A Framework for Integrated Communication and I/O Placement

Rajesh Bordawekar[1], Alok Choudhary[1] and J. Ramanujam[2]

[1] ECE Dept., 121, Link Hall, Syracuse University, Syracuse, NY 13244
[2] ECE. Dept., Louisiana State University, Baton Rouge, LA 70803

Abstract. This paper describes a framework for analyzing dataflow within an out-of-core parallel program. Dataflow properties of `FORALL` statement are analyzed and a unified I/O and communication placement framework is presented. This placement framework can be applied to many problems, which include eliminating redudant I/O incurred in communication. The framework is validated by applying it for optimizing I/O and communication in out-of-core stencil problems. Experimental performance results on an Intel Paragon show significant reduction in I/O and communication overhead.

1 Introduction

It is widely acknowledged in the high-performance computing circles that parallel input/output requires substantial improvement in order to make scalable computers truly usable. There are several reasons for a parallel application for performing input/output. These include real-time I/O, initial/final read-write, checkpointing and out-of-core computations [Bor96].

We focus on the problem of supporting *out-of-core* computations. Out-of-core computations are those computations whose primary data sets are stored on files in the secondary memory. Specifically, we concentrate on compiling out-of-core programs developed using High Performance Fortran (HPF) [Hig93].[3] HPF is a data parallel language which provides explicit language directives to partition data over processors in certain pre-defined decomposition patterns like `BLOCK` and `CYCLIC`. This data distribution results in each processor storing a *local array* associated with each array distributed in the HPF program. HPF also provides data-parallel program construsts like `FORALL` [Hig93].

In this paper, we describe a dataflow framework for optimizing communication in out-of-core problems. We focus on communication optimization within a single out-of-core `FORALL` construct. Unlike the available dataflow frameworks for optimizing inter-processor communication [KN94, KS95, GSS95], our framework takes an unified approach for placing I/O and communication calls *while preserving characteristics of these calls.* All the current frameworks focus on improving communication performance by vectorizing messages, eliminating redundant communication and overlapping communication with computation. How-

[3] Although the techniques are discussed with respect to HPF, they can be applied to compilation of data parallel programs in general.

ever, these frameworks do not directly extend to out-of-core problems. Another limitation of these frameworks is that they do not make efficient use of the *copy-in-copy-out* semantics of the HPF `FORALL` construct. We illustrate these points by applying two communication placement frameworks [KN94, KS95] to an out-of-core problem performing stencil computations (also called an *regular* problem). We then compare the results with an integrated I/O and communication placement framework which achieves substantial performance improvement by simultaneously reordering I/O and communication calls.

The paper is organized as follows: Section 2 introduces various dataflow definitions that will be used throughout the paper. In Section 3, we present an out-of-core regular problem and analyzes it's communication and I/O pattern. This problem is used as a running example throughout the paper. Section 4 presents an integrated I/O and communication framework and describes its application in eliminating extra file I/O from communication. Section 5 presents experimental performace results of optimizing out-of-core communication from stencil problems using our framework. Finally, we conclude in Section 6.

2 Background

Our program representation is based on [KS95]. Let $G{=}(N, E)$ be the interval flow graph representing an HPF program, with N nodes and E edges. Let s and e be the unique start and end nodes of G. Every edge in E can be classified as an entry, forward or backward edge. Let a Tarjan interval $\mathrm{T}(h)$ represent a set of program flow nodes that correspond to a loop in the program text. $\mathrm{T}(h)$ has a unique header h, where $h \notin \mathrm{T}(h)$. For every node n of the interval flow graph, G, we define $\textsc{Succ}(n)$ and $\textsc{Pred}(n)$ as a set of successor and predecessor nodes of n. The edges induce the following traversal order over G. Given a forward edge (m, n), a FORWARD order visits m before n and a BACKWARD order visits m after n. Let HEADER denote the header node of the interval $\mathrm{T}(n)$. [Bor96] describes the properties of the interval flow graph.

To anlyze dataflow properties of the `FORALL` statement, we use the classical dataflow definitions, i.e., USE, DEF, KILL. A variable is said to be USEd if it is referred in an expression. A variable is said to be DEFed if it is *initialized* in an expression. The variable is said to be LIVE until it is *defined* again (in other words, KILLed). We can extend these definitions for objects such as arrays. An array is said be INJURED, if some elements of the array are overwritten, otherwise the array can be considered LIVE. An array is said to be ACTIVE if some of its elements are either USEd or DEFed and these elements constitute the ACTIVE set of the array.

Recall that the `FORALL` statement has *copy-in-copy-out* semantics [Hig93]. Consequently, during the execution of a `FORALL` statement, old as well as new values of an array can be LIVE. In other words, the `FORALL` statement satisfies the DELAYED_KILL property [Bor96]. We use variable *DKILL* to represent an array which satisfies the DELAYED_KILL property.

We now define some dataflow variables that will be used for analyzing communication and I/O access patterns in out-of-core programs. Let $\textsc{Active}_n^p$ denote the set of elements that will be used in computation in processor p at a node n in the interval flow graph. Similarly $\textsc{Incore}_n^p$ denote the set of elements read by a processor p at node n. Definitions $\textsc{Active}_n^p$ and $\textsc{Incore}_n^p$ are used to compute the send-recv sets for each processor, $\textsc{Send}_n^p$ and $\textsc{Recv}_n^p$. Using $\textsc{Send}_n^p$ and $\textsc{Recv}_n^p$, we can compute the set of elements communicated at a node n, $\textsc{Comm}_n$ as $\bigcup_i \{\textsc{Send}_n^i + \textsc{Recv}_n^i\}$. Similarly, we compute the set of incore elements at node n, $\textsc{Incore}_n$, as $\bigcup_i \textsc{Incore}_n^i$. For every node n, for every processor p and $\textsc{Send}_n^p$, we define $\textsc{Eio}_n^p$ as a set of elements which will be sent by p but are not members of $\textsc{Incore}_n^p$. Formally, $\textsc{Eio}_n^p = \textsc{Send}_n^p - (\textsc{Incore}_n^p \bigcap \textsc{Send}_n^p)$.

For any data set $d \in$ INCORE or SEND or RECV, the following predicates are defined. Bit vectors are used to represent individual data sets.

- $\mathit{Used}(n,d) \overset{df}{=}$ TRUE *iff* a subset of d is referenced at node n.
- $\mathit{Kill}(n,d) \overset{df}{=}$ TRUE *iff* a subset of d is modified at n.
- $\mathit{Incore}(n,d) \overset{df}{=}$ TRUE *iff* a subset of d is in-core at n.

3 I/O and Communication Optimization: An Example

Figure 1:2 presents an HPF example in which an out-of-core array `a` is distributed over 4 processors in `BLOCK` fashion. This example will be used as a running example throughout the paper. Our running example performs one-dimensional relaxation using 3-point stencil computations. The interior points of the array `a` are updated using a `FORALL` construct. To preserve the `FORALL` semantics, it is necessary to use temporaries to store initial and intermediate data. Since the primary data sets are stored in files, it is necessary to use two different files, the source local array file (LAF) for reading initial data and a temporary LAF to store the updated intermediate data. After the computation is over, the temporary LAF can be renamed as the source LAF.[4]

Figure 1:3 shows the pseudo-code for the stripmined program (assuming per processor available memory as 10). There are two stripmined iterations, each iteration reads the initial data from the source file into an in-core local array (ICLA) `temp` and writes the intermediate results from an ICLA `temp1` to the temporary file. Each iteration, after reading the ICLA, performs communication (if required). For example, in the first iteration, processors 0,1 and 2 send elements `a(16)`,`a(32)` and `a(48)` to processors 1,2, and 3 respectively. In the second iteration, processors 1,2, and 3 send elements `a(17)`, `a(33)` and `a(49)` to processors 0, 1, and 2. Note that this is an example of the **Receiver-driven In-core communication** method [Bor96].

Figure 1:4 shows the initial communication and input/output placement. The communication and input/output sets for each processor are given in global name

[4] A more detailed description is provided in [Bor96].

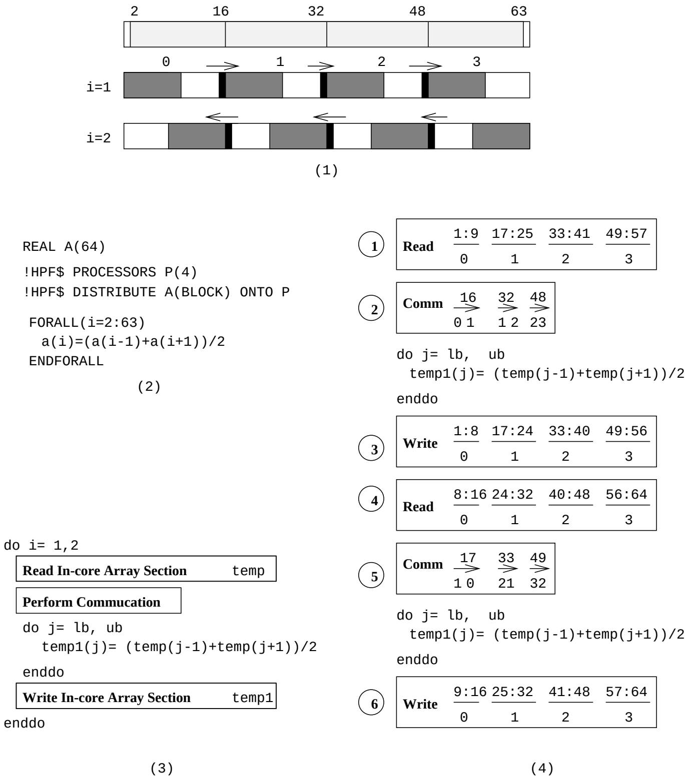

Fig. 1. Example program.

space while the bounds for the in-core computation are given in the local stripmined space (i.e., `lb`=1 and `ub`=8). For example, $\mathbf{Read}\frac{1:9}{0}$ means that processor 0 is reading elements `a(1)` to `a(9)`, **Comm** $0 \stackrel{16}{\rightarrow} 1$ represents communication of element `a(16)` from processor 0 to processor 1 and $\mathbf{Write}\frac{1:8}{0}$ means that processor 0 writing elements `a(1)` to `a(8)`.

From the computation pattern, it is easy to determine the communication pattern for each stripmined iteration [Bor96]. For example, in the first iteration, processor 0 needs to send element `a(16)` to processor 1. Since processor 0 does not have element `a(16)` in memory, however, it needs to read it from the LAF and send it to processor 1. Similarly, processors 2 and 3 need to read elements `a(32)` and `a(48)` from their LAFs and send them to their respective destinations. These file reads are termed as *extra* since the read elements are not required for computation by the owner processor. In the second iteration, processors 1, 2, and 3 perform also extra file accesses to read elements `a(17)`, `a(33)` and `a(49)` respectively. To prevent violation of `FORALL` semantics, *old values* of elements, `a(17)`, `a(33)`, and `a(49)`, are read from the source LAF and communicated to appropriate processors. It should be observed that elements `a(17)`, `a(33)` and a(49) are brought into memory in the first iteration and could be communicated *before* or *after* they are overwritten; thus minimizing extra file accesses. The example also performs redundant reads of some elements. For example, in the first iteration, processor 0 reads elements a(1) to a(9), but writes modified values of elements a(1) to a(8) *while retaining the old set of elements*, a(1) to a(9) *in form of the temporaries.*[5] In the second iteration, processor 0 again reads the *old* values of elements a(8) and a(9). Therefore, these two reads are *partially redundant*. These partially redundant reads can be eliminated if it is possible to determine which elements can be reused across iterations.

As observed before, for our running example, communication requires both inter-processor communication (i.e., communication of in-core data) and file I/O. To improve the communication cost, it is very important to minimize the file I/O cost (or the number of file accesses). The file accesses generated by the program can be classified into: (1) *Compulsory*: These accesses are required to read and write in-core data and (2) *Extra*: These accesses are required for communicating off-processor out-of-core elements. The file I/O cost can be reduced by (1) eliminating partially redundant compulsory file accesses and (2) minimizing extra file accesses by communicating *in-core* data whenever possible. The second optimization requires reordering computation and placing the communication calls so that only in-core data is communicated [Bor96]. In an out-of-core application, the computation order is decided by the data access pattern, that is, by *placement of the read/write calls*. Therefore, to minimize overhead due to file I/O in communication, it is important that both communication and I/O calls are placed at appropriate positions.

4 A Framework for Integrated I/O and Communication Placement

In Section 3, we describe the compilation of an out-of-core FORALL statement. We observe that the implementation of out-of-core FORALL requires extra file accesses during communication and a naive implementation results in reading redundant data. In this section, we propose an integrated I/O and communication placement framework that exploits the DELAYED_KILL property of the FORALL construct and applies the array access information for improving the overall performance. Note that the indeterminacy in FORALL execution order, allows our framework to freely reorder in-core computations. Specifically, our framework reorders in-core computation such that communication would involve only inter-processor communication. Consequently, all extra file accesses will be eliminated.

4.1 The Correctness Criteria

Our integrated framework imposes the following correctness requirements:

[5] Note that temporaries are marked LIVE during the FORALL computation.

- *Safety*: All data either communicated or read is used *immediately*.
- *Sufficiency*: Every in-core computation is preceded by an appropriate READ call and each non-local reference is preceded by appropriate communication.
- *Balance*: For every SEND, there is exactly one matching RECV. Note that this condition does not apply for READ.[6]

In the presence of the DELAYED_KILL type of computation, the definition of Safety is considerably weakened. Hence, it is more appropriate to term it as *Weak Safety*. Note that *Weak Safety* and *Sufficiency* are applicable for both file access and communication calls, while *Balance* is applicable only to communication calls. Therefore, our framework is able to take an unified approach for placing file access and communication calls while honoring their individual characteristics.

4.2 Eliminating Extra File Accesses in Communication

It should be observed that extra file accesses are generated because an array section[7] is used several times in the stripmined `FORALL` iterations; once by the processor that owns the section and in remaining cases, by other processors. If it is possible for the processors to perform computation on the common array section in the same iteration, the communication will involve only inter-processor data transfer and extra file accesses could be eliminated. To satisfy this condition, we add the following constraint in the correctness criteria.

Strict Safety Constraint

- *Strict Safety*: Everything that is *read* or *communicated* (i.e., *sent* and *received*) will be used only once.

Criteria *Safety* and *Strict Safety* require that the data read by processor i at node n, INCORE^i_n, should be used *immediately* and should not be used anywhere else in the computation. Computation in any processor, j, at node n', which requires elements of INCORE^i_n (in other words, $\text{RECV}^j_{n'} \subset \text{INCORE}^i_n$), should, therefore, be placed at node n. Then, processor i needs to send only the incore data ($\text{SEND}^i_n \subset \text{INCORE}^i_n$). Applying this condition to every processor, we can observe that if node n satisfies *Strict Safety*, COMM_n is subsumed by INCORE_n and therefore, set EIO is empty and all extra I/O is eliminated.

Processors i, j satisfying the above requirements exhibit one or both of the following *inclusion* properties

- $\text{RECV}^j_{n'} \subset \text{SEND}^i_n \rightarrow \text{RECV}^j_{n'} \subset \text{INCORE}^i_n$
- $\text{RECV}^i_n \subset \text{SEND}^j_{n'} \rightarrow \text{RECV}^i_n \subset \text{INCORE}^j_{n'}$

where n and n' are nodes of the interval flow graph denoting the initial placement of the computation (in other words, placement of READ calls). To

[6] We currently use synchronous I/O calls.
[7] An element can be considered as a special case of section.

find i, j and n, n', it is necessary to perform both FORWARD and BACKWARD flow analysis.

Let us now define a predicate $Incl^i_j(n, n')$ as follows:

– $Incl^i_j(n, n') \stackrel{df}{=}$ TRUE if $\text{RECV}^j_{n'} \subset \text{INCORE}^i_n$ or $\text{RECV}^i_n \subset \text{INCORE}^j_{n'}$

For a processor i, the solution of the $Incl^i_j(n, n')$, for any processor j ($j \neq i$), gives the node pair (n, n') satisfying the inclusion properties. The inclusion property is then verified for every INCORE and RECV set in the program. If all the INCORE and RECV sets satisfy the *inclusion* property, then the computation is said to be *balanced*. For balanced computation, one can eliminate extra I/O by reordering computations.

We illustrate this optimization by using our running example (Figures 1). Table 1 illustrates the values of various dataflow variables corresponding to the stripmined iterations (Figure 1). There are two stripmined iterations; for each iteration, INCORE gives the set of elements that are brought in memory by each processor (ICLA). Corresponding ACTIVE, SEND and RECV sets are also shown.

Table 1. Dataflow Variables for the running example.

Iter.	Processor	Node	INCORE	ACTIVE	SENT	RECV
1	0	2	1:9	1:9	16	-
	1	2	17:25	16:25	32	16
	2	2	33:41	32:41	48	32
	3	2	49:57	48:57	-	48
2	0	6	8:16	8:17	-	17
	1	6	24:32	24:33	17	33
	2	6	40:48	40:49	33	49
	3	6	56:64	56:64	49	-

Table 2 presents the solutions for the *Incl* predicate for all processors in form of the *Inclusion* matrix. An entry (n, n') in a position $[i, j]$ denotes the pair of nodes of the interval flow graph satisfying the inclusion equations for the processors i and j. This entry is called as a *solution* entry. In other words, it defines the INCORE sections of processors i and j that satisfy the inclusion property. For example, consider the solution at position [2,1]. The solution tuple (2,6) denotes that $\text{RECV}^2_2 \subset \text{INCORE}^1_6$, i.e., the data required by the ICLA of processor 2 at node 2 (first stripmined iteration) is part of the ICLA of processor 1 at node 6 (second stripmined iteration). The entries in the positions [0,0] and [3,3] denote that processors 0 and 3 do not perform communication at nodes 2 and 6 respectively (in other words, in the first and second stripmined iteration). Such entries are called *non-solution* entries. The number of *solution* entries in

i^{th} row or j^{th} column denotes the number of times a processor i or j performs communication.

Table 2. Inclusion matrix for the running example.

Processor	Processor			
	0	1	2	3
0	(2,2)	(2,6)	-	-
1	(2,6)	-	(6,2)	-
2	-	(2,6)	-	(6,2)
3	-	-	(2,6)	(6,6)

The information provided by Table 2 can be used to reorder the computation. This reordering is an iterative procedure; every iteration tries to schedule computation such that the inclusion equations are satisfied. The iterations stop when all ICLAs represented by the solution tuples are scheduled. Let us understand the reordering procedure using our running example and its inclusion matrix.

1. In the first step, choose a random processor i. For our problem, let us choose processor 2. For this processor, select a solution entry from the second row, e.g., entry [2,1] which corresponds to the solution tuple (2,6). It states that $\text{R}\textsc{ecv}_2^2 \subset \text{I}\textsc{ncore}_6^1$. Therefore, sections of local arrays of processors 2 and 1, corresponding to the nodes 2 and 6 (in the interval flow graph) should be brought in memory.
2. In the second step, using the inclusion matrix, determine if the ICLA of processor 1 requires any off-processor data. It can be easily found out by checking the first row of the inclusion matrix for solution entries containing node 6. The entry [1,2] corresponds to the solution tuple (6,2), which indicates that $\text{R}\textsc{ecv}_6^1 \subset \text{I}\textsc{ncore}_2^2$. Note that the array section of processor 2, corresponding to node 2, is already in memory. Therefore, the communication between processors 1 and 2 will involve only inter-processor communication.
3. The first two steps have scheduled ICLAs of processors 1 and 2. The third step tries to schedule ICLAs of the remaining processors so that there are no extra I/O accesses. Consider processor 0. In the 0^{th} row, the only solution entry involves processor 1 at node 2. Since ICLA of processor 1 at node 2 is already scheduled, this entry cannot be used. In this case, the non-solution entry, i.e., entry at position [0,0], [2,2], should be used. This non-solution entry suggests that the ICLA of processor 0 at node 2 does not require communication and therefore, can be scheduled along with ICLAs of processors 1 and 2. Applying the same principle to processor 3, we can see that ICLA of processor 3 at node 6 does not require communication. Hence,

this ICLA can be scheduled along with the ICLAs of processor 0, 1 and 2. For this ICLA schedule, only communication required will be inter-processor communication between processors 1 and 2.

4. Applying the same procedure, the remaining four ICLAs can be scheduled. This ICLA schedule will involve interprocessor communication between processors 0 and 1, and between processors 2 and 3. Therefore, the overall computation involves only inter-processor communication and the extra I/O accesses are eliminated. Figure 2:A illustrates the final placement of I/O and communication calls. Figure 2:B illustrates an alternative placement. This placement is obtained using a different choice of initial processor.

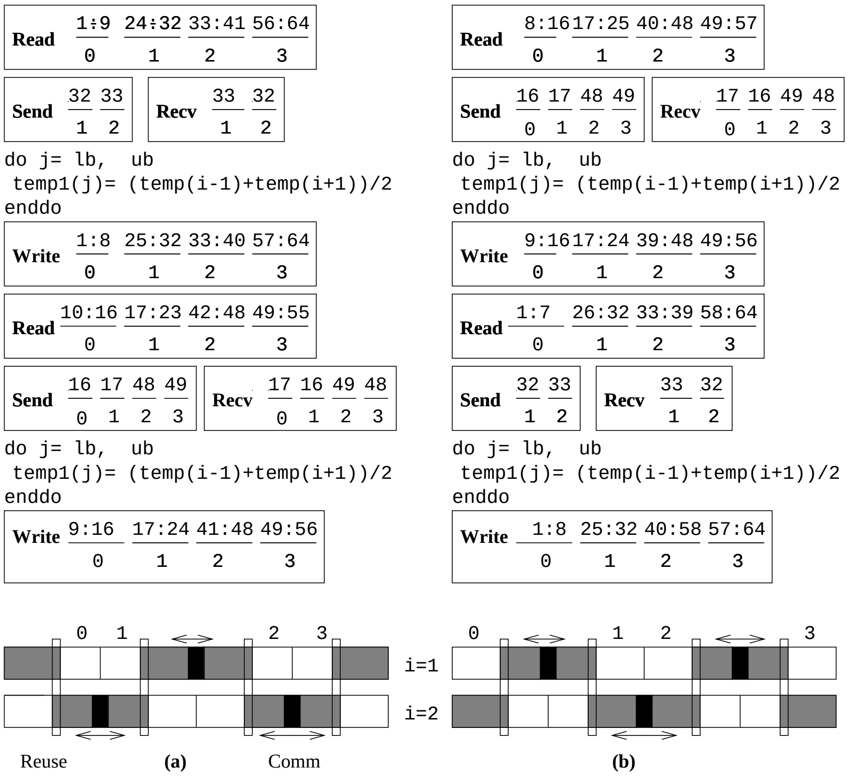

Fig. 2. Final placement of I/O and communication calls.

5 Applying Dataflow Framework to Stencil Problems

We now apply the communication and I/O placement framework to the stencil problems. We illustrate using the 5- and 9-point stencils (Figure 3 (1) and (2)).

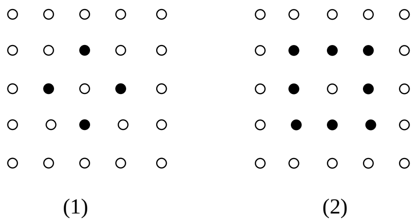

Fig. 3. 5- and 9-point Stencils

This section presents performance results of hand-coded out-of-core examples that use 5-and 9-point stencils. The experiments were performed for square real arrays of size 8K*8K (aggregate file sizes 256 Mbytes), distributed in `BLOCK-BLOCK` fashion over processors logically arranged as a square mesh. These experiments are performed using 16 and 64 nodes of an Intel Paragon.

Tables 3 present performance results for column, and square tiles. In each experiment, the amount of time required to read and write local data, LIO, and the time required for performing communication, COMM, were measured for unordered and ordered (after placing the I/O and communication calls) access patterns and the communication gain was computed. Each table presents LIO and COMM for 5- and 9-point stencils with different processor grids and different array sizes. Since the local computation time is negligible compared to LIO, we have not reported the computation cost. Each experiment was performed for the memory ratio of $\frac{1}{4}$ (i.e., the ratio of size of available memory to that of out-of-core array). Note that for the unordered cases, COMM includes the cost of inter-processor communication and extra file I/O.

From Table 3, we can observe that by reordering communication and I/O calls, the communication cost COMM is significantly reduced. For example, for a 9-point stencil problem running on 64 processors using 8K*8K array and column tiles, COMM without ordering is 2.06 seconds, and with ordering is 0.05 seconds (therefore, the communication gain is 39). For the same problem, if square tiles are used, the communication gain is 35992. This increase in the gain is due to the additional I/O cost incurred during accessing square tiles.

6 Conclusions

In this paper, we described a framework for optimizing communication and I/O costs in out-of-core problems. We focussed on communication and I/O optimization within a `FORALL` construct. We showed that existing frameworks do not extend directly to out-of-core problems and can not exploit the `FORALL` semantics. We presented a unified framework for the placement of I/O and communication calls and applied it for optimizing communication for stencil applications. Using the experimental results, we demonstrated that correct placement of I/O and communication calls can completely eliminate extra file I/O from communication and as a result, significant performance improvement can be obtained.

Table 3. Performance of the 5-and 9-point stencils. time in seconds.

Memory ratio	Procs.	**Unordered** COMM a	LIO b	**Ordered** COMM c	LIO d	Comm Gain e=(a/c)
5-point Stencil, Column Tiles, 8K*8K Array						
1/4	16	**1.38**	7.57	**0.04**	6.71	**35.38**
1/4	64	**1.16**	10.87	**0.05**	9.80	**25.21**
9-point Stencil, Column Tiles, 8K*8K Array						
1/4	16	**1.27**	7.70	**0.03**	7.05	**38.48**
1/4	64	**2.06**	10.20	**0.05**	10.15	**39.84**
5-point Stencil, Square Tiles, 8K*8K Array						
1/4	16	**175.11**	183.96	**0.03**	178.14	**5506.60**
1/4	64	**192.2**	197.57	**0.005**	196.40	**36061.79**
9-point Stencil, Square Tiles, 8K*8K Array						
1/4	16	**150.78**	175.88	**0.03**	182.01	**4569.09**
1/4	64	**192.2**	197.57	**0.005**	196.40	**35992.51**

Acknowledgments

The work of R. Bordawekar and A. Choudhary was supported in part by NSF Young Investigator Award CCR-9357840, grants from Intel SSD and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from Advanced Research Projects Agency(ARPA) administered by US Army at Fort Huachuca. R. Bordawekar is also supported by a Syracuse University Graduate Fellowship. The work of J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457768, an NSF grant CCR-9210422 and by the Louisiana Board of Regents through contract LEQSF(1991-94)-RD-A-09. This work was performed in part using the Intel Paragon System operated by Caltech on behalf of the Center for Advanced Computing Research (CACR). Access to this facility was provided by CRPC.

References

[Bor96] Rajesh Bordawekar. *Techniques for Compiling I/O Intensive Parallel Programs.* PhD thesis, Electrical and Computer Engineering Dept., Syracuse University, April 1996.

[GSS95] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 1995.

[Hig93] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[KN94] Ken Kennedy and Nenad Nedeljković. Combining Dependence and Data-Flow Analyses to Optimize Communication. Technical Report CRPC-TR94484-S, CRPC, Rice University, September 1994.

[KS95] Ken Kennedy and Ajay Sethi. A Constraint Based Communication Placement Framework. Technical Report CRPC-TR95515-S, CRPC, Rice University, February 1995. Revised May 1995.