트리구조의 계산을 위한 효율적인 동적 부하분산 전략

황 인 재[†] · 홍 동 권[††]

요 약

어떤 응용프로그램에서는 계산구조가 프로그램의 수행도중 동적으로 변한다. 이런 경우 정적으로 태스크를 분할하고 할당하는 것은 병렬컴퓨터에서 높은 성능을 얻는데 충분하지 못하다. 이 논문에서는 동적으로 변하는 트리구조를 가진 계산을 프로세서들에 효율적으로 분배하는 부하분산 알고리즘을 소개한다. 이 알고리즘의 메쉬구조상에서의 구현기법이 소개되고 복잡도가 분석된다. 실험을 통하여 이 알고리즘이 좋은 성능을 나타내는 것을 보인다.

An Efficient Dynamic Load balancing Strategy for Tree-structured Computations

Injae Hwang[†] · Dong-Kweon Hong[††]

ABSTRACT

For some applications, the computational structure changes dynamically during the program execution. When this happens, static partitioning and allocation of tasks are not enough to achieve high performance in parallel computers. In this paper, we propose a dynamic load balancing algorithm which efficiently distributes the computation with dynamically growing tree structure to processors. We present an implementation technique for the algorithm on mesh architectures, and analyze its complexity. We also demonstrate through experiments how our algorithm provides good quality solutions.

키워드 : 병렬컴퓨터(parallel computer), 부하분산(load balancing), 메쉬(mesh), 트리구조(tree structure)

1. Introduction

Parallel processing is one of the most promising approaches to solve computationally intensive problems. These problems arise in many different fields of science and engineering. Even though multiprocessor systems have such an enormous raw computing power, they can be utilized only when the problems are efficiently parallelized. To keep all the processors busy, we have to partition the problem into many components that can be executed in parallel by the processors in the systems. The term "load balancing" refers to the activity of distributing or redistributing the workload among the processors to achieve high performance.

In this paper, we propose an efficient load balancing algorithm for executing algorithms with dynamically changing workload on parallel computers. When the task allocation is performed for parallel computers, task interaction graph is constructed for the application. The task interaction graph represents the amount of computational workload and the communication needed between the tasks. For some algorithms, the task graph is allowed to change during the course of program execution. In this case, static partitioning and allocation of task graph is not enough to achieve high performance in parallel computers. Examples of such algorithms are those that have dynamically growing tree structures. Searching algorithms often employ trees to explore solution spaces. These algorithms are useful methods for solving optimization problems. Adaptive mesh refinement[2] is another example of such algorithms where fine mesh is imposed on those regions with steep curve as the computation proceeds. For these algorithms, it is necessary to reassign workload dynamically in response to the changes in the computational structure of the program. In this paper, we present an efficient dynamic load balancing algorithm which tries to balance the workload among the processors while keeping the communication cost under the acceptable limit. The interconnection networks of parallel

† 종신회원 : 충북대학교 컴퓨터교육과 교수
†† 종신회원 : 계명대학교 컴퓨터 · 전자공학부 교수
논문접수 : 2001년 3월 15일, 심사완료 : 2001년 10월 25일

computers are assumed to be mesh.

2. Related Work

Load balancing and task allocation problems in both distributed and parallel computing environment were extensively studied by many researchers. In this section, we survey some of the related works to this paper.

Hanxleden and Scott [5] developed a testbed for different balancing techniques including scatter decomposition, binary decomposition and some dynamic balancing strategies. They also introduce a simple decentralized balancing strategy in which each node computes the amount of its own workload, and broadcasts it to other nodes as load distribution information ; since this is just a single number, there is little communication overhead. These approaches were implemented on Intel iPSC/2 distributed memory parallel computer and their effects on the message size and the number of messages were observed.

Kopidakis [6] proposed two heuristic algorithms for the problem of task allocation in heterogeneous distributed systems. The objective is the minimization of the sum of processor execution and intertask communication costs. They transform the problem to a maximization one, where they try to determine and avoid large communication costs and inefficient allocations. Their performance is evaluated through an experimental study.

So far, we surveyed a few load balancing methods and most of them are based on static strategies. There are also many papers which propose load balancing algorithms for dynamically changing workload. we survey a few of them in the rest of this section. Pilkington and Baden discussed a partitioning strategy for non-uniform scientific computations running on distributed memory MIMD parallel computers[8]. They considered the case of a dynamic workload distributed on a uniform mesh, and compared their method against other two methods. It was shown that their method is superior to the other two in rendering balanced workloads.

A parallel method for the dynamic partitioning of unstructured meshes was developed by Walshaw and Cross [9]. The method introduced a new iterative optimization technique known as relative gain optimization. Experiments indicated that the algorithm provided partitions of an equivalent or higher quality to static partitioners and much more rapidly. The algorithm also resulted in only a small fraction of the amount of data migration compared to the static partitioners.

Most of the approaches discussed so far fall into one of the two categories ; centralized and decentralized. In centralized schemes, load balancing decisions are made by a central processor. In decentralized schemes, each processor has to make its own decisions about load balancing after collecting the necessary status information from only a subset of all the processors. It also takes less time to collect the information from the subset of processors, and it is not necessary to broadcast the results of load balancing decisions. Centralized schemes however, have the advantage of making more accurate decisions over decentralized schemes.

In the approach proposed in this paper, global workload information is used to make the decision on the redistribution of workload since it is more accurate than local workload information used by decentralized methods. However, by making all the processors work on making load balancing decision, we can obtain the quicker solution. In addition to that, the results of the decision need not be broadcast to other processor. The actual workload migration occurs after making the decision on workload redistribution, and usually this step can take more time than the load balancing activity itself. In our approach, the workload migration can take place while the task distribution is being computed, which results in reduced load balancing overhead. The above advantages are the major motivations for developing the load balancing strategy proposed in this paper.

3. Problem Formulation

There are many algorithms which have tree-structured task graphs. In such algorithms, the computation starts with root node and the tree grows dynamically as each node produces its children. A node in the tree corresponds to a task and an edge corresponds to communication between the parent node (task) and the child node (task). After producing children, the parent waits until it receives results from their children, then terminates. When the child tasks are executed on the processors different from those on which the parent task is executed, inter-processor communication is necessary for migrating the child tasks and receiving messages from them.

With the above tree structure, the problem is assigning the nodes of a dynamically growing tree to processors. This is a very difficult problem since we can not predict the future growth of the tree. To make the problem more tractable, we synchronize the computation on all the processors at each level i of the tree. The tasks at level i in the tree are generated and assigned to processors at the same time. The

child task should be migrated from the processor where it was produced to the processor where it is to be executed. After the execution of the child task is done, the result should be sent back to the parent task. If we assume that the computational cost of each task and the size of message between the parent and child are known, then our problem becomes one of distributing the child tasks to processors, so that computational workload is balanced among processors and the maximum communication cost is minimized. To formulate the execution time of tasks at level i, we introduce the following notations :

$G=(V,E)$: processor graph where V is the set of processor nodes and E the set of communication links
S : set of tasks generated at level i
E_a : execution time of task a
M_a : cost of sending the message generated by task a to an adjacent processor
B_a : cost of sending task a to an adjacent processor
$d(p_1, p_2)$: communication distance between processors p_1 and p_2 in G
$f : S \rightarrow V$: $f(a)$ is the processor where task a was generated
$g : S \rightarrow V$: task assignment function ; $g(a)$ is the processor where task a is executed
$h : V \rightarrow 2^S$: inverse of g ; $h(p)$ is the set of tasks assigned to processor p

Then the total execution time of tasks at level i is given as follows.

$$T_i = \max_{p \in V}\left(\sum_{a \in h(p)} E_a\right) + \max_{a \in S}(d(f(a), g(a)))(B_a + M_a)$$

The first term represents the sum of computational workloads of the tasks assigned to processor p and the second term represents the inter-task communication cost. Then our load balancing problem can be stated as follows : Given G, S, f determine g (and hence h) such that T_i is minimized.

This problem can easily be shown to be intractable. If we ignore communication cost and assume that there are only two processors, partitioning problem can be reduced to this problem. Since partitioning problem was already shown to be NP-complete [4], this problem is NP-hard.

4. Proposed Approach

In the approach we propose, we treat inter-task communication cost as a constraint and try to find an allocation of tasks which minimizes the maximum computational cost while satisfying the constraint. With this strategy, the formulation of our load balancing problem is modified as follows :

Find f (and hence g) such that $\max_{p \in V} \sum_{p \in h(p)} E_a$ is minimized with the constraint that $\max_{a \in s} d(f(a), g(a))(B_a + M_a) \leq C_{\max}$, where $C_{\max}$ is the acceptable limit for inter-task communication cost. The parameter $C_{\max}$ provides the desired trade-off between computational and communication costs and can be set based on the nature of the application and the architectural parameters such as ratio of communication bandwidth to computational capacity in the system. When the execution time of each task E_a and the number of processors N are given, $C_{\max}$ is set to $\frac{\sum_a E_a}{N}$ so that the communication cost cannot exceed the average computational cost per processor. In our heuristic algorithm, $C_{\max}$ will be used to limit the maximum distance that each task can migrate from the processor where it was generated.

In the heuristic we propose, we first assign a label t_a to each task a that indicates the maximum distance it can migrate from its currently assigned processor. Initially, $t_a = \lfloor \frac{C_{\max}}{B_a + M_a} \rfloor$, that is, t_a is inversely proportional to the amount of communication required for the task. Each time a task needs to move by distance d during a balancing step, t_a decreases by d and when t_a becomes zero it remains assigned to its current location.

For balancing the computational load, we use a recursive procedure, that is, first balancing load between two halves of processors and then applying the procedure recursively to the two halves. In order to balance the workload between two halves, we order the tasks in each half in non-increasing order of t_a values (only tasks with strictly positive t_a values are considered). We move tasks from overloaded half to under-loaded half in the above order. The intuition behind this order is that if tasks with larger t_a values are allowed to migrate first, they can also migrate during the later iterations of the algorithm allowing a better chance to balance the workload. Also from the point of view of task migration cost, smaller sized tasks are preferred over larger sized tasks.

We consider an approach for implementation of this heuristic on multiprocessor systems. In this approach, each processor first creates a "token" or a "packet" for each task that it has. This token contains information about the task such as the amount of communication required, presently assigned location(processor), t_a value etc. During the load

balancing procedure, every processor first broadcasts all the tokens it has, to every other processor in the system. During the first iteration, each processor does the same computation, namely attempt to balance the load between two halves of processors using the proposed heuristic. During the second iteration, all the processors in a half do the same computation, namely attempt to balance the load between two quadrants of that half. Thus this process proceeds up to $\log N$ (assuming N to be a power of 2) iterations, where during the i-th iteration ($0 \le i \le \log N-1$), a group of $\frac{N}{2^i}$ processors performs the same task of balancing workload between two halves of the group. In the next section, we give details of the algorithm and analyze its time complexity.

5 Load balancing Algorithm

In describing our algorithm, we use a mesh architecture though the algorithm is easily applicable to other regular topologies such as hypercubes. The number of processors is assumed to be N ($\sqrt{N}$ processors in each row and column labeled from 0 to $\sqrt{N}-1$). Each processor is denoted by $P_{i,j}$ ($0 \le i \le \sqrt{N}-1,\ 0 \le j \le \sqrt{N}-1$). The detailed description of the algorithm is as follows.

1. Generate a packet $\langle E_a, t_a, C_a \rangle$ for each task a to be allocated where
   E_a : Execution time of task a
   t_a : The maximum distance task a can migrate from its current location
   C_a : Index of the processor to which task a is currently assigned Initially, C_a is set to the index of the processor where task a was generated
2. Each processor broadcasts the packets generated in previous step to all other processors using multinode broadcast algorithm [3].
3. After processors receive the packets, they execute the following algorithm.

Procedure LoadBalancing
1. $V\_cut = 1$ // When V_cut is 1, workload is balanced between left and right sets of processors //
2. for $k = 0$ to $\log N - 1$ do
3. if $V\_cut = 1$ then
4. for each $P_{x,y}$ do in parallel
5. Let A be the left set of processors
6. Let B be the right set of processors
7. Call **MigrateTasks**(A, B)
   else // if $V\_cut = 0$ (When V_cut is 0, workload is balanced between upper and lower sets of processors in the same way) //
8. $V\_cut = (V\_cut+1) \bmod +2$

end Procedure LoadBalancing

Procedure MigrageTasks (A, B)
1. Let L_A be $\sum_{C_a \in A} E_a$
2. Let L_B be $\sum_{C_a \in B} E_a$
3. $\triangle_L = \frac{|L_a - L_b|}{2}$
4. if $\triangle_L > \varepsilon$ then // ε is a small number denoting load imbalance tolerance limit //
5. if $L_A > L_B$ then // Move tasks from A to B. //
6. let $S = \{a \mid C_a \in A, E_a \le \triangle_L \text{ and } t_a > 0\}$
7. Find $D_{\min} = \min\{t_a \mid a \in S\}$ and $D_{\max} = \max\{t_a \mid a \in S\}$
8. Partition S into $S_{D_{\min}}, \cdots, S_{D_{\max}}$ where $S_j = \{a \in S \mid t_a = j\}$.
9. for $n = D_{\max}$ to $D_{\min}$ do
10. Let $T_n = \{a_1, \cdots, a_l\}$ where $a_i \in S_n$.
11. Call **SelectTask** ($T_n, \triangle_L, r$)
12. for each task $a_i \in T_n$
13. if $E_{a_i} < E_{a_r}$ then
14. Send task a_i to the nearest processor in B, and change C_{a_i} to the index of the processor.
15. $L_A = L_A - E_{a_i}$, $L_B = L_B + E_{a_i}$
16. $\triangle_L = \frac{|L_A - L_B|}{2}$
    endif
17. if $\triangle_L \le \varepsilon$ then return
    endfor
    endfor
    else // if $L_A < L_B$ //
18. do similar task migration from B to A.
    endif

end Procedure MigrageTasks

Procedure SelectTask ($W, \triangle, r$)
Given a set of tasks $W = \{a_1, a_2, \cdots a_l\}$, and a positive integer $\triangle$, return an index r such that the following is true : $\sum_{i \in W_r} E_i < \triangle$ and $\sum_{i \in W_r'} E_i < \triangle$
where $W_r = \{1 \le i \le l \mid E_{a_i} < E_{a_r}\}$ and $W_r' = \{1 \le i \le l \mid E_{a_i} \le E_{a_r}\}$
This procedure is just a weighted selection problem and can be solved by a divide-and-conquer algorithm just as in the selection problem [1] ; we omit the details here.

end Procedure SelectTask

In the above procedure, all the processors compute the same task distribution between the left and right halves of the processors during the first iteration of the loop. During the second iteration, each half of the processors compute the task distribution between upper and lower quadrant of the processors. During the final iteration, each pair of processors compute task distribution between them. By having many duplications of the same computation among processors, the result of the computation does not have to be broadcast to

other processors. After each iteration, all the processors have the necessary information for task migrations between the two sets of processors. All the processors use the same tie-breaking policy to get the same result for identical computation. At line 14 in procedure **MigrateTasks**, the selected task can be sent to any processor in the set B. Since it can further migrate to other processor in the later iteration, sending it to a particular processor is not necessary. And, the processor does not need to wait until the execution of the algorithm is completed. Since computing task distribution needs only packets not actual tasks, computation can proceed while tasks are being migrated. Therefore, task migration and computing task distribution can overlap in our algorithm. If task a was selected to be migrated but it is not available yet, the processor just need to remember its destination, and it is sent to that destination when available.

The analysis of time complexity of our load balancing algorithm is as follows. In step 1, a packet is generated for each task. This will take $O(m_{\max})$ time where $m_{\max}$ is the maximum number of packets a processor generates. For step 2, following result will be useful. Suppose there is a linear array of N processors, each having a certain number (not the same number) of packets with a total of M packets. Broadcasting can be achieved in $O(M+N)$ steps where at each step, a processor can send (and receive) a packet to (from) its neighbors. From this result, step 2 takes $O(M+N)$. Let's derive the time complexity of step 3, namely the procedure **LoadBalancing**. Lines 1 - 8 in procedure **MigrateTasks** takes $O(M)$ time. Since procedure **SelectTask** takes $O(|W|)$ time, lines 9 - 17 take $O(M)$ time on for all the iterations of loop 9. Procedure **MigrateTasks** is called $\log N$ times. Hence, step 3 takes $O(M\log N)$ time. Including all the four steps, the total time complexity of our load balancing algorithm is $O(N+M\log N)$.

6 Experimental Results

We performed simulation to test the accuracy of solutions provided by our load balancing algorithm. We randomly generated the trees, and applied the proposed load balancing strategy to assign the tasks to processors. For the first few generations, the number of tasks can be much smaller than the number of processors, and load balancing is not important during that time. Therefore, we assumed that the processors initially have a certain number of tasks (No_Tasks) and the number of child tasks created has an exponential distribution with a mean value of 1. We also assumed that the computational costs of tasks have a uniform distribution in the range [MIN, MAX]. The communication cost between a parent task and a child task is assumed to be proportional to the computational cost of the child task with the parameter C-COST being the constant of proportionality.

Since finding optimal solutions takes too much time, we used the following four yardsticks to compare our algorithm against : (a) average amount of computation per processor which is a trivial lower bound (referred to as **LB**) for any solution, (b) solution value for list scheduling heuristic with communication constraint, (c) solution value for decreasing fit heuristic, (d) solution value for Kopidakis' task assignment algorithm[6]. In the rest of the section, we use L_1, L_2, L_3 and L_4 to denote respectively our load balancing algorithm, list scheduling, decreasing-fit heuristics and Kopidakis' algorithm.

In our experiment, we started with 5 tasks per processor and then went through 1000 generations. We obtained the average objective function values for the heuristics under consideration. The following tables respectively indicate how the performance of the heuristics vary with C-COST, number of processors, variance of execution times, initial number of tasks per processor (No_tasks) and average execution time of tasks.

As can be seen in <table 1>, L_1 and L_4 give good solutions regardless of C-COST while L_3 performs better only for very small values of C-COST when balancing workload plays a more important role than reducing communication costs. When C-COST is larger than 0.1 (which means that communication cost is more than 10% of computational cost), L_1 and L_4 perform better than L_2 and L_3. When the distance between the two processors is relatively large, as in meshes, L_1 has the advantage over L_4 because the communication cost is constrained under a certain value while workload is balanced.

With respect to sensitivity to number of processors, L_1, our heuristic performs the best in most of the cases as can be seen in <Table 2>. This can be attributed in part to the ability of the algorithm to keep the communication cost to a minimum while retaining the opportunity to balance the workload among the processors.

<Table 1> Solution values for different C-COST values

C-COST	LB	L_1	L_2	L_3	L_4
0.01	51.6	59.5	58.5	57.1	57.5
0.1	51.6	61.7	61.1	59.6	60.2
0.2	51.6	63.5	64.4	72.7	63.1
0.3	51.6	65.9	69.5	77.5	66.7
0.4	51.6	68.6	74.0	90.9	69.1
0.5	51.6	70.6	76.8	97.2	72.8

〈Table 2〉 Solution values for different numbers of processors

No.of Proc.	LB	L_1	L_2	L_3	L_4
32	45.0	54.3	58.2	61.5	54.9
64	46.3	58.5	60.1	64.7	58.1
128	51.6	65.9	69.5	77.5	66.4
256	64.6	76.9	80.4	89.1	77.7
512	68.2	83.4	89.3	105.8	84.1

〈Table 3〉 Solution values for different variances of execution times

MIN-MAX	LB	L_1	L_2	L_3	L_4
8-12	51.5	62.2	64.9	75.6	63.0
6-14	51.6	62.7	65.2	75.8	62.2
4-16	51.6	65.9	69.5	77.5	66.7
2-18	51.7	66.6	70.1	77.7	66.8
0-20	51.7	68.3	71.8	78.9	68.1

With respect to variance of execution times, we see that L_3 is not much influenced while the objective function values obtained by L_1, L_2 and L_4 increase with large variation in execution times of tasks. But since L_1 keeps the communication cost small, the total cost is smaller than that of L_2 or L_3. L_4 performs similarly as L_1. The reason for smaller communication cost is attributed in part to the fact that communication cost for a task is proportional to its execution time and with wider variation in execution times, tasks with heavy communication costs are not allowed to migrate too far while tasks with light communication costs can migrate further.

7 Conclusions and Future Work

In this paper, we discussed a dynamic load balancing problem that arises in mapping computations with tree-structured task graphs onto multiprocessor systems. We formulated the objective function which includes both computation cost and communication cost between tasks. The heuristic algorithm we proposed in this paper tries to minimize the maximum computation time among the processors while keeping the communication time under a certain limit. In addition to analyzing its complexity, we also experimentally analyzed the accuracy of solutions provided by our heuristic algorithm.

Our future work will be concerned with the more difficult task of developing an efficient load balancing algorithm which can be applied to arbitrary task graphs where communication can take place between any pair of tasks. Especially, we are interested in developing load balancing algorithms which are useful for massively parallel architectures.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company, 1974.

[2] I. Altas and J. Stephenson, "A Two-Dimensional Adaptive Mesh Generation Method," Journal of Computational Physics, Vol.94, pp.201-224, 1991.

[3] D. P. Bertsekas and J. N. Tsitsiklis, Parallel and Distributed Computation, Prentice-Hall, Englewood, New Jersey, 1989.

[4] M. R. Garey and D. S. Johnson, "Computers and Intractability," W. H. Freeman and Company, San Francisco, CA, 1983.

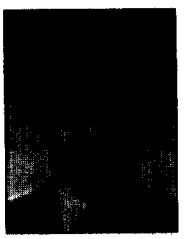
[5] R. Hanxleden and L. R. Scott, "Load Balancing on Message Passing Architectures," Journal of Parallel and Distributed Computing, Vol.13, pp.312-324, 1991.

[6] Y. Kopidakis, M. Lamari and V. Zissimopoulos, "On the Assignment Problem : Two New Efficient Heuristic Algorithms," Journal of Parallel and Distributed Computing, Vol. 42, pp.21-29, 1997.

[7] C. Leangsuksun and J. Potter, "Designs and Experiments on Heterogeneous Mapping Heuristics," Proceedings of 1995 Workshop on Heterogeneous Processing, pp17-22, 1994.

[8] J. R. Pilkington and S. B. Baden, "Dynamic Partitioning of Non-Uniform Structured Workloads with Spacefilling Curves," IEEE Transactions on Parallel and Distributed Systems, Vol.7, No.3, 1996.

[9] C. Walshaw, M. Cross and M. G. Everett, "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes," Journal of Parallel and Distributed Computing, Vol.47, No.2, pp102-108, 1997.



황 인 재

e-mail : lh@ghost.chungbuk.ac.kr

1986년 충북대학교 컴퓨터공학과 졸업 (공학사)

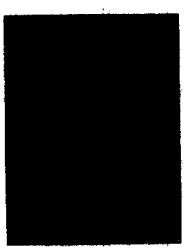
1981년 University of Florida. Computer & Information Sciences 졸업(공학석사)

University of Florida. Computer & Information Sciences 졸업(공학박사)

1986년~1987년 한국 전자통신연구소 연구원

1995년~현재 충북대학교 컴퓨터교육과 부교수

관심분야 : 병렬처리, 병렬컴퓨터 구조, 병렬 알고리즘, 디지털 시스템설계, VLSI 설계 알고리즘



홍 동 권

e-mail : dkhong@kmucc.keimyung.ac.kr

1985년 경북대학교 전자공학과 졸업(학사)

1992년 University of Florida 전자계산학과 졸업(석사)

1995년 University of Florida 전자계산학과 졸업(박사)

1985년~1990년 한국전자통신연구원

1996년~1997년 한국전자통신연구원

1997년~현재 계명대학교 컴퓨터·전자공학부 전임강사

관심분야 : 능동 실시간 데이터베이스, 병렬 처리, 성능 평가, 시뮬레이션, 멀티미디어 처리