Multi-List Scheduling on Nectar: Software and Experiment

(Abstract)

I-Chen Wu

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

H. T. Kung

Division of Applied Sciences, Harvard University, Cambridge, MA 02138

April 1993

1 Summary

One of the two primary applications for the Nectar testbed has been mathematical optimizations such as the traveling salesperson problem over networks. Unlike many other numerical computations, mathematical optimization involves large-scale searching using methods such as best-first search and divide-and-conquer.

We have found that these searching processes are highly dynamic in the sense that it is impossible to make useful *a priori* estimates on the number of nodes the search tree will explore and the sizes of the nodes. This indeterminacy, plus the possible large load fluctuations in the network, makes the problem of performing mathematical optimization over networks extremely challenging. Efficient run-time, instead of static or compile-time, task scheduling must be used for both balancing loads and minimizing interprocessor communication costs. In the past it would take a large effort to implement any of these run-time scheduling methods. For example, just a few years ago, it took us several months to write thousands of lines of load-balancing code in C when we were parallelizing a solid modeling program (called Noodles [1]) developed by the CMU Engineering Design Research Center.

This paper reports a new approach that has greatly reduced the effort of developing efficient run-time task scheduling software for performing large, dynamic computations over networks such as those mentioned above. The approach, called "multi-list scheduling," is based on a unifying scheduling model involving the use of multiple scheduling lists. The model can capture a large class of scheduling policies, especially those related to searching such as best-first search, divide-and-conquer, and alpha-beta search. Based on the multi-list scheduling model, we have developed software capable of automating the load-balancing operation for the user. This software is operational on the Nectar network at CMU.

As described below, we have successfully applied this multi-list scheduling approach to real-world mathematical optimization problems such as the set covering problem. Using this approach the user needs to write only 10 to 20 lines of code to specify the scheduling policy for run-time load-balancing. When applied to an 8-workstation Nectar network, the approach yielded a speedup of 7.71 for the set covering problem.

The multi-list scheduling approach is, in particular, well-suited for implementing an efficient parallel divide-and-conquer method [6] which employs a provably minimum number of interprocessor communication messages. Measured performance on the Nectar network suggests that, when this efficient divide-and-conquer method is used, the near-linear speedup result above is expected to scale up well for large networks with many fast workstations and with large propagation delays.

In summary, we have made significant progress in solving large mathematical optimization problems over networks. The new multi-list scheduling approach not only substantially reduces programming efforts but also yields excellent performance in terms of speedups. The result is expected to change the way people solve these problems over networks.

2 Multi-List Scheduling Model

Any task scheduling on a single processor is basically a list of runnable tasks possibly ordered in terms of their "priorities" or "costs". However, as this paper will show, when scheduling tasks for a set of processors over a network, the ability of using *multiple* task lists by each processor is important. This multi-list scheduling model is depicted in Figure 1.

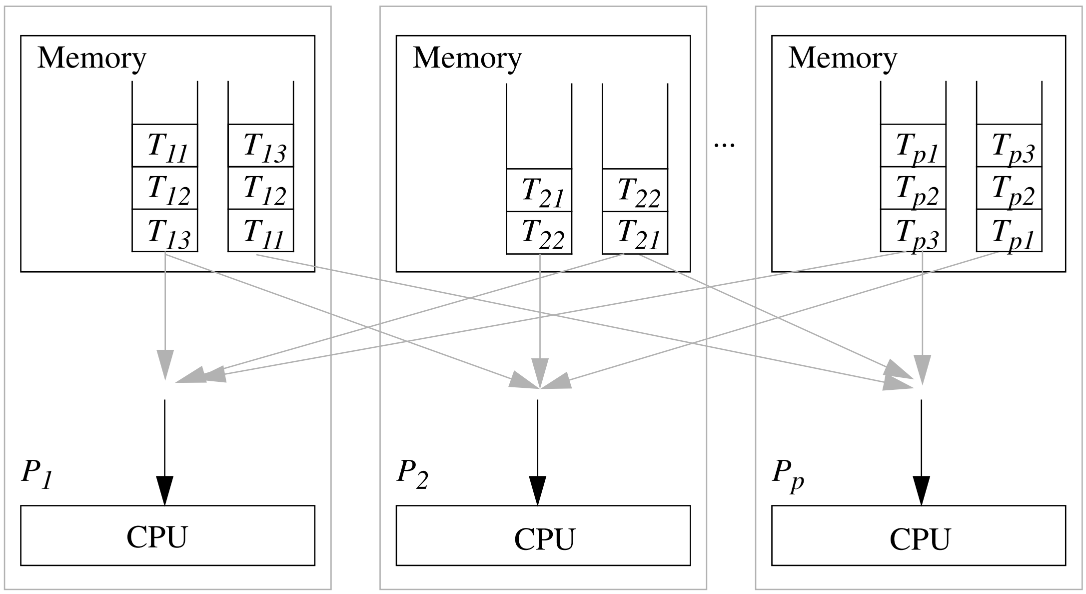

Figure 1 Multi-list scheduling model

As depicted in Figure 1 tasks generated by processor P_i, denoted by T_{ij}, are simultaneously inserted into multiple lists maintained by the processor. There are "list merging patterns", which map the lists to the processors, indicating from which lists each processor receives tasks for execution. For example, the list merging patterns of the Figure 1 indicate that each processor gets its next task for execution from the left list in the local memory or from the right list in the local memory of any other processor. Associated with each list merging patter is a "task selection procedure" which defines how a task is selected from these lists.

It turns out that this simple multi-list scheduling model easily captures a large class of scheduling policies, which in fact include all the interesting ones that we know of. These include divide-and-conquer (D&C) (for e.g., quadtree/octree maintenance problems), best-first search (BFS) (for e.g., traveling salesperson problems), alpha-beta game search problems (for e.g., computer Chess), and synchronous network simulation (for e.g., circuit simulation problems). In the following section, we illustrate the idea using a less obvious case, namely, D&C.

3 Divide and Conquer (D&C) and its Implemention in Multi-List Scheduling

A D&C computation can be viewed as a process of expanding and shrinking a tree. Each node in the tree corresponds to a problem instance; children of the node correspond to its subproblems. During the computation, each internal (non-leaf) node goes through two phases. The first phase is the *divide phase* during which the problem instance associated with the node is divided into subproblems. The second phase is the *combine phase* during which the solution of the problem instance associated with the node is derived by combining solutions of the subproblems associated with the node's children. Each leaf after its creation will perform some computation and return the results to its parent. At a given time, nodes on a wavefront that cuts across all paths from the root to leaves can be active in performing divide, combine, or compute operations. Along each path the wavefront first moves down from the root to its leaf and then up from the leaf to the root. For simplicity of discussion, we will ignore the shrinking phase; let a *frontier node* be defined as a node which has been generated but has not been expanded and a *local frontier node* of a processor be defined as a frontier node whose parent was expanded (or executed) on this processor.

In order to perform D&C efficiently in parallel, Wu and Kung [6] proposed an efficient scheduling algorithm, called the *PDC* scheduling algorithm in this abstract. The *PDC* scheduling algorithm schedules nodes with the following rules. First, if a processor has local frontier nodes, it must schedule the deepest among them. This is the "depth-first" search which can minimize the local memory requirement and avoid wasteful interprocessor communications. Second, when a processor runs out of its local frontier nodes, it schedules a frontier node closest to the root, from all other processors. Note that the node closest to the root is likely to contain the largest subtree, which will have the most locality and therefore will need the least communication.

Wu and Kung also proved that, among all the scheduling algorithms which can split the load nearly evenly, their algorithm is optimal with respect to the *"communication cost*," which is defined to be equal to the total number of "cross nodes". (A cross node is a node which is generated by one processor but expanded by another processor.) Suppose that the network system has p processor, the D&C computation tree has N nodes and height h, and the execution time for each node is one time unit. Then using the scheduling algorithm, the parallel D&C will take time $(N/p) + h$, while the communication cost is $O(ph)$. In general, N grows exponentially in h. Thus, communication cost can be kept negligibly small comparing to the parallel computation time.

The *PDC* scheduling algorithm described above can be easily implemented in the multi-list scheduling model. Each processor will insert nodes (which are tasks) it generates into two lists, *local list* (*LL*) and *glocal list* (*GL*), which are priority queues. A node is inserted to LL with priority $\pi_L = d$ and to GL with priority $\pi_G = -d$, where d is the depth of the node in the tree. The list merging patterns are those shown in Figure 1 with LL and GL being the left and right lists in the memory of each processor. The task selection procedure for each processor is as follows. If list LL is non-empty, select the highest-priority task in the list; otherwise, select the highest-priority task from GLs of all other processors. Note that because $\pi_L = d$, the processor always follows depth-first scheduling to schedule a local task, if one exists; otherwise, because $\pi_G = -d$, the processor always schedules the task closest to the root from processors. Thus, the multi-list scheduling scheme described here realizes the *PDC* scheduling algorithm.

4 Experimental Results

In our experiments for the multi-list scheduling system, we have parallelized a set covering program on Nectar with up to 8 processors. (The number of processors is not large, but is sufficient to validate the approach.) The set covering problem is a real application in industry [4], and the original program we use in this experiment was written by Harche and Thompson [5] of the Graduate School of Industrial Administration at CMU. The problem can be formulated as an integer linear programming using branch-and-bound (B&B) tree search to locate the optimal solution. Both D&C and BFS can be used to implement B&B tree search. Thus this is a good application example to test both parallel best-first search (*PBFS)* and parallel divide-and-conquer (*PDC*) scheduling algorithms in our experiments.

The experiments use four problems, called Problems 1, 2, 3, and 4, of increasing problem sizes. In these four problems, the heights of the search tree are 30, 36, 66, and 71, respectively; the numbers of tree nodes are 1135, 2107, 7097, and 23767, respectively.

To express performance, we use *speedup* and *efficiency* as defined below. Let T_{seq} be the execution time for the original sequential program, and T_p the time for the parallel version of the program using p processors. We define the speedup as $S_p = T_{seq} / T_p$ and the efficiency as $E_p = S_p / p$.

Table 1 lists the performance results for Problems 1 through 4 with the *PDC* scheduling algorithm. These results show that the larger a problem the better the speedup. In particular, on eight processors, the largest problem (Problem 4) has a very good speedup of 7.71.

	p	1 (seq)	1	2	4	8
	T	16.01	16.29	8.51	4.66	3.05
Problem 1	*S*	1.00	0.98	1.88	3.44	5.25
	E	1.00	0.98	0.94	0.86	0.66
	T	31.60	31.97	16.48	8.73	5.01
Problem 2	*S*	1.00	0.98	1.92	3.62	6.31
	E	1.00	0.99	0.96	0.91	0.79
	T	176.58	177.36	89.24	45.79	24.64
Problem 3	*S*	1.00	1.00	1.98	3.86	7.17
	E	1.00	1.00	0.99	0.96	0.90
	T	464.45	466.74	233.38	118.00	60.24
Problem 4	*S*	1.00	1.00	1.99	3.94	7.71
	E	1.00	1.00	1.00	0.98	0.96

Table 1 Measured performance results for the *PDC* scheduling algorithm.

In order to understand the performance results of Table 1, it suffices to examine the overhead and the idle time for Problems 1 and 2. There are two main kinds of overhead: the overhead for scheduling and that for communication. According to our measurements, the scheduling overhead for each task is about 100 microseconds. Since the average task computation time is about 15 milliseconds on average (which is 16.01 seconds/1135), the scheduling overhead is less than 1%. Table 2 shows the average scheduling overhead on each processor for Problems 1 and 2. The scheduling overhead is parallelized well for the following reason. Since we always search the same complete tree no matter how many processors are used, we encounter the same number of tasks in the tree. Since we schedule each task exactly once, we always schedule tasks a fixed number of times. Therefore, the scheduling overhead can be distributed over processors nearly evenly.

	1	2	4	8
Problem 1	0.09	0.04	0.02	0.01
Problem 2	0.18	0.08	0.04	0.02

Table 2 Average scheduling overhead in seconds when using the *PDC* scheduling algorithm.

For the communication overhead, we measure the number of sends/receives for Problems 1 and 2 in Table 3. Since each send/receive is roughly 200 microseconds [2], we can roughly calculate the aggregate communication overhead. For *PDC*, the ratio of the number of sends/receives to the product of the tree height and the number of processors is about 3 to 5, close to a constant. This result is close to the theoretical communication cost [6], which is $O(ph)$ for the worst case.

	Height	1	2	4	8
Problem 1	30	15	188	405	1154
Problem 2	36	8	265	502	1366

Table 3 Number of sends/receives for the *PDC* scheduling algorithm.

Processors become idle mainly in the situations of lack of parallelism and long latency for responding task requests. In our implementation, we use a scheme in which a processor schedules the next task while executing the current task. Since the task granularity is quite large when compared with the latency of task request (each requiring about 3 to 5 sends/receives), processors become idle mainly due to lack of parallelism. Note that if the task request latency is very large (e.g., when using a wide-area network), our system can schedule several tasks at a time so that the latency can be hidden.

	1	2	4	8
Problem 1	0.00	0.30	0.47	0.76
Problem 2	0.00	0.35	0.61	0.76

Table 4 Average idle times in seconds when using the *PDC* scheduling algorithm.

Table 4 shows the idle times for Problems 1 and 2. In general, if the more processors are used, the more likely processors become idle. For the tree in the set covering problem of our experiments, the average branching factor (defined as $N^{1/h}$) is very low, about 1.1 to 1.3. So, for example, when we initially execute the part of the tree near the root, not much parallelism can be exploited. When the number of processors increases, we expect the maximum value of the average idle time to be close to the execution time of all the nodes on the critical path from the root to leaves. Since number of the nodes on the critical path (about $O(h)$) is small compared with the total number of nodes (N), the idle time for the *PDC* scheduling algorithm can be negligibly small.

	p	1(seq)	1	2	4	8
Problem 1	T	16.79	17.71	9.42	5.1	3.52
	S	1.00	0.95	1.78	3.25	4.77
	E	1.00	0.95	0.89	0.81	0.60
Problem 2	T	33.48	34.83	17.85	9.44	5.68
	S	1.00	0.96	1.88	3.55	5.89
	E	1.00	0.96	0.94	0.89	0.74
Problem 3	T	181.32	185.62	94.87	48.31	25.83
	S	1.00	0.98	1.91	3.75	7.02
	E	1.00	0.98	0.96	0.94	0.88
Problem 4	T	475.31	486.84	247.03	125.77	64.81
	S	1.00	0.98	1.92	3.78	7.33
	E	1.00	0.98	0.96	0.95	0.92

Table 5 Measured performance results for the *PBFS* scheduling algorithm.

Table 5 lists the corresponding measured performance for the *PBFS* scheduling algorithm. Note that the *PBFS* results are not as good as the *PDC* results. This is partly because as we noted before, the *PDC* scheduling algorithm employs a provably minimum number of interprocessor communication messages.

For *PBFS*, the idle time and the scheduling overhead are very similar to those for *PDC* and their presence is for the same reasons discussed before. So, we will only discuss the communication overhead for *PBFS* here. Table 6 shows the measured numbers of send/receive pairs for Problems 1 and 2. The numbers for *PBFS* are much larger than those for *PDC*. This is because for *PBFS* whenever a task is scheduled, the minimum-cost task needs to be recomputed across all processors.

	#Nodes	1	2	4	8
Problem 1	1135	770	1190	1695	1846
Problem 2	2107	1672	1963	2716	3543

Table 6 Measured number of send/receive pairs when using the *PBFS* scheduling algorithm.

5 Programming Simplicity

As depicted in Figure 2 and Figure 3, using the multi-list scheduling approach described in this paper, the code for implementing the *PDC* or *PBFS* scheduling algorithm can be very simple. It has only about 10 to 20 lines in our experiments. A program of such a size can be written within tens of minutes.

This is in sharp contrast with previous dynamic load balancing programs, which would typically require thousands of lines in C. This was the case in our earlier experience in parallelizing Noodles [3], a solid modeling program developed in the Engineering Design Research Center at CMU. It took us several months to write such a program!

```
/*** following are in the PBFS.h file ***/
extern void PBFS_Schd();   /* scheduler for PBFS */
ML_List_p      LT;

/* interface for the programmer writing BFS application code. */
#define DECLARE_NODE(T,c) LT->Insert_Task_Pri(T,-c)
       /* T: tree node, c: cost of node T. */
#define INIT_PBFS(g) ML_Init(PBFS_Schd, g, INTKEY)
       /* Initialize the system in which the scheduler
          is PBFS_schd(), the minimum grain size is g,
          and the type of priority key is integer. */

/*** following are in the PBFS.c file ***/
void PBFS_Schd () {
    /* create a list */
    PQ = MP_Create_List("LT");
    /* merge all lists (named "LT") in the entire system */
    MP_Merge_All("LT");
}
```

Figure 2 The code for PBFS

```
/*** following are in the PDC.h file ***/
extern void PDC_Schd();   /* scheduler for PDC */
ML_List_p      LLT, GLT;

/* interface for the programmer writing D&C application code. */
#define DECLARE_NODE(T,d) \
    { LPQ->Insert_Task_Pri(T,d); GLT->Insert_Task_Pri(T,-d); }
       /* T: tree node, d: the tree depth of node T. */
#define INIT_PDC() MP_Init(PDC_Schd, DC_GRAIN, INTKEY)
       /* Intialize the system in which the scheduler is PDC_schd(),
          the minimal grain size is DC_GRAIN (a large number),
          and the type of priority key is integer. */

/*** following are in the PDC.c file ***/
void PDC_Schd () {
    /* create two Lists */
    LLT = MP_Create_List("LLT");
    GLT = MP_Create_List("GLT");
    /* merge the local LLT and all GLTs */
    MP_Merge_Local("LLT");
    MP_Merge_All("GLT");
}
```

Figure 3 The code for PBFS

References

[1] Choi, Y. *Vertex-based Boundary Representation of Non-Manifold Geometric Models*. Engineering Design Research Center, Carnegie Mellon University, 1989.

[2] Cooper, E., Steenkiste, P., Sansom, R., and Zill, B. *Protocol Implementation on the Nectar Communication Processor*. **Proceedings of the SIGCOMM '90 Symposium on Communications Architectures and Protocols**, ACM. Philadelphia, 1990, pp. 135–143.

[3] Fahringer, T., Gubitoso, M., Kung, H., Prinz, F., and Wu, I.-C. *Parallelizing Noodles on Nectar*. internal document, 1989.

[4] Fisher, M. and Kedia, P. *Optimal Solution of Set Covering/Partitioning Problems Using Dual Heuristics*. **Management Science**, vol. 36 (1990), pp. 674–688.

[5] Harche, F. and Thompson, G. *The Column Subtraction Algorithm: An Exact Method for Solving Weighted Set Covering, Packing and Partitioning Problems*. To appear in *Computers and Operations Research*, 1993.

[6] Wu, I.-C. and Kung, H. *Communication Complexity for Parallel Divide-and-Conquer*. **1991 Symposium on Foundations of Computer Science**. San Juan, 1991, pp. 151–162.