병렬계산을 위한 부하분산 알고리즘의 병렬화

Parallelization of A Load balancing Algorithm for Parallel Computations

황 인 재*
In-Jae Hwang*

요 약

본 논문에서는 병렬프로그램을 효율적으로 수행하는데 필수적인 부하분산을 위한 기존 알고리즘의 부하분산 오버헤드를 최소화하기 위하여 이 알고리즘의 병렬화 방법을 제시한다. 병렬계산 모델로는 동적으로 변하는 트리구조를 들었으며 이러한 계산은 많은 응용분야에서 찾아볼 수 있다. 부하분산 알고리즘은 통신비용을 정해진 한도 이내로 유지하면서 프로세서간 계산부하를 최대한 균등하게 분산시키고자 시도한다. 이 알고리즘이 메쉬와 하이퍼큐브 구조에서 어떻게 병렬화될 수 있는가를 상세히 보이고 각각의 경우에 대하여 시간상 복잡도를 분석하여 기존의 알고리즘보다 여러 가지 오버헤드가 개선되었음을 증명한다.

Abstract

In this paper, we propose an approach to parallelize a load balancing algorithm that was shown to be very effective in distributing workload for parallel computations. Load balancing algorithms are required in executing parallel program efficiently. As a parallel computation model, we used dynamically growing tree structure that can be found in many application problems. The load balancing algorithm tries to balance the workload among processors while keeping the communication cost under certain limit. We show how the load balancing algorithm is effectively parallelized on mesh and hypercube interconnection networks, and analyzed the time complexity for each case to show that parallel algorithm actually reduced the various overhead.

Key words : Parallel computation, Load Balancing, Tree Structure, Mesh, Hypercube

Ⅰ. Introduction

Load balancing is the activity of distributing or redistributing the computational load among the processors of a multiprocessor system in order to achieve high performance. In multiprocessor computer systems with distributed memory, load balancing should also take into consideration the inter processor communication cost that arises when a computation in a processor needs data located in another processor's memory.

In this paper, we propose an approach to parallelize a load balancing algorithm that was shown to be very effective in distributing workload while keeping communication cost under a certain limit.[3] The parallel computation model used for the above load balancing algorithm is dynamically growing tree structure that can be found frequently in many application problems. In this model, the computation structure changes dynamically as the computation proceeds, hence ordinary static load balancing approaches cannot be applied. The computation structure can be described with task interaction graph

*충북대학교 컴퓨터교육과
접수 일자 : 2004. 6. 26 수정 완료 : 2004. 7. 22
논문 번호 : 2004-1-18

that represents the amount of computational workload and the communication needed between the tasks. The communication pattern also changes with the change of computation structure, and the problem of finding the allocation of tasks and communication path becomes extremely complicated. In this case, static partitioning and allocation of task graph is not enough to achieve high performance in parallel computers.

To this problem, we present a parallel load balancing algorithm based on the approach previously published. The approach was shown to be very effective in distributing workload among processors, but it is basically a serial algorithm so that a considerable overhead can incur when the number of tasks is large. We present a parallelized load balancing algorithm that can substantially reduce the overhead by having all the processors work on finding the redistribution of workload. The algorithm is shown to be implemented on mesh and hypercube that are most widely used interconnection networks for parallel computers.

II. Related work

Most of the load balancing and task allocation algorithms can be classified as centralized or distributed by the way the workload information is gathered and load balancing dicision is made. In this section, we survey some of them.

Legrand, Renard and Robert [4] proposed a heuristic for mapping iterative algorithms onto heterogeneous clusters. The application data is partitioned over the processors, which are arranged along a virtual ring. At each iteration, independent calculations are carried out in parallel, and some communications take place between consecutive processors in the ring. They established a complexity result that assesses the difficulty of the problem.

Genaud, Giersch and Vivien [2] presented solutions to statically load balance scatter operations in parallel codes run on grids. Their strategy is based on the modification of the data distributions used in scatter operations. They used seismic tomography application to illustrate the benefits of their scheme.

These load balancing methods are based on static strategies. There are also many papers which propose load balancing algorithms for dynamically changing workload. Pilkington and Baden [5] discussed a partitioning strategy for non-uniform scientific computations running on distributed memory MIMD parallel computers. 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 [7]. 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.

Both static and dynamic schemes are either centralized or 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 processors.

III. Problem Formulation and Proposed Approach

The detailed formulation of the problem is given in [3], and we briefly summarize it here. In the dynamically growing tree structure, the problem is assigning the nodes representing computations 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. Then our problem is 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 an 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.

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

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.

In the approach proposed in [3], inter-task communication cost is treated as a constraint, and an allocation of tasks is found that 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 $\max_{a\in s} d(f(a),g(a))(B_a+M_a) \le C_{\max}$, where $C_{\max}$ is the acceptable limit for inter-task communication cost. 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 the heuristic algorithm, first a label t_a is assigned 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, a recursive procedure is used, 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, the tasks in each half are ordered in non-increasing order of t_a values. Tasks are moved from overloaded half to under-loaded half in the above order. 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. In the next section, we describe how the above approach can be efficiently parallelized and analyze its time complexity.

Ⅳ. A Parallel Load Balancing Algorithm

Our parallel load balancing algorithm will be described based on hypercube interconnection network first, and the modification for mesh will be discussed in the following section. In the algorithm, the processors initially generate tokens or packets one for

$\langle E_a, t_a, C_a \rangle$ 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. First, the packets are evenly distributed among the processors. This token distribution step is important for distributing among the processors the workload associated with the remaining steps of the load balancing algorithm. We perform $\log N$ iterations where during the i-th iteration load balancing is performed between two halves of a sub-cube of dimension $\log N - i$; we call each such half a "segment". In each iteration, the following major steps are performed : (a) sort the packets such that all packets whose new processor locations belong to the same segment reside in a set of contiguous processors; within the same segment they are in the order in which the corresponding tasks will be considered for migration, (b) selection of tasks (i.e. packets) for migration. At the end of the algorithm, a processor that holds a packet informs the processor in which the task was generated about its new location. The actual migration of tasks happens only after the load balancing algorithm is completed.

Below we give a formal description of our algorithm. In the description, we use parallel algorithms for well-known computational problems such as segmented scan (prefix), summing, sorting etc. and packet-routing problems such as token distribution, broadcasting etc.. We will refer to the literature for efficient algorithms for these problems.

Algorithm Load balance;

(a) Each processor creates for each task a it has, a packet with the following information : (i) M_a, size of the task, (ii) t_a, maximum distance it can migrate from its currently assigned location, (iii) $f(a)$, index of processor where it was generated and (iv) Q_a, index of the segment containing the processor the task is currently assigned to; this is initially set to the most significant bit of $f(a)$.

(b) Distribute the packets evenly among the processors using **procedure TokenDistribution.**

(c) Perform $\log N$ iterations where in the l-th iteration, call **procedure AdjustDistribution}(l).**

(d) Now each processor may contain packets whose Q_a fields (that indicate final locations of the tasks) are different from their $f(a)$ fields (source locations); we "mark" such packets. Now these marked packets need to be sent to the processors where the corresponding tasks were generated. This is a many-to-many routing problem for which efficient solutions exist when two packets destined for the same processor are allowed to be combined. We call **procedure InformSource** to perform the above task.

end Load balance

Procedure TokenDistribution;

1. Each processor P_i numbers its packets from 1 to m_i where m_i is the number of packets it has.
2. Determine $M_{\max} = \max_i m_i$
3. Broadcast $M_{\max}$ to other processors
4. Let $k = 0$
5. For $j = 1$ to $M_{\max}$ do steps 6-10.
6. Broadcast k to other processors
7. Each P_i sets $a_i = 1$ if it has a packet indexed j, else sets $a_0 = 0$.
8. Compute prefix on $(a_{i\,i=0\,N-1}, b_{i\,i=0\,N-1})$; b_i's are obtained.
9. Each P_i sends its j-th packet to the processor indexed $(k + b_i - 1) \bmod N$
10. P_{N-1} sets $k = (b_{N-1} + k) \bmod N$.

end TokenDistribution

Procedure AdjustDistribution (l)

1. Sort packets according to lexicographical order of the fields $\langle Q_a, t_a, M_a \rangle$ where Q_a and M_a are in increasing order and t_a is in decreasing order. For packet with equal values of Q_a, t_a and M_a, their relative order after packing can be arbitrary. For this, we use a parallel sorting scheme for hypercubes.
2. Each processor numbers its packets from 1 up to $\left|\frac{M}{N}\right|$. Let PS_j be the set of contiguous processors that have packets a with $Q_a = j$, $(0 \le j \le 2^l - 1)$. Assume these sets are disjoint.

3. Concurrently compute workload difference on each processor. Now each processor in P_i knows Δj, half the load difference between two halves of its segment namely P_i and $P_{j+(-1)^j}$; let $g_j = 1$ if the load in PS_j is greater is greater than the load in $PS_{j+(-1)}$ and $\Delta_j > \varepsilon$, 0 otherwise. Let $PS_j^k \subseteq PS_j$ be the set of contiguous processors that have packets a with $Q_a = j$ and $t_a = k$, where $0 \le j \le 2^{l-1}$ and $0 \le k \le \log N$.
4. Compute prefix on M_a's to obtain M_a^{jk}, for all $0 \le j \le 2^l - 1$ and $1 \le k \le \log N$. Here M_a^{jk} is the sum of M_b's of packets b with $t_b = k$ and which exist in those processors of PS_j^k indexed smaller than the processor containing packet a and of those packets indexed no greater than a in the same processor containing a.
5. Mark packets to be migrated to the other halves of their segment.
6. Each processor does the following for each marked packet a it has: complement the l-th most significant bit of Q_a and decrease t_a.
7. if $l \neq \log N$ then each processor updates the segment number of each packet a that it has, by concatenating the $(l+1)$-th most significant bit of $f(a)$ to the right of Q_a.

end AdjustDistribution;

Procedure InformSource;

1. Sort the marked packets according to their $f(a)$ fields using a parallel sorting scheme for hypercubes. For $0 \leqq j \leqq N-1\$$ let T_j be the set of contiguous processors holding packet a with $f(a) = j$
2. Compute prefix on these packets where the associative operation is defined to be merging of two packets with the same destination into one packet. After this step, the least indexed processor in T_j keeps the merged packet while others in T_j discard their outputs. Now we have at most one packet destined for a processor but a processor may have more than one such packet to send. Each processor P_i that has at least one such packet, indexes the packets (in the order of their destinations) from 1 to m_i' where $1 \le m_i' \le \frac{M}{N}$
3. for $j = 1$ to $\frac{M}{N}$ do the following :

   Each processor routes its j-th packet if it exists to its destination.

end InformSource

Time complexity analysis :

Procedure **TokenDistribution** takes $O(M_{\max} \log N)$ time derived as follows : Step 1 takes $O(M_{\max})$ time and steps 2 and 3 take $O(\log N)$ time; steps 6, 8 and 9 all take $O(\log N)$ time for each iteration of step 5. Computing workload difference takes $O(\frac{M}{N} + \log N)$ time as the operations performed in it are summing and segmented scan. By the same reasoning, selecting packets to be migrated also takes $O(\frac{M}{N} + \log N)$ time.

Now the complexity of procedure **AdjustDistribution** is analyzed as follows : step 1 takes $T_s(M, N)$ time which is the time to sort M items on N processors with approximately equal number of items (in fact $O(\frac{M}{N})$ items) per processor. Step 2 takes $O(\frac{N}{M})$ time and each of the steps 3, 4 and 5 takes $O(\frac{M}{N} + \log N)$ time; steps 6 and 7 take $O(\frac{M}{N})$ time. Hence the complexity of **AdjustDistribution** is $O(T_s(M, N) + \frac{M}{N} + \log N)$ time. When $M \le N$, we can use odd-even merge sort [1] to sort the packets in $O(\log^2 N)$ time. When $M > N$, M packets can be sorted in $O(\frac{M \log M}{N})$ time on a hypercube using the algorithm given in [1].

The complexity of procedure **InformSource** is derived next. Step 1 takes $O(T_s(M', N))$ time where M' is the number of marked packets; note that the number of marked packets per processor is at most $\frac{M}{N}$. Hence $T_s(M', N) = O(\log N)$ when $M \le N$ and equal to $O(\frac{M \log M}{N})$ otherwise. Step 2 takes $O(\frac{M}{N} + \log N)$ time and step 3 takes $O(\frac{M}{N} \log N)$

time. Hence the complexity of InformSource is $O(T_s(M,N)+\frac{M}{N}\log N)$.

Now we are ready to analyze the complexity of our load balancing algorithm. Step (a) takes $O(M_{max})$ time and step (b) takes $O(M_{max}\log N)$ time. Step (c) takes $O(T_S(M,N)+\frac{M}{N}+\log N)$ time for each of $\log N$ iterations. Thus when $M\leq N$, the algorithm has complexity $O(M_{max}\log N+\log^3 N+\frac{M}{N}\log N)$. When $M>N$, the algorithm has complexity $O(M_{max}\log N+\frac{M\log M}{N}\log N+\frac{M}{N}\log N)$. This time complexity is much lower than $O(N+M\log N)$ which was given in [3]. The improvement is substantial especially when the number of processors and the number of tasks are large.

V. Mesh Network

Our parallel load balancing algorithm can be implemented on a mesh with only a few changes. The workload is balanced between left and right halves of processors, then between upper and lower quadrants of processors and so on. We need to go through $\log N$ steps as in a hypercube. Other parts of the algorithm remain unchanged. For the analysis of time complexity of our load balancing algorithm on a mesh, we first consider parallel algorithms for well-known computational and routing problems used in our load balancing algorithm and their time complexities on a mesh. The number of processors is assumed to be N. Also we assume that the processors in the mesh are ordered in a snake-like row major order.

Time complexity analysis for a mesh :

Broadcasting a value to all the N processors by a processor takes $O(\sqrt{N})$ time. Prefix computation of M segmented inputs with approximately $\frac{M}{N}$ inputs per processor takes $O(\frac{M}{N}+\sqrt{N})$ time [1]. Finding the maximum or sum of M values and storing it in some processor also takes $O(\frac{M}{N}+\sqrt{N})$ time when there are approximately $\frac{N}{M}$ inputs per processor. Sorting N numbers (with one in each processor) takes $O(\sqrt{N}\log N)$ time using shearsort. [1]

Now we analyze the time complexity of our load balancing algorithm as follows. Procedure TokenDistribution takes $O(M_{max}\sqrt{N})$ time derived the same way as for a hypercube. Computing workload difference and selecting packets to be migrated takes $O(\frac{M}{N}+\sqrt{N})$ time. The complexity of procedure AdjustDistribution is analyzed as follows : step 1 takes $T_s(M,N)$ time which is the time to sort M items on N processors with approximately equal number of items per processor. Sorting on a mesh takes $O(\frac{M}{\sqrt{N}}\log N)$ time. Step 2 takes $O(\frac{M}{N})$ time and each of the steps 3, 4 and 5 takes $O(\frac{M}{N}+\sqrt{N})$ time; steps 6 and 7 take $O(\frac{M}{N})$ time. Hence the complexity of AdjustDistribution is $O(T_s(M,N)+\frac{M}{N}+\sqrt{N})$ time.

The complexity of procedure InformSource is derived next. Step 1 takes $O(T_s(M',N))$ time where M' is the number of marked packets; note that the number of marked packets per processor is at most $\frac{M}{N}$. $T_s(M',N)=O(\frac{M}{\sqrt{N}}\log N)$. Step 2 takes $O(\frac{M}{N}+\sqrt{N})$ time and step 3 takes $O(\frac{M}{\sqrt{N}})$ time. Hence the complexity of InformSource is $O(T_s(M,N)+\frac{M}{\sqrt{N}})$.

Now we are ready to analyze the complexity of our load balancing algorithm for a mesh. Step (a) takes $O(M_{max})$ time and step (b) takes $O(M_{max}\sqrt{N})$ time. Step (c) takes $O(T_s(M,N)+\frac{M}{N}+\sqrt{N})$ time for each of $\log N$ iterations. The algorithm has complexity $O(M_{max}\sqrt{N}+\frac{M}{\sqrt{N}}\log^2 N+\frac{M}{\sqrt{N}})$.

VI. Conclusions

In this paper, we proposed an approach to parallelize a load balancing algorithm that was shown to be very effective in distributing workload while keeping communication cost under a certain limit.[3] The

parallel computation model used for the load balancing algorithm is dynamically growing tree structure that can be found frequently in many application problems. We formulated the objective function which includes both computation cost and communication cost between tasks. The parallel load balancing algorithm we proposed, can substantially reduce the load balancing overhead compared to the previous approach.[3] This goal was achieved by having all the processors cooperate to find the new distribution of the workload and communication path to be used. The algorithm was described based on hypercube and mesh interconnection networks, and its time complexity was analyzed for each case.

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

[2] S. Genaud, A Giersch and F. Vivien, "Load Balancing Scatter Operations for Grid Computing," Proceedings of the International Parallel and Distributed Processing Symposium, April 2003.

[3] I. Hwang, D. Hong, "An Efficient Dynamic Load Balancing Strategy for Tree-structured Computations," Korea Information Processing Society Journal, Vol 8-A, No. 4, pp.455-460, 2001. 12.

[4] A. Legrand, H. Renard, Y. Robert, "Mapping and Load Balancing Iterative Computations," IEEE Transactions on Parallel and Distributed Systems, Vol. 15, No. 6. June 2004, pp. 546-558.

[5] 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.

[6] L. Prechelt and S. U. Hanssgen, "Efficient parallel execution of irregular recursive programs," IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 2. February 2002 pp. 167-178.

[7] 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, 1997, pp102-108.

황 인 재 (In-Jae Hwang)
1986년 충북대학교 컴퓨터공학과 공학사
1991년 Univ. of Florida 전산학과 공학석사
1994년 Univ. of Florida 전산학과 공학박사
1995년~현재 충북대학교 컴퓨터교육과 부교수
관심분야 : 병렬 및 분산처리, 알고리즘