Efficient Program Partitioning based on Compiler Controlled Communication[1]

Ram Subramanian and Santosh Pande

Dept. of ECECS, P.O. Box 210030, Univ. of Cincinnati, Cincinnati, OH45221
E-mail: {rsubrama, santosh}@ececs.uc.edu, correspondence: santosh@ececs.uc.edu

Abstract. In this paper, we present an efficient framework for intra-procedural performance based program partitioning for sequential loop nests. Due to the limitations of static dependence analysis especially in the inter-procedural sense, many loop nests are identified as sequential but available task parallelism amongst them could be potentially exploited. Since this available parallelism is quite limited, performance based program analysis and partitioning which carefully analyzes the interaction between the loop nests and the underlying architectural characteristics must be undertaken to effectively use this parallelism.
We propose a compiler driven approach that configures underlying architecture to support a given communication mechanism. We then devise an iterative program partitioning algorithm that generates efficient program partitioning by analyzing interaction between effective cost of communication and the corresponding partitions. We model this problem as one of partitioning a directed acyclic task graph (DAG) in which each node is identified with a sequential loop nest and the edges denote the precedences and communication between the nodes corresponding to data transfer between loop nests. We introduce the concept of **behavioral edges** between edges and nodes in the task graph for capturing the interactions between computation and communication through parametric functions. We present an efficient iterative partitioning algorithm using the behavioral edge augmented PDG to incrementally compute and improve the schedule. A significant performance improvement (factor of 10 in many cases) is demonstrated by using our framework on some applications which exhibit this type of parallelism.

1 Introduction and Related Work

Parallelizing compilers play a key role in translating sequential or parallel programs written in high level language into object code for highly efficient concurrent execution on a multi-processor system. Due to the limitations of static dependence analysis especially in the inter-procedural sense, many loop nests are identified as *sequential*. However, available task parallelism could be potentially exploited. Since this type of available parallelism is quite limited, performance

[1] This work is supported by the National Science Foundation grant no. CCR-9696129 and DARPA contract ARMY DABT63-97-C-0029

based program analysis and partitioning which carefully analyzes the interaction between the loop nests and the underlying architectural characteristics (that affect costs of communication and computation) must be undertaken to effectively use this parallelism. In this work, we show that the compiler, if given control over the underlying communication mechanism, can significantly improve runtime performance through careful analysis of program partitions.

When the object code executes on the underlying hardware and software subsystems, a complex interaction occurs amongst them at different levels. These interactions could be from a much finer level such as the multi-function pipelines within a CPU, to a more coarser level such as at the interconnection network, cache and/or at the memory hierarchy level. Unfortunately, parallelizing compilers have very little or no knowledge of the actual run time behavior of the synthesized code. More precisely, most optimizing parallel compilers do not have a control over the underlying architecture characteristics such as communication mechanism to match their sophisticated program analysis and restructuring ability. Due to this reason, the actual performance of the code may be different (often worse) than the one anticipated at compile time [11, 12]. The above problem of exploiting task parallelism amongst sequential loop nests is especially susceptible to this limitation since the amount of available parallelism is small. However, it can be effectively exploited if the compiler is given control over key architecture features. The compiler could then undertake program partitioning driven by performance model of the architecture. This is the motivation behind our work in this paper.

Most of the reported work in the parallelizing compilers literature focuses on analyzing the program characteristics such as the dependences, loop structures, memory reference patterns etc. to optimize the generated parallel code. This is evident from the theory of loop parallelization [1] and other techniques for detecting, transforming and mapping both the loop and task parallelism [3, 5, 8, 6, 13].

Current approaches to performance enhancement involve the use of a separate performance evaluation tool such as Paradyn [15], to gather performance statistics. Another important effort in this area is Pablo [16] which uses a more traditional approach of generating and analyzing the trace of program execution to identify performance bottlenecks. Wisconsin Wind Tunnel (WWT) project has contributed an important discrete event simulation technique that results in accurate and fast performance estimation through simulation [20]. However, the drawback of these tools is that they demand that a user be responsible for guiding the estimator tool to elicit high performance.

Performance estimation of a parallel program has been the focus of research by parallelizing compiler researchers for a while. Sarkar [7] first developed a framework based on operation counts and profile data to estimate the program execution costs for Single Assignment Languages. Another important effort from Rice University involved development of *Parascope Editor* [17] in which the user annotates the program and is given a feed-back based on performance estimates to help the development process. However, the use of compilation techniques for

estimation was a complex problem due to the complex run time behaviors of the architectures. The communication overhead is especially tricky to model and the complexity of modeling its software component has been shown [18].

Some studies have indicated the potential of compiler analyses for performance prediction. The P3T system based on Vienna Fortran [14] was the first attempt in using compiler techniques for performance estimation for such systems. However, the approach focussed more on the effect of the compiler optimizations and transformations on the performance rather than on relating it to the behavior of the underlying architectural components. For example, P3T based on Vienna Fortran Compiler (VFC) estimates the effect of loop unrolling, tiling, strip-mining and a whole variety of loop transformations to choose their best ordering on a given architecture. Our work can be contrasted with theirs in that we study the relationship between architecture configuration and partitioning to effectively map a limited amount of task parallelism available across sequential loop nests. Our approach allows the interactions between costs and partitions to be analyzed to refine partitions. The input to our scheme can be any scheduling algorithm such as STDS [2], Chretienne's [9], DSC (Dominant Sequence Clustering) [10], and so on.

The organization of this paper is as follows: section 2 presents a motivating example, section 3 describes our solution methodology, section 4 explains our algorithm, section 5 includes results and discussion, followed by conclusions in section 6.

2 Motivating Example

We now introduce the motivation behind our work through an example. Figure 1(a) shows a code segment consisting of loop nests. The loop nests execute sequentially due to the nature of data dependencies inside the loops. Figure 1(b) gives the task graph (DAG) such that each node $v_i \in V$ represents a task and the directed edge $e(v_i, v_j) \in E$ represents the precedence constraint from v_i to v_j, that corresponds to the actual data dependence edge. The node computation costs and the edge communication costs are shown next to each node and edge respectively. The computation cost of node v_i is denoted by $t(v_i)$ and the communication cost of edge $e(v_i, v_j)$ is denoted by $c(v_i, v_j)$. The node and edge costs are found using operation counts and the sizes of the messages and assuming a certain cost model. Typical cost models for message passing use a fixed start-up cost and a component proportional to size of message. We first show how to find optimal partitioning assuming an ideal architecture in which cost of each message is independent of any other concurrent messages. We then show how a compiler could choose a communication protocol and improve the performance.

In order to minimize the completion time, it is desirable to schedule the successor of a node on the same processor to eliminate the communication cost on the edge. We, however, still continue to pay communication costs on the other edges. The completion time found by using this strategy is given by

$$t(v_i) + \min_{j:e(v_i,v_j)\in E} \max_{k:k\neq j, e(v_i,v_k)\in E} (t(v_j), t(v_k) + c(v_i, v_k)) \qquad (1)$$

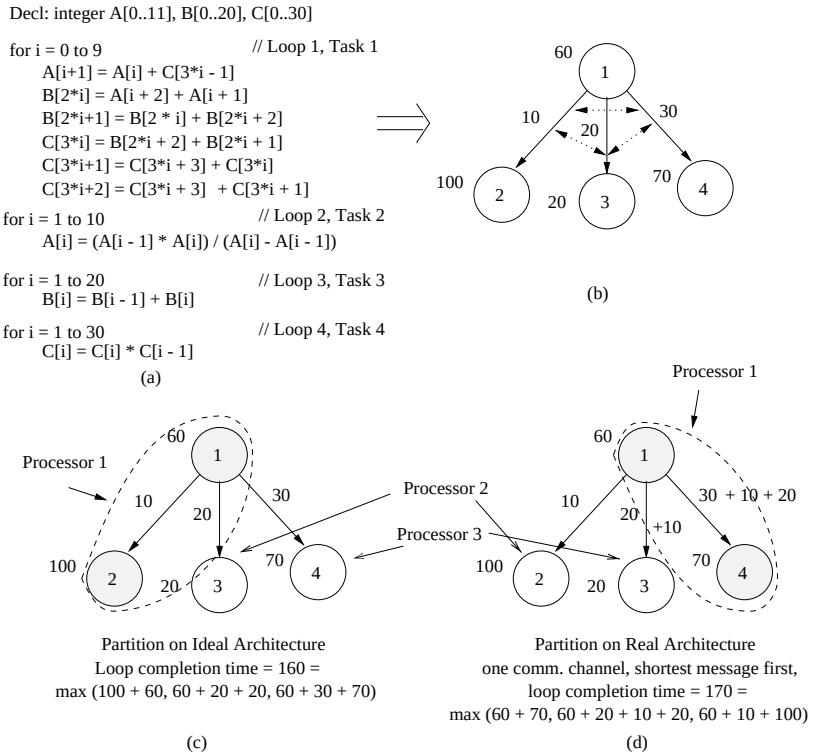

Fig. 1. An Example Graph

Using the ideal architecture (in which the cost of each message is independent of any other concurrent messages), the optimal partition found using the above strategy is given in Figure 1(c). However, on a real machine, the costs of concurrent messages could be heavily influenced by limitations of number of channels, communication protocol used, and so on. The behavior is shown by dotted lines in Figure 1(b). For example, if there is only one communication channel available per processor such as in a cluster of workstations topology, it will be time-multiplexed between different concurrent messages which will have an influence on costs of communication. In such cases, to increase throughput, the compiler could choose a *Shortest Message First* protocol. Under this condition, the optimal partition changes to the one shown in Figure 1(c). The actual completion time of the partition shown in Figure 1(c) using this protocol is 180 as against 160 on ideal machine; whereas the completion time of partition shown in Figure 1(d) found using actual message-passing protocol used (shortest message first) is 170, which is optimal. This shows that the choice of communication protocol also allows the compiler to refine its partition to get better performance.

3 Outline of Our Approach

In order to efficiently partition tasks on a set of processors, an accurate model of program behavior should be known. Accurate modeling of program behavior on a given architecture is a very complex task. The complexities involve latencies, cache behaviors, unpredictable network conflicts and hot spots. A factor which affects partitioning decisions are relative costs of communication and computation which are decided by loop bounds and data sizes. In many cases, loop bounds could be estimated through symbolic tests [19]; similarly the maximal data footprint sizes could be determined at compile time using Fourier-Motzkin elimination and bounds on static array declarations. As illustrated in the motivating example, these costs are themselves dependent on partitioning decisions. We assume that suitable analysis outlined as above is done to expose the desired costs for a series of sequential loop nests.

We propose a unified approach that tends to bring the program behavioral model and the architectural behavioral model as close as possible for accurate program partitioning. We develop an Augmented Program Dependence Graph that can represent a detailed execution behavioral description of the program on a given parallel architecture using *behavioral edges* that represents the dependencies and the interactions of the computation or communication. We then develop an extensive framework for efficient performance-based program partitioning method for the task parallelism represented in the augmented PDG.

3.1 Intermediate form for performance analysis

Sequential loops in the program are identified as tasks to be executed on processors. A dependence relationship between various tasks (sequential loops) is captured by the task graph, or the Program Dependence Graph (PDG). The PDG is a tuple, G(V,E,C(V), C(E)), where, V is the set of nodes, E is the set of edges, C(V) is the set of node computation costs and C(E) is the set of edge communication/synchronization costs. These costs are primarily based on the operation counts and the number of data items to be communicated and hence are simplistic and error prone. We develop a more accurate model of each architectural component such as buffer set-up delays, network set-up delays and transmission delays and based on this model, add several parametric annotations to the basic PDG to reflect the complex inter-dependent behaviors. In order to capture the dependences of these parameters on each other, we introduce an extra set of edges called **behavioral edges** that denote the direction and degree of the interaction. The degree of interaction is captured by a partial function of all the involved variables. Behavioral edges are shown in Figure 2 by dotted lines.

Behavioral edges could be of four types which denote the interaction of computation on both computation(nodes) and communication(edges), and interaction of communication on both computation and communication. Thus, a behavioral edge could have a source or sink on a PDG computation node or on a PDG precedence edge. For example, if communication on a PDG precedence

edge is affected by communication on another PDG precedence edge, we add a behavioral edge between the two. This edge bears a partial function involving the number of available channels, buffers, network delays, sizes of the messages and the routing parameters that reflects the interactions of these two communications. If the interaction is symmetric, the edge added is bidirectional with the same function, otherwise, it is unidirectional and there are two such edges and two partial functions. The partial functions map the original cost on an edge or a node to an appropriately modified cost using the underlying parameters and the other interacting costs. For example, a partial function $f(a, b, c)$ on a behavioral edge going from precedence edge $e1$ to $e2$, designates the modification in the cost of edge $e2$ due to the cost of edge $e1$ and the underlying parameters a, b and c. The augmented PDG is thus, a tuple $G(V, E, C(V), C(E), B(f))$, where $B(f)$ denotes a set of behavioral edges with the respective partial functions. These behavioral edges are used only for augmenting the PDG to perform cost-based program partitioning. They are not part of the precedence edges of the PDG. Thus, through this elaborate model of augmented PDG, we capture the interactions of computation and communication. Analyzing these interactions, a compiler is able to control a key architectural feature such as communication mechanism and perform program partitioning accurately. In the next subsection, we develop an example to illustrate the Augmented PDG.

3.2 Augmented PDG Example

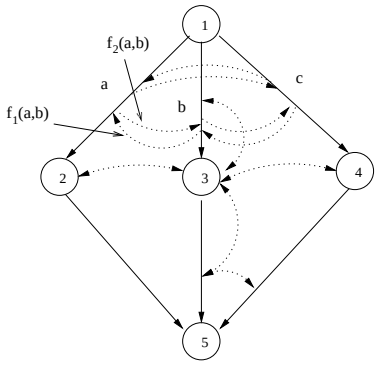

Fig. 2. An Example Augmented PDG

The solid edges in Figure 2 represent the precedence constraints on the computation and the dotted ones represent the behavioral edges between two nodes or two precedence edges or between a node and an edge. In this example, the communication costs on edges (1,2), (1,3) and (1,4) are inter-dependent since the communication may occur concurrently. This inter-dependence is designated by six directed behavioral edges between these edges. We need to have two directed

behavioral edges instead of a bidirectional behavioral edge between the precedence edges since the interaction is not symmetric. Here, the partial functions f_1 and f_2 are shown representing two of these behavioral edges. If edges (1,2), (1,3) and (1,4) have costs a, b and c respectively, then the partial functions are functions of these variables. For example, in case the underlying architecture is configured to select the underlying message passing mechanism as "shortest message first", then

$$f_1(a,b) = \begin{cases} a+b \text{ if } b < a \\ a \quad \text{otherwise} \end{cases} \text{ and } f_2(a,b) = \begin{cases} a+b \text{ if } a < b \\ b \quad \text{otherwise} \end{cases}$$

The incoming communication on node 3 can perturb its ongoing computation and thus, there is an edge between edge (1,3) and node 3. Similarly the cache performance of computation at node 2 will be affected if merged with 3 and the same is true with the merger of nodes 3 and 4. This is designated by the respective behavioral edges between these nodes. However, no such performance degradation possibly occurs for nodes 2 and 4 and hence there is no edge. Similarly, maybe the outgoing communication on edge (3,5) can be improved if the data is directly written in the communication buffer and thus, there is a behavioral edge between the node 3 and edge (3,5). There is also a behavioral edge between edge (3,5) and (4,5) to denote the interaction of communication between them.

3.3 Program Partitioning

The key goal of this phase is to partition the PDG using the results of the previous analysis so that partitions are tuned to communication latencies.

Being a precedence constrained scheduling problem, the problem of finding the optimum schedule for the complete PDG in general is NP-hard [21]. Our goal here is to use architecture specific cost estimates to refine the partition.

4 An Iterative Algorithm for Partitioning

One of the most important reasons for changes in schedule times of different nodes at run-time, is the change in the communication costs of the edges. As mentioned before, edge costs vary dramatically depending on the communication mechanisms of the architecture. Concurrent communication may be affected by *Shortest Message First, First Come First Served (FCFS)* protocols etc. A compiler may choose a particular communication scheme to configure the architecture depending upon the communication pattern. For example, in case of concurrent communication being known at compile time, a shortest first ordering may be useful to minimize the average message passing delays. On the other hand, if at compile time, it is not known whether concurrent communication will occur, then an FCFS scheme may be relied upon to increase throughput. The choice of communication mechanism potentially affects the edge costs and hence

affects the task schedules. The schedule chosen is also closely related to the hardware support. An example of this can be seen in the IBM SP2 scalable cluster. It is a cluster of processors connected by a single high performance switch (HPS). This means that if a processor sends two messages to two processors concurrently, the messages will not be routed the same way, but instead, will be routed sequentially, depending on the underlying communication layer. This changes the communication costs on the edges depending on the message size. On the other hand, on a CRAY T3E, the 3D torus interconnect makes it possible for each processor to communicate to 6 neighboring processors concurrently. Once these communications are fixed, the compiler attempts to refine the schedule. Here, the costs of the communication edges will not change as long as there is no contention. Hence, our algorithm tries to involve these communication cost changes to iteratively determine a schedule tuned to the costs.

In our algorithm, pairs of concurrent communication edges are identified and behavioral edges are associated with each pair. The effect of the underlying communication model (shortest message first or FCFS) is used to determine the communication costs of these edges. For example, assume that the compiler chooses shortest message first communication mechanism. Let an edge e_1 in the PDG be connected to edges $e_2, e_3, \ldots, e_i$ by means of behavioral edges. If edge cost $c(e_1)$ is the least among the above edges, e_1 would be communicated first and the communication costs of the rest of the edges will increase by $c(e_1)$. The increase in edge costs would, in turn, worsen the schedule. A recomputation of the schedule has to be done under this circumstance, and the new schedule could in turn affect the behavioral edges. A two way dependence exists between the schedule and the edge costs. Thus, this algorithm iteratively refines the dependence between behavioral edges and schedule, by computing a schedule using behavioral edges and uses this schedule in turn to compute the effect of cost changes due to behavioral edges. In this way the algorithm arrives at a stable schedule. A given scheduling algorithm thus, can be used as an input to our iterative algorithm, which our algorithm invokes to compute the schedule. Other inputs to our iterative refinement algorithm are: the task graph G, message passing model M chosen by the compiler analyzing the communication pattern, and the convergence limit Ψ [2]. Our algorithm demonstrates an effective method for finding operational behavioral edges, changing costs appropriately and iteratively minimize the schedule for an input task graph.

Algorithm:

`Iterative_Partition()`
`Input:` A $\equiv$ scheduling algorithm, G $\equiv$ task graph - $G(V, E, C(V), C(E)$
M $\equiv$ message passing model (SMF, FCFS), $\Psi \equiv$ limit
`Output:` schedule S
`Begin`
C1(E) $\leftarrow$ C(E); {*Initial edge costs are denoted as C1(E)*}

[2] The convergence of this algorithm is slow in some cases; thus, we impose Ψ as a limit imposed on the differences between successive schedule lengths.

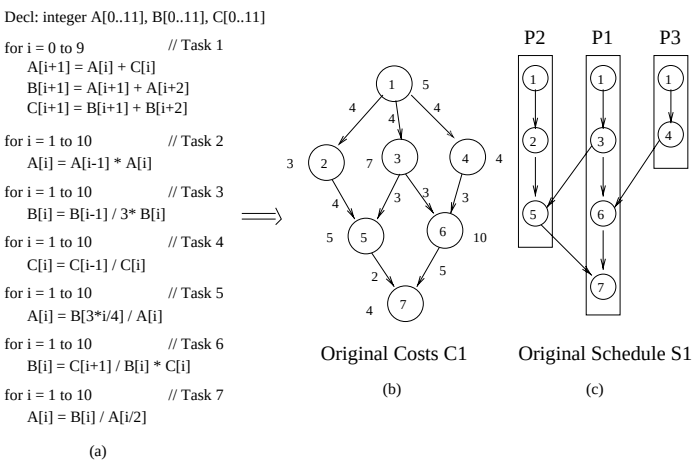

Fig. 3. (a) An example code segment, (b) its task graph with costs C1 and (c) its schedule S1

S1 = call_A ($G(V, E, C(V), C1(E))$); {*call A, with cost C1 for sche. S1*}
call iter_schedule($G(V, E, C(V), C1(E)$, S1); {*iteratively refine schedule*}
End Iterative_Partition

iter_schedule()
Input: S1 ≡ schedule, G ≡ task graph - $G(V, E, C(V), C1(E)$
Output: S2 ≡ Final schedule, L2 ≡ Final schedule length
Denotation: *An edge from node i to node j is denoted by (i,j)*
Begin
{*First find behavioral edge pairs from schedule S1. beh_edge_pairs[i] is a list of edges connected to edge i by a behavioral edge. List ended with -1* }
repeat
for (all edges (i, j) in G) do
for (all other edges (u, v) in G) do
{ *A directed edge* (i, j) *has behavioral edges to all directed edges* (u, v) *such that node* $u \neq j$ *and u is not the successor of node j*}
if $((u \neq j) \wedge (u \neq SUCC(j)))$ then
if $((ect(i) \leq ect(u) < last(j)) \vee (ect(u) \leq ect(i) < last(v)))$ then
add edge (u, v) to beh_edge_pairs$[(i, j)]$
endif
endif
endfor
endfor
{ *SUCC(j) - successor node of node j; ect - earliest completion time;* }
{ *last - latest allowable start time of node* }
{*modify costs of edges connected by beh. edges*}

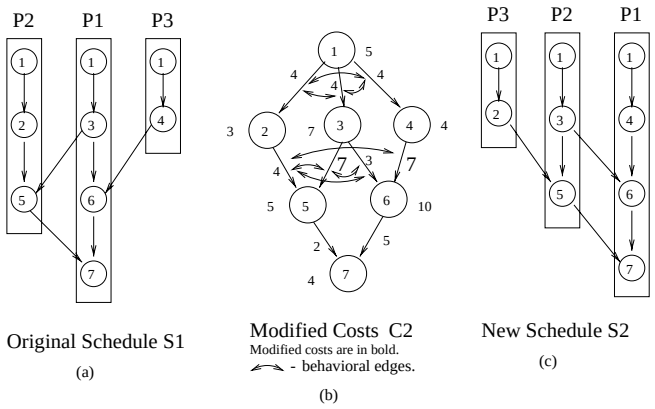

Fig. 4. Illustration of Algorithm

```
    for (each row r of beh_edge_pairs) do
      If (M = SMF) then
        sort edges of row r in ascending order of costs
      endif
      for (each edge (u, v) in beh_edge_pairs[(i, j)]) do
        {modify costs if u and v not assigned to the same processor }
        If ( Processor(u) ≠ Processor(v) ) then
            c((u, v)) = c((u, v)) + c((i, j));
        endif
      endfor
    endfor
    C2(E) ← C1(E); {Denote modified cost of edges as C2}
    S2 = call_A ( G(V, E, C(V), C2(E)) ); {call A, using C2 to get sche. S2}
    L1 = schedule_length(S2, C2);
    L2 = schedule_length(S1, C2); {Using C2 and S1 find schedule length L2}
  until ((convergence) ∨ (L2 - L1 > Ψ))
End iter_schedule
```

4.1 An Example

We now illustrate the working of the above algorithm through an example.

An example code segment consisting of sequential loops, its PDG and its associated schedule is shown in Figure 3(a), Figure 3(b) and Figure 3(c) respectively. It can easily be seen that each loop in the code segment is sequential and cannot be parallelized. When the graph in Figure 3(b) is input to the algorithm with costs C1 and assuming that the compiler has chosen First Come First Served to increase throughput, the schedule obtained is S1, using a scheduling algorithm A. In the figures shown, P1, P2 and P3 are 3 processors on which the corresponding nodes are scheduled. The arrows between the nodes in different

processors represent communication edges. Once schedule S1 is computed, the algorithm identifies behavioral edges between the task graph edges using the schedule S1. These behavioral edges are found depending on whether there exists a concurrent communication amongst the task graph edges. In this case, the behavioral edges found for S1 are shown in Figure 4(b). These behavioral edges are used by the algorithm to modify edge costs from C1 to C2. For the example DAG, the edges that have the costs modified are edges $(3, 5)$ and $(4, 6)$, with new costs of 7 each. The modified costs are also shown in Figure 4(b) in bold. Using the new costs C2, a schedule S2 is determined by the scheduling algorithm A, as illustrated in Figure 4(c).

The schedule S1 has schedule length 26 and uses 3 processors. Suppose the costs of the edges change to costs C2 due to interaction of communications. The schedule length obtained when S1 operates with C2 costs is 30. Our algorithm uses the modified costs C2 and gives a schedule length of 29, which is 1 less than the schedule S1 operating with costs C2. When the iteration progresses to completion, the costs are further refined and schedule lengths are minimized. The final schedule length obtained is 26.

The complexity of finding the pairs of behavioral edges is $O(e^2)$. If there are e edges in the graph, we have a worst case of $(e-1) + (e-2) + \ldots + 1 = e(e-1)/2$ behavioral edges. Since the algorithm examines every pair of edges, its complexity is $O(e^2)$. Modifying the costs incurs a worst case complexity of $O(e^2)$. The step of finding out concurrent communication edges using the earliest and latest, start and completion times, reduces the number of actual behavioral edges. This improves the average case of the algorithm to a large extent. Assuming the algorithm converges in k iterations, the overall complexity of the algorithm is $O(ke^2)$.

5 Results and Discussion

In this work, we used the STDS algorithm developed by Darbha et al. [2] as an input to our iterative algorithm for performance evaluation. We have chosen STDS algorithm due to its relative insensitivity to cost changes as demonstrated in [11]. Our motivation here is that the use of such a stable algorithm would demonstrate the essential merit of our iterative based approach, by factoring out the effect of sensitivity of the schedule to the algorithm.

Our algorithm was implemented on a SPARCstation 5. We first tested the algorithm with the BTRIX test program taken from NAS benchmarks [22] as input in the form of a task graph. The BTRIX test program consists of a series of sequential loop nests with one potentially parallel loop nest. Figure 5 shows the task graph obtained from the program. Due to limitations of inter-procedural analysis in the presence of procedure calls, loops corresponding to tasks 2 and 3 are marked as sequential loop nests. The loop corresponding to task 6 can be parallelized. Hence, we assume that the processor allocated to task 6 is responsible for scatter and gather of data to perform task 6. The computation time for task 6 is taken as the effective parallel time if all the steps are executed in

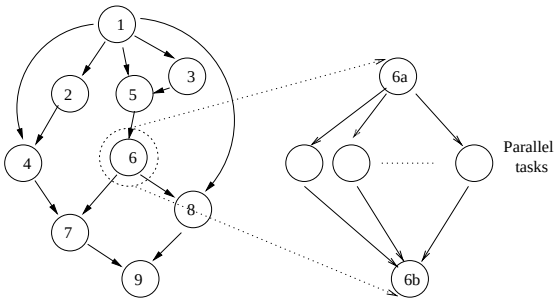

Fig. 5. Task Graph obtained from NAS benchmark code given in Appendix A

BTRIX Test Program					
Loop	L2	NP2	L1	NP1	Factor
30	38225	3	41225	3	1.08
50	66725	3	72825	3	1.09
80	111350	3	123600	3	1.11
100	142350	3	159700	3	1.12
130	190725	3	217225	3	1.14

Table 1. NAS kernel benchmark program: BTRIX - with varying loop counts

parallel. This is clearly illustrated in the graph in Figure 5. Task 6 has been expanded as synchronization nodes `6a` and `6b`, and intermediate parallel nodes, which can be partitioned using any partitioning approach. Results were obtained for the FCFS model by varying the loop counts in BTRIX test program These results are tabulated in Table 1. In all result tables, the following notations and derivations have been made: S1 is the schedule obtained from the STDS algorithm and C1 is the original costs of the task graph. When the costs change to C2 in one iteration of our algorithm, **L1** is the schedule length obtained when **S1** operates on **C2**. The rational for this is that in the STDS algorithm, without our iterative tuning, the tasks are scheduled in the order given by S1, but the actual underlying costs are C2. S2 is the schedule obtained by our algorithm, with costs C2 and the schedule length is L2. NP1 is the number of processors required to schedule the tasks for schedule length L1 and NP2 for schedule length L2. The *Factor* column shows the factor of improvement of the new schedule S2 over S1 using the new costs C2, as discussed before.

We tested our algorithm with different types of task graphs with different node and edge sizes. Examples of the graphs that were given as input are illustrated in Figure 6. These graphs occur in various numerical analysis computations [4]. The node sizes were varied from 50 to 400 nodes and the average values for the schedule lengths have been tabulated in Table 2 and Table 3.

The NAS benchmark test program shows improvement in the schedule lengths for all loop counts. It can also be seen that the improvements increase as loop

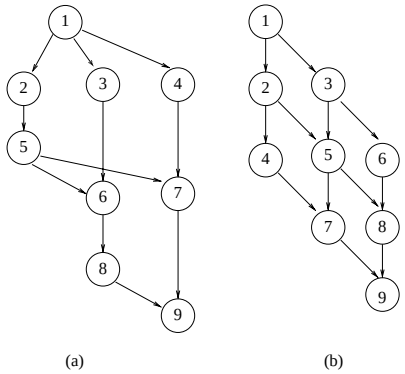

Fig. 6. (a) Cholesky Decomposition, (b) a Laplace Equation solver task graphs

Laplace Solver Graphs							Cholesky Decomposition Graphs						
Node	Edges	L2	NP2	L1	NP1	Factor	Node	Edges	L2	NP2	L1	NP1	Factor
49	84	237	12	1214	12	4.12	54	89	624	24	10099	24	16.18
100	180	402	27	3000	27	6.46	119	209	2149	69	36473	69	16.97
144	264	539	38	4455	38	7.26	152	271	3144	94	64466	94	20.50
225	420	768	55	6907	55	8.00	230	419	5970	156	166490	156	27.88
324	612	1050	78	10969	78	9.44	324	599	10124	234	358428	234	35.40
400	760	1267	100	15952	100	11.59	405	755	14265	303	585928	303	41.07

Table 2. FCFS model: Laplace Equation solver and Cholesky Decomposition graphs

counts increase. Typically for such numerical computing programs, loop counts are very high, and hence, the algorithm generates a better schedule. Table 2 shows results for the FCFS model of communication when the various graphs are input. Table 3 shows results for the SMF model of communication. Table 2 and Table 3 show results for Laplace Equation solver graph and Cholesky Decomposition graph inputs. It can be seen from the tables that the larger the size of the input, the better is the performance improvement due to our algorithm. This could be attributed to the fact that a larger graph size would increase the possibility of concurrent communication and this effect is captured more efficiently by our algorithm leading to a performance enhancement. The drawback, of course, is that more analysis is needed during each iteration. The results also show that the SMF model of communication results in lesser schedule lengths than the FCFS model. This can be easily explained by the fact that the increase in communication costs due to concurrent communication is minimum, if the messages are sent in shortest message first manner.

When the costs of the edges are modified, the schedule also changes and vice versa. Hence, a convergence is not always guaranteed but rather the schedules may oscillate between two values. The reason is that the edge costs are modified using the previous schedule and behavioral edges, and the new schedule is in

Laplace solver graphs							Cholesky Decomposition Graphs						
Node	Edges	L2	NP2	L1	NP1	Factor	Node	Edges	L2	NP2	L1	NP1	Factor
49	84	237	12	1214	12	4.12	54	89	193	24	1255	24	5.50
100	180	402	27	3000	27	6.46	119	209	443	65	5647	64	11.74
144	264	539	38	4455	38	7.26	152	271	571	88	9193	88	15.10
225	420	768	55	6907	55	8.00	230	419	875	146	13316	146	14.22
324	612	1050	78	10969	78	9.44	324	599	1243	216	37485	216	29.16
400	760	1267	100	15952	100	11.59	405	755	1561	279	53560	279	33.31

Table 3. SMF model: Laplace equation solver and Cholesky Decomposition graphs

turn dependent on the modified edge costs. In such a situation, the algorithm chooses the better of two schedules and terminates.

As can be seen, in all cases, the algorithm adapts schedules and costs to each other using behavioral edges and minimizes schedule length. The improvement in many schedules is of the order of 10.

6 Conclusion

We have developed a framework based on behavioral edges and have also developed an iterative algorithm to successively refine the schedule to minimize schedule lengths. The compiler also analyzes the communication behavior of the program and appropriately chooses the communication mechanism. Using this mechanism and the changed costs, the schedule is further refined by the algorithm. Using a performance study undertaken, it can be seen that an improvement of over 10 times is possible for many graphs encountered in numerical computations. Thus, if the compiler is given control over choosing the underlying communication mechanism, it can effectively analyze the partitioning decisions and significantly improve the performance.

References

1. Banerjee, Utpal, *Loop Parallelization*, Kluwer Academic Publishers, 1994 (Loop Transformations for Restructuring Compilers Series).
2. Darbha S. and Agrawal D. P., "Optimal Scheduling Algorithm for Distributed-Memory Machines", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 1, January 1998, pp. 87–95.
3. High Performance Fortran Forum. High Performance Fortran Language Specification, Version 1.0, Technical Report, CRPC–TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised January 1993).
4. Kwok Y-K and Ahmad I., "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, May 1996, Vol. 7, No. 5, pp. 506–521.
5. Bau D., Kodukula I., Kotlyar V., Pingali K. and Stodghill P., "Solving Alignment Using Elementary Linear Algebra", *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*, LNCS 892, 1994, pp. 46–60.

6. Gerasoulis A. and Yang T., "On Granularity and Clustering of Directed Acyclic Task Graphs", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, Number 6, June 1993, pp. 686-701.
7. Sarkar V., *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, Mass. 1989.
8. Subhlok Jaspal and Vondran Gary, "Optimal Mapping of Sequences of Data Parallel Tasks", *Proceedings of Principles and Practice of Parallel Programming (PPoPP) '95*, pp. 134–143.
9. Chretienne P., 'Tree Scheduling with Communication Delays', *Discrete Applied Mathematics*, vol. 49, no. 1-3, p 129-141, 1994.
10. Yang, T. and Gerasoulis, A., 'DSC: scheduling parallel tasks on an unbounded number of processors', *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, 951-967, 1994.
11. Darbha S. and Pande S. S., 'A Robust Compile Time Method for Scheduling Task Parallelism on Distributed Memory Systems', *Proceedings of the 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pp. 156–162.
12. Pande S. S. and Psarris K., 'Program Repartitioning on Varying Communication Cost Parallel Architectures', *Journal of Parallel and Distributed Computing 33*, March 1996, pp. 205–213.
13. Pande S. S., Agrawal D. P., and Mauney J., 'A Scalable Scheduling Method for Functional Parallelism on Distributed Memory Multiprocessors', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 4, April 1995, pp. 388–399.
14. Fahringer, T., 'Estimating and Optimizing Performance of Parallel Programs', *IEEE Computer: Special Issue on Parallel and Distributed Processing Tools*, Vol. 28, No. 11, November 1995, pp. 47–56.
15. Miller Barton P., Callaghan M., Cargille J., et. al. The Paradyn Parallel Performance Measurement Tool', *IEEE Computer: Special Issue on Parallel and Distributed Processing Tools*, Vol. 28, No. 11, November 1995, pp. 37–46.
16. Reed D. A., et. al.,'Scalable Performance Analysis: The Pablo Performance Analysis Environment', *Proceedings of Scalable Parallel Libraries Conference*, IEEE CS Press, 1993, pp. 104–113.
17. Balasundaram V., Fox G., Kennedy K. and Kremer U., 'A Static Performance Estimator to Guide Data Partitioning Decisions', *Proceedings of 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1991, pp. 213–223.
18. Karamcheti V. and Chien A., 'Software Overhead in Messaging Layers: Where Does the Time Go?', *Proceedings of the 6th ACM International Conference on Architectural Support for Programming Languages and Systems (ASPLOS VI)*, pp. 51–60.
19. Blume W. and Eigenmann R., 'Symbolic Range Propagation', *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.
20. Reinhardt, S., Hill M. D., Larus J. R., Lebeck A. et. al., 'The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers', *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 48–60, May 1993.
21. Garey, M.R. and Johnson, D.S., 'Computers and Intractability: A guide to the theory of NP-Completeness', *Freeman and Company*, 1979.
22. NAS Parallel Benchmarks, `http://science.nas.nasa.gov/Software/NPB/`