Analysis of scheduling parallel tasks on hypercube systems

J.-F. Lin and S.-J. Chen

Indexing terms: Multiprocessing systems, Parallel processing, Scheduling

The problem of scheduling independent parallel tasks with the consideration of setup time on a d -dimensional hypercube system is investigated. The objective of this problem is to find a schedule with minimum finish time; such a scheduling problem is NP-hard. Therefore, a heuristic algorithm is proposed and its performance bound derived.

Introduction: The problems of scheduling tasks on multiprocessor systems have been extensively studied for a long time. Conventional approaches were proposed for the problem of scheduling each task on only one processor at a time, which is referred to as the conventional task scheduling problem. Finding a minimum finish time nonpre-emptive schedule for this type of problem on an m -processor system, where $m \geq 2$, is NP-complete. Recently, the problem of parallel task scheduling [1, 4, 6] has been introduced. Consider a set of n independent parallel tasks $T = (T_1, T_2, \dots, T_n)$ to be processed on a d -dimensional hypercube, each task T_i having a computational requirement t_i . Then assume that each task T_i is associated with a maximum parallelism dimension Δ_i , that is, each task T_i can be processed at most on a Δ_i -dimensional subcube and this parallelism dimension, once decided for T_i , will not be altered during its processing. If a task T_i is scheduled to run on a d -dimensional subcube $0 \leq d_i \leq \Delta_i \leq d$, then d_i will be called the scheduled dimension of T_i and the execution time required by T_i will be $(t_i/2^{d_i})$. For this problem type, a schedule is feasible if the scheduled dimension d_i of each task T_i is no greater than its maximum parallelism dimension Δ_i . A feasible schedule is called an optimal schedule if it has the minimum finish time. A heuristic algorithm has a performance bound of β if $(S_H/S_O) \leq \beta$ for all problem instances, where S_H and S_O denote the finish times of a heuristic schedule and that of an optimal schedule, respectively.

Under the linear speedup assumption, Wang and Cheng [4] proposed an ECT (earliest completion time) algorithm for scheduling parallel tasks and derived a performance bound of $(3 - 2/m)$ in which the number of processors required by each task and the number of processors m in systems are arbitrary. Also, Zhu and Ahuja [6] proposed an LDF (largest dimension first) algorithm for scheduling parallel tasks on a d -dimensional hypercube system and derived a performance bound of $(2 - 1/2^d)$.

Maximum parallel dimension policy: Under the ideal assumption, previous parallel task scheduling algorithms [4, 6] assumed that a task can be partitioned into equal size subtasks and that no overhead is required. That is, the setup time caused by switching from processing a task to processing another task need not be included in their performance analysis. But in fact, the overhead of switching is high and must be taken into consideration. In this Letter, the setup time is assumed to be sequence independent and constant [2, 3, 5], i.e. the time taken to switch a processor to a new task is independent of the task last processed and is a constant s . Thus, the total processing time $P(x, t_i)$ required for processing a task T_i is equal to $[(t_i/2^x) + s]$, where $1 \leq x \leq \Delta_i$. By simple calculus, we then have $\partial P(x, t_i)/\partial x = -[(\ln 2)/2^x]t_i$ and $\partial^2 P(x, t_i)/\partial x^2 = [(\ln 2)^2/2^x]t_i$. As $\partial P(x, t_i)/\partial x < 0$ and $\partial^2 P(x, t_i)/\partial x^2 > 0$ for $1 \leq x \leq d$, $P(x, t_i)$ is a decreasing function which reaches its minimum value at Δ_i . Based on the above result, an intuitive strategy is to let the scheduled dimension d_i of task T_i be assigned to its maximum dimension of parallelism Δ_i , which will lead to a minimum processing time $P(d_i, t_i) = [(t_i/2^{\Delta_i}) + s]$.

Procedure maximum-parallel-dimension: { Assign Δ_i to the scheduled dimension d_i of each task T_i , for $i = 1, 2, \dots, n$. }

With the above scheduled dimension d_i for each parallel task T_i , a nonpre-emptive scheduling algorithm, called the largest scheduled dimension first (LSDF) algorithm for the parallel task scheduling problem is proposed. The major policy of LSDF, similar to the LDF algorithm [6], is that the larger the scheduled dimension

of a task is, the earlier it will be assigned.

Algorithm LSDF:

{ Call the maximum-parallel-dimension procedure;

Arrange parallel tasks in nonincreasing order of their scheduled dimension;

Assign parallel tasks to subcubes according to this nonincreasing order; }

Lemma 1: The number of free processors in a hypercube system will always be a multiple of the number of processors required by a task which is the next to be scheduled in the LSDF algorithm.

Lemma 2: If a task T_i , $1 \leq i \leq n$, is finished at time S_H , then there will be no processor idle before $(S_H - P(d_i, t_i))$, where S_H and t_i denote the finish time of a task set T scheduled by LSDF and the computation requirement of task T_i , respectively.

Theorem 1: $S_H \leq (2 - 1/2^d)S_O + (1 - 1/2^d)ns$, where S_H and S_O denote the finish times of the LSDF schedule and that of an optimal schedule of T , respectively.

Proof: By Lemmas 1 and 2, we have

$$\begin{aligned} (2^d \times S_H) &= \sum_{i=1}^n [2^{\Delta_i} \times P(\Delta_i, t_i)] + (2^d - 2^{\Delta_i})P(\Delta_i, t_i) \\ (2^d \times S_H) &= \sum_{i=1}^n \left[2^{\Delta_i} \left(\frac{t_i}{2^{\Delta_i}} + s \right) \right] + (2^d - 2^{\Delta_i}) \left(\frac{t_i}{2^{\Delta_i}} + s \right) \\ S_H &\leq \frac{\sum_{i=1}^n [2^{\Delta_i} (\frac{t_i}{2^{\Delta_i}} + s)] + (2^d - 1)(\frac{t_i}{2^{\Delta_i}} + s)}{2^d} \end{aligned}$$

It is obvious that $[\sum_{i=1}^n (t_i + s)]/2^d \leq S_O$ and $[(t_i/2^{\Delta_i}) + s] \leq S_O$. We have

$$S_H \leq \left(2 - \frac{1}{2^d} \right) S_O + \frac{\sum_{i=1}^n (2^{\Delta_i} - 1)s}{2^d}$$

That is, $S_H \leq (2 - 1/2^d)S_O + (1 - 1/2^d)ns$.

Fig. 1 LSPF schedule and optimal schedule

a LSPF schedule
b Optimal schedule

Example: Let $t_i = t$ and $\Delta_i = d$ for $i = 1, 2, \dots, 2^d - 1$, and $t_{2^d} = t$ and $\Delta_{2^d} = 1$. Fig. 1a and b illustrate that the finish times of an LSDF schedule and that of an optimal schedule are $(2t + 2^d s - t/2^d)$ and $(t + s)$, respectively. Thus, $S_H = (2 - 1/2^d)S_O + (2^d - 2 - 1/2^d)s \approx (2 - 1/2^d)S_O + (1 - 1/2^d)ns$, where $n = 2^d$.

Corollary 1: When the setup time is negligible, say $s = \epsilon/n$, the worst performance bound of LSDF will approach $(2 - 1/2^d)$ [6].

Acknowledgments: This research was supported by the National Science Council under grant NSC 83-0408E002-129.

References

- 1 DU, J., and LEUNG, J.Y.: 'Complexity of scheduling parallel task systems', *SIAM J. Discrete Math.*, 1989, **2**, pp. 473-487
- 2 MONMA, C.L., and POTTS, C.N.: 'On the complexity of scheduling with batch setup times', *Operations Researches*, 1989, **37**, pp. 798-804
- 3 MONMA, C.L., and POTTS, C.N.: 'Analysis of heuristics for preemptive parallel machine scheduling with batch setup times', *Operations Researches*, 1993, **41**, (5), pp. 981-992
- 4 WANG, Q., and CHENG, K.H.: 'A heuristic of scheduling parallel tasks and its analysis', *SIAM J. Comput.*, 1992, **21**, (2), pp. 281-294
- 5 WONGINGER, G.J., and YU, Z.: 'A heuristic for preemptive scheduling with setup times', *Computing*, 1992, **49**, pp. 151-158
- 6 ZHU, Y., and AHUJA, M.: 'On job scheduling on a hypercube', *IEEE Trans.*, 1993, **C-4**, (1), pp. 62-69

Efficient implementation of heuristic search

N. Cremelie and J.-P. Martens

Indexing terms: Search problems, Artificial intelligence

The heuristic search algorithm presented by Nilsson is commonly implemented with lists. However, the computational complexity of this implementation grows quadratically with the number of nodes being visited. The implementation presented has a computational complexity which is proportional to that number. Moreover, the storage requirements can be kept modest by introducing a late allocation strategy.

Introduction: Heuristic search is an attractive method for solving search problems. Instead of exhaustively investigating all possibilities, heuristic search relies on a best-first strategy: at any time the search proceeds in the direction that seems most likely to result in a solution. The decision of which path to examine first is based on heuristics.

Brief review of Nilsson algorithm: Search problems can be described as follows: given a set A of nodes, a set T of transitions between nodes, a set $S \subset A$ of source nodes and a set $G \subset A$ of goal nodes, find the path of successive transitions, starting in S and ending in G , exhibiting the smallest accumulation of transition costs. Nilsson [1] suggests ordering the nodes by means of an evaluation function $f(n)$, estimating the cost of the best path from S to G passing through node n . The Nilsson algorithm uses two lists of nodes: an OPEN list of visited nodes still waiting for further expansion, and a CLOSED list of expanded nodes ('expanding a node n ' means examining the transitions from n to its successor nodes). On initialisation, the source nodes are put in the OPEN list. Then the following sequence is executed repeatedly, until a stop criterion is reached:

- If OPEN is empty then stop the search (no solution!)
else select from OPEN the node n with the smallest $f(n)$, remove it from OPEN and put it in CLOSED.
- If n is a goal node then stop the search (solution found!)
- Expand n ; for all successor nodes n_i of n , compute $f(n_i)$, and:
if n_i is not in OPEN nor CLOSED then put n_i in OPEN, store the path to n_i
else if the new $f(n_i) <$ the old one then replace the old path to n_i by the new one and put n_i in OPEN.

Obviously, several list searches are required at each execution of the loop. Because the computational cost of searching in an unordered list is of the order p (the number of entries in the list), the above algorithm has a computational complexity of the order p^2 . By sorting the lists, this complexity becomes proportional to $p \log p$.

Implementation without lists: Suppose that the search space consists of a finite set of N nodes, each having a unique index between 1 and N (assume that this index is easy to compute from the representation of the node). It is then possible to set up an array of N entries, with entry n containing all characteristics of node n : $f(n)$, $g(n)$, a backpointer to the predecessor node, and a flag to indicate whether the node is open, closed or not yet visited. In this way, list searches are avoided. Information is now retrieved by an array indexing operation.

Fig. 1 K -ary minimum tree

The terminals $\square$ represent the search space nodes

To determine the open node having the smallest cost $f(n)$, a K -ary tree is introduced. The nodes of the search space constitute the terminals of the tree (Fig. 1) and each non-terminal a holds a pointer to the best open node (terminal) reachable from a . Consequently, the root of the tree points to the best open node in the entire search space. It is useful to define the non-terminals on the branches connecting a terminal n with the root as the 'ancestors' of n . The 'children' of a non-terminal a are the K non-terminals or terminals on the branches leaving a .

Initialising the tree is straightforward: all except the source nodes are new, the pointers in the source nodes' ancestors are set appropriately, and the remaining non-terminals are set to point to some dummy node with an excessively high cost. Whenever a node changes in some way, only small portions of the tree need an update. Two cases can be distinguished:

Case 1: Node n is selected for expansion and becomes closed: Because n was the open node with the smallest $f(n)$, all ancestors of n (a_1, a_2, a_3 in the case of Fig. 1) were referring to n and have to be updated as n becomes closed. The update process starts at the nearest ancestor of n (a_3 in Fig. 1): determine the minimum cost of the open nodes reachable from a_3 and let a_3 point to the node yielding this minimum. This requires $K + \alpha K$ tests, with $\alpha \leq 1$ (always test on open status, test on cost if open). Updating the remaining ancestors of n (a_2, a_1 in Fig. 1) requires $K - 1$ tests for each ancestor: only the minimum cost of the terminals pointed to by its children is to be determined. Consequently, the total number of tests for this case is $(K + \alpha K) + (K - 1)(\log_K N - 1) = T_1$.

Case 2: The value of f in a successor n_i of n is modified: In this case, node n_i becomes or remains open. The new $f(n_i)$ must be compared to the cost f in the nodes referred to by the ancestors of n_i , starting at the nearest ancestor of n_i and going up in the tree as long as the original cost was larger than $f(n_i)$. The number of tests for this update is $\beta \log_K N = T_2$ ($\beta \leq 1$).

Case 1 occurs once per node expansion, whereas case 2 can apply for all successors. If the expected number of applications of case 2 is called γ , then the total number of tests per expansion is given by $T_1(K) + \gamma T_2(K) = T(K)$. Note that α and β can only be estimated *a posteriori*, whereas γ is problem-dependent. However, assuming the worst case values for α, β and γ (an upper bound for γ follows from the problem description), an upperbound on $T(K)$ can be determined. Clearly this number does not depend on the number of expansions already made, and therefore the computational complexity of the proposed implementation is of the order p .

An adequate implementation for the K -ary tree is one using separate arrays for the different tree levels. Furthermore, K should be chosen so as to minimise $T(K)$. A study of $T(K)$ determined that N, α, β and γ have little impact on the location of the minimum. Moreover, $T(K)$ does not increase too quickly if K exceeds its optimum value (Fig. 2). Hence even an inaccurate estimate of α, β and γ can yield a near optimal K . Notice that a larger K also implies a lower storage requirement.