In Proceedings of the 1998 International Parallel Processing Symposium

A Generalized Framework for Global Communication Optimization

M. Kandemir*† P. Banerjee‡§ A. Choudhary‡† J. Ramanujam¶ N. Shenoy‡†

Abstract

In distributed-memory message-passing architectures reducing communication cost is extremely important. In this paper, we present a technique to optimize communication globally. Our approach is based on a combination of linear algebra framework and dataflow analysis, and can take arbitrary control flow into account. The distinctive features of the algorithm are its accuracy in keeping communication set information and its support for general alignments and distributions including block-cyclic distributions. The method is currently being implemented in the PARADIGM compiler. The preliminary results show that the technique is effective in reducing both number as well as volume of the communication.

1. Introduction

Distributed-memory multiprocessors are attractive for high performance computing in that they offer potentially high levels of flexibility, scalability and performance. But the need for explicit message passing resulting from the lack of a globally shared address space renders programming these machines a difficult task. The main objective behind the efforts such as High Performance Fortran (HPF) is to raise the level of programming by allowing the user write programs with a shared address space view augmented with directives that specify data mapping. The compilers for such languages are responsible for partitioning computation, inserting necessary commands that implement required message passing for access to non-local data.

On such machines, cost to access non-local data is usually orders of magnitude higher than accessing local data. Therefore, it is imperative to reduce the frequency and volume of non-local accesses as much as possible. Several efforts have been aimed at reducing communication overhead incurred. The most common optimization technique used by previous researchers is message vectorization [7, 3]. In message vectorization, instead of naively inserting `send` and `recv` operations just before references to non-local data, communication is hoisted to outer loops. Essentially this optimization replaces many small messages with one large message reducing the number of messages. Some researchers [7] have also considered message coalescing which combines messages due to different references to the same array, and message aggregation which combines messages due to references to different arrays to the same destination processor into a single message. Many of these techniques are limited to a single loop nest.

Recently a number of authors have proposed techniques based on dataflow analysis to optimize communication across multiple loop nests [5, 6, 10]. Almost all of the approaches use a variant of Regular Section Descriptors (RSD) introduced by Callahan and Kennedy [4]. For each array referenced in the program, an RSD is defined which describes the portion of the array that is referenced. Although this representation is convenient for simple array sections such as those found in pure block or cyclic distributions, it is hard to embed alignment and general distribution information into it. Apart from inadequate support for block-cyclic distributions, working with section descriptors may sometimes result in overestimation of the communication sets.

In this paper, we show that the problem of global communication optimization can be cast in a linear algebra framework. This allows the compiler to easily apply traditional loop-based optimization techniques such as message vectorization, message coalescing as well as global optimizations such as redundant communication elimination and communication hoisting. Using the linear algebra framework proposed by Ancourt et al. [2], our technique is able to handle the optimization problem at the granularity of individual array elements. The task of code generation is made easier by our use of the Omega library [9] from the University of Maryland. The global communication sets resulted from our dataflow analysis can be manipulated by the Omega system. Our approach gives compiler the capability to represent communication sets globally as equalities and inequalities as well as to use polyhedron scanning techniques to perform optimizations such as redundant communication elimination and global message coalescing which

*EECS Dept., Syracuse University, Syracuse, NY 13244. e-mail:`mtk@ece.nwu.edu`

†Supported in part by NSF Young Investigator Award CCR-9357840 and NSF grant CCR-9509143

‡ECE Dept., Northwestern University, Evanston, IL 60208. e-mail:`{banerjee,choudhar,nagaraj}@ece.nwu.edu`

§Supported in part by DARPA under contract DAAH-04-G-0273 administered by the Army Research Office

¶ECE Dept., Louisiana State University, Baton Rouge, LA 70803. e-mail:`jxr@ee.lsu.edu`. Supported in part by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422.

were not possible under the loop nest based communication optimization schemes. We assume reader's familiarity with basic dataflow concepts such as control flow graph (CFG), basic block, and interval analysis as well as basic HPF directives. We also assume that prior to our analysis, the compiler has performed all loop-level transformations to enhance parallelism (e.g., loop permutation, loop distribution). Our technique is based on interval analysis performed on the CFG. The interval analysis [1] consists of a contraction phase and an elimination phase. In this paper, an interval corresponds to a loop and depending on the context the term 'node' is used for a statement, a basic block or a reduced interval.

2. Dataflow analysis

Consider the following program fragment annotated by HPF-like directives.

```
        real X(a_l:a_u)
!HPF$ template T(t_l:t_u)
!HPF$ processors PROC(p_l:p_u)
!HPF$ align X(j) with T(α*j+β)
!HPF$ distribute T(cyclic(C)) onto PROC

        do i = i_l, i_u
           X(γ_L*i+θ_L) = ··· X(γ_R*i+θ_R) ···
        enddo
```

Let $\mathcal{R_L} = \mathtt{X(}\gamma_L\mathtt{*i+}\theta_L\mathtt{)}$ and $\mathcal{R_R} = \mathtt{X(}\gamma_R\mathtt{*i+}\theta_R\mathtt{)}$. Assuming p and q denote two processors, the following sets can be defined:

$$
\begin{aligned}
\mathtt{Own}(\mathtt{X},\mathtt{q}) &= \{\mathtt{d} \mid \mathtt{d} \in \mathtt{X} \text{ and is owned by } \mathtt{q}\}\\
\mathtt{Compute}(\mathtt{X},\mathcal{R_L},\mathtt{q}) &= \{\mathtt{i} \mid \gamma_L * \mathtt{i} + \theta_L \in \mathtt{Own}(\mathtt{X},\mathtt{q})\\
&\qquad \text{and } i_l \le \mathtt{i} \le i_u\}\\
\mathtt{View}(\mathtt{X},\mathcal{R_R},\mathtt{q}) &= \{\mathtt{d} \mid \exists\imath \text{ st. } \imath \in \mathtt{Compute}(\mathtt{X},\mathcal{R_L},\mathtt{q}) \text{ and}\\
&\qquad \mathtt{d} = \mathtt{X}(\gamma_{\mathcal{R}} * \imath + \theta_{\mathcal{R}}) \text{ and } i_l \le \imath \le i_u\}\\
\mathtt{CommSet}(\mathtt{X},\mathcal{R_R},\mathtt{p},\mathtt{q}) &= \mathtt{Own}(\mathtt{X},\mathtt{q}) \cap \mathtt{View}(\mathtt{X},\mathcal{R_R},\mathtt{p}).
\end{aligned}
$$

Intuitively, the set Own(X,q) refers to the elements mapped onto processor q through compiler directives. The set of iterations to be executed by q due to a LHS reference $\mathcal{R_L}$ is given by Compute(X,$\mathcal{R_L}$,q). Of course, during the execution of this iteration set, some elements (local or non-local) accessed by RHS reference $\mathcal{R_R}$ will be required; the set View(X,$\mathcal{R_R}$,q) defines these elements. Finally, CommSet(X,$\mathcal{R_R}$,p,q) defines the elements that should be communicated from processor q to processor p due to reference $\mathcal{R_R}$. It should be noted that in general there may be more than one RHS reference, and the computation may involve multi-dimensional arrays and a multi-level nest in which case d and i denote data and iteration vectors respectively. Also in the most general case, α, γ_L and γ_R are matrices, and β, θ_L and θ_R are vectors.

The definition of the Own set above is rather informal. For a more precise definition, we take into account the block-cyclic distribution and define the Own set as

$$
\begin{aligned}
\mathtt{Own}(\mathtt{X},\mathtt{q}) \;=\; &\{\mathtt{d} \mid \exists \mathtt{t},\mathtt{c},\mathtt{l} \text{ such that } \mathtt{t} = \alpha * \mathtt{d} + \beta \text{ and}\\
&\mathtt{t} = \mathcal{C} * P * \mathtt{c} + \mathcal{C} * \mathtt{q} + \mathtt{l} \text{ and } a_l \le \mathtt{d} \le a_u \text{ and}\\
&p_l \le \mathtt{q} \le p_u \text{ and } t_l \le \mathtt{t} \le t_u \text{ and } 0 \le \mathtt{l} \le \mathcal{C}-1\},
\end{aligned}
$$

where P = $p_u - p_l + 1$. In this formulation, $\mathtt{t} = \alpha * \mathtt{d} + \beta$ represents alignment information and $\mathtt{t} = \mathcal{C}*P*\mathtt{c}+\mathcal{C}*\mathtt{q}+\mathtt{l}$ denotes the distribution information. Simple BLOCK and CYCLIC(1) distributions can easily be handled within this framework by setting c=0 and l=0, respectively. See [2] for the details.

A *communication descriptor* can be defined as a pair $\langle\mathcal{R},\mathcal{S}\rangle$, where $\mathcal{R}$ is an array identifier (name) and $\mathcal{S}$ is the *communication set* associated with $\mathcal{R}$. The exact definition of a communication set depends on the context in which it is used. Throughout our analysis, a set is defined as $\{\bar{d} \mid \bar{d}$ is owned by q and is required by (or should be transferred to or has already been transferred to) p$\}$ except for the KILL set, which defines the set of elements written (killed) by q. Since we define a communication set as a list of equalities and inequalities, it can be represented as $\mathcal{S} = \{\bar{d} \mid \mathcal{P}(\bar{d})\}$ where $\mathcal{P}(.)$ is a predicate. Let $\{\bar{d} \mid \mathcal{P}(\bar{d})\}$ and $\{\bar{d} \mid \mathcal{Q}(\bar{d})\}$ be two communication sets. We define the operations $+_c$, $-_c$, and $\cap_c$ on communication sets as follows:

$$
\begin{aligned}
\{\bar{d} \mid \mathcal{P}(\bar{d})\} +_c \{\bar{d} \mid \mathcal{Q}(\bar{d})\} &= \{\bar{d} \mid \mathcal{P}(\bar{d}) \text{ or } \mathcal{Q}(\bar{d})\}\\
\{\bar{d} \mid \mathcal{P}(\bar{d})\} -_c \{\bar{d} \mid \mathcal{Q}(\bar{d})\} &= \{\bar{d} \mid \mathcal{P}(\bar{d}) \text{ and not } (\mathcal{Q}(\bar{d}))\}\\
\{\bar{d} \mid \mathcal{P}(\bar{d})\} \cap_c \{\bar{d} \mid \mathcal{Q}(\bar{d})\} &= \{\bar{d} \mid \mathcal{P}(\bar{d}) \text{ and } \mathcal{Q}(\bar{d})\}
\end{aligned}
$$

Note that the operations 'or', 'and' and 'not' can easily be performed by the corresponding Omega operations on sets which contain equalities and inequalities.

Let $\mathcal{D} = \langle\mathcal{R},\mathcal{S}\rangle$ be a communication descriptor. We define two functions: a function $\mathcal{N}$ from communication descriptors space to array identifiers space; and a function $\mathcal{M}$ from communication descriptors space to communication sets space such that $\mathcal{N}(\mathcal{D}) = \mathcal{R}$ and $\mathcal{M}(\mathcal{D}) = \mathcal{S}$.

Suppose $\mathcal{DS}_1$ and $\mathcal{DS}_2$ are two communication descriptor sets. Three operations, namely $+_d$, $-_d$, and $\cap_d$, are defined on these sets as follows:

$$
\begin{aligned}
\mathcal{DS}_1 +_d \mathcal{DS}_2 \;=\; &\{\mathcal{D} \mid \mathcal{D} \in \mathcal{DS}_1 \text{ and } \forall\mathcal{D}' \in \mathcal{DS}_2\ \mathcal{N}(\mathcal{D}) \ne \mathcal{N}(\mathcal{D}')\}\\
&\cup\{\mathcal{D} \mid \mathcal{D} \in \mathcal{DS}_2 \text{ and } \forall\mathcal{D}' \in \mathcal{DS}_1\ \mathcal{N}(\mathcal{D}) \ne \mathcal{N}(\mathcal{D}')\}\\
&\cup\{\mathcal{D} \mid \exists\mathcal{D}' \in \mathcal{DS}_1, \mathcal{D}'' \in \mathcal{DS}_2 \text{ st.}\\
&\mathcal{N}(\mathcal{D}) = \mathcal{N}(\mathcal{D}') = \mathcal{N}(\mathcal{D}'') \text{ and}\\
&\mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}') +_c \mathcal{M}(\mathcal{D}'')\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{DS}_1 -_d \mathcal{DS}_2 \;=\; &\{\mathcal{D} \mid \mathcal{D} \in \mathcal{DS}_1 \text{ and } \forall\mathcal{D}' \in \mathcal{DS}_2\ \mathcal{N}(\mathcal{D}) \ne \mathcal{N}(\mathcal{D}')\}\\
&\cup\{\mathcal{D} \mid \exists\mathcal{D}' \in \mathcal{DS}_1, \mathcal{D}'' \in \mathcal{DS}_2 \text{ st.}\\
&\mathcal{N}(\mathcal{D}) = \mathcal{N}(\mathcal{D}') = \mathcal{N}(\mathcal{D}'') \text{ and}\\
&\mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}') -_c \mathcal{M}(\mathcal{D}'')\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{DS}_1 \cap_d \mathcal{DS}_2 \;=\; &\{\mathcal{D} \mid \exists\mathcal{D}' \in \mathcal{DS}_1, \mathcal{D}'' \in \mathcal{DS}_2 \text{ st.}\\
&\mathcal{N}(\mathcal{D}) = \mathcal{N}(\mathcal{D}') = \mathcal{N}(\mathcal{D}'') \text{ and}\\
&\mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}') \cap_c \mathcal{M}(\mathcal{D}'')\}
\end{aligned}
$$

In this paper, we sometimes use $\cup_c$ and $\cup_d$ instead of $+_c$ and $+_d$, respectively. It should be noted that although these operations are similar to those presented by Gong et al. [5], there is an important difference. Since we keep the communication sets accurately in terms of equalities and inequalities, we can optimize (e.g., coalesce) communication messages even if the messages do not have the same communication pattern (e.g., broadcast, point-to-point) or identical sender/receiver sets.

2.1. Local (Intra-Interval) Analysis

The local analysis part of our framework computes KILL, GEN and POST_GEN sets for each interval. Then the interval is reduced to a single node and annotated with this information.

Let $\mathcal{R}_\mathcal{L}(\bar{\imath})$ and $\mathcal{R}_\mathcal{R}(\bar{\imath})$ be the data elements obtained from references $\mathcal{R}_\mathcal{L}$ and $\mathcal{R}_\mathcal{R}$ respectively with a specific iteration vector $\bar{\imath}$. The computation of the KILL set proceeds in the forward direction; that is, the nodes within the interval are traversed in topological sort order. Let KILL(i,q) be the set of elements written (killed) by processor q in node i, and Modified(i,q) be the set of elements that may be killed along any path from the beginning of the interval to node i (including node i). Then,

$$\begin{aligned}
\mathtt{KILL(i,q)} &= \{\bar{d} \mid \bar{d} \in \mathtt{Own(X,q)} \text{ and } \exists \bar{\imath} \text{ st.} \\
&\qquad \bar{d} = \mathcal{R}_\mathcal{L}(\bar{\imath}) \text{ and } \bar{i_l} \le \bar{\imath} \le \bar{i_u}\}, \\
\mathtt{Modified(i,q)} &= [\bigcup_{j \in pred(i)} \mathtt{Modified(j,q)}] \cup \mathtt{KILL(i,q)}
\end{aligned}$$

assuming that Modified(pred(first(i)),q) = $\emptyset$ where first(i) is the first node in i. If last(i) is the last node in i, then

$$\mathtt{KILL(i,q)} = \mathtt{Modified(last(i),q)}.$$

This last equation is used to reduce an interval into a node.

GEN(i,p,q) is the set of elements required by processor p from processor q at node i with no preceding write (assignment) to them. The computation of the GEN proceeds in the backward direction, i.e., the nodes within each interval are traversed in reverse topological sort order. The elements that can be communicated at the beginning of a node are the elements required by any RHS reference within the node except the ones that are written before being referenced.

Assuming $\bar{\imath} = (\imath_1, ..., \imath_n)$ and $\bar{\imath}' = (\imath'_1, ..., \imath'_n)$, let $\bar{\imath}' \prec \bar{\imath}$ mean that $\bar{\imath}'$ is lexicographically less than or equal to $\bar{\imath}$; and $\bar{\imath}' \prec_k \bar{\imath}$ mean that $\imath'_j = \imath_j$ for all $j < k$, and $(\imath'_k, ..., \imath'_n) \prec (\imath_k, ..., \imath_n)$. Since a node can refer to multiple RHS references, we define

$$\mathtt{GEN(i,p,q)} = \bigcup_{\mathcal{R}_\mathcal{R}} \mathtt{GEN}(\mathtt{i}, \mathcal{R}_\mathcal{R}, \mathtt{p}, \mathtt{q}).$$

For the sake of simplicity, we assume one RHS reference per node, and use GEN(i,p,q) in the following. Let Comm(i,p,q) be the set of elements that may be communicated at the beginning of interval i to satisfy communication requirements from the beginning of i to the last node of i. Then, assuming that Comm(succ(last(i)),q) = $\emptyset$, we have

$$\begin{aligned}
\mathtt{GEN(i,p,q)} &= \{\bar{d} \mid \exists \bar{\imath} \text{ st. } \bar{i_l} \le \bar{\imath} \le \bar{i_u} \text{ and } \bar{d} \in \mathtt{Own(X,q)} \text{ and} \\
&\qquad \bar{d} = \mathcal{R}_\mathcal{R}(\bar{\imath}) \text{ and } \mathcal{R}_\mathcal{L}(\bar{\imath}) \in \mathtt{Own(X,p)} \text{ and not} \\
&\qquad (\exists \bar{\jmath}, \mathcal{R}_\mathcal{L}' \text{ st. } \bar{i_l} \le \bar{\jmath} \le \bar{i_u} \text{ and } \bar{d} = \mathcal{R}_\mathcal{L}'(\bar{\jmath}) \text{ and} \\
&\qquad \bar{\jmath} \prec_{\mathtt{level(i)}} \bar{\imath})\}, \\
\mathtt{Comm(i,p,q)} &= [\bigcap_{s \in succ(i)} \mathtt{Comm(s,p,q)}] \cup \mathtt{GEN(i,p,q)}.
\end{aligned}$$

In addition, we use the following equation to reduce an interval into a single node:

$$\mathtt{GEN(i,p,q)} = \mathtt{Comm(First(i),p,q)}.$$

In the definition of GEN, $\mathcal{R}_\mathcal{R}$ denotes the RHS reference, and $\mathcal{R}_\mathcal{L}$ denotes the LHS reference of the same statement. $\mathcal{R}_\mathcal{L}'$, on the other hand, refers to any LHS reference within the same interval. Notice that while $\mathcal{R}_\mathcal{L}'$ is a reference to the same array as $\mathcal{R}_\mathcal{R}$, $\mathcal{R}_\mathcal{L}$ can be a reference to any array. level(i) gives the nesting level of the interval (loop), 1 corresponding to the outermost loop in the nest.

After the interval is reduced, the GEN set for it is recorded, and an operator $\mathcal{F}$ is applied to the last part of this GEN set to propagate it to the outer interval:

$$\mathcal{F}(\bar{\jmath} \prec_\mathtt{k} \bar{\imath}) = \bar{\jmath} \prec_{(\mathtt{k}-1)} \bar{\imath}.$$

We should emphasize that computation of the GEN sets gives us all the communication that can be vectorized above a loop nest; that is, our analysis easily handles message vectorization. Finally, POST_GEN(i,p,q) is the set of elements required by processor p from processor q at node i with no subsequent write to them:

$$\begin{aligned}
\mathtt{POST_GEN(i,p,q)} &= \{\bar{d} \mid \exists \bar{\imath} \text{ st. } \bar{i_l} \le \bar{\imath} \le \bar{i_u} \text{ and } \bar{d} \in \mathtt{Own(X,q)} \\
&\qquad \text{and } \bar{d} = \mathcal{R}_\mathcal{R}(\bar{\imath}) \text{ and } \mathcal{R}_\mathcal{L}(\bar{\imath}) \in \mathtt{Own(X,p)} \\
&\qquad \text{and not } (\exists \bar{\jmath}, \mathcal{R}_\mathcal{L}' \text{ st. } \bar{i_l} \le \bar{\jmath} \le \bar{i_u} \text{ and} \\
&\qquad \bar{d} = \mathcal{R}_\mathcal{L}'(\bar{\jmath}) \text{ and } \bar{\imath} \prec_{\mathtt{level(i)}} \bar{\jmath})\}.
\end{aligned}$$

The computation of POST_GEN(i,p,q) proceeds in the forward direction. Its computation is similar to those of KILL(i,q) and GEN(i,p,q) sets.

2.2. Dataflow Equations

We concentrate on the computation of the recv sets only. Similar analysis applies to send sets as well except for the fact that the definition of the communication set should be modified accordingly. Our dataflow analysis framework consists of a backward and a forward pass. In the backward pass, the compiler determines sets of data elements that can safely be communicated at specific points. The forward pass, on the other hand, eliminates redundant communication and determines the final

Backward Analysis:

$$\mathtt{SAFE\_OUT(i,p,q)} = \bigcap_{s \in succ(i)} \mathtt{SAFE\_IN(s,p,q)} \tag{1}$$

$$\mathtt{SAFE\_IN(i,p,q)} = \begin{cases} \mathtt{GEN(i,p,q)} & \text{if } \mathcal{P}(i) \\ (\mathtt{SAFE\_OUT(i,p,q)} -_{\mathtt{d}} \mathtt{KILL(i,q)}) +_{\mathtt{d}} \mathtt{GEN(i,p,q)} & \text{otherwise} \end{cases} \tag{2}$$

Forward Analysis:

$$\mathtt{RECV\_IN(i,p,q)} = \bigcap_{j \in pred(i)} \mathtt{RECV\_OUT(j,p,q)} \tag{3}$$

$$\mathtt{RECV(i,p,q)} = \begin{cases} \mathtt{GEN(i,p,q)} -_{\mathtt{d}} \mathtt{RECV\_IN(i,p,q)} & \text{if } \exists\, k \in succ(i) \text{ and } k \notin dom(i) \\ \mathtt{SAFE\_IN(i,p,q)} -_{\mathtt{d}} \mathtt{RECV\_IN(i,p,q)} & \text{otherwise} \end{cases} \tag{4}$$

$$\mathtt{RECV\_OUT(i,p,q)} = \begin{cases} \mathtt{RECV\_IN(i,p,q)} -_{\mathtt{d}} \mathtt{KILL(i,q)} & \text{if } \exists\, k \in succ(i) \text{ and } k \notin dom(i) \\ ((\mathtt{RECV(i,p,q)} +_{\mathtt{d}} \mathtt{RECV\_IN(i,p,q)}) -_{\mathtt{d}} \mathtt{KILL(i,q)}) +_{\mathtt{d}} \mathtt{POST\_GEN(i,p,q)} & \text{otherwise} \end{cases} \tag{5}$$

Figure 1. Dataflow equations for optimizing communication.

set of elements (if any) that should be communicated at the beginning of each node `i`. The input for the equations consists of the `GEN(i,p,q)`, `KILL(i,q)` and `POST_GEN(i,p,q)` sets as computed during the local analysis. The dataflow equations for the backward analysis are given by Equations (1) and (2) in Figure 1. Note that $\cap$ in this figure denotes $\cap_d$.

`SAFE_IN(i,p,q)` and `SAFE_OUT(i,p,q)` are the sets of communication descriptors which denote the elements that *can* safely be communicated at the beginning and end of node `i`, respectively. Equation (1) says that an element should be communicated at a point if and only if it will be used in all of the following paths in the CFG. This is the fundamental rule that our dataflow analysis as well as some of the previous approaches like [10] adheres to. Equation (2), on the other hand, gives the set of elements that can safely be communicated at the beginning of node `i`, and makes use of the `GEN` and `KILL` sets. Intuitively, an element can be communicated at the beginning of node `i` if and only if it is either required (generated) by node `i` or it reaches at the end of node `i` and is not overwritten (killed) in it. $\mathcal{P}(i)$ is a predicate used to control how far the communications should be hoisted. It should be noted that if the elements contained in `SAFE_IN` sets are directly communicated without any further analysis, there would be significant amounts of redundant communication. The task of the forward analysis phase is to eliminate redundant communication.

The dataflow equations for the forward analysis are given by Equations (3), (4) and (5) in Figure 1. `RECV_IN(i,p,q)` and `RECV_OUT(i,p,q)` denote the set of communication descriptors which contain the elements that *have been* communicated so far (at the beginning and end of the node `i` respectively) from `q` to `p`. On the other hand, `RECV(i,p,q)` denotes the set of communication descriptors which contain elements that *should* be communicated from `q` to `p` at the beginning of node `i` and is finally used by the communication generation portion of the compiler to generate the actual `recv` commands. Equation (3) simply says that the communication set arriving in a join node can be found by intersecting the sets for all the joining paths. Equation (4) is used to compute the `RECV` set which corresponds to the elements that can be communicated at the beginning of the node except the ones that have already been communicated (`RECV_IN`). The elements that have been communicated at the end of node `i` (that is, `RECV_OUT` set) are simply the union of the elements communicated up to the beginning of `i`, the elements communicated at the beginning of `i` provided that the condition in equation (5) is not satisfied (except the ones which have been overwritten (killed) in `i`) and the elements communicated in `i` and not written subsequently (`POST_GEN`), again provided that the condition in the equation is not satisfied. It should be emphasized that all these sets are communication descriptor sets, and the order of operations as indicated by the parentheses is important.

2.3 Global Dataflow Analysis

Our approach starts by computing the `GEN`, `KILL` and `POST_GEN` sets for each node. Then the contraction phase of the analysis reduces the intervals from innermost to outermost and annotates them with `GEN`, `KILL` and `POST_GEN` sets. When a reduced CFG with no cycles is reached, the expansion phase starts and `RECV` sets for each interval is computed, this time from outermost to innermost. There is one important point to note: before starting to process the next inner graph, the `RECV_IN` set of the first node in this graph is set to the `RECV` set of the interval that contains it. More formally, in the expansion phase, we set

$$\mathtt{RECV\_IN(i,p,q)}^{k^{th} pass} = \mathtt{RECV(i,p,q)}^{(k-1)^{th} pass.}$$

Table 1. Results of the static analysis for `swim256` on 8 nodes.

program	number of messages			communication volume (in array elements)		
	(BLOCK,*)	(CYCLIC(4),*)	(CYCLIC,*)	(BLOCK,*)	(CYCLIC(4),*)	(CYCLIC,*)
base	4,691	5,212	5,212	2,188,032	34,277,622	135,778,800
opt	2,489	2,770	2,770	1,270,230	18,936,460	73,517,520
% imprv.	47%	47%	47%	42%	45%	46%

This assignment then triggers the next pass in the expansion phase. Of course, before the expansion phase starts $\mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q})^{1^{st}pass}$ is set to $\emptyset$. The `send` sets are generated analogously during the same process. We refer the reader to [8] for the details of our approach.

3. Preliminary results

We measure the effectiveness of our approach statically in terms of number of messages and communication volume across all processors. Table 1 presents the results of static analysis for `40` iterations of the `swim256` benchmark (from Spec benchmark suite) on `8` processors with an input size of `512` for `(BLOCK,*)`, `(CYCLIC(4),*)` and `(CYCLIC,*)` distributions. tt base refers to the message vectorized version whereas `opt` is the program resulted from our global optimization framework. On the average, there is a `47%` reduction in the number of messages, and a `44%` reduction in the communication volume. The results show that our approach is successful at reducing the number of communication calls and the communication volume. We refer the reader to [8] for more experimental results.

4. Conclusions

In this paper, we presented a global communication optimization scheme based on two complementary techniques: dataflow analysis and linear algebra framework. The combination of these techniques allows us to optimize communication globally for HPF-like alignments and distributions including block-cyclic distributions. The cost of the analysis is managed by keeping the communication sets symbolically until the end of the dataflow analysis where the Omega library is called to generate actual sets in terms of equalities and inequalities. The preliminary experimental results show the effectiveness of our approach in reducing the number of messages and the volume of the data to be communicated.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[2] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proc. 4th International Workshop on Compilers for Parallel Computers*, Delft, the Netherlands, 1993.

[3] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[4] D. Callahan, and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *J. Parallel Dstrib. Comput. 5*, 5 (Oct. 1988), pp. 517–550.

[5] C. Gong, R. Gupta, and R. Melhem. Compilation techniques for optimizing communication on distributed-memory systems. In *Proc. International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[6] M. Gupta, E. Schonberg, and H. Srinivasan. A unified dataflow framework for optimizing communication. In *Proc. 7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca NY, August 1994.

[7] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. In *Communications of the ACM*, 35(8):66–80, August 1992.

[8] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. Optimizing communication using global dataflow analysis. Technical Report, CPDC-TR-97-02, Northwestern University, October 1997.

[9] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, CS Dept., University of Maryland, College Park, March 1995.

[10] K. Kennedy, and A. Sethi. A constrained-based communication placement framework, Technical Report CRPC-TR95515-S, CRPC, Rice University, 1995.