Runtime Parallel Incremental Scheduling of DAGs

Min-You Wu, Wei Shu
The University of New Mexico

Yong Chen
Seagate Technology

Abstract

A runtime parallel incremental DAG scheduling approach is described in this paper. A DAG is expanded incrementally, scheduled, and executed on a parallel machine. A DAG scheduling algorithm is parallelized to scale to large systems. In this approach, a large DAG can be executed without consuming large amount of memory space. Inaccurate estimation of task execution time and communication time can be tolerated. This runtime approach can also execute dynamic DAGs. Implementation of this parallel incremental system demonstrates the feasibility of this approach. Preliminary results show that it is superior to other approaches.

1. Introduction

Task parallelism is essential for applications with irregular structures. With computation partitioned into tasks, load balance can be achieved by scheduling the tasks, either dynamically or statically. Dynamic algorithms are able to balance the load well [4, 9]. For example, the Runtime Incremental Parallel Scheduling (RIPS) algorithm can provide high-quality load balancing [9]. In RIPS, all processors cooperate to schedule work, accurately balancing the load by using global load information at runtime.

Most dynamic algorithms schedule independent tasks, that is, a set of tasks that do not depend on each other. On the other hand, static task scheduling algorithms consider the dependences among tasks. The Directed Acyclic Graph (DAG) is a task graph that models task parallelism as well as dependences among tasks. Currently, most DAG scheduling algorithms are static, which allocate tasks to Processing Elements (PEs) for a balanced load while minimizing communications. As the DAG scheduling problem is NP-complete in its general form [5], many heuristic algorithms have been proposed to produce satisfactory performance [7, 12, 14].

Current DAG scheduling algorithms have drawbacks which may limit their usage. Some important issues to be addressed are:

- They are slow since they run on a single processor machine. A scheduler may require tens of hours of computation time of modern workstations to generate a schedule for 1K processors.
- They require a large memory space to store the graph and are not scalable thereafter. As an example, to schedule a parallel program to 1K processors, a graph of a few millions of nodes may require memory space of hundreds of Mbytes.
- The quality of the obtained schedules relies heavily on accurate estimation of execution times. Without this information, sophisticated scheduling algorithms cannot deliver satisfactory performance.
- The application program must be recompiled for different problem sizes since the number of tasks and the estimated execution time of each task varies with the problem size.
- It is static as the number of tasks and dependences among tasks in a DAG must be known at compile-time. Therefore, it cannot be applied to dynamic problems.

These problems limit applicability of current DAG scheduling techniques and have not yet received substantial attention. Thus, many researchers consider the static DAG scheduling unrealistic.

A new approach was proposed in [11] to solve these problems. It suggests a parallel scheduling method to solve the first problem, and an incremental scheme to solve the last four problems. A parallel algorithm was published in [13] and another parallel algorithm was published in [6]. The complete parallel incremental system has not been implemented until recently, which is presented in this paper. A different approach, the supervisor and executor approach, was described in [3] where the memory space limitation and the recompiling problem can be eliminated by generating and executing tasks at runtime. The system is called PTGDE, where a scheduling algorithm runs on a supervisor processor, which schedules the DAG to a number of executor processors. When a task is generated, it is sent to an executor processor to execute. This method solves the memory limitation problem because only a small portion of the DAG is in the memory at a time. However, the scheduling algorithm is still sequential and not scalable. Because there is no feedback from the executor processors, the load
imbalance caused by weight estimation cannot be adjusted. It cannot be applied to dynamic problems either. Moreover, a processor resource is solely dedicated to scheduling. If scheduling runs faster than execution, the supervisor processor will be idle; otherwise, the executor processors will be idle.

In this paper, we report the implementation result of the parallel incremental scheduling system proposed in [11]. A scheduling algorithm can run faster and is more scalable when it is parallelized. By incrementally scheduling and executing DAGs, the memory limitation can be alleviated and inaccurate weight estimation can be tolerated. It can also be used to solve dynamic problems. This parallel incremental DAG scheduling scheme is based on general static scheduling and is extended from our previous project, Hypertool [12]. The new system is named Hypertool/2. Different from runtime incremental parallel scheduling for independent tasks [9], Hypertool/2 takes care of dependences among tasks and uses the DAG as its computation model. This system consists of two major components, parallel scheduling, and incremental scheduling/execution. In Section 2, the DAG and CDAG models are discussed. An incremental execution model is presented in Section 3. The parallel scheduling algorithm used in this implementation is presented in Section 4. The system organization is discussed in Section 5. Performance results are shown in Section 6. Section 7 concludes the paper.

2 DAG and Compact DAG

A DAG, or a macro dataflow graph, consists of a set of nodes \( \{n_1, n_2, ..., n_n\} \) connected by a set of edges, each of which is denoted by \( e_{i,j} \). Each node represents a task, and the weight of node \( n_i \), \( w(n_i) \), is the execution time of the task. Each edge represents a message transferred from node \( n_i \) to node \( n_j \) and the weight of edge \( e_{i,j} \), \( w(e_{i,j}) \), is equal to the transmission time of the message. In a DAG, a node that does not have any parent is called an entry node whereas a node that does not have any child is called an exit node. A node cannot start execution until it gathers all of the messages from its parent nodes. A node sends messages to its child nodes after completion of its execution. The edge weight between two nodes that are assigned to the same PE is assumed to be zero. Figure 1 shows a DAG generated from a program shown in Figure 2 [12]. This program is a parallel Gaussian elimination algorithm with partial pivoting, which partitions a given matrix by columns. Node \( n_0 \) is the INPUT procedure and \( n_{10} \) the OUTPUT procedure. The procedures FindMax and UpdateMtx are called several times. Nodes \( n_1, n_9, n_{12} \) and \( n_{16} \) are FindMax and other nodes are UpdateMtx. The control dependencies in the program are ignored, so that a procedure call can be executed whenever all input data of the procedure are available. Data dependencies are defined by the single assignment of parameters in procedure calls. Communications are invoked only before and after procedure execution. In other words, a procedure receives messages before it begins execution, and it sends messages after it has finished the computation. Data dependencies among the procedural parameters define the DAG. The size of the DAG is proportional to \( N^2 \), where \( N \) is the matrix size.

![Figure 1. A DAG (Gaussian elimination).](image-url)

This single-assignment programming makes compile-time analysis easy and more accurate. However, it seems to require much more memory space. By using incremental execution described below, the memory is allocated at runtime. The actual memory consumption is even less than static non-single-assignment programs. The technique to runtime allocating memory will be discussed later.

In a static system, a DAG is generated from the user program and scheduled at compile time. Then the scheduled DAG is loaded to PEs for execution. In a runtime scheduling system, the DAG is not generated all at once. Instead, it is generated incrementally. For this purpose, a compact form of the DAG (Compact DAG or CDAG) is generated at compile time. It is then expanded to the DAG incrementally at runtime. The size of a CDAG is proportional to the program size while the size of a DAG is proportional to the problem size or the matrix size.

A CDAG is defined by its communication rules similar to the parameterized task graph in [2]. A communication rule is in the format of

\[
\text{source node} \rightarrow \text{destination node}: \text{message name} | \text{guard}.
\]
Program GaussianElimination
/* matrix[N+1][N][N+1] */
/* stores single-assigned \(N \times (N+1)\) matrix \(A\) and column of equation \(Ax = y\); */
/* vector[N+1].index[N] stores single-assigned row permutation; */
/* vector[N+1].m[N] stores single-assigned coefficients; */

/*************************** Main Program ***************************
/* a serial part of computation */
/* initialize vector[0].index[N]; */
/* initialize matrix[0][N][N+1]; */
call INPUT(vector[0], for \(i=0\) to \(N\) matrix[0][i]);

for \(i = 0\) to \(N-1\) do /* perform \(N\) iterations in parallel */
call FindMax(matrix[i][i], vector[i], vector[i+1], i);
    /* it can be executed if matrix[i][i] and vector[i] */
    /* are available; vector[i+1] becomes available at */
    /* the end of this procedure execution */
for \(j = i\) to \(N\) do /* do parallel operations on \(N-i+1\) columns */
call UpdateMtx(matrix[i][j], matrix[i+1][j], vector[i+1], i);
    /* it can be executed if matrix[i][j] and vector[i+1] */
    /* are available; matrix[i+1][j] becomes available at */
    /* the end of this procedure execution */

/* a serial part of computation */
/* do back substitution */
call OUTPUT(vector[N], for \(i=0\) to \(N-1\) matrix[i+1][i], matrix[N][N]);
End

/**************************** Procedure FindMax ***************************/
Procedure FindMax(inColumn, inVec, outVec, k)
/* Input: inColumn column \(k\) where max pivot will be found; */
/* inVec permutation index and coefficients; */
/* k iteration number; */
/* Output: outVec vector of output values; */
/* find maximum */
max = inColumn[inVec.index[k]];
n = k;
for \(i = 1\) to \(N-1\) do /* copy inColumn to inVec.index */
    if max < inColumn[inVec.index[i]]
        max = inColumn[inVec.index[i]];
    n = i;
for \(i = 0\) to \(N-1\) do /* copy inVec.index to outVec.index */
    outVec.index[i] = inVec.index[i];
if (n <> k) /* permute row index */
    tmp = outVec.index[k];
    outVec.index[k] = outVec.index[n];
    outVec.index[n] = tmp;
for \(i = k+1\) to \(N-1\) do /* calculate multiplying factors */
    j = outVec.index[i];
    outVec.m[j] = inColumn[j] / max;
End

/**************************** Procedure UpdateMtx ***************************/
Procedure UpdateMtx(outColumn, inColumn, inVec, k)
/* Input: inColumn column to be updated; */
/* inVec permutation index and coefficients; */
/* k iteration number; */
/* Output: outColumn column of output values; */
for \(i = 0\) to \(k\) do /* copy inColumn to outColumn */
    j = inVec.index[i];
    outColumn[j] = inColumn[j];
pivot = inColumn[inVec.index[k]];
for \(i = k+1\) to \(N-1\) do /* update the column */
    j = inVec.index[i];
    outColumn[j] = outColumn[j] - inVec.m[j] * pivot;
End

Figure 2. A parallel Gaussian elimination algorithm.
The communication rules of the Gaussian elimination code are shown in Figure 3. The corresponding CDAG is shown in Figure 4. The DAG and CDAG can be automatically generated from an annotated C program, such as the one shown in Figures 2. A compiler, which is not described here, translates the program to the CDAG. Actually, this compiler is much simpler than the one described in [2] because the procedure parameters are single-assigned.

### 3 The Incremental Execution Model

Different models can be used to execute a CDAG at runtime. We proposed an incremental execution model. It is shown in Figure 5. In the incremental execution model, only a subgraph is scheduled in each system phase. The size of the subgraph to be generated each time is normally limited by the available memory space. The system scheduling activity alternates with the underlying computation work. It starts with a system phase where only a part of the DAG is generated and scheduled. It is followed by a user computation phase to execute the scheduled tasks. The PEs will execute until most tasks have been completed and transferred to the next system phase to generate and schedule the next part of the DAG.

A policy decides when to transfer from a user computation phase to the next system phase. It can be triggered when any PE runs out of tasks. The PE initiates the scheduling activity by broadcasting a pause signal to all other PEs. A PE, upon receiving the pause signal, completes the current task execution and switches from this user phase to the next system phase. In the next system phase, another part of the DAG is generated. The newly generated tasks are scheduled together with the old tasks. This way, load imbalance due to inaccurate estimation can be tolerated. Then the scheduled tasks are sent to PEs to start the next user phase.

This incremental execution model has many advantages:

- This approach can relax the memory space requirement because the task graph is generated incrementally.
- The demand for an accurate estimation of execution time becomes less critical. While an inaccurate estimate may result in load imbalance, the load imbalance can be adjusted when scheduling the next set of tasks.

---

**Figure 3. Communication rules for the Gaussian elimination code.**

**Figure 4. A CDAG (Gaussian elimination).**

**Figure 5. Incremental scheduling system.**

```plaintext
INPUT → FindMax(i) : vector[0], matrix[0, 0] | i = 0
INPUT → UpdateMtx(0, j) : matrix[0, j] | 0 ≤ j ≤ N
FindMax(i) → FindMax(i + 1) : vector[i + 1] | 0 ≤ i ≤ N - 2
FindMax(i) → OUTPUT : vector[N] | i = N - 1
FindMax(i) → UpdateMtx(i, j) : vector[i + 1] | 0 ≤ i ≤ N - 1, i ≤ j ≤ N
UpdateMtx(i, j) → UpdateMtx(i + 1, j) : matrix[i + 1, j] | 0 ≤ i ≤ N - 2, i + 1 ≤ j ≤ N
UpdateMtx(i, j) → FindMax(i + 1) : matrix[i + 1, j] | 0 ≤ i ≤ N - 2, j = i + 1
UpdateMtx(i, j) → OUTPUT : matrix[i + 1, j] | 0 ≤ i ≤ N - 1, j = i
UpdateMtx(i, j) → OUTPUT : matrix[N, N] | i = N - 1, j = N
```
tasks. An estimator is still needed for a rough estimation, but accuracy of estimation will have less impact on the overall performance.

- The application programs need not be recompiled for each problem size since this scheme adapts to different problem sizes. The cost of this approach is the runtime overhead, which can be minimized by grain-size control.
- It can be applied to dynamic problems. In general, the structure of dynamic problems cannot be known at compile-time. An incremental scheduler can adapt to this property by scheduling subgraphs for execution when partial information becomes available.

4 The Parallel Scheduling Algorithm

A scheduling algorithm produces a schedule for a given DAG or a partial DAG. A scheduling algorithm should have the following features:

- **High quality** — it is able to minimize the completion time of a given DAG.
- **Low complexity** — it is able to minimize the time for scheduling a given DAG.

Since the scheduling time is a part of the runtime system time, minimizing scheduling time becomes extremely important in incremental scheduling. Assume an algorithm spends $T_s$ time to schedule a DAG to a $T_e$ timespan (completion time). Another algorithm spends $T_s + \delta_s$ time to schedule the same DAG to a $T_e - \delta_e$ timespan. We can benefit from the second algorithm only when $\delta_s$ is smaller than $\delta_e$. An algorithm should be selected to minimize the sum of scheduling time and timespan.

Here we describe a simple DAG scheduling algorithm, the MCP algorithm [12]. This algorithm was designed to schedule a DAG on a bounded number of PEs. First, the as-late-as-possible (ALAP) time of a node is defined as $T_L(n_i) = T_{critical} - level(n_i)$, where $T_{critical}$ is the length of the critical path counting both node and edge weights, and $level(n_i)$ is the length of the longest path from node $n_i$ to the end point, including node $n_i$. In fact, high-quality scheduling algorithms more or less rely on the ALAP time or level.

**MCP Algorithm**

1. Calculate the ALAP time of each node.
2. Sort the node list in an increasing ALAP order. Ties are broken by using the smallest ALAP time of the successor nodes, the successors of the successor nodes, and so on.
3. Schedule the first node in the list to the PE that allows the earliest $start\_time$ with insertion. Delete the node from the list and repeat Step 3 until the list is empty.

In step 3, when determining the $start\_time$, idle time slots created by communication delays are also considered. A node can be inserted to the first idle time slot that can fit. This method is called an **insertion algorithm**. The MCP algorithm has been compared to four other well-known scheduling algorithms under the same assumption. It has been shown that MCP performed the best [1].

The complexity of the MCP algorithm is $O(n^2 \log n)$, where $n$ is the number of nodes in a graph. In the second step, the ties can be broken randomly to have a simplified version of MCP. The scheduling quality only varies a little, but the complexity is reduced to $O(n^2)$. However, for a runtime system, $O(n^2)$ is still too large. Therefore, in this implementation, we further reduce the complexity by using the non-insertion version of the MCP algorithm. Its complexity is $O(e + n \log n + np)$, where $e$ is the number of edges and $p$ the number of PEs. Our experiment shows that for coarse-grain partitioning, this algorithm produces schedule lengths at most 3% longer than the original MCP but its scheduling time is reduced by one to two orders of magnitude.

The parallel scheduling algorithm, used in our system is a parallel version of the modified MCP algorithm described above. The PEs that execute a parallel scheduling algorithm are called the **physical PEs (PPEs)** in order to distinguish them from the **target PEs (TPEs)** to which the DAG is to be scheduled. The basic idea behind parallel scheduling algorithms is that instead of identifying one node to be scheduled each time, we identify a set of nodes that can be scheduled in parallel. The quality and speed of a parallel scheduler depend on data partitioning. There are two major data domains in a scheduling algorithm, the source domain and the target domain. The source domain is the DAG and the target domain is the schedule for target processors. A horizontal scheme [13] is illustrated in Figure 6, where three PPEs schedule the graph to six TPEs and each PPE holds a portion of schedules of six TPEs. In this scheme, each PPE is assigned a set of graph nodes using time domain partitioning. The resultant schedule is also partitioned so that each PPE maintains a portion of the schedule of every TPE. Each PPE schedules its own portion of the graph before all PPEs exchange information with each other to determine the final schedule.

The horizontal parallel MCP is named the HPMCP algorithm [13], which is shown in Figure 7. After the graph is partitioned, each PPE uses MCP to schedule its portion to
produce its sub-schedule. When applying MCP, we ignore the dependencies between partitions so that each partition can be scheduled independently. A node is treated as an entry node in its partition if all of its parent nodes are not local. Nodes are scheduled in the order of their ALAP priorities. Each PPE schedules its partition starting from its local time 0. Then adjacent sub-schedules are concatenated to form the final schedule.

1. Partition the nodes, with each partition assigned to a PPE.
2. Each PPE applies the MCP algorithm to its partition to produce a sub-schedule, ignoring the edges between a node and its remote parent nodes. Schedule the first node in the list to the TPE that allows the earliest start time. Delete the node from the list and repeat this scheduling step until the list is empty.
3. Concatenate each pair of adjacent sub-schedules.

**Figure 7. The HPMCP Algorithm.**

5 Runtime System Organization

The runtime system of Hypertool/2 is organized as modules including graph generation, scheduling, node execution, communication handling, and incremental execution handling modules.

5.1 Graph generation

In Hypertool/2, the CDAG is expanded incrementally. In the first system phase, a set of nodes \( S_0 \) is generated from the CDAG. Assume set \( S_0 \) has \( B \) nodes and there are \( P \) PPEs. Each PPE generates \( B/P \) nodes. The CDAG is available to every PPE and nodes can be independently generated. Then, \( S_0 \) is scheduled and executed. When user phase 0 pauses, all the residual nodes that have not been executed will remain in a subset \( S'_0 \). In the next system phase, the newly expanded nodes together with \( S'_0 \) form a new set \( S_1 \). Generally, in system phase \( k \), the newly expanded nodes together with residual set \( S'_{k-1} \) that contains all nodes that have not been executed from the previous phase form set \( S_k \). A partial DAG \( G_k = \{ S_k, E_k \} \), is constructed. An edge \( e_{i,j} \) in \( E_k \), if node \( n_i \) and \( n_j \) are both in \( S_k \). Any outgoing edge from a node in \( S_k \) becomes a *future* message if its destination node is not in \( S_k \). This partial DAG \( G_k \) is then scheduled to PEs and executed in user phase \( k \).

5.2 Scheduling

The scheduling module will establish a mapping for each node in \( S_k \) from its logic ID to its physical ID, which consists of a target PE number and a local ID at the target PE. For each edge \( e_{i,j} \) in \( E_k \), node \( i \) also has the physical ID of node \( j \). Thus, when node \( i \) finishes its execution, all of its outgoing messages can be immediately directed to their destinations.

Once a PPE generates its subset of nodes, it independently schedules these nodes to form a subschedule using the MCP algorithm. When a DAG is scheduled, it is called scheduled DAG. The scheduled DAG is then loaded to TPEs for execution. Each TPE obtains a list of nodes sorted in the order of execution (using local IDs).

5.3 Execution

In the execution module, a dispatch routine is responsible to pick a node and prepare it for execution. In the scheduled DAG, the nodes in the list are to be executed in order. The dispatch routine picks the next node in the list and check its incoming messages. When all of its incoming messages have arrived, the node is ready and executed.

The dispatch routine allocates memory and prepares parameters for the node’s execution. The node procedure is then invoked. On completion of node execution, output parameters are processed by communication handling module. All memory space allocated for the node is also deallocated. This message-driven, macro-dataflow execution model can efficiently utilize memory.

5.4 Communication handling

Message receiving is handled either when the PE does not have a ready node to execute, or during the period between completion of one node execution and start of the next ready node execution.

Since a push scheme is applied with every outgoing message incorporated with its destination PE number as well as the local ID, an arrived message can be easily attached to the corresponding node. Once all of its incoming messages have arrived, the node itself becomes ready to execute.

A message is sent for each outgoing edge after node execution. If a message has a single destination, it is classified as *unicast*. If a message has multiple receipts, it is classified as a *multicast*. Although a multicast message can be sent to different destinations one by one, it may require unacceptable communication time. The communication module uses a multicasting tree for an efficient multicast.

5.5 Incremental execution handling

Every PE that has received a *pause* massage will finish its current node execution and pauses its current user phase \( k \). Before entering system phase \( k + 1 \), the residual nodes that have not been executed as well as corresponding messages need to be processed to incorporated into the phase \( k + 1 \).

Before entering phase \( k + 1 \), the current mapping from node logical IDs to their physical IDs has to be invalidated,
since the physical IDs are meaningful only to a particular phase. When the residual nodes are sent back for rescheduling, the messages that have already arrived at those nodes are detached and converted into the future messages to be appended at the blocking queue, so that delivery of these messages are postponed until the new mapping becomes available for phase \( k + 1 \). If that message is a multicast message, it is deleted since the multicast message will be remulticast in every later phase.

6 Experimental Study

The Hypertool/2 has been implemented on an Intel Paragon machine. We have tested three application programs, Gaussian elimination, Exhaustive search for N-queens, and GROMOS. Gaussian elimination is a static problem, N-queens is a dynamic problem, and GROMOS is a problem with an irregular structure defined by input data.

The Gaussian elimination code has been partitioned by defining a grain-size parameter \( SN \). When \( SN \) is equal to 1, each task includes a single column, same as in Figure 2. When \( SN \) is larger than 1, \( SN \) columns are merged into a single task. This parameter controls the grain size so that the overhead to handle a task is relatively smaller than the computation time of the task. The problem size (matrix size) is \( 2K \times 2K \). \( SN \) is set to be 8 and the total number of tasks is 32966. The number of tasks to be generated in each system phase is \( 1000 \times P \), where \( P \) is the number of PPEs. The weight of each task has been estimated by an estimator for simple loops.

The N-Queen problem in exhaustive search has an irregular and dynamic structure. The test program used here is 15-Queens. The number of tasks generated and the grain size in each task are unpredictable. After execution, we know that 15941 tasks have been generated. In the N-Queens problem, different from Gaussian elimination, the number of tasks that can be generated in each system phase depends on the number of ready tasks. The ready tasks are the newly created tasks that can be scheduled and executed. This problem is very dynamic and the task weights are difficult to predict. Thus, a unit weight is assumed for each task.

GROMOS is a molecular dynamics program [10]. The test data for GROMOS is the bovine superoxide dismutase molecule (SOD), which has 6968 atoms [8]. The cutoff radius is predefined to 16 \( \AA \). GROMOS has an irregular structure. Given input data, the number of tasks is known, and the task weight can be estimated. The total number of tasks for the given test data is 4986. The number of tasks to be generated in each system phase is set to be \( 100 \times P \).

We use these problems to test the system behavior from static to dynamic, completely irregularly to slightly irregularly structured problems. First, we study the system performance, especially the system overhead which includes task generation, scheduling, and other overhead. Gaussian elimination is a typical problem with heavy overhead. Gaussian elimination is a problem with heavy dependency and therefore it is used for this study. Table 1 shows an analysis of execution time on 16 TPEs: task generation time, scheduling time, other runtime overhead and computation time on different number of PPEs. The total parallel execution time is the sum of above items. The time unit is second. The computation time is the time of executing tasks. It can be seen from the table that the system overhead is relatively large since the task graph of Gaussian elimination has many dependences. By parallelizing graph generation and scheduling, the system overhead (including task generation, scheduling, and other overhead) has been reduced from 126.1 seconds on a single PE to 64.7 seconds on eight PEs. On sixteen PEs, the benefit of parallelizing graph generation and scheduling has been overwhelmed by parallelization overhead. The system overhead increases to 66.5 seconds.

The execution times and speedups of three application programs are shown in Table 2. In this table, the number of PPEs is the same as that of TPEs except that Gaussian elimination on 16 and 32 TPEs uses only eight PPEs for best performance. The speedup is defined as \( S = T_1 / T_p \), where \( T_1 \) is the execution time on a single TPE and \( T_p \) is the parallel execution time on \( p \) TPEs. We do not compare our result to hand-written sequential code since Hypertool/2 is of a different execution model. In general, the system overhead

<table>
<thead>
<tr>
<th>Number of PPEs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task generation</td>
<td>16.8</td>
<td>8.6</td>
<td>4.6</td>
<td>3.2</td>
<td>2.8</td>
</tr>
<tr>
<td>Scheduling</td>
<td>31.2</td>
<td>15.1</td>
<td>9.9</td>
<td>6.0</td>
<td>5.8</td>
</tr>
<tr>
<td>Other overhead</td>
<td>78.1</td>
<td>61.0</td>
<td>58.2</td>
<td>55.5</td>
<td>57.9</td>
</tr>
<tr>
<td>Computation</td>
<td>347</td>
<td>351</td>
<td>354</td>
<td>355</td>
<td>356</td>
</tr>
<tr>
<td>Total</td>
<td>473</td>
<td>436</td>
<td>427</td>
<td>420</td>
<td>423</td>
</tr>
</tbody>
</table>

Table 2. Hypertool/2 Performance.

<table>
<thead>
<tr>
<th>Number of TPEs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaus. elim.</td>
<td>5542</td>
<td>2788</td>
<td>1431</td>
<td>766</td>
<td>420</td>
<td>268</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.99</td>
<td>3.87</td>
<td>7.23</td>
<td>13.2</td>
<td>20.7</td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>331</td>
<td>166</td>
<td>82.8</td>
<td>41.8</td>
<td>21.0</td>
<td>10.9</td>
</tr>
<tr>
<td>15-Q</td>
<td>360</td>
<td>181</td>
<td>91.1</td>
<td>46.0</td>
<td>23.2</td>
<td>12.1</td>
</tr>
<tr>
<td>Gromos</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 ( \AA )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PPEs is the same as that of TPEs except that Gaussian elimination on 16 and 32 TPEs uses only eight PPEs for best performance. The speedup is defined as \( S = T_1 / T_p \), where \( T_1 \) is the execution time on a single TPE and \( T_p \) is the parallel execution time on \( p \) TPEs. We do not compare our result to hand-written sequential code since Hypertool/2 is of a different execution model. In general, the system overhead
on a single TPE is less than 5%. Considering the flexibility of this execution model, the overhead paid is acceptable.

The performance of Gaussian elimination is good when the number of TPEs is less than 16. On a large number of TPEs, the overhead increases due to its heavy dependency. Its performance is not as good as other two applications which do not have many dependences. In 15-Queens, since the weight is difficult to estimate, a simple estimation of task weight is used, that is, every task has the same weight. The performance study shows that the largest task is tens times larger than the smallest task. However, since incremental scheduling can adjust the load imbalance due to estimation errors, the load can be eventually balanced. The resultant speedup of this application is quite high. GROMOS also shows good performance. Its dependency is not heavy and it can scale up to 32 TPEs well. Compared to 15-Queens, the grain sizes of tasks are larger and the number of tasks is smaller.

PTGDE is a work similar to ours [3]. PTGDE runs on a simulator, which does not reflect the real situation. For comparison purpose, we have re-implemented it on Intel Paragon based on description in [3]. Since PTGDE is unable to run dynamic problems such as N-Queen and GROMOS, only Gaussian elimination has been implemented. PTDGE requires at least two PEs to run because the supervisor and the executor are running on different PEs. Table 3 compares the execution time and speedup of the Gaussian elimination code. For two PEs, only one of them executes the user program and there is no speedup. PTGDE uses a dynamic scheduling algorithm, PTGDS, to schedule tasks at runtime, which is in a depth-first-search style. This scheduling algorithm does not balance the load well, especially for a large number of PEs. Its performance is not as good as the MCP algorithm. Furthermore, PTGDE schedules tasks at a single PE so that it is not scale up to more processors.

### Table 3. Comparison of Gaussian Elimination on Hypertool/2 and PTGDE.

<table>
<thead>
<tr>
<th>Number of PEs</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypertool/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (S)</td>
<td>2788</td>
<td>1431</td>
<td>766</td>
<td>423</td>
<td>268</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.99</td>
<td>3.87</td>
<td>7.23</td>
<td>13.1</td>
<td>20.7</td>
</tr>
<tr>
<td>PTGDE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (S)</td>
<td>5574</td>
<td>1998</td>
<td>1042</td>
<td>808</td>
<td>590</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.00</td>
<td>2.79</td>
<td>5.35</td>
<td>6.90</td>
<td>9.45</td>
</tr>
</tbody>
</table>

7 Conclusion and Future Works

This paper describes a new approach which schedules DAGs in parallel and incrementally executes DAGs at runtime. Our experimental results show that the major goals of this research have been achieved, that is, the DAG can be used for large problems, parallel scheduling can scale well, inaccurate estimation can be tolerated, the system can run dynamic problems, and the application program needs not to be recompiled for different problem sizes.

Acknowledgments

We are very grateful to Reinhard Hanxleden for the GROMOS program, Terry Clark for the SOD data, and M. Feeley for the elegant N-Queen program. This research was partially supported by NSF grants CCR-9505300 and CCR-9625784.

### References