

# A Runtime System for Dynamic DAG Programming

Min-You Wu<sup>1</sup>, Wei Shu<sup>1</sup>, and Yong Chen<sup>2</sup>

<sup>1</sup> Department of ECE, University of New Mexico  
{wu, shu}@eece.unm.edu

<sup>2</sup> Department of ECE, University of Central Florida

**Abstract.** A runtime system is described here for dynamic DAG execution. A large DAG which represents an application program can be executed on a parallel system without consuming large amount of memory space. A DAG scheduling algorithm has been parallelized to scale to large systems. Inaccurate estimation of task execution time and communication time can be tolerated. 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. 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. As the DAG scheduling problem is NP-complete in its general form [4], many heuristic algorithms have been proposed to produce satisfactory performance [6, 3, 9].

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.
- They require a large memory space to store the graph and are not scalable thereafter.
- The quality of the obtained schedules relies heavily on the estimation of execution time. Accurate estimation of execution time is required. 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.
- They are static as the number of tasks and dependences among tasks in a DAG must be known at compile-time. Therefore, they 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.

The memory space limitation and the recompiling problem can be eliminated by generating and executing tasks at runtime, as described in PTGDE [2], 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 inaccurate estimation of execution time 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.

We have proposed a *parallel incremental* scheduling scheme to solve these problems [5]. 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 [6]. The new system is named Hypertool/2. Different from runtime incremental parallel scheduling for independent tasks, Hypertool/2 takes care of dependences among tasks and uses DAG as its computation model.

## 2 DAG and Compact DAG

A DAG, or a macro dataflow graph, consists of a set of nodes  $\{n_1, n_2, \dots, n_n\}$  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. Figure 1 shows a DAG generated from a parallel Gaussian elimination algorithm with partial pivoting, which partitions a given matrix by columns. Node  $n_0$  is the INPUT procedure and  $n_{19}$  the OUTPUT procedure. The size of the DAG is proportional to  $N^2$ , where  $N$  is the matrix size.

In a static system, a DAG is generated from the user program and scheduled at compile time. Then this *scheduled DAG* is loaded to PEs for execution. In a runtime scheduling system, the DAG is generated incrementally and each time only a part of the DAG is generated. 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 CDAG is similar to the parameterized task graph in [2]. 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.

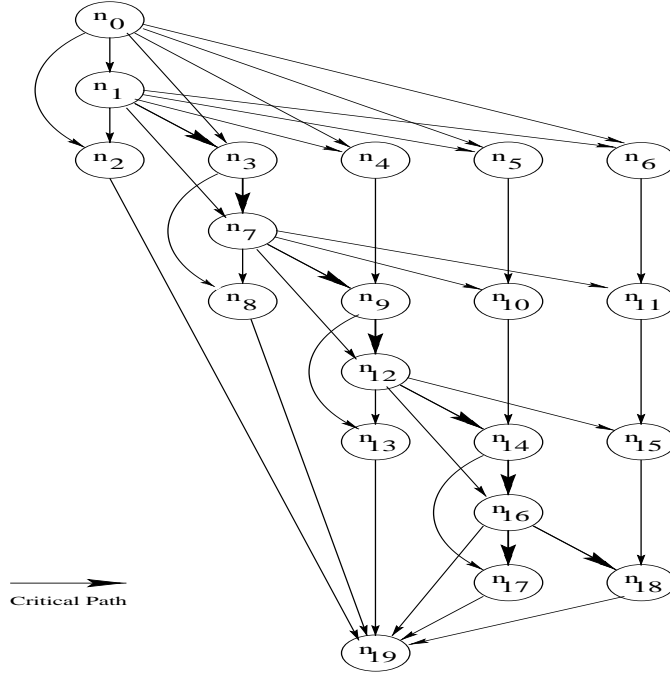


Fig. 1. A DAG (Gaussian elimination).

A CDAG is defined by its communication rules. A communication rule is in the format of

*source node*  $\rightarrow$  *destination node*: *message name* | *guard*.

The communication rules in Figure 2 is generated from an annotated C program of Gaussian elimination. For details, refer to [8]. The corresponding CDAG is shown in Figure 3. The runtime system takes the CDAG as its input.

---

```

INPUT  $\rightarrow$  FindMax(i) : vector[0], matrix[0, 0] | i = 0
INPUT  $\rightarrow$  UpdateMtx(0, j) : matrix[0, j] |  $0 \leq j \leq N$ 
FindMax(i)  $\rightarrow$  FindMax(i + 1) : vector[i + 1] |  $0 \leq i \leq N - 2$ 
FindMax(i)  $\rightarrow$  OUTPUT : vector[N] | i = N - 1
FindMax(i)  $\rightarrow$  UpdateMtx(i, j) : vector[i + 1] |  $0 \leq i \leq N - 1, i \leq j \leq N$ 
UpdateMtx(i, j)  $\rightarrow$  UpdateMtx(i + 1, j) : matrix[i + 1, j] |  $0 \leq i \leq N - 2, i + 1 \leq j \leq N$ 
UpdateMtx(i, j)  $\rightarrow$  FindMax(i + 1) : matrix[i + 1, j] |  $0 \leq i \leq N - 2, j = i + 1$ 
UpdateMtx(i, j)  $\rightarrow$  OUTPUT : matrix[i + 1, j] |  $0 \leq i \leq N - 1, j = i$ 
UpdateMtx(i, j)  $\rightarrow$  OUTPUT : matrix[N, N] | i = N - 1, j = N

```

---

Fig. 2. Communication rules for the Gaussian elimination code.

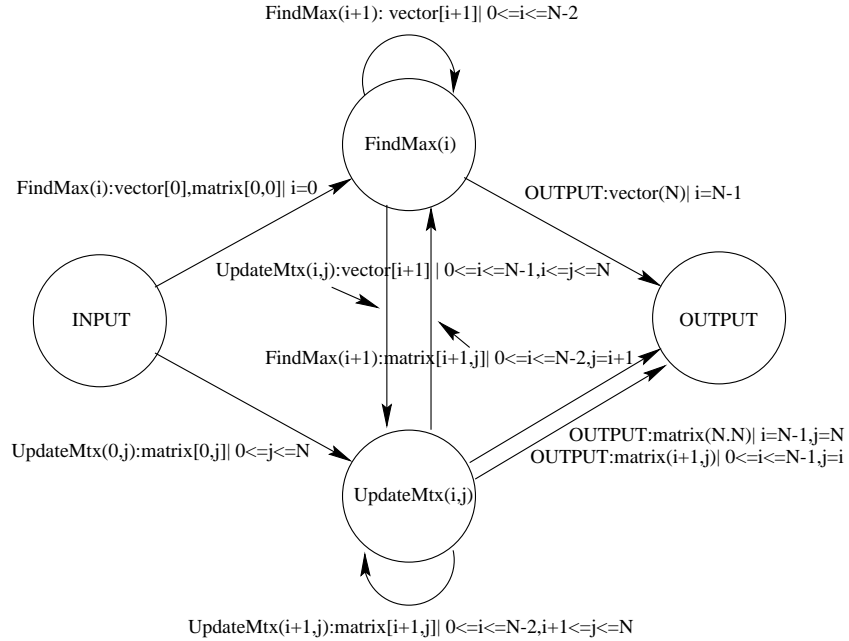


Fig. 3. A CDAG (Gaussian elimination).

### 3 The Incremental Execution Model

Different models can be used to execute a CDAG at runtime. We proposed an incremental execution model. In this 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 transfer to the next system phase to generate and schedule the next part of the DAG. For details, refer to [8]. The 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.
- The application programs need not be recompiled for each problem size since this scheme adapts to different problem sizes.
- 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

Since the scheduling time is a part of the runtime system time, minimizing scheduling time becomes extremely important in incremental scheduling. Here we describe a simple DAG scheduling algorithm, the MCP algorithm [6]. 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  $n_i$ .

### 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.
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.

A comparison study has shown that MCP performed better than many other scheduling algorithms [1]. The parallel scheduling algorithm, used in our system is a parallel version of the MCP algorithm. 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 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 TPEs. A *horizontal* scheme [7] is illustrated in Figure 4, 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

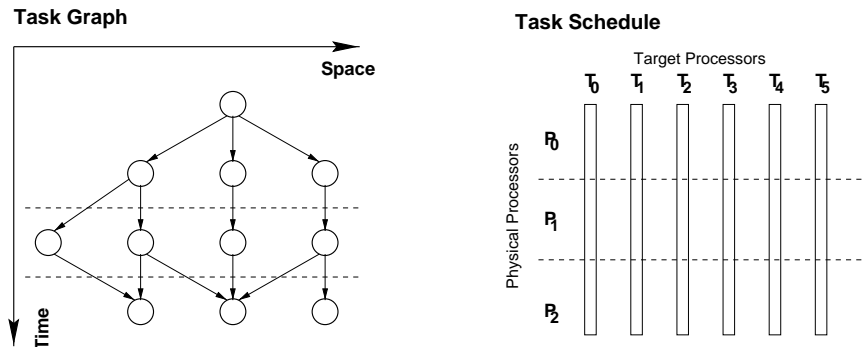


Fig. 4. The Horizontal Scheme.

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 [7], which is shown in Figure 5. After the graph is partitioned, each PPE uses MCP to schedule its partition to produce its sub-schedule. When applying MCP, we ignore the dependences between partitions so that each partition can be scheduled independently. Then adjacent sub-schedules are concatenated to form the final schedule.

- 
1. Partition the nodes and each partition is 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.
- 

**Fig. 5.** 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.

### Graph generation

The CDAG is expanded incrementally. In the first system phase, a subset of nodes is generated from the CDAG. Assume that the subset has  $B$  nodes and there are  $P$  PPEs. Each PPE generates  $B/P$  nodes independently. Then, this subset is scheduled and executed. In the next system phase, the newly expanded nodes together with the residual nodes from the last phase form a new subset. Any outgoing edge from a node in the subset becomes a *future* message if its destination node is not in the subset. This partial DAG is then scheduled to PEs and executed in a new user phase.

### Scheduling

The scheduling module will establish a mapping for each node from its logic ID to its physical ID, which consists of a target PE number and a local ID. Once a PPE generates its subset of nodes, it independently schedules these nodes to form a subschedule using the MCP algorithm. Obviously, an accurate ALAP time cannot be obtained if there exists any future edge in the subset. Instead, we use quasi-ALAP binding based on the subgraph only and ignoring all future edges. We expect that strategy to achieve sub-optimal results. These subschedules are then concatenated to form a schedule for the current phase.

### 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 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.

**Communication handling**

Message receiving is handled at the idle time or between completion of one node execution and start of the next ready node execution. 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*.

**Incremental execution handling**

Every PE that has received a *pause* message will finish its current node execution and pauses its current user phase. Before entering the next system phase, the residual nodes as well as corresponding messages need to be processed to incorporated into the new phase. Once a system phase is completed, a new mapping from logic IDs to physical IDs is made available to all target PEs.

**6 Experimental Study**

In this experimental study, we use the Gaussian elimination code as an example. It has been partitioned by defining a grain-size parameter *SN*. that is, *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 × 2K. *SN* is set to be 8 and the total number of tasks is 32966.

The Hypertool/2 has been implemented on an Intel Paragon machine. First, we study the system performance, especially the system overhead, which includes task generation, scheduling, and other overhead. Table 1 shows an analysis of execution time on 16 TPEs. The time unit is second. The computation time is the time of executing tasks. By parallelizing graph generation and scheduling, the system overhead has been reduced from 126.1 seconds on a single PE to 64.7 seconds on eight PEs.

**Table 1.** Timing on Various Numbers of PPEs (Number of TPE is 16).

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

PTGDE is a work similar to ours [2]. 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 [2]. Table 2 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.

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

Number of PEs		1	2	4	8	16	32
Hypertool/2	Time (S)	5542	2788	1431	766	423	268
	Speedup	1.00	1.99	3.87	7.23	13.1	20.7
PTGDE	Time(S)	—	5574	1998	1042	808	590
	Speedup	—	1.00	2.79	5.35	6.90	9.45

## 7 Conclusion

This paper described a new approach, which schedules DAGs in parallel and incrementally executes DAGs at runtime. Our experimental results showed that DAGs can be used for large problems, parallel scheduling can scale well, and inaccurate estimation can be tolerated.

### Acknowledgments

This research was partially supported by NSF grants CCR-9505300 and CCR-9625784.

### References

1. I. Ahmad, Y.K. Kwok, and M. Y. Wu. Performance comparison of algorithms for static scheduling of DAGs to multiprocessors. In *Second Australasian Conference on Parallel and Real-time Systems*, pages 185–192, September 1995.
2. M. Cosnard, E. Jeannot, and L. Rougeot. Low memory cost dynamic scheduling of large coarse grain task graphs. In *International Parallel Processing Symposium*, April 1998.
3. H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, June 1990.
4. M.R. Gary and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
5. M. Y. Wu. Parallel incremental scheduling. *Parallel Processing Letters*, 5(4):659–670, December 1995.
6. M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel and Distributed Systems*, 1(3):330–343, July 1990.
7. M. Y. Wu and W. Shu. On parallelization of static scheduling algorithms. *IEEE Transactions on Software Engineering*, 23(8):517–528, August 1997.
8. M. Y. Wu, W. Shu, and Y. Chen. Incremental scheduling and execution of dags. In *IASTED International Conference on Parallel and Distributed Computing Systems*, November 1999.
9. T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed System*, 5(9):951–967, September 1994.