PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors

Tao Yang and Apostolos Gerasoulis
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
Email: tyang or gerasoulis@cs.rutgers.edu

Abstract

We describe a parallel programming tool for scheduling static task graphs and generating the appropriate target code for message passing MIMD architectures. The computational complexity of the system is almost linear to the size of the task graph and preliminary experiments show performance comparable to the "best" hand-written programs.

1 Introduction

In this paper, we consider *static* scheduling and code generation for message passing architectures. There are generally three distinct ways in addressing the programming difficulties for distributed memory architectures.

The first approach considers the problem of automatic parallelization and scheduling from sequential programs. The emphasis has been in the development of compilers or software tools that will assist in programming parallel architectures [2, 16, 18, 19]. Since message passing architectures require coarse grain parallelism to be efficient, one difficulty is the identification of parallelism especially at the procedural level and another is the scheduling and synchronization.

The second approach emphasizes the development of languages with parallel constructs such as KALI [14] and FORTRAN D [13]. Most of those compiler systems have not addressed the scheduling issues.

The third approach uses existing sequential languages, with the addition of communication library functions, to distribute the data and programs, and schedule the execution manually. This approach has been the most successful with respect to utilization of message passing architectures.

As far as we know no automatic parallelization compiler exists that can produce code for message passing machines which is competitive to manually programmed code. This is because of the difficulties in precise dependence analysis and scheduling. However, manually programming is tedious and debugging parallel programs is very hard.

Here, we describe a software system named PYRROS[1] for automatic scheduling and code generation. PYRROS takes as an input parallel program tasks with precedence constraints and produces code for message passing architectures such as nCUBE-2 and INTEL-2.

The paper is organized as follows. Section 2 discusses the macro-dataflow model task graph used to represent parallel computation. Section 3 describes the organization of PYRROS. Section 4 discusses the program and data partitioning for PYRROS along with an example. Section 5 presents a two-step method for scheduling a task graph. Section 6 describes code optimization algorithms for the target code generation. Section 7 presents some preliminary experiments. Section 8 discusses related work.

2 Macro Dataflow Task Graphs

A directed acyclic task graph (DAG) is defined by tuple $G = (V, E, \mathcal{C}, \mathcal{T})$ where V is the set of tasks and $v = |V|$ is the number of tasks, E is the set of communication edges and $e = |E|$ is the number of edges, $\mathcal{C}$ is the set of edge communication costs and $\mathcal{T}$ is the set of task computation costs. The value $c_{i,j} \in \mathcal{C}$ is the communication cost incurred along the edge $e_{i,j} = (n_i, n_j) \in E$, which becomes zero if both tasks n_i and n_j are mapped onto the same processor. The value $\tau_i \in \mathcal{T}$ is the execution time of task $n_i \in V$.

The macro-dataflow model of execution works as follows, e.g. Wu and Gajski [24], Sarkar [22]. The data flow goes through the graph and triggers the execution of tasks. A task receives all data from its predecessors

[1] The name PYRROS comes from PYRRHUS the legendary king of Epirus.

in *parallel*. It then executes without interruption and immediately after completion it sends the data to all successors in *parallel*. We do not allow task *duplication* in separate processors since it results in an increase of the space complexity. This model can overlap communication with computation which is extremely important for message passing architectures.

3 System Organization of PYRROS

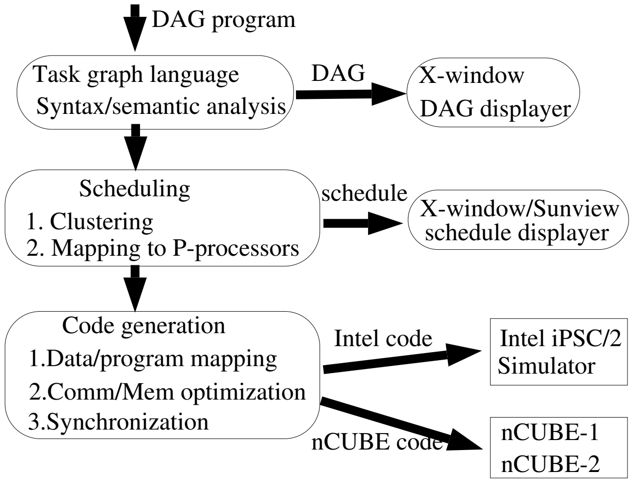

Figure 1: The system organization of PYRROS.

The organization of PYRROS is shown in Figure 1. The current tool has the following components: a task graph language, with an interface to C, allowing users to define partitioned programs and data. A scheduling system for clustering, load balancing and physical mapping, and communication/computation ordering. A graphic displayer for displaying task graphs and scheduling results. A code generator that inserts synchronization primitives and performs code optimization for the INTEL iPSC/2 simulator and the nCUBE-1 and nCUBE-2 hypercube machines.

4 Program and Data Partitioning for PYRROS

Program and data partitionings are given as an input to PYRROS. Let us consider as an example the kji form without pivoting of the Gauss-Jordan (GJ) algorithm with interior loop partitioning shown in Figure 2. The task T_k^j defined in Figure 2 uses column k to modify column j of the matrix A. The coarse grain data dependence graph is given in Figure 3. Task T_k^j is a functional task with input data being columns k and j and output data column j. Task T_k^{k+1} is a broadcasting node, sending the same column $k+1$ to all T_{k+1}^j, $j = k+2 : n+1$. The data units are the columns of the matrix on which the tasks operate.

for $k = 1$ **to** n
  for $j = k+1$ **to** $n+1$
    T_k^j : { $a_{k,j} = a_{k,j}/a_{k,k}$
      for $i = 1 : n$ **and** $i \neq k$
        $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
      end}
  end
end

Figure 2: The kji form for Gauss-Jordan without pivoting and with interior loop task partitioning.

The communication and computation weights are estimated as follows. We can either symbolically compute the time that it takes to execute and communicate the data for each task or perform an average case estimation, Sarkar [23]. For T_k^j there are about $n\omega$ operations for each task, where ω is the time that it takes for each operation, $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$. For the communication cost we use the linear model $\alpha + n\beta$ time to transfer a vector of size n between two neighbors processors, where α is the start up time and β is the transmission rate. These communication and computation parameters can be found in Dunigan [7]. For example, for the nCUBE-2 hypercube, $\omega = 2.38\mu s$, $\alpha = 160\mu s$ and $\beta = 2.4\mu s$ per word transfer for single precision arithmetic.

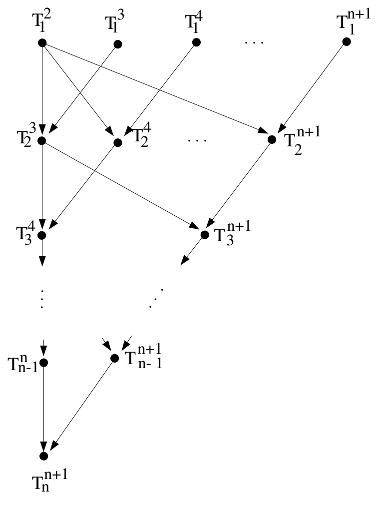

Figure 3: The Gauss-Jordan DAG. The node computation cost is $\tau = n\omega$ and edge communication cost is $c = \alpha + n\beta$.

4.1 Task graph language

The PYRROS system uses a simple language for defining task graphs. The program code in Figure 4 is a description of the GJ DAG shown in Figure 2 and 3. The key words are boldfaced. The semantics of the loop is the same as in Figure 2. The interior loop body contains the data dependence and weight information for a task T_k^j along with the specification of task computation. Task T_k^j has **weight** $n\omega$ and it receives column k and j from T_{k-1}^k and T_{k-1}^j respectively if $k > 1$; otherwise it receives data from the host. The *c_update* is an external C function which defines the task T_k^j. After the *c_update* is executed, then if $k < n$ this task sends column j to T_{k+1}^j and also performs a broadcast to other tasks if $k = j - 1$. When $k = n$, it sends the solution column $k + 1$ to the host.

```
for k=1 to n
   for j = k+1 to n+1
      task T(k,j)
         weight n*w
         receive if(k>1) column[k] T(k-1,k) weight al-
pha+beta*n
                  column[j] T(k-1,j) weight alpha+beta*n
               else
                  column[k] host weight alpha+beta*n
                  column[j] host weight alpha+beta*n
               endif
        perform c_update(k,j)
        send if (k<n)
               if (k != j-1)
                     column[j]      T(k+1,j)      weight
alpha+beta*n
               else
                 for b = j+1 to n+1
                     column[j]      T(k+1,b)      weight
alpha+beta*n
                 endfor
               endif
             else
               column[k+1] host weight alpha+beta*n
         endif
      endtask
   endfor
endfor
```

Figure 4: PYRROS task specification for the GJ DAG.

PYRROS will read this program and perform lexical and semantics analysis to generate an internal representation of the DAG. Then using the X-window DAG displayer we can verify whether the definition of the task graph is correct.

5 Scheduling in PYRROS

PYRROS uses the two-step method for scheduling:

1. Clustering on an unbounded number of processors of a completely connected architecture.
2. Scheduling the clusters on a fixed number of p processors of the physical architecture.

We have chosen this method for several reasons. First of all it has been shown to be a good approach to scheduling, particularly for coarse grain parallelism, Kim and Browne [15], Sarkar [22], Ortega [17]. Second we can devise low complexity algorithms without affecting performance.

5.1 Clustering

Clustering is defined as a mapping of tasks onto clusters. This is an important part in scheduling, since once tasks are assigned in a cluster they execute in the same processor in the actual architecture. Sarkar [22] was one of the first to recognize the importance of clustering, which he calls internalization pre-pass. A detailed discussion of the clustering problem is in Gerasoulis and Yang [10].

PYRROS uses the Dominant Sequence Algorithm (DSC) to automatically determine the clustering for task graphs. It performs a sequence of clustering refinement steps and at each refinement step, it tries to zero an edge to reduce the parallel time (PT). The parallel time is determined by the longest path in the scheduled graph. We call this path the *Dominant Sequence* (DS).

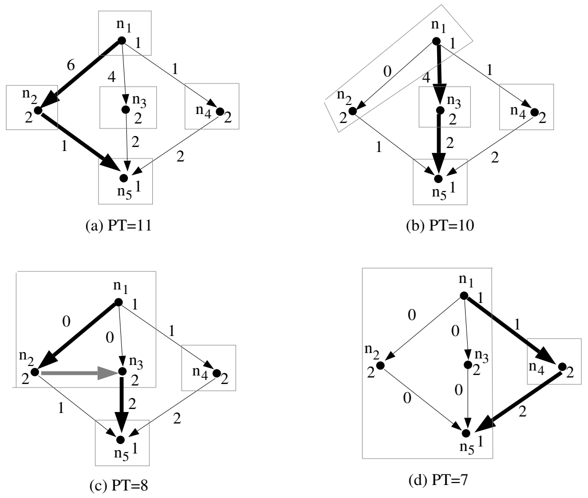

Figure 5: The clustering refinements in DSC.

In Figure 5 (c) three clusters are given, $\{ n_1, n_2, n_3 \}$, $\{ n_4 \}$ and $\{ n_5 \}$. If this clustered graph is scheduled then an execution ordering between the independent tasks n_2 and n_3 must be determined. If we execute n_2 before n_3 then PT is determined by the dominant sequence DS $=< n_1, n_2, n_3, n_5 >$. The DSC algorithm is outlined below:

```
Initially each task is a cluster.
Compute the initial DS. Mark all edges unexamined.
WHILE there is an edge unexamined DO
     Zero an edge in DS if PT does not increase.
     Mark this edge examined.
     Find a new DS.
ENDWHILE
```

For example, Figure 5(a) is the initial clustering. The DS is shown with bold face arrows and PT=11. The first edge in DS is zeroed and PT is reduced to 10, shown Figure 5(b) along with the new DS$=< n_1, n_3, n_5 >$. A zeroing is accepted as long as PT does not increase from the previous step. Thus this zeroing is accepted. Next the edge (n_1, n_3) is zeroed and the new DS$=<$

$n_1, n_2, n_3, n_5 >$ is shown in Figure 5(c) with PT=8, so this zeroing is also accepted. Since the only unexamined edge in DS is (n_3, n_5) this edge is picked up for zeroing in the next step and the parallel time of the clustering shown in Figure 5(d) is PT=7. In the final step (n_1, n_2) is chosen for zeroing, but this zeroing cannot be accepted since the parallel time increases to PT= 8.

A detailed description of the DSC algorithm is in Yang and Gerasoulis [25]. DSC has been shown to be superior to several other clustering algorithms from the literature in the following sense:

- DSC has an "almost linear" complexity, $O((v + e)\log v))$ time complexity and $O(v+e)$ space complexity. The key idea to keep the complexity low is to find the DS nodes incrementally from one step to the next in $O(\log v)$ time. Most of the algorithms in the literature have complexity higher than $O(v^2)$ which makes them impractical for large task graphs. For the GJ DAG, $v = n^2$ where n is the size of the matrix. Using an $O(v^2)$ algorithm will result in $O(n^4)$ complexity which is higher than the $O(n^3)$ sequential complexity of GJ.

- Experimental results using random graphs and numerical computation DAGs show that DSC outperforms several higher complexity algorithms from the literature [15, 22, 24] in terms of parallel time.

- The general problem of finding the optimum clustering is NP-complete. However there are few classes of task graphs such as join, fork and coarse grain tree for which the optimal solutions are tractable in a polynomial time complexity. DSC is the only general clustering algorithm which finds the optimum for these task graphs. When graph structures become slightly more complicated such as fork/join concatenated graphs and fine grain trees, the time complexity of optimal scheduling jumps to NP-complete, Chretienne [3]. This indicates that DSC has reached a high degree, in terms of parallel time optimality, that a polynomial-time complexity algorithm can attain.

5.2 Scheduling a clustered graph on p processors

Let us assume that the number of clusters is u and the number of available processors is p. We schedule the clusters to p processors in the following three stages:

1. Merge the u clusters into p completely connected virtual processors if $u > p$.

2. Map the p virtual processors into p physical processors.

3. Order the execution of tasks in each processor.

Sarkar [22] has proposed an incremental scheduling method that combines all three stages by performing v iterations. Each iteration of the Sarkar's algorithm selects a task and assigns this task, along with other tasks in the same cluster, to the one of p processors that gives the minimum parallel time increase from the previous iteration. The cost of his algorithm is $O(vp(v+e))$ where $O(v+e)$ comes from the cost in computing the parallel time at each iteration. This complexity is too high for large graphs. Next we describe a low complexity algorithm used in PYRROS.

5.2.1 Cluster merging

PYRROS uses a *work profiling method*, George et. al. [12] for cluster merging. This method is simple, with low complexity and has been shown to work well in practice, e.g. Saad [20], and Ortega [17].

1. Compute the arithmetic load LM_j for each cluster.
2. Sort the clusters in an increasing order of their loads.
3. Use a load balancing algorithm to map u clusters to p virtual processors so that every processor has approximately the same load.

For example, assume there are four clusters as depicted in Figure 6(a) and each task has weight 1. First the clusters are ordered in terms of their loads:

$$M_1 = \{T_1^3\},\ M_2 = \{T_1^4, T_2^4\},$$
$$M_3 = \{T_1^5, T_2^5, T_3^5\},\ M_4 = \{T_1^2, T_2^3, T_3^4, T_4^5\}.$$

Then M_1 is merged with M_4, M_2 is merged with M_3 and the arithmetic load is balanced.

5.2.2 Physical mapping

At this stage, we have p virtual processors (or clusters) and p physical processors. Since physical processors are not completely connected, we have to take the processor distance into account. Let $TC_{i,j}$ be the total communication which is the summation of costs of all edges between virtual processor i and j. And Let $CC = \{TC_{i,j} | TC_{i,j} \neq 0\}$ and $m = |CC|$. In general we expect that $m << e$.

The goal of the physical mapping is to determine the physical processor number $P(V_i)$ for each virtual processor V_i that minimizes the following cost function:

$$F(CC) = \sum_{TC_{i,j} \in CC} distance(P(V_i), P(V_j)) \times TC_{i,j}.$$

Figure 6 is a special case of the GJ DAG when $n = 4$. A clustering for this DAG is Figure 6(a) and each cluster corresponds to a virtual processor. Assuming that the edge cost is equal to 3 time units then the total communication between four virtual processors is shown in Figure 6(b).

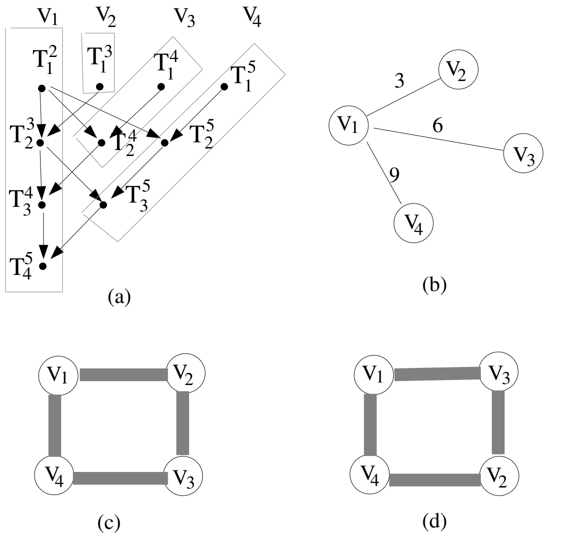

Figure 6: An example of physical mapping. Each nonzero edge cost is 3 time units.

In Figure 6(c) we show one physical mapping to a 4-node hypercube with $F(CC) = 24$ and another mapping is shown in (d) with $F(CC) = 21$.

Determining an optimal physical mapping is very hard. Currently PYRROS uses a heuristic algorithm due to Bokhari [1]. This algorithm starts from an initial assignment, then performs a series of pairwise interchanges so that the $F(CC)$ is reduced monotonically. The complexity is $O(p^3)$.

5.2.3 Task ordering

After the physical mapping has been fixed, the interprocessor communication delay between tasks can be determined. We still need to order the execution of tasks within each processor such that the total parallel time is minimized. Unfortunately, this problem is NP-complete.

PYRROS uses the RCP (ready critical path) algorithm with complexity $O(v \log v + e)$. One major difference between RCP and the classical critical path algorithm is that it maintains a *ready* task list instead of a *free* task list[2]. Our experiments have verified the advantages of the ready list scheduling. A detailed analysis of the task ordering problem is given in Yang and Gerasoulis [26]. The RCP algorithm is outlined below.

1. Adjust the communication edges of the DAG using the information of task processor assignment and processor physical distances.
2. Compute the level priority for each task in the DAG, which is the length of the longest path from it to an exit node, including both computation and nonzero communication costs in that path.
3. Each processor maintains a ready priority list of tasks. The ready task with the highest priority is executed as soon as the processor where this task is assigned becomes available.

Consider an example in Figure 7(a) where six tasks have been assigned to two processors. If the execution ordering is $\{n_4, n_3\}$ on processor 0 and is $\{n_2, n_1, n_5, n_6\}$ on processor 1, then PT is 12 as shown in Figure 7(c). The schedule produced by RCP is shown in Figure 7(b) with PT=7. Task n_4 has level priority 4 and n_3 has level priority 3. At time 1, n_4 and n_3 are both free but not ready on processor 0. At time 2, only n_3 is ready, thus it is scheduled for execution on processor 0. At time 3, n_4 is ready and it is scheduled. If n_4 had been scheduled before n_3 since n_4 was free with higher priority, an idle gap would have been created from time 1 to 3 on processor 0. This is because n_4 could not be executed until time 3.

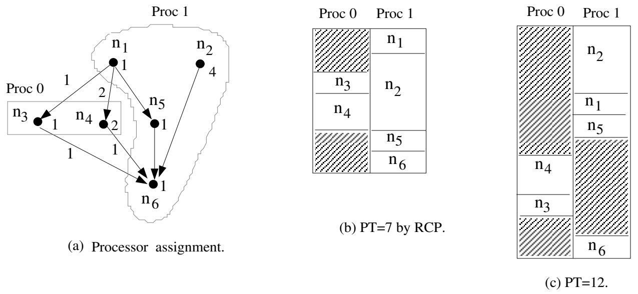

Figure 7: An example of task execution ordering.

6 Code Generation

In order to execute tasks in a target machine, we need to generate the appropriate code based on the previous scheduling result. The complexity of the algorithms described in this section is $O((v+e)\log p)$.

6.1 Machine model

The target machines are MIMD message passing local memory architectures. Each processor has a unique identification and can execute different programs on different data. Data communication and synchronization are implemented by message exchange. The message passing primitives are the following:

- *send(m,i)*
  The message m is sent to processor i. Sending is nonblocking in the sense that this procedure does not have to wait for the receiving processor to get this message.

[2] A task is free if the predecessors of this task have completed their executions and a task is ready if the data items sent from its predecessors have arrived at the local memory.

- *receive(m,i)* and *receive(m)*
  By executing the first procedure a processor will wait until a message from processor i arrives at this processor. It is a blocking receive. For the second "receive" function, a processor will wait until *any* message is received by this processor.

We number processors as $0, 1, 2, \cdots, p-1$. There is a special function *mynode* that returns the processor number where the program resides. There is a special host node, named *host* where the PYRROS host program is executed. The host performs the following functions: (1) allocates the required number of processors, (2) loads node programs, (3) sends the initial data to nodes and (4) collects the final result from nodes.

6.2 Program distribution and data distribution

The scheduling part of PYRROS determines the processor assignment for each task. We define

- $Proc(x)$ – the processor assignment of task x.
- $Tasks(i)$ – the ordered set of tasks executed in processor i.

Using the DAG information, we define the following sets for each task x:

- $InMsg(x)$ – the data items that task x must use to perform a computation. It is expressed by a set of message-to-source pairs:

$$InMsg(x) = \{(t_1, s_1) \cdots (t_k, s_k)\}$$

  where each pair (t_i, s_i) indicates a message t_i sent from processor s_i.

- $Procedure(x)$ – the operational specification of task x.
- $OutMsg(x)$ – the data items produced by x and sent to some successor tasks. It is expressed by a set of message-destination pairs:

$$OutMsg(x) = \{(t_1, s_1) \cdots (t_k, s_k)\}$$

  where each pair (t_i, s_i) indicates a message t_i sent to processor s_i.

We also define

- $fst((t, s)) = t$, the first item of a pair.
- $scnd(t, s)) = s$, the second item of a pair.
- $FirstSet(\{(t_1, s_1), \cdots, (t_k, s_k)\}) = \{t_1, \cdots, t_k\}$, the set composed of the first elements of the message items.

The program distributed to processor i is determined by $Tasks(i)$. The initial data used in processor i is distributed by the host as:

$$InitData(i) = \bigcup_{x \in Tasks(i), m \in InMsg(x)} fst(m) | scnd(m) = host.$$

The data items used in processor i are

$$Data(i) = \bigcup_{x \in Tasks(i), m \in InMsg(x)} fst(m) | scnd(m) \neq i.$$

The final result for processor i to send back to the host is

$$Result(i) = \bigcup_{x \in Tasks(i), m \in OutMsg(x)} fst(m) | scnd(m) = host.$$

Thus the host will collect the following result from the parallel architecture:

$$Result(host) = \bigcup_{0 \le i \le p-1} Result(i).$$

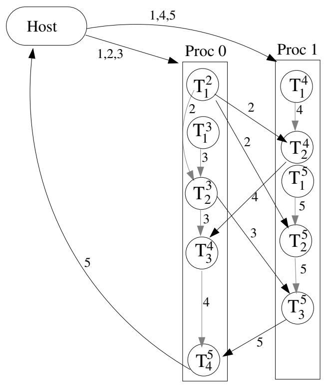

Figure 8: The GJ DAG with $n = 4$ mapped onto 2 processors.

Example: Figure 8 depicts the mapping of GJ DAG of Figure 3 with $n = 4$ onto two processors. Precedence edges are marked with the data items (columns) to be communicated. The dashed arrows within each processor indicate the intra-processor communication. The following table gives examples of $InMsg$ and $OutMsg$ sets:

	$InMsg$	$OutMsg$
T_1^4	$\{(1, host), (4, host)\}$	$\{(4, 1)\}$
T_2^4	$\{(2, 0), (4, 1)\}$	$\{(4, 0)\}$

The following table summarizes the information on data and program distribution for this DAG:

	processor 0	processor 1
$Tasks$	$(T_1^2, T_1^3, T_2^3, T_3^4, T_4^5)$	$(T_1^4, T_2^4, T_1^5, T_2^5, T_3^5)$
$Data$	$\{1, 2, 3, 4, 5\}$	$\{1, 2, 3, 4, 5\}$
$InitData$	$\{1, 2, 3\}$	$\{1, 4, 5\}$
$Result$	$\{5\}$	$\emptyset$

6.3 Program generation

Program generation includes arithmetic computation generation and communication generation. The schedule derived according to the DAG precedence ensures the correctness of the computation ordering. In order to execute each task properly, we need to make sure that each data item referred by this task is available in the local memory of its processor before its execution. After the completion of its computation, each new data item produced by this task should be sent to appropriate processors whose tasks need this data item for further computation.

A naive approach for the program generation for the macro dataflow model is to use the code in Figure 9 for each processor i.

```
FOR each task x in Tasks(i) scanned in the execution
order DO
   FOR each message m in InMsg(x) DO
      IF scnd(m) ≠ i THEN
         allocate some space if necessary and receive this
message.
      ENDIF
   ENDFOR
   Execute procedure(x).
   FOR each message m in OutMsg(x) DO
      IF scnd(m) ≠ i THEN
         send this message to processor scnd(m).
      ENDIF
   ENDFOR
ENDFOR
```

Figure 9: A naive approach to program generation.

There are two problems that we must address to improve the code quality.

- **Communication optimization**

  A task could send the same messages to many tasks in the same processor. Thus we need to eliminate the unnecessary communication operations. Furthermore, one message could be sent to many processors. Using one-to-one message sending repeatedly creates network contention. Thus, a broadcast algorithm is needed that takes advantage of the network topology.

- **Memory utilization improvement**

  Since the size of a local memory is still limited, we need to reuse the memory space. Thus if some data items are no longer used after certain execution point, the space freeing should be performed so that the memory utilization can be improved.

In the next subsection, we discuss how PYRROS addresses the first problem. The memory optimization will not be presented here.

6.4 Communication optimization

6.4.1 Communication procedure selection

The selection of communication procedures depends on the topology of the target machine. Li and Chen [16] have discussed an approach for recognizing some communication patterns such as one-to-all broadcast, uniform-shift, all-to-one-reduce and affine-transform. They have not, however, addressed the optimization and implementation of a specific communication procedure.

Here, we will focus on the selection and optimization of message broadcasting on a *hypercube* architecture. The algorithm for one-to-all spanning tree broadcasting on a hypercube has been developed by Saad and Schultz [21]. This algorithm requires all processors to participate in the message routing, which is not appropriate when many processors do not need this message. Thus, we need a one-to-some multicasting scheme to reduce the number of unnecessary processors involved in message forwarding to alleviate the communication contention.

PYRROS uses a static elimination scheme to reduce the broadcasting range from the entire hypercube to several small subgraphs which sufficiently cover all destinations. The space overhead for storing a subgraph information in a communication packet is $O(1)$. Two types of basic subgraph broadcasting are used:

- *Linear path broadcasting*

  If all nodes belong to a shortest path between the broadcasting node and the farthest receiving node then the message is broadcasted along this linear path. The advantage of this communication is its simplicity. The disadvantage is that there are few instances for which a path covers all destinations.

- *Subcube broadcasting*

  A more complicated broadcasting is to use a spanning tree of a subcube. The advantage is that it can reach scattered nodes in a more efficient manner within a smaller broadcasting scope comparing with the entire hypercube broadcasting. The representation and determination of a subcube will be described below.

PYRROS partitions the destinations into several subsets such that each subset is covered by a path or a subcube. The partitioning method will not be discussed here.

Assume that the source processor is s and the destination processors are $p_1, p_2, \cdots p_k$. One possible linear path for covering these processors can be explored by checking whether a destination p_i is in the path from s to the farthest receiving processor. The nodes in this path can be found by reversing the bits of node binary code numbers from right to left (or from left to right). To determine a subcube where the one-to-some broadcasting is conducted, the following procedure is performed:

> Let
> $$h = EOR(s, p_1, p_2, \cdots, p_k)$$
> where EOR is the bitwise exclusive OR function with $k+1$ arguments, i.e. the i-th bit of h is one if there are different values among the i-th bits of those $k+1$ arguments.
> h is called a *subcube mask*. The number of non-zero bits in the subcube mask is the subcube dimension.

Example: Figure 10 shows a part of a DAG mapped onto a hypercube of dimension 3. Task n_1 broadcasts a data message to n_2, n_4 and n_5. This communication can be accomplished using subcube broadcasting within the cube composed of processor 000, 001, 011 and 010 where the subcube mask is $h = 011$. The broadcasting from n_2 to n_3, n_6 and n_7 can be implemented as a linear-path communication along processor path 011, 001 and 101.

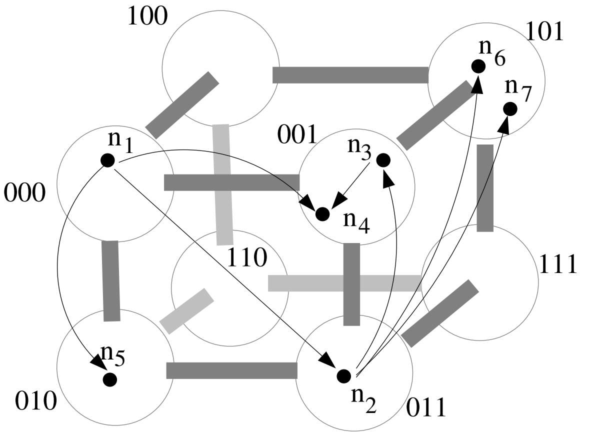

Figure 10: Part of a DAG mapped to a hypercube with 8 nodes.

6.4.2 Redundant communication elimination

The naive approach for generating a program shown in Figure 9 strictly ensures that before the execution of a task, all messages it needs will be received. After the completion of this task, it will send the messages it produces to the tasks that need them. Some redundant communication might be generated by this approach. For instance, in Figure 10 assume that n_2 sends the same message to n_6 and n_7. In the sending site, n_2 only needs to send this message once since n_6 and n_7 are scheduled in the same processor 101. In the receiving site, only one receiving operation is needed to be performed in processor 101. Once a processor receives the same data message, another could get this data item from the local memory. Furthermore, if the redundant receiving is conducted but no message sending corresponds to this receiving, a *deadlock* is created.

Since the communication operation is inserted with respect to each task, it is easy for sending sites to avoid the redundant sending because the produced messages can be sent with respect to the processors that need them instead of tasks. In the receiving site, however, we need to inform the availability of data items in the local memory to the local tasks whose input data items have already been received by a previously executed task.

We will perform a dataflow information computation to determine whether an insertion of receiving operation is necessary before a task execution. Assume that for processor i, the ordered task execution list is $Tasks(i) = (n_1, n_2, \cdots, n_k)$. Define $MsgExist(n_j)$ to be a set of messages that are available in the local memory of processor i before processing task n_j. Then

> $MsgExist(n_1) = \emptyset$
> **FOR** $j = 2$ **TO** k **DO**
> $OutMsg1(n_{j-1}) = FirstSet(OutMsg(n_{j-1})) \odot i.$
> $MsgDef(n_{j-1}) = InMsg(n_{j-1}) \ominus OutMsg1(n_{j-1}) \cup OutMsg1(n_{j-1}).$
> $MsgExist(n_j) = (MsgExist(n_{j-1}) \ominus MsgDef(n_{j-1})) \cup MsgDef(n_{j-1}).$
> **ENDFOR**

where the operator $\ominus$ is defined as:
$X \ominus Y = \{msg \mid msg \in X \wedge fst(msg) \notin FirstSet(Y)\}$
and operator $\odot$ is defined as:
$\{m_1, \cdots, m_k\} \odot s = \{(m_1, s), \cdots, (m_k, s)\}$.

Thus, for each data message in $InMsg(n_j)$, say msg, that message is already in the local memory if $msg \in MsgExist(n_j)$. As a result, a receiving operation can be eliminated.

Example: We show the above dataflow computation for processor 1 in Figure 8:

	$MsgExist$
T_1^4	$\emptyset$
T_2^4	$\{(1, host), (4, 1)\}$
T_1^5	$\{(1, host), (2, 0), (4, 1)\}$
T_2^5	$\{(1, host), (2, 0), (4, 1), (5, 1)\}$
T_3^5	$\{(1, host), (2, 0), (4, 1), (5, 1)\}$

We can eliminate receiving data item 2 from processor 0 for T_2^5 since this item is already in set $MsgExist(T_2^5)$.

6.4.3 Deadlock free communication protocol

Without incorporating the efficient communication procedures, the execution graph is purely acyclic and no deadlock could be caused. But by using specialized communication procedures, each processor not only will execute its tasks but also will be involved in the message routing. As a result, the entire multiprocessor system no longer is executing an acyclic graph. A deadlock could be caused when there is a cycle in message waiting and forwarding.

For example in Figure 10 processor 000 broadcasts a message produced by n_1 to n_4, n_5 and n_2. In order to let n_2 receive this message, processor 001 would have to receive this message from processor 000 and forward this message to processor 011. However, before doing that, processor 001 could be performing the message receiving for n_3 and must wait for the message produced by n_2 at processor 011. This gives rise to a deadlock.

In order to resolve the conflict in message waiting and forwarding, we use an asynchronous communication scheme to allow each processor to receive any arbitrary message that arrives at the message buffer of this processor, to store this message if it is useful and to forward this message if necessary. The routing direction of the message forwarding is recognized according to a communication protocol described below.

For each message, there is a control head indicating the following information: the type of message passing routing, all additional information needed for this routing procedure, the message identification and the message size. The type of message passing could be *one-to-one*, *one-to-all tree broadcasting*, *one-to-some linear path broadcasting*, and *one-to-some subcube broadcasting*. After recognizing that information, a message could be routed accordingly.

For the subcube broadcasting, the following message routing method is then used:

Let *mynode* be the current processor number and m be the received subcube mask.
WHILE $m > 0$ **DO**
  Let j be the right most bit of one in the binary code of m.
  Forward the message to node $EOR(mynode, 2^{j-1})$ with subcube mask $EOR(m, 2^{j-1})$.
  Reset $m = EOR(m, 2^{j-1})$.
ENDWHILE

Example: In Figure 10, task n_1 in processor 000 broadcasts a message in the subcube with mask 011. The routing tree is depicted in Figure 11.

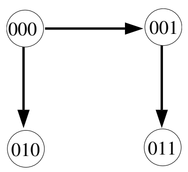

Figure 11: Subcube broadcasting with subcube mask=011.

For linear path broadcasting, the forwarding is terminated if the end point of this linear path is reached otherwise the following routing method is used:

Let s be the starting node in this path, *mynode* be the current node number and j be the leftmost bit of one in the binary code of $EOR(s, mynode)$.
Forward the message to node $EOR(mynode, 2^{j-1})$.

7 Performance Results

We report our experiments on the BLAS-3 GJ algorithm. The LINPACK BLAS-3 based programs are widely used for benchmarking parallel architectures, see Dongarra et. al. [6]. The task graph definition is similar to the one defined for the regular GJ algorithm in Figure 4.

We have also manually written a program that uses natural clustering and wrap mapping along the gray code ring, following the approach by Saad [20]. The sequential time is 1150 seconds for 1000×1000 linear algebraic system on nCUBE-2.

In the following table we compare the performance of automatically-generated code by PYRROS with the hand-written manual program when $n = 1000$. PYRROS first simulates the execution on the host using the nCUBE-2 parameters ($\omega = 2.38\mu s$, $\alpha = 160\mu s$ and $\beta = 2.4\mu s$ per word for single precision arithmetic) and gives a predicted performance. The predicted performance by PYRROS is very close to the real performance in nCUBE-2. Moreover, the performance of the automatically-generated code is comparable with that of our hand-written version. The speedup entry is computed by dividing the sequential time (1150 seconds) by the actual PYRROS code time.

	Pyrros(auto)		Hand	speedup
	predicted	nCUBE		
p=4	299.0	283.4	288.7	4.0
p=8	155.7	150.5	150.4	7.6
p=16	84.5	81.0	81.4	14.1
p=32	50.0	48.7	47.4	24.3
p=64	33.0	31.8	32.8	35.1

8 Related Work

There are several other systems related to PYRROS. PARAFRASE-2 [18] by Polychronopoulos et. al., is a parallelizing compiler system that performs dependence analysis, program partitioning and dynamic scheduling on shared memory machines. SCHEDULER by Dongarra and Sorensen [5] uses centralized dynamic scheduling for the shared memory model of computation. The KALI system by Koelbel and Mehrota [14] addresses parallel code generation and is currently targeted at DOALL parallelism. Kennedy's group [13] is also working on code generation for FORTRAN D on distributed-memory machines. The PARTI system by Saltz's group [4] focuses on irregular dependence graphs determined at run-time and optimizes the performance by precomputing the data accessing pattern. The current FORTRAN D system, KALI and PARTI do not address program scheduling for optimizing the mapping of program and data and the ordering of code execution.

HYPERTOOL by Wu and Gajski [24] uses the same model of computation and produces codes for the hypercube simulator SIMON. TASKGRAPHER by El-Rewini and Lewis [8] schedules task graphs but does not generate codes. The time complexity of the two systems is over $O(v^2)$.

Acknowledgments

Jack Dongarra from University of Tennessee gave us the initial inspiration for PYRROS and Mel Ciment from NSF the initial financial support DMS-8706122 and encouragement to look at parallel computing issues. Partial support has been provided by the Air Force Office of Scientific Research and the Office of Naval research under grant N00014-90-J-4018. We thank Weining Wang for developing the task graph language parser, Milind Deshpande for the X window schedule displayer, and the nCUBE corporation for giving us free time on the nCUBE-2 and also a booth at Supercomputing 91 to demonstrate PYRROS.

References

[1] S.H. Bokhari, *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic Publisher, 1990.

[2] D. Callahan, K. Kennedy, *Compiling programs for distributed-memory multi-processors,* The Journal of Supercomputing, (2): 151-169, 1988.

[3] Ph. Chretienne, *Task Scheduling over Distributed Memory Machines*, Proc. of the International Workshop on Parallel and Distributed Algorithms, (North Holland, Ed.), 1989.

[4] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis, *Distributed Memory Compiler Methods for Irregular Problems: Data Copy Reuse and Runtime Partitioning*, ICASE Report No. 91-73, 1991.

[5] J. J. Dongarra, and D.C. Sorensen, *SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs*, in D.B. Gannon, L.H. Jamieson and R.J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pp. 363-394. The MIT Press, Cambridge, Massachusetts, 1987.

[6] J. J. Dongarra, J. Du Croz, I. Duff and S. Hammarling, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. on Math. Software, 16(1):1-17, Mar. 1990.

[7] T.H. Dunigan, *Performance of the INTEL iPSC/860 and nCUBE 6400 Hypercube*, ORNL TM-11790, Oak Ridge National Laboratory, Oak Ridge, TN, Nov. 1991.

[8] H.El-Rewini and T.G. Lewis, *Scheduling Parallel Program Tasks onto Arbitrary Target Machines*, J. of parallel and Distributed Computing 9, 138-153(1990).

[9] A. Gerasoulis and T. Yang, *On the Granularity and Clustering of Directed Acyclic Task Graphs*, LCSR-TR-153, Dept. of Computer Science, Rutgers Univ., 1990.

[10] A. Gerasoulis and T. Yang, *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors*, LCSR-TR-169, Dept. of Computer Science, Rutgers Univ., 1991.

[11] A. Gerasoulis and T. Yang, *Compile-time scheduling and code generation for message-passing architectures*, Report.

[12] A. George, , M.T. Heath, and J. Liu, *Parallel Cholesky Factorization on a Shared Memory Processor*, Lin. Algebra Appl., 77(1986), pp. 165-187.

[13] S. Hiranandani, K. Kennedy and C.W. Tseng, *Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines*, Supercomputing 91, pp. 86-100.

[14] C. Koelbel and P. Mehrotra, *Compiling global name-space parallel loops for distributed execution*, IEEE Trans. on parallel and Distributed Systems 2(4), 1991.

[15] S.J. Kim and J.C Browne, *A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures*, Int'l Conf. on Parallel Processing, vol 3, pp. 1-8, 1988.

[16] J. Li and M. Chen, *Compiling Communication-Efficent Programs for Massively Parallel Machines.*, May 1991, accepted in IEEE Trans. on Distributed and Parallel Computing.

[17] J.M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, New York:Plenum, 1988.

[18] C. D. Polychronopoulos, M.B. Girkar, M. R.Haghighat, C.L. Lee, B.P. Leung, and D.A. Schouten, *The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran*, in D. Gelernter, A. Nicolau and D. Padua (Eds.), *Languages and Compilers for Parallel Computing*, 1990.

[19] A. Rogers, K. Pingali, *Process Decomposition Through Locality of Reference*, SIGPLAN 1989, pp. 69-80.

[20] Y. Saad, *Gaussian Elimination on Hypercubes*, Parallel Algorithms and Architectures, Cosnard, M. et al. Eds., Elsevier Science Publishers, North-Holland, 1986.

[21] Y. Saad and M. H. Schultz, *Data Communication in Hypercubes*, DCS/RR-428, Research Report, Yale Uiverstiy, 1985.

[22] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, The MIT Press, 1989.

[23] V. Sarkar, *Determining Average Program Execution Times and their Variance*, SIGPLAN 1989, pp. 298-312.

[24] Min-You Wu and D. Gajski, *A Programming Aid for Hypercube Architectures*, The Journal of Supercomputing, vol. 2, pp. 349-372, 1988.

[25] T. Yang and A. Gerasoulis, *A Fast Scheduling Algorithm for DAGs on an Unbounded Number of Processors*, The Proceeding of IEEE Supercomputing 91, pp. 633-642.

[26] T. Yang and A. Gerasoulis, *List Scheduling with and without Communication Delay*, Report.