segment in $\mathcal{S}(C,p)$ a message from a remote processor, $nt^p(C)$, the number of transfers for C with respect to a processor p, is defined as

$$nt^p(C) = |\mathcal{S}(C,p)| \tag{1}$$

Assuming a loosely synchronous programming model for a communication statement C, the processor that receives the maximum number of data messages dominates the performance of C. Therefore, we define the number of transfers induced by an independent communication statement C as

$$nt(C) = \max_{p \in P^A}(nt^p(C)) \tag{2}$$

5.2 Nested Communication

Nested and innermost communication statements represent the *dependent* communication model where processors communicate not necessarily at the same time. Dependent communication is commonly caused by true dependences. Nested communication could be partially vectorized to a higher loop nest level, whereas innermost communication directly precedes its associated array reference. Whether or not a transfer is invoked (see Section 3) during execution of a program depends on the current loop iteration being executed, and the array element accessed.

In general, *nested* communication is considerably more expensive than communication which could be hoisted out to a higher loop nest level by vectorizing communication. The following code shows the main loop nest of a parallel Successive Over-Relaxation (SOR) [24] as created under *VFCS* by 2Dblock-wise distributing array U and F to a 4x4 processor array P:

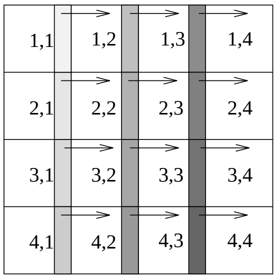


Figure 2: Communication pattern for a parallel SOR

```
        PARAMETER (N=128)
        PROCESSORS P(4,4)
        REAL U(N,N), F(N,N) DIST(BLOCK,BLOCK)
        ...
        DO K = 1, TIME
           ...
C1:        EXSR U(:,:) [0/0,0/1]
C2:        EXSR U(:,:) [0/1,0/0]
L1:        DO J=2,N-1
C3:           EXSR U(:,J-1) [0/0,1/0]
L2:              DO I=2,N-1
C4:                 EXSR U(I-1,J) [1/0,0/0]
S:                  OWNED(U(I,J))→ U(I,J) =
     *           (1-OMEGA)*U(I,J)+OMEGA*0.25*(F(I,J)+
     *              U(I-1,J)+U(I+1,J)+U(I,J+1)+U(I,J-1))
                 ENDDO
              ENDDO
           ...
        ENDDO
```

S is a masked assignment statement – implementing the owner computes rule – which guarantees that S is executed by a processor iff this processor owns U(I,J). C1 and C2 are independent communication statements. We have shown how to model such statements in Section 5.1. C4 illustrates an innermost communication statement (not vectorizable), which will be discussed in Section 5.3. The analysis in the rest of this section focuses on a model for nested communication statements such as C3 which could be partially vectorized and whose overlap area depends on loop variable J. Figure 2 shows the 2Dblock-wise distributed array U and the communication pattern induced by C3. Each processor receives the data in its overlap area from the left neighboring processor. The overlap areas for different processors are shaded with different colors.

If we map the array subscript functions (dashed lines in Figure 3) based on the overlap area for array U of a specific processor – for instance processor P(3,2) with local segment U(65:96,33:64) – into the loop iteration space – which yields an intersection polytope PO (dotted area) – then we can make the following observations:

- L1 is the only loop which has an impact on computing the number of transfers as implied by P(3,2). P(3,2) evaluates at runtime for each iteration of L1 whether or not a data element is accessed in its overlap area. If yes, then P(3,2) receives a data message containing the non-local data required from its left neighboring processor P(3,1), otherwise no transfer (receive operation) is invoked.
- By mapping the overlap area of P(3,2) into the iteration space it can be determined whether any element of the overlap area is actually accessed.
- The number of data messages received by P(3,2) can be approximated by the length of PO along the J axis in the iteration space.
- L1, the loop immediately enclosing C3 specifies the loop iteration space axis which is used to compute the length of PO. This value is at the same time an estimate for the number of transfers induced by C3 with respect to P(3,2). L1 is called the *communication loop*, and the associated axis in the loop iteration space is referred to as the *communication axis* with respect to C3.
- In order to extend the previous analysis for cases where L1 in turn is nested inside of enclosing loops we proceed as follows: For every single loop iteration of

all loops enclosing L1, the array subscript functions based on the local array segments extended by the overlap area have to be mapped into the loop iteration space of the entire loop nest. For each of these iterations we build an intersection polytope and derive the length along the communication axis. The sum of all such lengths across all iterations of loops enclosing L1 specifies $nt_e^p(C)$, an approximation for $nt^p(C)$. Obviously, this last step of the algorithm is computationally expensive. In [11, 10] we have proposed several alternatives to significantly improve this algorithm.

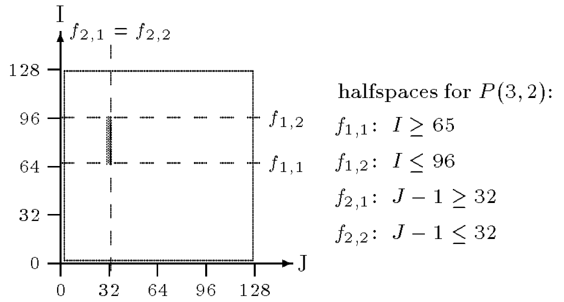


Figure 3: Loop iteration space intersection for processor $P(3,2)$

Note that the subscript expressions of U in C3 are uniquely associated with a distinct reference to U in a specific statement inside of the loop body. In this case it is $U(I, J-1)$ in statement S. The distinct association of a communication statement with an array reference in a parallel program is based on the underlying compilation model which is described in detail in [4].

Finally, we define the number of transfers for a nested communication statement C. Let array A be the reference that causes C, and P^A the set of processors to which A is distributed, then

$$nt(C) = \sum_{p \in P^A} nt^p(C) \tag{3}$$

which assumes that a nested communication sequentializes a loop and all processors communicate consecutively. This assumption is realistic as data dependences causing nested communication, frequently prevent message vectorization and parallel execution of a loop.

5.3 Innermost Communication

In general, a communication statement is most expensive if it cannot be vectorized at all due to a true dependence. Such a communication directly precedes its associate nonlocal array reference inside of a loop. Every access to a single data element in the overlap area results in a data exchange. Most compilers try to optimize innermost communication by vectorizing, fusing and aggregating communication. An exception to this objective is message pipelining [16], which – under certain conditions – initiates send operations as soon as possible (inside of loops) in order to overlap communication with computation.

Let C be an innermost communication statement which exchanges data elements of a distributed array A that is distributed onto a set of processors P^A. C results in a separate transfer for every single data element in the overlap area of A with respect to a processor $p \in P^A$. As a consequence the associated number of transfers invoked by C with respect to p is basically equivalent to its associated amount of data transferred which accounts for all innermost communication statements. The reader may, therefore, refer to Section 6 on how to compute the number of transfers for innermost communication statements.

In what follows, we describe how to extend $nt(C)$ to loop nests: Let $\mathcal{F}_L$ be the set of procedure call statements contained in a loop nest L. $proc(q)$ is the associated procedure of a procedure call $q \in \mathcal{F}_L$. $call(E)$ is the set of all associated procedure calls to a procedure E. ntE is the accumulated number of transfers for E. L contains a set of communication statements $\mathcal{C}_L$, which are partitioned into $\mathcal{C}_i$ (independent communication) and $\mathcal{C}_d$ (dependent communication), then the number of transfers induced by all $C \in \mathcal{C}_L$ can be expressed as

$$\begin{aligned} ntL(L) \;\; &= \sum_{C \in \mathcal{C}_i} nt(C) + \sum_{C \in \{\mathcal{C}_L - \mathcal{C}_i\}} \sum_{p \in P} nt^p(C) + \\ &+ \sum_{q \in \mathcal{F}_L} \frac{ntE(proc(q)) * freq(q)}{\sum_{g \in call(proc(q))} freq(g)} \end{aligned} \tag{4}$$

The first term in Eq. (4) is based on Eq. (2), and the second term on Eq. (3). The third term is explained as follows: In order to simplify matters for modeling procedure calls (Fortran subroutine and function calls), the parameter outcome of a single procedure call instantiation is supposed to be independent of the call site. In other words, the parameter outcome at a particular call site is the same as the parameter outcome of the procedure over all call sites, which is a common assumption made for performance estimators [22]. Codes (e.g. multigrid codes) that do not satisfy this assumption are likely to imply a loss in estimation accuracy. Recursions are not supported in procedure calls. The previous two assumptions are made for all communication parameters as described in this paper.

Similarly, we define the number of transfers for a procedure or an entire program. Let E be a procedure or an entire program. E contains a set of nested loops $\mathcal{L}_E$. $freq(L)$ is the execution count of $L \in \mathcal{L}_E$ as determined by *Weight Finder* (see Section 4). $\mathcal{F}_E$ is the set of procedure calls – outside of loops – in E, then the number of transfers induced by E is defined by

$$ntE(E) = \sum_{L \in \mathcal{L}_E} ntL(L) * freq(L) + \sum_{q \in \mathcal{F}_E} \frac{ntE(proc(q)) * freq(q)}{\sum\limits_{g \in call(proc(q))} freq(g)} \quad (5)$$

Note that ntL_e and ntE_e respectively denote the approximated values for ntL and ntE. ntL_e and ntE_e are obtained by respectively replacing $nt(C)$ and $nt^p(C)$ in Eq. (4) with $nt_e(C)$ and $nt_e^p(C)$, and $ntL(L)$ with $ntL_e(L)$ in Eq. (5).

6 Amount of Data Transferred

In this section, we describe how to estimate the number of data elements transferred by a communication statement of a parallel program. The current generation of DMMPs reduces the impact of the message length on the communication overhead. For applications that transmit small data volumes, the startup cost is the predominate communication cost factor. However, for increasing data volumes transmitted, the message transfer time per byte becomes the first order performance effect.

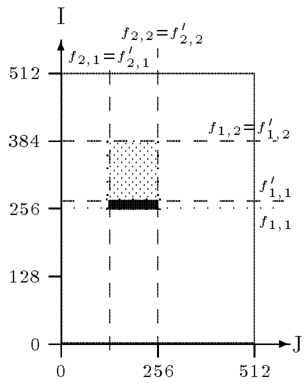
In order to model the amount of data transferred as induced by a parallel program, we consider the data distribution, array access pattern, loop iteration space and compiler communication optimizations such as message vectorization and fusion. As the compiler specifies the communication pattern at the source code level, the target architecture can be for the most part – except data type sizes – ignored. Consequently, this parameter ports easily to a large class of distributed memory architectures.

We define $td(C)$ to be the amount of data transferred by $C \in \mathcal{C}_L$ for a single instantiation of a loop L. $td^p(C)$ denotes the corresponding amount of data transferred with respect to a processor $p \in P$. $td^p(C)$ is defined to be equal to the number of data elements received during execution of C. For simplicity we do not differentiate among various types of data elements transferred such as integers and reals. The actual implementation, however, fully models different types of data messages by computing message lengths in terms of number of bytes. Data type sizes obviously depend on the target architecture but they can be properly separated from machine independent parameters.

For the sake of brevity, here we shall give a short outline of the algorithm to compute $td_e^p(C)$, an approximated value for $td^p(C)$. Details of this algorithm are given in [11, 10]. First, we map the array subscript expressions of an array A referenced in C – based on the extension segment of p – into the loop iteration space. The resulting intersection with the loop iteration space yields a geometric object *PO1*. Second, we map the array subscript functions – based on the local array segment boundaries of p – into the iteration space, which results in a geometric object *PO2*. The precise value for $td^p(C)$ is given by the number of n-dimensional integer-valued vectors in *PO1* $-$ *PO2* (*PO1* without *PO2*) which describe the loop iterations in which non-local data elements of A are accessed by p during execution of C. $td_e^p(C)$ is computed as the volume of *PO1* $-$ *PO2*. Experiments in Section 9.1 validate the quality of this approximation. Consider the following code excerpt:

```
      PARAMETER(N=512)
      PROCESSORS P(4,4)
      REAL U(N,N) DIST(BLOCK,BLOCK)
      ...
L:     DO J=2,N - 1
         DO I=2,N - 1
C4:         EXSR U(I-1,J) [1/0,0/0]
            ...
         ENDDO
      ENDDO
      ...
```

Processor $P(3,2)$ owns $U(257:384, 129:256)$. Figure 4 shows *PO2* (dotted area) and *PO1* (dotted + dark area). The dotted area displays *PO1* $-$ *PO2*, whose volume serves as an approximation for $td^p(C4)$ where $p = P(3,2)$.



halfspaces of $P(3,2)$ based on extension segment:

$f_{1,1}$: $I-1 \geq 256$

$f_{1,2}$: $I-1 \leq 384$

$f_{2,1}$: $J-1 \geq 129$

$f_{2,2}$: $J-1 \leq 256$

halfspaces of $P(3,2)$ based on local array segment:

$f'_{1,1}$: $I-1 \geq 257$

$f'_{1,2}$: $I-1 \leq 384$

$f'_{2,1}$: $J-1 \geq 129$

$f'_{2,2}$: $J-1 \leq 256$

Figure 4: Loop iterations accessing non-local data with respect to C4 for processor $P(3,2)$

As we extensively use the volume of polytopes for our performance model, we briefly outline the volume algorithm: The volume of a n-dimensional polytope is computed by a triangulation of the corresponding polytope into a set of disjoint n-dimensional convex simplices. A n-dimensional convex simplex contains exactly $n+1$ vertices. According to [21] the volume of a n-dimensional convex simplex is given by

$$\frac{1}{n!} Det(\vec{v}_2 - \vec{v}_1, \ldots, \vec{v}_{n+1} - \vec{v}_1)$$

where $\{v_1, \ldots, v_{n+1}\}$ is the set of vertices of the corresponding simplex. The sum of the volumes across all simplices of the polytope defines the volume of the polytope. For detailed mathematical analyses of this volume algorithm, the reader may refer to [11, 10].

In order to statically estimate $td(C)$, we must know whether C is an independent ($C \in \mathcal{C}_i$) or dependent communication ($C \in \mathcal{C}_d$) in accordance with the communication class model of Section 3.

$$td(C) = \begin{cases} \max\limits_{p \in P}(td^p(C)) & : \quad C \in \mathcal{C}_i \\ \sum\limits_{p \in P} td^p(C) & : \quad C \in \mathcal{C}_d \end{cases} \tag{6}$$

where $td_e(C)$, the approximate value for $td(C)$, is computed by replacing $td^p(C)$ by $td_e^p(C)$ in Eq. (6). This model assumes for independent communication statements that the processor which receives the maximum data volume dominates this communication statement. Dependent (nested and innermost) communication statements commonly sequentialize the execution of a loop. For this case, the sum of all data volumes exchanged across all processors reflects a realistic communication behavior.

The amount of data transferred as induced by a loop nest can be expressed as:

$$\begin{aligned} tdL(L) \quad = \quad & \sum_{C \in \mathcal{C}_i} \max_{p \in P} td^p(C) \; + \sum_{C \in \mathcal{C}_d} \sum_{p \in P} td^p(C) \; + \\ & + \sum_{q \in \mathcal{F}_L} \frac{tdE(proc(q)) * freq(q)}{\sum\limits_{g \in call(proc(q))} freq(g)} \end{aligned} \tag{7}$$

where $\mathcal{C}_L$ denotes the set of all communication statements contained in a loop L, and $\mathcal{F}_L$ is the set of procedure call statements contained in L. This model can be viewed as separating a loop's communication overhead into three parts: independent communication, dependent communication, and communication induced by procedure calls.

Finally, we model the amount of data transferred by all communication statements in E – a procedure or an entire program – as

$$\begin{aligned} tdE(E) \quad = \quad & \sum_{L \in \mathcal{L}_E} tdL(L) * freq(L) \; + \\ & + \sum_{q \in \mathcal{F}_E} \frac{tdE(proc(q)) * freq(q)}{\sum\limits_{g \in call(proc(q))} freq(g)} \end{aligned} \tag{8}$$

where E, $\mathcal{L}_E$, $freq(L)$, and $\mathcal{F}_E$ have the same meaning as in Eq. (5). The first term in Eq. (8) corresponds to the loops contained in E. Each loop is weighted by its execution count to reflect its impact on the overall program performance. The second term models procedure calls outside of loops. To simplify matters, we assume that the same amount of data is transferred for every instantiation of L. A more accurate modeling of tdE requires distinct values of $freq(L)$ for every instantiation of L at the price of a considerable larger computational effort to deduce the communication parameters.

In order to compute $tdL_e(C)$ and $tdE_e(E)$ – the estimated values for $tdL(C)$ and $tdE(E)$, respectively – we approximate $td^p(C)$ by $td_e^p(C)$ in Eq. (7) and $tdL(C)$ by $tdL_e(C)$ in Eq. (8).

7 Transfer Times

Estimating the number of transfers and the amount of data transferred helps the parallelizer to evaluate a parallel program with respect to communication overhead. However, an interesting problem arises when trade-offs between these two parameters need to be resolved. For instance, experiments [9] have been shown that – under certain conditions – for small message sizes the number of transfers is the first order performance effect which is caused by high startup times, whereas for larger message sizes the amount of data transferred is more important than the number of transfers due to the increasing impact of transfer times per byte. In this section we present a performance model for transfer times (tt), which is a sensible measure combining number of transfers, amount of data transferred and various machine specific indices such as message startup overhead, message transfer time per byte, sizes for different data types, and processor distances. For the transfer time parameter we assume DMMPs based on static and fixed routing schemes. Machine specific parameters are separated from the machine independent modeling aspects to enhance portability of our analysis. Network contention is not considered in our transfer times but computed as a separate performance parameter in Section 8.

An important aspect affecting the time required to exchange a message, is the number of network hops. According to [6] the network distance between the sender and receiver of a message can have a significant impact on the communication behavior of most existing DMMPs, including iPSC/860 hypercube, Intel Paragon, IBM SP-2 and Meiko CS-2. The distance (number of network hops) between two different target architecture processors $p1'$ and $p2'$ is defined by $dist(p1', p2')$ which inherently is a machine specific function. On the iPSC/860 hypercube the path to be traversed by a message is statically fixed by the underlying e-cube routing policy [5]: starting with the right

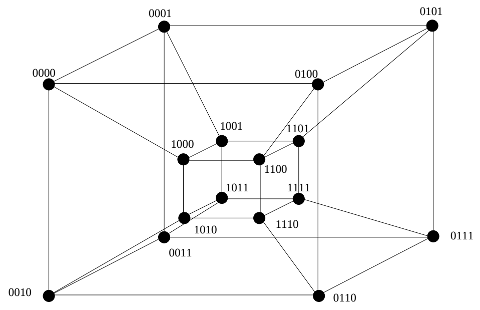


Figure 5: 4-dimensional hypercube topology

hand side of the binary label of the current processor, a message moves to the processor whose label most closely matches the label of the destination processor. Most current DMMPs are based on static and fixed routing policies. Adapting *dist* to other DMMPs should be quite straight forward.

Figure 5 illustrates a four dimensional hypercube with the binary labels of all 16 processors. If a message is sent from processor 0000 to processor 0111, then the following path – assuming e-cube routing – has to be traversed: 0000 $\rightarrow$ 0001 $\rightarrow$ 0011 $\rightarrow$ 0111 which implies 3 network hops.

The number of network hops between a sending and a receiving processor $p1'$ and $p2'$ can be easily computed by using exclusive OR operations.

The transfer time of $C \in \mathcal{C}_L$ for a single instantiation of L with respect to a processor $p \in P$ is expressed by a $tt^p(C)$. $tt(C)$ specifies the transfer time for C with respect to a single instantiation of L.

In order to compute $t(p1', p2', m)$, the required transfer time to send a single data message of length m (in bytes) from $p1'$ to $p2'$, we use the well-known linear communication model [5]:

$$t(p1', p2', m) = \alpha + \beta * m + \gamma * dist(p1', p2') \qquad (9)$$

where α is the message startup time, β the transfer time per message byte, and γ the overhead for each network hop. For all our experiments conducted on the iPSC/860 hypercube, we used characteristic values for α, β, and γ - which depend on the message length – given in [5].

We incorporate two different algorithms in order to approximate $tt(C)$: one for independent communication and another for dependent communication.

7.1 Independent Communication

In the following, we describe how to derive $tt_e(C)$, an estimate for $tt(C)$, where C is an independent communication statement and A is the distributed array referenced by C:

1. Find processor $q \in P^A$ with the maximum $|\mathcal{S}(C,q)|$.

2. Classify the non-local segments in $\mathcal{S}(C,q)$ with respect to the number of network hops which yields $\mathcal{S}(C,q) = \{H_1, \ldots, H_k\}$. H_i $(1 \le i \le k)$ is the set of non-local segments in $\mathcal{S}(C,q)$ that require i network hops to be sent to q.

3. $tt_e(C)$ is computed as:

$$tt_e(C) = nt^q(C) * \sum_{H_i \in \mathcal{S}(C,q)} (\alpha + \beta * \frac{td^q(C)}{nt^q(C)} + \gamma * i) * \frac{|H_i|}{nbs} \quad (10)$$

where $td^q(C)$ is the amount of data transferred and $nt^q(C)$ the number of transfers induced by q with respect to C, and

$$nbs = \sum_{H_i \in \mathcal{S}(C,q)} |H_i|$$

Let R be the set of data messages received by q with respect to C. $snd(r)$ and q are, respectively, the sending and receiving processor of a data message $r \in R$ with length $len(r)$ (in bytes), then the precise transfer time for C, is defined as

$$tt(C) = \sum_{r \in R} t(snd(r), q, len(r)) \qquad (11)$$

which assumes that processor q dominates the communication behavior of C, and all its transfers occur consecutively in accordance with the communication class model for independent communication (see Section 3).

7.2 Dependent Communication

We now derive $tt_e(C)$, an estimate for $tt(C)$, where C is a dependent communication statement and A the distributed array referenced by C:

1. Classify the actually accessed non-local segments of C over all $p \in P^A$ with respect to the number of network hops which yields $H = \{H_1, \ldots, H_u\}$. H_i $(1 \le i \le u)$ is the set of non-local segments that require i network hops for the associated message to be exchanged.

2. $tt_e(C)$ is computed as:

$$tt_e(C) = \frac{nt(C)}{nbs} * \sum_{H_i \in H} (\alpha + \beta * \frac{td(C)}{nt(C)} + \gamma * i) * |H_i| \quad (12)$$

nbs is defined as:

$$nbs = \sum_{H_i \in H} |H_i|$$

Let R be the set of data messages received by C with respect to all $p \in P^A$. $snd(r)$ and $rcv(r)$ are, respectively, the sending and receiving processor of a data message $r \in R$ with length $len(r)$ (in bytes). Then the precise transfer time for C is given by

$$tt(C) = \sum_{r \in R} t(snd(r), rcv(r), len(r)) \qquad (13)$$

which assumes that all transfers occur consecutively in accordance with the communication class model for dependent communication.

In the following we define a single accumulated transfer time, which models all communication statements in a loop, a procedure and a parallel program, respectively. Let $\mathcal{C}_L$ denote the set of all communication statements contained in a loop L, and $\mathcal{F}_L$ the set of procedure call statements in L. Then the transfer time induced by all communication statements in $\mathcal{C}_L$ can be expressed as

$$ttL(L) = \sum_{C \in \mathcal{C}_L} tt(C) + \sum_{v \in \mathcal{F}_L} \frac{ttE(proc(v)) * freq(v)}{\sum\limits_{g \in call(proc(v))} freq(g)} \qquad (14)$$

The first term in Eq. (14) corresponds to the communication statements in L, and the second term reflects the communication cost of the procedure calls inside of L.

If E is a procedure or an entire program, then the transfer time implied by all communication statements in E is given by

$$\begin{aligned} ttE(L) &= \sum_{L \in \mathcal{L}_E} ttL(L) * freq(L) + \\ &+ \sum_{v \in \mathcal{F}_E} \frac{ttE(proc(v)) * freq(v)}{\sum\limits_{g \in call(proc(v))} freq(g)} \qquad (15) \end{aligned}$$

The first term corresponds to the communication overhead induced by loops in E, and the second term models the communication costs of procedure calls – outside of loops – in E.

The estimated values for $ttL(L)$ and $ttE(E)$ are referred to as $ttL_e(L)$ and $ttE_e(E)$, respectively. These values are computed by substituting $tt_e(C)$ for $tt(C)$ in Eq. (14) and $ttL_e(L)$ for $ttL(L)$ in Eq. (15).

8 Network Contention

It is well known [5, 20] that network contention, which refers to the sharing of a communication channel by two or more transfers, can severely degrade all network traffic. For example, Bokhari [5, 6] reports that on the iPSC/860 hypercube and the Intel Paragon channel contention can increase the time to communicate by several 100 %. Bokhari also presents experiments [6] about the considerable sensitivity for network contention of the Meiko CS-2 and IBM SP-2. Most existing DMMPs do not exhibit a measurable impact on communication caused by simultaneous sending of two messages on the same channel in opposite direction. Furthermore, they seem insensitive regarding node contention, the sharing of a node by two or more transfer paths.

A programmer requires thorough knowledge of the underlying architecture and the parallelizing compiler in order to evaluate the network contention of a parallel program. Hence, providing the programmer with information on the network contention behavior of a parallel program can be very useful. In this section, we present a novel approach to estimate network contention of parallel programs for DMMPs at compile time. In order to overcome expensive simulation approaches we will compute upper bounds for the number of channel contentions based on a classification of communication statements. In general our network contention parameter can be applied to DMMPs with circuit switched static and fixed routing policies. For such architectures the exact network traversal path for each transfer can be statically deduced. For the sake of demonstrating our approach, we use the e-cube routing policy (see Section 7) for most examples in this section.

We assume that network contention occurs, iff two or more transfers occupy at least one specific communication channel at the same time in the same direction (*channel contention*), which is a realistic assumption for many DMMPs such as the iPSC/860 hypercube [5].

The *traversal direction* of a channel can be easily defined by the binary representation of processors adjacent to a channel. For example, traversing a channel from the processor with the smaller binary label to the one with higher binary label can be defined as the *positive traversal direction*; the opposite direction as the negative one.

8.1 Network Contention Algorithm

We define network contention induced by $C \in \mathcal{C}_L$ for a single instantiation of L by $nc(C)$ which is the number of channel contentions incurred by C. If n ($n \geq 1$) transfers use the same channel at the same time in the same direction, then $nc(C)$ is incremented by $n-1$.

The following describes how to derive $nc_e(C)$, an estimated value for $nc(C)$, where A is the distributed array referenced by C; $ct^+(c)$ and $ct^-(c)$, the number of transfers respectively traversing a channel c of the target architecture in the positive and negative channel direction. The transfers induced for every $p \in P^A$ with respect to C are supposed to be evenly distributed across all non-local segments of p in $\mathcal{S}(C,p)$.

1. Initialize NC, a network contention counter, and $ct^+(c)$, $ct^-(c)$ for all channels c of the target architecture with zero

2. For every $p \in P^A$ do the following:

   (a) Compute $\mathcal{S}(C,p)$, the set of all non-local segments of p, $nt^p(C)$, the number of transfers for p induced by C, and $w = nt^p(C)/\mathcal{S}(C,p)$.

   (b) For each $h \in \mathcal{S}(C,p)$ – which refers to a data portion to be received by p – find the owning processor p' (responsible to send h to p), and derive the exact message transfer path between p' and p based on circuit switched static and fixed routing. For each network channel c traversed, the traversal direction is evaluated; For a positive channel traversal direction: if $ct^+(c) \equiv 0$ (channel is free) then $ct^+(c) := w$ (w messages are supposed to traverse c consecutively in the positive direction) without changing NC; otherwise c is assumed to be already occupied by other transfers. Therefore, NC is incremented by $\min(w, ct^+(c))$ – number of channel conflicts – and $ct^+(c) := |ct^+(c) - w|$ in this order; For a negative channel traversal direction this step is done for $ct^-(c)$ only;

3. if C is a dependent communication statement then NC is multiplied by a probability value φ ($0 \leq \varphi \leq 1$)

The case where two transfers imply channel contention by traversing a channel in *opposite* direction at the same time, can be easily modeled by omitting the phase for determining the traversal direction and ct^- in the previous algorithm. Every time a specific channel is traversed and already occupied, NC is incremented accordingly without considering the traversal direction.

For independent communication statements the loosely synchronous programming model is assumed. The invocation of a dependent communication (inside of loops) is controlled by the current loop iteration, data distribution and array element accessed. Hence, dependent communication frequently occurs consecutively separated by computation, and consequently, network contention is less likely to occur as compared to independent communication. This behavior is modeled by multiplying NC with a probability factor φ for dependent communication statements. $nc_e(C)$ is defined by the value of NC at the end of the previous algorithm.

The following describes how to extend the above algorithm for a loop nest L containing communication. Let $\mathcal{F}_L$ be the set of procedure call statements contained in L. $proc(q)$ is the associated procedure of a procedure call $q \in \mathcal{F}_L$. $call(E)$ is the set of all associated procedure calls to a procedure E. $ncE(E)$ refers to the accumulated network contention implied by E.

1. for every independent C proceed as follows:

   (a) same as item 2 in the network contention algorithm for a single communication statement, replacing NC by NC1.

2. initialize $NC2$, a network contention counter with zero

3. for every dependent communication statement C proceed as follows:

   (a) same as item 2 in the network contention algorithm for a single communication statement, replacing NC by NC2.

4. $NC2$ is multiplied by φ

Based on $NC1$ and $NC2$ of the previous algorithm, we can express $ncL_e(L)$, an approximated value for $ncL(L)$, as

$$ncL_e(L) = NC1 + NC2 + \sum_{q \in \mathcal{F}_L} \frac{ncE(proc(q)) * freq(q)}{\sum_{g \in call(proc(q))} freq(g)} \quad (16)$$

In [11, 10] we describe how to experimentally determine a realistic value for φ once for every different target architecture of interest.

In what follows, we show how to extend the network contention parameter to procedures and an entire program. Let E be a procedure or an entire program, then the network contention induced by all communication statements in E is defined as

$$\begin{aligned} ncE(L) &= \sum_{L \in \mathcal{L}_E} ncL(L) * freq(L) \; + \\ &+ \sum_{q \in \mathcal{F}_E} \frac{ncE(proc(q)) * freq(q)}{\sum_{g \in call(proc(q))} freq(g)} \end{aligned} \quad (17)$$

The estimated values for $ncL(L)$ and $ncE(E)$ are referred to as $ncL_e(L)$ and $ncE_e(E)$, respectively. $ncE_e(E)$ is computed by substituting $ncL_e(L)$ for $ncL(L)$ in Eq. (17).

Note that the estimated number of channel contentions is an upper bound because time constraints are not considered in our analysis. If two transfers ever occupy the same channel in the same direction then they are assumed to induce channel contention. This assumption ignores the fact that channel contention only occurs, if both transfers traverse the same channel at the same time.

In the appendix, we present a necessary condition for the existence of a channel contention between two different transfers on a DMMP assuming e-cube routing, and a proof about the absence of network contention for an important class of communication statements.

8.2 Experiments

We have not yet found a reasonable way to measure the exact number of channel contentions for real programs on our experimental system, the iPSC/860 hypercube. For this reason we tried to validate the estimated network contention by transfer time measurements which correlate with the actual network contention behavior.

In order to demonstrate the impact of network contention on a parallel program we used *VFCS* to compile the following synthetic code:

```
      PARAMETER(N=1600)
      PROCESSORS P(16)
      DOUBLE PRECISION Z(N,N),Y(N,N),X(N,N) DIST(:,BLOCK)
C1:  EXSR Z(:,1001) [0/0,0/401]
L1:  mask → DO 10 I=1,N
        OWNED(Z(I,501))→ Z(I,501) = Z(I,501) + Z(I,1001)
10   CONTINUE

C3:  EXSR Y(:,1101) [0/0,0/101]
C1:  EXSR Z(:,1001) [0/0,0/401]
L2:  mask → DO 11 I=1,N
        OWNED(Z(I,501))→ Z(I,501) = Z(I,501) + Z(I,1001)
        OWNED(Y(I,901))→ Y(I,901) = Y(I,901) + Y(I,1101)
11   CONTINUE

C2:  EXSR X(:,1101) [0/0,200/0]
C1:  EXSR Z(:,1001) [0/0,0/401]
L3:  mask → DO 12 I=1,N
        OWNED(Z(I,501))→ Z(I,501) = Z(I,501) + Z(I,1001)
        OWNED(X(I,1301))→ X(I,1301) = X(I,1301) + X(I,1101)
12   CONTINUE
```

All arrays are distributed column-wise to 16 processors. *VFCS* applies message vectorization but not yet message aggregation for communication statements that refer to different arrays. For this code a compiler can easily determine the non-local segments required for each processor including the remote processors owning these segments without using overlap areas. Buffer communication (see Section 3) is used to implement this code. The network contention parameter has been implemented primarily for communication based on overlap areas. This parameter can handle buffer communication if non-local segments and their ownership can be determined at compile time, which applies for the code shown above. Note that the network contention parameter as well as all other communication parameters can perfectly handle this code based on overlap areas which implies both neighboring and non-neighboring communication.

The code contains three loops – $L1$, $L2$, and $L3$ – and each loop induces a subset of communication statements in $\{C1, C2, C3\}$. The communication pattern – traversal path – of every communication in $\{C1, C2, C3\}$ is the same, independent of its location in the code. The following describes the traversal path for each different communication statement:

$C1:\quad t1: 10 \rightarrow 11 \rightarrow 9 \rightarrow 13 \rightarrow 5$

$C2:\quad t2:\quad 11 \rightarrow 9 \rightarrow 13$

$C3:\quad t3:\quad 11 \rightarrow 9$

Figure 6 plots the transfer time ttL (in μsecs) – measured on the iPSC/860 hypercube – for all three loops for various values of message length. Note that ttL is not the predicted but the actually measured transfer time. The message length can be changed by varying the problem size N. Loop $L1$ (solid line) implies only one communication statement – $C1$ – which obviously does not induce any network contention as it implies only a single communication between processor 10 and 5. As expected, the communication time increases for larger message lengths.

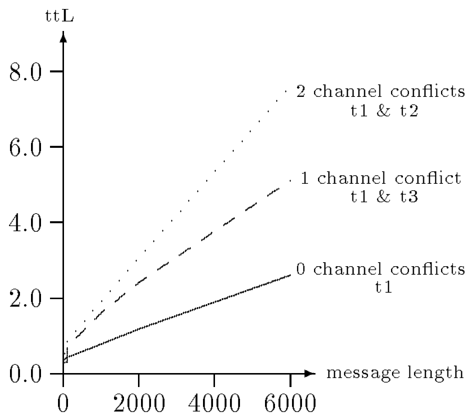


Figure 6: Channel conflicts and transfer times for various message lengths

Loop $L2$ (dashed line) implies two communication statements, $C1$ and $C3$, both of which traverse the channel $11 \rightarrow 9$, thus inducing a single channel contention. Even though both transfers $t1$ and $t3$ occur simultaneously and have different sending and receiving processors, they do not overlap. The associated communication time for this experiment doubles which is caused by the contention at channel $11 \rightarrow 9$.

If we consider the communication pattern of loop $L3$, then we notice that both $t1$ and $t2$ use channels $11 \rightarrow 9$ and $9 \rightarrow 13$. These two channel conflicts have a clear impact on the communication time of $C1$ and $C2$ as shown by the timings given in Figure 6. We observe that for each channel conflict the communication time increases by a certain constant factor. By using our network contention cost model, we can precisely estimate the number of channel conflicts for every loop in code shown above.

9 Experiments

In this section, we present two experiments. We begin by validating the estimation accuracy of our communication cost functions for 7 different small sized codes for various problem sizes and processor numbers. Then we show how these cost functions can be used to support the performance tuning of an ADI code excerpt under *VFCS*. For all experiments we used *VFCS* Version 1.1. to generate parallel program versions by applying a variety of program transformations and data distribution strategies, P^3T to statically compute the communication cost functions, and the iPSC/860 hypercube to execute the parallel programs and measure their corresponding runtimes for validation of our predicted results. The iPSC time measurements were obtained on a 16 node Intel iPSC/860 hypercube. It is running under release 3.3.1 of the Intel software. Time measurements were made using dclock(), a microsecond timer.

9.1 Accuracy of Communication Cost Functions

In this section, we evaluate the estimation accuracy of the following three cost functions: number of transfers (ntE), amount of data transferred (tdE), and transfer time (ttE). In what follows, we describe the codes that have been selected for this experiment. For each code we indicate the number of source code lines (excluding comments), loop nests, procedure calls (only if included), optimization strategies applied, and communication type (nearest neighbor, multicast, or broadcast) induced. The term basic optimization refers to a sequence of transformations: elimination of redundant loop iterations [4], communication vectorization, fusion, and aggregation.

1. *LIVERMORE 11* [19] is a partial sums code that comprises 11 lines of code, and 1 loop nest. Basic optimization has been applied which results in nearest neighbor communication.

2. *LIVERMORE 23* [19] is a 2D implicit hydrodynamics fragment code that comprises 18 lines of code, and 1 loop nest. Basic optimization has been applied which results in nearest neighbor communication.

3. RED-BLACK [24] is an implementation of a pointwise relaxation using the red-black checkerboard algorithm. This code comprises 42 lines of code and 5 loop nests. Basic optimization has been applied which results in nearest neighbor communication.

4. JACOBI [24] is a code which implements the Jacobi relaxation iterative method to approximate the solution of a partial differential equation discretized on a grid. This code comprises 98 lines of code, 7 loop nests, and 4 procedures. Procedure calls occur both inside and outside of loop nests. Basic optimization has been applied which induces nearest neighbor communication.

5. *ADI* (alternating direction implicit) [24] is a well known and effective method for solving partial differential equations in two or more dimensions. This code (main part of ADI) comprises 51 lines of code and 4 loop nests. Basic optimization and loop distribution but no scalar forward substitution has been applied, which implies broadcast, multicast and nearest neighbor communication.

6. *EFLUX* is a core routine of the FLO52 program of the Perfect Benchmarks [7], which analyzes the transonic flow past an airfoil by finding a solution of the unsteady Euler equations. This code comprises 81 lines of code and 7 loop nests. Only basic optimization but no scalar forward substitution has been applied, which implies broadcast and nearest neighbor communication.

7. *SHALLOW* is a weather prediction code that uses finite-difference methods to solve a system of shallow-water equations. This code has been written by Paul Swarztrauber at the National Center for Atmospheric Research, Boulder, Colorado, and comprises 150 lines of code and 10 loop nests. Basic optimization, loop interchange, distribution and inter loop communication fusion has been implied which implies multicast and nearest neighbor communication.

We deliberately did not fully optimize all codes under *VFCS* in order to examine the estimation accuracy of our analysis for both optimized and non-optimized codes and to analyze various types of communication including nearest neighbor, multicast and broadcast communication.

Table I presents the results for several machine and problem sizes. P indicates the set of processors. N describes the total problem size and its dimensionality. $N/|P|$ yields the problem size on an individual processor.

ntE, ttE, and tdE correspond to the actually measured performance results with respect to the entire program as taken on the iPSC/860 hypercube, while ntE_e, ttE_e, and tdE_e are the associated estimates.

The estimation accuracy of a parallel program parameter $\psi \in \{ntE, tdE, ttE\}$ is expressed by the relative difference ϵ which is defined as follows:

$$\epsilon = \begin{cases} \left|\frac{\psi - \psi_e}{\psi}\right| & : \text{ if } \psi > 0 \\ |\psi - \psi_e| & : \text{ if } \psi = 0 \end{cases}$$

TABLE I
Measured and Estimated Communication Costs

Kernel	$\|P\|$	N	ntE	ntE_e	ϵ	tdE	tdE_e	ϵ	ttE	ttE_e	ϵ
LIVERMORE 11	4	64	3	3	0.0	23	24	0.0434	0.00023	0.00024	0.0434
		128	3	3	0.0	23	24	0.0434	0.00023	0.00024	0.0434
		256	3	3	0.0	23	24	0.0434	0.00023	0.00024	0.0434
	8	64	7	7	0.0	54	56	0.0370	0.00057	0.00058	0.0175
		128	7	7	0.0	54	56	0.0370	0.00057	0.00058	0.0175
		256	7	7	0.0	54	56	0.0370	0.00057	0.00058	0.0175
	16	64	15	15	0.0	118	120	0.0169	0.00127	0.00128	0.0078
		128	15	15	0.0	118	120	0.0169	0.00127	0.00128	0.0078
		256	15	15	0.0	118	120	0.0169	0.00127	0.00128	0.0078
LIVERMORE 23	4	64	178	187	0.0505	1968	2031	0.0320	0.0150	0.0156	0.0400
		128	367	379	0.0326	3994	4079	0.0212	0.0307	0.0315	0.0260
		256	753	763	0.0132	8119	8174	0.0067	0.0623	0.0633	0.0160
	8	64	421	435	0.0332	3969	4064	0.0239	0.03607	0.0370	0.0258
		128	861	883	0.0255	8022	8160	0.0172	0.07325	0.0749	0.0225
		256	1763	1779	0.0090	16313	16351	0.0023	0.1496	0.1506	0.0066
	16	64	905	931	0.0287	7978	8130	0.0190	0.07845	0.0803	0.0235
		128	1865	1891	0.0139	16251	16320	0.0042	0.1614	0.1628	0.0086
		256	3784	3811	0.0071	32687	32704	0.0005	0.3261	0.3278	0.0052
RED-BLACK	4	64x64	88	88	0.0	10833	11072	0.0220	0.24631	0.2488	0.0102
		128x128	88	88	0.0	22106	22328	0.0100	0.2946	0.2971	0.0084
		256x256	88	88	0.0	44630	44854	0.0051	0.3934	0.3938	0.0010
	8	64x64	88	88	0.0	10748	11092	0.0321	0.27227	0.2784	0.0220
		128x128	88	88	0.0	22108	22352	0.0111	0.32369	0.3267	0.0092
		256x256	88	88	0.0	44674	44876	0.0045	0.42255	0.4234	0.0020
	16	64x64	88	88	0.0	16019	16276	0.0160	0.32755	0.3302	0.0080
		128x128	88	88	0.0	22080	22356	0.0125	0.3539	0.3563	0.0067
		256x256	88	88	0.0	44667	44882	0.0048	0.4522	0.4529	0.0014
JACOBI	4	64x64	75	75	0.0	136116	138975	0.0210	0.74298	0.7543	0.0150
		128x128	75	75	0.0	539274	548442	0.0170	2.48381	2.5089	0.0101
		256x256	75	75	0.0	2176425	2178384	0.0009	9.49455	9.4974	0.0003
	8	64x64	119	119	0.0	158126	160973	0.0180	0.9522	0.9628	0.0111
		128x128	119	119	0.0	631789	637476	0.0091	2.9899	3.0050	0.0051
		256x256	119	119	0.0	2535391	2536664	0.0005	11.1474	11.1486	0.0001
	16	64x64	207	207	0.0	171683	171975	0.0170	1.22993	1.2396	0.0078
		128x128	207	207	0.0	677938	682006	0.0060	3.41874	3.4256	0.0021
		256x256	207	207	0.0	2715555	2715827	0.0001	12.1455	12.1468	0.0001
ADI	4	64x64	274	285	0.0401	144651	147262	0.0180	0.1066	0.1096	0.0281
		128x128	529	541	0.0226	584178	589436	0.0090	0.3252	0.3312	0.0184
		256x256	1037	1053	0.0154	2353107	2358520	0.0023	1.10679	1.1173	0.0094
	8	64x64	430	443	0.0302	282228	285051	0.0100	0.18096	0.1967	0.0869
		128x128	814	827	0.0159	1137159	1143528	0.0056	0.59852	0.6095	0.0183
		256x256	1579	1595	0.0101	4571198	4580798	0.0021	2.0929	2.1039	0.0052
	16	64x64	617	635	0.0291	543003	546696	0.0068	0.3362	0.3396	0.0101
		128x128	1122	1147	0.0222	2194064	2199330	0.0024	1.0873	1.0961	0.0081
		256x256	2159	2171	0.0055	8815301	8822354	0.0008	3.8958	3.8997	0.0010
EFLUX	4	64x64	124	124	0.0	664380	679661	0.0230	0.2807	0.2865	0.0206
		128x128	124	124	0.0	2733706	2768151	0.0126	1.0926	1.1026	0.0058
		256x256	124	124	0.0	11104465	11172203	0.0061	4.36837	4.3802	0.0027
	8	64x64	232	232	0.0	834510	848697	0.0170	0.37107	0.3752	0.0111
		128x128	232	232	0.0	3416102	3450264	0.0100	1.37977	1.3909	0.0081
		256x256	232	232	0.0	13853092	13912661	0.0043	5.4555	5.4720	0.0030
	16	64x64	448	448	0.0	923922	933254	0.0101	0.44751	0.4509	0.0075
		128x128	448	448	0.0	3772529	3791392	0.0050	1.56955	1.5727	0.0020
		256x256	448	448	0.0	15281509	15283038	0.0001	6.05389	6.0545	0.0001
SHALLOW	4	64x64	15	15	0.0	3694	3820	0.0341	0.0044	0.0045	0.0227
		128x128	15	15	0.0	7458	7660	0.0270	0.00589	0.0060	0.0186
		256x256	15	15	0.0	15128	15340	0.0140	0.00889	0.0090	0.0123
	8	64x64	15	15	0.0	3699	3822	0.0330	0.0047	0.0048	0.0212
		128x128	15	15	0.0	7467	7662	0.0260	0.00619	0.0063	0.0177
		256x256	15	15	0.0	15130	15342	0.0140	0.00919	0.0093	0.0110
	16	64x64	15	15	0.0	3702	3825	0.0330	0.0051	0.0052	0.0196
		128x128	15	15	0.0	7468	7663	0.0260	0.00659	0.0067	0.0166
		256x256	15	15	0.0	15145	15342	0.0130	0.0096	0.0097	0.0104

For all the codes which contain only independent communication, we achieve 100 % estimation accuracy (ϵ = 0.0) with respect to nt. ADI and LIVERMORE 23 also contain nested and innermost communication which induce a slight estimation inaccuracy due to the approximation of nt by the volume of a polytope. All other results of the measurements are within 5 % of the predicted results. For td and tt we are respectively off by 4 % and 8 % in the worst case.

The loop bounds never change in all 4 procedures of the JACOBI program, which implies that the communication behavior at a particular call site is the same as the communication behavior of the procedure over all call sites. As our interprocedural communication cost model is based on a similar assumption (see Section 5.3), the estimation error caused by procedure calls in the JACOBI program is negligible. Other codes, such as multigrid codes, commonly pass loop bounds as formal procedure parameters to procedures. In the latter case loop bounds may drastically change across different procedure calls and consequently may imply high estimation errors.

A main observation for the majority of experiments conducted is that the estimates consistently improve for increasing problem sizes.

9.2 Usefulness of Communication Cost Functions

In this section, we show an experiment of the ADI code in order to demonstrate the usefulness of the communication cost functions to support the programmer and compiler in finding performance efficient program transformations and in selecting profitable data distribution strategies. We will validate our estimates by actually measuring the runtime of various ADI program versions on the target architecture. The corresponding code represents the main part of an ADI variation as shown in [24]:

```
      PROGRAM ADI
      PARAMETER(N=512)
      REAL U(N,N),A(N,N),B(N,N)
      REAL C(N,N),R(N,N),GAM(N,N)
      INTEGER I,J
      REAL BET

C:    INITIALIZATION OF ARRAYS NOT SHOWN

L1:   DO 101 J=1,N
C:       TRIDIAG COLUMN-WISE:
         BET=B(1,J)
         U(1,J)=R(1,J)/BET
         GAM(2,J)=C(1,J)/BET
         DO 111 I=2,N
            BET=B(I,J)-A(I,J)*GAM(I,J)
            U(I,J)=(R(I,J)-A(I,J)*U(I-1,J))/BET
            IF(I.LT.N) GAM(I+1,J)=C(I,J)/BET
111      CONTINUE
         DO 112 I=N-1,1,-1
            U(I,J)=U(I,J)-GAM(I+1,J)+U(I+1,J)
112      CONTINUE
101   CONTINUE

L2:   DO 201 I=1,N
C:       TRIDIAG ROW-WISE:
         BET=B (I,1)
         U(I,1)=R(I,1)/BET
         GAM(I,2)=C(I,1)/BET
         DO 211 J=2,N
            BET=B(I,J)-A(I,J)*GAM (I,J)
            U(I,J)=(R (I,J)-A(I,J)*U(I,J-1))/BET
            IF(J.LT.N) GAM(I,J+1)=C (I,J)/BET
211      CONTINUE
         DO 212 J=N-1,1,-1
            U(I,J)=U(I,J)-GAM(I,J+1)+U(I,J+1)
212      CONTINUE
201   CONTINUE
      END
```

9.2.1 Selecting a Data Distribution Strategy

The runtime of the original sequential ADI code – without initialization part – for a specific problem size N=512 was measured to be 9.459 seconds on a single node of the iPSC/860 hypercube. The original code is parallelized by applying three different data distribution strategies:

v1: Distribute the first dimension of all arrays onto 16 processors which corresponds to a row-wise distribution.

v2: Distribute the arrays 2-dimensional block-wise onto a 4x4 processor array.

v3: Distribute the second dimension of all arrays onto 16 processors which corresponds to a column-wise distribution.

Figure 7 displays the three program versions – one for each different data distribution – with their associated measured runtime mt (indicated in seconds) and the predicted parallel program parameters. Note that mt is not an estimate but reflects the actually measured runtime as taken on the iPSC/860 hypercube. According to the parameter priorities for the iPSC/860 hypercube as given in Section 4 – transfer time (tt) parameter has highest priority on the iPSC/860 hypercube – we deduce that program version $v1$ has the best performance outcome of all three different data distributions. The same ranking is determined by the corresponding measured runtimes. Note that the estimated transfer times are larger than the associated measured runtimes which is due to the fact that P^3T sums up the transfer times for all communication statements inside of loops. This parameter does not account for potential communication overlapping with respect to communication statements inside of loop nests across all processors. Note also that the transfer time is about two orders of magnitude larger than the computation time ct (Section 4) which is another performance parameter of

P^3T. This difference between transfer and computation time clearly indicates that $v1$, $v2$, and $v3$ imply a significant communication overhead. There are two reasons for that. First, most of the communication cannot be hoisted out of loops L1 and L2 (see ADI code) due to data dependences. Second, scalar assignments, which have to be executed by all processors, require the exchange of all local array segments across all processors.

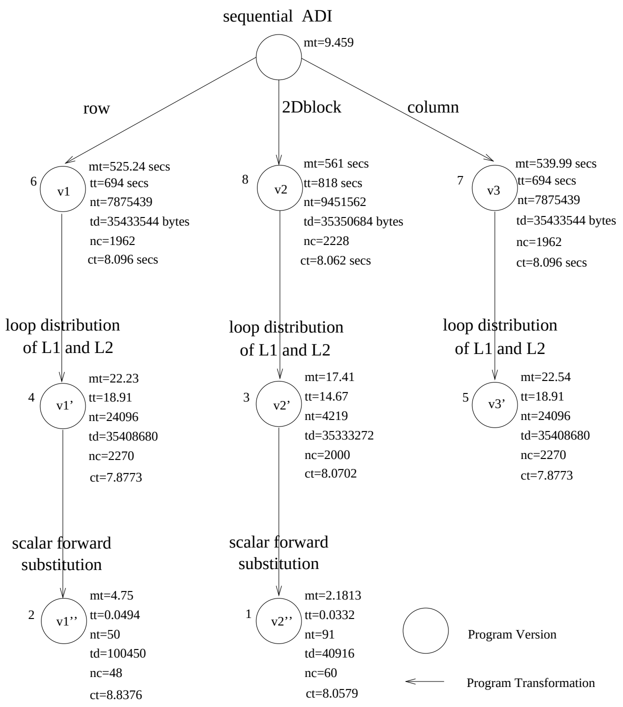


Figure 7: Performance tuning of the ADI program

9.2.2 Loop distribution of loop L1 and L2

In order to gain performance, L1 and L2 are distributed such that only perfectly nested loops are contained in the new program version. *VFCS* provides an interactive feature for selective loop distribution.

This transformation reduces the amount of communication drastically as indicated by the corresponding transfer time figures of $v1'$, $v2'$ and $v3'$. The resulting improvement in performance is clearly confirmed by the associated measured runtimes mt. The similarity of both row-wise and column-wise distribution regarding all performance parameters is due to the opposite performance behavior of loop L1 and L2. Column-wise distribution parallelizes loop L1 and sequentializes L2, while row-wise distribution parallelizes L2 and sequentializes L1. For this reason only row-wise and 2Dblock-wise distribution are compared for all subsequent optimizations. Note that after loop distribution the 2D block-wise distribution outperforms the row-wise distribution, which is correctly reported by the transfer time parameter.

9.2.3 Scalar forward substitution

In order to reduce redundant computation in the ADI code, scalars are incorporated as temporary variables to store common subexpressions. These scalar variables are used as part of the right-hand side of array assignment statements. The underlying compilation model requires all processors to execute all instantiations of scalar assignment statements. As the right hand-side of the scalar assignment statements contain references to distributed arrays, all processors have to exchange their corresponding local array segments. The consequence is a significant communication overhead. In order to prevent this fall-off in performance, *VFCS* provides the programmer with an important program transformation: *Scalar Forward Substitution*. This transformation replaces – under appropriate conditions [4] – an applied occurrence of a scalar variable by the expression on the right-hand side of an assignment to that variable.

After applying scalar forward substitution (see ADI code – assignments to variable BET) an approximate speed-up of 4 and 2 for $v1''$ and $v2''$ can be respectively achieved. P^3T confirms this performance improvement by a vast reduction of the associated transfer time. P^3T would choose $v2''$ to be the best program version created so far due to its superior performance behavior with respect to transfer time. This program version uses 16 processors and achieves an approximate speed up of 4.33 with respect to the runtime of the original sequential ADI code. The integers (from 1 to 8) at the left-hand side of each program version node in Figure 7 indicate the ranking of all parallel program versions as analyzed by P^3T which is determined based on the parameter priorities. $v1$ is superior to $v3$ and $v1'$ superior to $v3'$ due to a better cache performance which is a separate parameter of P^3T (see Section 4) not shown in Figure 7.

Note that the computation time parameter reflects only sequential computation overhead. Communication and blocking time due to message synchronization is not modeled by this parameter. It accounts for the original computation contained in the input program plus the extra overhead implied by parallelizing the code; for instance, more complex loop bounds, statement masks, and array subscript functions. The computation time parameter should, therefore, be compared to mt of the sequential ADI version as opposed to the mt value of the associated parallel ADI version. In addition, this parameter is sensitive to the improved cache behavior due to the application of loop distribution. The improved cache behavior of $v2''$ is a potential reason why the computation time of $v2''$ is slightly better than the actual runtime of the original ADI code.

10 Conclusion

In this paper we described a performance estimator which provides compile-time estimates of four critical communication characteristics: number of transfers, amount of data transferred, transfer time, and network contention. We have demonstrated the accuracy and effectiveness of our cost functions for a variety of small sized regular scientific codes. We believe that our major contributions to the problem of automatic communication prediction are:

- Reasonable large class of regular codes: Our approach is quite general and handles programs containing procedure calls, loop bounds and array subscript functions as linear functions of all enclosing loop variables, communication inside and outside of loops which covers both loosely synchronous and asynchronous programming model.
- High estimation accuracy: The estimations are very close to the actual communication costs due to carefully modeling parallel program (loop iteration spaces, array subscript functions, etc.), parallelizing compiler (data distributions, communication statements, etc.) and target machine (message startup overhead, message transfer time per byte, sizes for different data types, routing strategy, etc.). It is assumed that problem size and machine parameters including number of processors, are known at compile time. A particular interesting effect of our model reveals that its accuracy consistently improves for increasing problem sizes.
- Profiling for program unknowns: In order to handle larger classes of programs and improve estimation accuracy, we apply a single profile run of the original sequential program to determine characteristic values for branching probabilities, statement execution counts and unknown parameters in loop bounds. The profile data is automatically adapted for code transformations, changing number of processors and data distributions. The author believes that if conventional techniques such as distribution functions and guessing would be used instead of profiling both the estimation accuracy as well as the class of regular programs – that can be handled by the described estimation model – would be reduced considerably.
- Support parallelization process: Experiments have shown that our communication cost functions can successfully support the programmer in the effort of performance tuning small sized programs which includes the selection of data distributions and program transformations.

The analytical cost models as described in this paper have been fully implemented under P^3T, which is a performance estimator integrated in a parallelizing compiler (*VFCS*).

Ongoing work extends the communication cost models with symbolic analysis [12] in order to determine communication cost functions that are parameterized with problem and machine sizes. We are currently also working on extending P^3T for reduction operations. Modeling array redistribution, cyclic distributions, and other work distribution concepts such as those associated with FORALL loops in Vienna Fortran will be addressed in future work. We believe that estimating the communication costs of data parallel codes at compile-time as presented in this paper is an important step towards automatic compilation for distributed memory multiprocessor systems.

APPENDIX

In what follows, we present - assuming e-cube routing - a necessary condition for the existence of a channel contention between two different transfers on a DMMP, and a proof about the absence of network contention for an important class of communication statements.

The binary representation of a processor identification on the target architecture is specified by an array of bits. A processor identification for a processor s is defined by $s[1:d]$ where $s[1]$ and $s[d]$ are the least and highest significant bits, respectively. d (e.g. hypercube dimensionality) is constraint by the number of processors incorporated for the parallel program. A transfer t from a sending processor s to a receiving processor r is described by $t: s \rightarrow r$.

Let $t1, t2$ be two transfers induced by a communication statement C on a DMMP based on e-cube routing, then a channel contention occurs, iff $t1$ and $t2$ traverse a channel in the same direction at the same time.

> *LEMMA.* Let $t1 : s1 \rightarrow r1$ and $t2 : s2 \rightarrow r2$ be two transfers implied by a communication statement C on a DMMP based on e-cube routing, then for a necessary channel contention condition between $t1$ and $t2$, there exists a k $(1 \leq k \leq d)$ such that
>
> 1. $s1[k:d] = s2[k:d]$, and
> 2. $s1[k] \neq r1[k]$ and $s2[k] \neq r2[k]$, and
> 3. $r1[1:k] = r2[1:k]$

Intuitively the above lemma can be explained as follows: The path between a source processor s and the target processor r is uniquely defined by the underlying e-cube routing. The $k-1$ lowest significant bits are already switched from the source processor to the target processor binary representation. A switch of a single processor binary representation bit yields the binary representation of an immediate neighboring processor. In the k-th bit both transfers $t1$ and $t2$ have the same source processor and target processor binary representation, where the entire source and target binary representations are different. The $d-k$ highest significant bits are identical for both source processors in $t1$ and $t2$. When switching the k-th bit from the source to the target processor a channel contention may occur with respect to both transfers. Note that there might be several channel contentions because k is arbitrary. For instance, if a message is sent from processor 0 to processor 7 in the hypercube of Figure 5 on page 10, then the following path has to be traversed: $0 \rightarrow 1 \rightarrow 3 \rightarrow 7$. This path implies 3 network hops. Let $t1 : 0001 \rightarrow 0011 \rightarrow 0111$ and $t2 : 0000 \rightarrow 0001 \rightarrow 0011 \rightarrow 1011$. For this example $k = 2$ and the channel conflict between $t1$ and $t2$ occurs in $0001 \rightarrow 0011$.

PROOF. Let the path for $t1$ be described by $a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_r$ and for $t2$ by $b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_s$ where $a_1 = s1$, $a_r = r1$, $b_1 = s2$, $b_s = r2$, and $1 \leq s, r \leq d$. If there is a channel contention induced by $t1$ and $t2$ based on e-cube routing, then there exists a triple (i, j, k) with $1 \leq i \leq r$, $1 \leq j \leq s$ and $1 \leq k \leq d$ such that:

1. $a_i[k] = b_j[k]$, and
2. $a_{i+1}[k] = \neg a_i[k]$, and
3. $b_{j+1}[k] = \neg b_j[k]$, and
4. all bit positions besides the k-th one in a_i, b_i, a_{i+1} and b_{i+1} are identical, and
5. $a_i[k : d] = b_j[k : d] = a_1[k : d] = b_1[k : d]$, and
6. $a_i[k] = a_1[k]$ and $b_j[k] = b_1[k]$ and $a_r[k] \neq a_1[k]$ and $b_s[k] \neq b_1[k]$, and
7. $a_i[1 : k] = b_j[1 : k] = a_r[1 : k] = b_s[1 : k]$.

In the following an important class of communication statements will be discussed for which we can proof the absence of network contention on DMMPs based on e-cube routing. Note that the proof is based on target processors as opposed to virtual processors.

THEOREM. Let $t1 : s1 \rightarrow r1$ and $t2 : s2 \rightarrow r2$ be two transfers that are implied by a communication statement C on a DMMP based on e-cube routing, then there is no channel contention induced between $t1$ and $t2$, iff $r1-s1 = r2-s2$.

This theorem claims that a communication statement does not imply channel contention if all of its transfers induce a constant difference between sending and receiving processor with respect to their processor identification. For example, let us assume two transfers $t1 : 7 \rightarrow 4$, and $t2 : 11 \rightarrow 8$. Their difference between sending and receiving processors is 3. Hence, there is no network contention between these two transfers. The above theorem covers many important stencil and wavefront computations, where each processor receives a single data message from a specific neighboring processor with a constant processor distance. Note that if both transfers have the same source and target destination, then we assume that there is no network contention, as such transfers occur consecutively.

PROOF. For a channel contention the following holds:

1. $s1[k : d] = s2[k : d]$ and $r1[1 : k] = r2[1 : k]$ according to the previous lemma, and
2. $s1 \neq s2$ or $r1 \neq r2$

If $s1 \neq s2 \Longrightarrow s1[1 : k-1] \neq s2[1 : k-1] \Longrightarrow r1[1 : k-1] - s1[1 : k-1] \neq r2[1 : k-1] - s2[1 : k-1] \Longrightarrow r1 - s1 \neq r2 - s2$.

If $r1 \neq r2 \Longrightarrow r1[k+1 : d] \neq r2[k+1 : d] \Longrightarrow$ if $r1 - s1 = r2 - s2$ has to be true, then $s1[1 : k-1] \neq s2[1 : k-1] \Longrightarrow r1[1 : k-1] - s1[1 : k-1] \neq r2[1 : k-1] - s2[1 : k-1] \Longrightarrow r1 - s1 \neq r2 - s2$.

COROLLARY. Let C be a communication statement, A the array referenced by C and distributed according to the general block distribution as specified in Section 3, then C does not induce any network contention if the following two conditions are satisfied:

1. The overlap area of A extends the local area in exactly one dimension side in the same direction such that the overlap area does not extend the local area across more than one non-local segment, and
2. assuming a virtual to physical processor mapping strategy such that the absolute physical processor identification difference of all pairs of communicating processors with respect to C is a constant value.

PROOF. The conditions of the above corollary guarantee that every processor $p \in P^A$ communicates with at most one neighboring processor and the distance between source and target for each transfer is constant. The previous theorem states the absence of network contention for such a communication statement.

For example, every specific communication statement in the Gauss/Seidel, Jacobi and Red-black relaxation kernel do not imply any network contention based on the previous corollary. Note that this corollary refers only to single communication statements. The absence of network contention is not guaranteed for more than one of such a communication statement in consecutive order.

ACKNOWLEDGMENTS

The author is grateful to Prof. H. Zima for his support of this research and to the Vienna Fortran Compiler research group, in particular to Alex Celic, for implementing the software infrastructure of P^3T and to Hans Moritsch for providing the ADI code. The author thanks the anonymous referees for their constructive comments.

References

[1] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] S. P. Amarasinghe, J. M. Anderson, and M. S. Lam. An Overview of a Compiler for Scalable Parallel Machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 253–272, Portland, OR, August 1993. Springer Verlag.

[3] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *Proceedings of the First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, December 1994.

[4] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H. Zima. *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.

[5] S. Bokhari. Communication overhead on the intel iPSC-860 Hypercube. ICASE Interim Report 10, NASA Contractor Report 182055, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665, May 1990.

[6] S. Bokhari. Communication Overhead on the Intel Paragon, IBM SP2 and Meiko CS-2. ICASE Interim Report No. 28, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665, August 1995.

[7] M. B. et al. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, pages 3(3): 5–40, 1989.

[8] T. Fahringer. The Weight Finder, An Advanced Profiler for Fortran Programs. In *Automatic Parallelization, New Approaches to Code Generation, Data Distribution, and Performance Prediction.* Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, March 1993.

[9] T. Fahringer. Using the P^3T to Guide the Parallelization and Optimization Effort under the Vienna Fortran Compilation System. In *IEEE Proc. of the Scalable High Performance Computing Conference 1994, Knoxville, TN*, May 1994.

[10] T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11):47 – 56, November 1995.

[11] T. Fahringer. *Automatic Performance Prediction of Parallel Programs* . Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.

[12] T. Fahringer. Toward Symbolic Performance Prediction of Parallel Programs. In *IEEE Proc. of the 1996 International Parallel Processing Symposium*, pages 474–478, Honolulu, Hawaii, April 15 - 19, 1996.

[13] T. Fahringer and H. Zima. A Static Parameter based Performance Prediction Tool for Parallel Programs. In *Proc. of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993. best paper award.

[14] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. and 2. Prentice Hall, Englewood Cliffs, NY, 1988.

[15] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proc. Sixth International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.

[16] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, Aug 1991.

[17] High Performance FORTRAN Language Specification. Technical Report, Version 1.0, Rice University, Houston, TX, May 1993.

[18] U. Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Rice University, Technical Report CRPC-TR95559-S, October 1995.

[19] F. McMahon. *The Livermore Fortran Kernels Test of the Numerical Performance Range*, pages 143–186. Elsevier Science B.V.,North Holland, Amsterdam, 1988.

[20] G. Pfister and V. Norton. Hot Spot Contention and Combining in Multistage Interconnection Networks. In *Proc. of the 1985 International Conference on Parallel Processing*, pages 790–797, 1985.

[21] B. A. Rosenfeld. *Multi-dimensional Spaces (in russian)*, chapter 5, pages 169–173. Nauka, Moscow, 1966.

[22] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessor*. The MIT Press, Cambridge, Massachusetts, 1989.

[23] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[24] W. Vetterling, S. Teukolsky, W. Press, and B. Flannery. *Numerical Recipes: Example Book (FORTRAN)*. Cambridge University Press, 1990.

[25] K. Wang. A Framework for Static, Precise Performance Prediction for Superscalar-Based Parallel Computers . In H. Sips, editor, *In Proceedings of the 4th International Workshop on Compilers for Parallel Computers*, pages 413–427, Delft, The Netherlands, December 1993.

[26] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a language specification. Technical report, ICASE, Hampton,VA, 1992. ICASE Internal Report 21.

THOMAS FAHRINGER is an Assistant Professor of Computer Science at the Institute for Software Technology and Parallel Systems, University of Vienna. His research focuses on parallelizing compilers, multithreaded run-time systems, software tools for multiprocessor architectures, performance analysis, prediction and visualization tools for parallel programs and architectures. Fahringer received a M.S. and a Ph.D. degree in Computer Science from the Technical University of Vienna in 1988 and 1993, respectively. From 1988 through 1990 he was a visiting research assistant at Carnegie Mellon University. Readers may contact Fahringer at the Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria; WWW: http://www.par.univie.ac.at and e-mail: tf@par.univie.ac.at