Allocating Tasks in Multi-core Processor based Parallel Systems

Yi Liu[1], Xin Zhang[2], He Li[2], Depei Qian[1,2]
[1]*(School of Computer, Beihang University, BeiJing 100083, China)*
[2]*(Department of Computer, Xi'an Jiaotong University, Xi'an 710049, China)*
yi.liu@jsi.buaa.edu.cn

Abstract

After a discussion of the task allocation problem in multi-core processor based parallel system, this paper gives the task allocation model, and proposes an iteration-based heuristic algorithm, which is composed of two rounds of operations, in which the processes are assigned to processing nodes in the first round and threads in process are assigned to processor cores in the second round respectively. Each round of operation partitions the Task Interaction Graph by iterations with backtracking. Evaluation result shows that the algorithm can find near-optimal solutions in reasonable time, and behaves better than genetic algorithm when the number of threads increases, since it can find solutions in much less time than genetic algorithm.

1. Introduction

Multi-core processors are becoming popular in recent years. And with the deployment of multi-core processors in massively parallel systems, the promotion of system performance will mainly relies on the increasing number of processor cores. To utilize these huge number of cores efficiently, there will be much more processes or threads in an application than ever. At the same time, the hierarchy of the system becomes more complex due to the using of multi-core processors.

This paper discusses the task allocation problem in multi-core processor based massively parallel systems, builds the task allocation model, and proposes a heuristic algorithm for task allocation. The algorithm is composed of two rounds of operations which assign processes to processing nodes and threads to processor cores respectively. Each round of operation is composed of multiple iterations with backtracking, and finally, a partition of the task interaction graph is obtained. The algorithm is evaluated with exhaustive searching and genetic algorithms. The results show that the algorithm can find near-optimal solutions in reasonable time, and behaves better than genetic algorithm when the number of threads increases, since it can find solutions in much less time than genetic algorithm.

The rest of this paper is organized as follows. Section 2 discusses the task allocation problem in multi-core based parallel system, and gives the task allocation model; section 3 presents the heuristic algorithm; section 4 evaluates the algorithm with genetic and exhaustive searching algorithms; section 5 summarizes the related research works, and section 6 concludes the paper.

2. Task Allocation Model

2.1 Task allocation in multi-core based parallel system

To simplify the discussion, task allocation is considered only for message-passing parallel systems, in which one processing node is equipped with one multi-core processor, and processing nodes are connected by interconnection network.

In this kind of systems, the multiple cores in processor are tightly coupled by sharing cache or by interconnecting with high-speed channel. On the other hand, processing nodes are loosely coupled, since the communication overhead between them is much higher than the overhead between processor cores.

For parallel applications based on message-passing programming model such as MPI and PVM, the program is composed of multiple tasks (processes) that communicating each other by sending / receiving messages. In multi-core processor based parallel systems, to utilize computation resources more efficiently, the process may be multithreaded, and these threads run on different cores. Threads belonging to the same process communicate each other by sharing memory, while those threads belonging to different processes communicate each other by passing

This work is supported by National Science Foundation of China under grant No. 60673180

messages due to their independent memory spaces.

Based on the above discussions, the rules of task allocation in multi-core based parallel systems include: firstly, to reduce communication overhead, assign threads that communicating frequently to the same processing node; secondly, all the threads belonging to the same process must be assigned to the same processing node, the reason is that threads in the same process share the same memory space, and each processing node have independent memory space in message-passing systems; thirdly, to reduce the context-switching overheads of processes, it's better not to assign multiple processes to the same core; fourthly, keep load balancing among processing nodes and cores.

2.2 The task allocation model

Suppose the multi-core parallel system is composed of N_{node} processing nodes $D_0, D_1, \ldots D_{Nnode-1}$, and each processor is composed of N_{core} processor cores $C_0, C_1, \ldots C_{Ncore-1}$; the parallel application to be allocated is composed of N_{proc} processes $P_0, P_1, \ldots P_{Nproc-1}$, and process P_i is composed of Mi threads $T_0, T_1, \ldots, T_{Mi-1}$.

The total number of threads: $N_{thread} = \sum_{k=0}^{Nproc-1} M_k$.

The application to be allocated can be represented as a task interaction graph $G = (V, E)$, where V is the set of vertices $\{V_i\}$, and V_i corresponds to an ordered pair $<\{T_i\}, \{P_i\}>$, where T_i is the corresponded thread and Pi is the process of the thread; E is the set of undirected edges $\{E_{ij}\}$.

Each edge E_{ij} connecting vertices V_i and V_j represents the communication or data sharing between thread T_i and T_j. The weight of the edge E_{ij} is represented by W_{ij}. A bigger W_{ij} means that the communication between the two threads is relatively frequent. $W_{ij}=0$ means that there is no communication between T_i and T_j. Considering the threads belonging to the same process share single memory space, the weight of these edges are set to maximum.

Figure.1 shows a task interaction graph containing 4 processes (12 threads), in which threads belonging to the same process are enclosed with dashed line.

The task allocation can be represented as a function f, which maps each thread to a processor core, formally:

$$f : \{T_i\} \to \{C_j\}, i = 0,1,\ldots N_{thread}-1, j = 0,1,\ldots,N_{node} \times N_{core}-1$$

The goal of the task allocation algorithm is to partition the task interaction graph into $N_{node} \times N_{core}$ sub-graphs. Vertices contained in sub-graph G_{ij} correspond to the threads assigned to processor core C_j in node P_i.

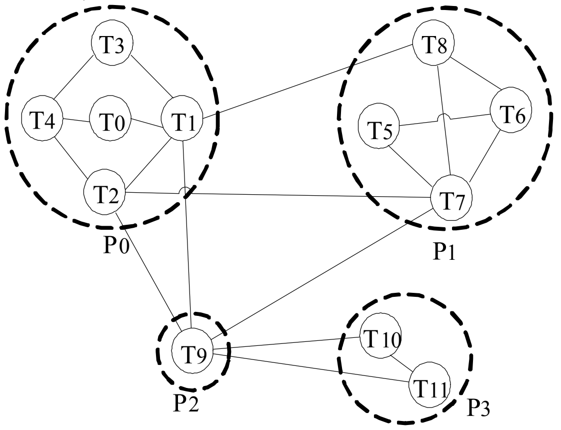


Figure. 1 Example of task interaction graph

According to the task allocation rules discussed in section 2.1, conditions for graph partitioning are listed as follows in priority order:

(1) Threads belonging to the same process are assigned to the same processing node; (condition-1)

(2) The number of vertices in each sub-graph are approximately equal; (condition-2)

(3) The sum of weights of edges between threads assigned to different processing nodes is the minimum among all of the partitions of the graph; (condition-3)

(4) It is better not to assign multiple processes to the same processor core. (condition-4)

The partition of the graph which satisfies the above four conditions is the optimal solution. It has been proved in [1] that the problem of finding an optimal schedule for a set of tasks is NP-complete in general cases. Instead, we propose a heuristic algorithm to find near-optimal solutions.

To make the description more clear, we introduce the concept of compound-vertex. A compound-vertex can be a single vertex, or a set of vertices in graph, and it is formally defined as:

(1) A vertex V_i is a compound-vertex which can be formally represented as $V_i = < TC_i, PC_i >$, where $TC_i=\{T_i\}, PC_i=\{P_i\}$;

(2) Two compound-vertices can be merged to form a new compound-vertex. Formally the merging operation $V_i + V_j \to V_k$ is represented as:

$$< TC_i, PC_i > + < TC_j, PC_j > \to < TC_k, PC_k >$$

where $TC_k = TC_i \cup TC_j$, $PC_k = PC_i \cup PC_j$. The weight of edge from V_k to a vertex V_a equals to the sum of weight of edges from V_i and V_j to V_a, that is, $W_{ka} = W_{ia} + W_{ja}$, $a=0,1,\ldots,N_{thread}-1$.

3. Heuristic Algorithm

3.1 The Idea of the algorithm

The idea of the heuristic algorithm for task allocation is: the task allocation is divided into two rounds of operations; the first-round of operation allocates processes to processing nodes; and the second-round allocates threads to cores in the processing nodes.

Each round of operations is started with the input task interaction graph, and composed of multiple iterations. During each iteration, the edge with maximum weight in the current task interaction graph is selected, and the two end-points of the edge are merged, in other words, two compound-vertices are merged to form a new compound-vertex. To satisfy the load balancing requirement in condition-2 given in section 2.2, the number of threads in the new compound-vertex must not exceed a threshold value. Repeat the selecting-merging procedure until the number of compound-vertices in the task interaction graph equals to the number of processing nodes or cores. The resulted compound-vertices correspond to the partition of the graph. Vertices included in compound-vertex V_i are to be assigned to processing node P_i or core C_i.

For the first round of operation, the original task interaction graph is pre-partitioned by processes, that is, the number of vertices in the graph equals to the number of processes, and each compound-vertex corresponds to a process. The first-round of operation terminates when the number of compound-vertices <= number of processing nodes N_{node}. By the end of first-round, each compound-vertex is a sub-graph, which corresponds to a processing node, and threads included in the sub-graph are to be assigned to the corresponded processing node.

The second round of operation is done for each of sub-graphs partitioned by the first-round, that is, the input task interaction graph for second-round of operation is the sub-graph obtained in first-round. The second-round terminates when the number of compound-vertices <= number of cores N_{core}. By the end of second-round, each compound-vertex corresponds to a processor core, and threads included in the sub-graph are to be assigned to the corresponded processor core.

In the operations of the algorithm, the condition-1 introduced in section 2.2 is satisfied by dividing the task allocation into two rounds, and allocating processes to processing nodes in the first round; condition-2 is satisfied by limit the number of vertices in sub-graph by a threshold during each iteration; condition-3 is satisfied by partitioning edges with bigger weight into sub-graphs in each iteration; as for condition-4, vertices belonging to the same process are primarily allocated to the same processor core, since their weights are set to maximum value at the beginning.

The threshold mentioned above is used to control load balancing. It can be defined as the average number of threads for one processing node or core, and fluctuating within a bounded value. The threshold values for the first and second round of operations can be calculated as follows respectively:

$$Threshold_{first\_round} = Max(\left\lceil \frac{N_{thread}}{N_{proc}} \right\rceil, M_{\max}) \times (1+\alpha) \quad (0 \le \alpha \le 1) \qquad (3\text{-}1)$$

$$Threshold_{second\_round} = \left\lceil \frac{Threshold_{first\_round}}{N_{core}} \right\rceil \qquad (3\text{-}2)$$

The M_{max} in expression (3-1) is the maximum number of threads in a process, and α is a ratio value used to achieve tradeoff between load balancing and communication minimization. The bigger the α is, the bigger the threshold is. The result is that the algorithm tends to reduce communication overhead, since edges with bigger weight will be selected for merging. And if the α is smaller, edges with bigger weight may not be selected due to threshold limitation, at this time the algorithm tends to load balancing.

3.2 Backtracking mechanism

Normally, in each round of operations, solution can be found after times of iterations. If the load-balancing limitation is relatively strict, the solution may not be found. At this time, the algorithm should increase the threshold and repeat the operation.

In each iteration of the algorithm, the edge with the maximum weight is primarily selected from the edges satisfying the condition of load balancing. The selection is a kind of "local optimal", and in some situations, is not "global optimal", even leads to no-solution-found for the algorithm.

To make the algorithm searching solutions in wider range, a backtracking mechanism is added to the operation. That is, memorize all the possible selections in each iteration. In the situations of no-solution-found, the algorithm goes back and recalls the memory, makes another selection and tries again. To avoid too many iterations caused by backtracking, it is necessary to limit the number of backtracking steps in a small value.

The backtracking can be easily implemented by a stack. During each iteration, all the edges in the queue and associated status are pushed into the stack, and popped out when backtracking is needed. The step number of backtracking can be limited by the size of the stack.

3.3 The algorithm

Based on the above discussions, the heuristic algorithm is presented as follows:

```
/* adjust the number of processing node if it is bigger than
   the needed number by the program (1 thread per core) */
if   ($N_{node}$ * $N_{core}$ > $N_{thread}$ )
     $N_{node}$ = $N_{thread}$ / $N_{core}$;
/* First-round operation*/
Operation( Task Interaction Graph, $N_{node}$ ,$Threshold_{first\_round}$);
/*Second-round operation*/
for   ( each sub-graph $G_i$ outputed by the first-round operation ) {
     Operation( $G_i$, , $N_{core}$ , $Threshold_{second\_round}$);
}

procedure Operation( Task Interaction Graph, Number of
                       processing elements, Threshold )
{
     Sort edges in the graph by the weight in decreasing order and put them into a queue;
     while   ( number of compound-vertices Nc > number of
                                    processing elements){
          i←0;
          while (vertices for merging not found) {
               fetch the i-th edge $E_{ab}$ from the queue;
               if   ( thread number in vertex $V_a$ +thread number
                              in vertex $V_b$ ≤Threshold )
                    found vertices for merging;
               else
                    i←i+1;
          }/*while*/
     if (found vertices for merging ) {
          push all the edges in the queue and current graph into
                                              the stack;
          merge vertex $V_a$ and $V_b$ to form new vertex $V_c$, delete
          vertex $V_a$ and $V_b$ from the graph and add vertex $V_c$,
          operations include:
               $V_a + V_b \rightarrow V_c$    ( definition of merge-operation is
                                              shown in 2.2 )
               for (j=0,1,...,Nc-1)
                    delete edges $E_{ai}$ and $E_{bi}$ from the queue, and
                    insert $E_{ci}$ into the queue according to its weight;
               Nc←Nc–1;
     }/*if*/
     else {
          if (stack is not null)
               pop the queue and graph from stack;
          else
               return "no-solution-found";
          }
}
```

3.4 Considerations for SMP-based processing node

If the processing node is symmetric multi-processing (SMP), the hierarchy of the system becomes more complex. The reason is that processors in processing node share main memory, and they coupled looser than cores inside of processor, but tighter than processing nodes.

Based on the above considerations, the task allocation in SMP & multi-core based systems can be done by a revision of the above heuristic algorithm. At this time, the algorithm is composed of three rounds of operations in stead of two rounds, that is, processes are allocated to processing nodes in the first round, threads are allocated to processors in the second round, and threads are allocated to processor cores in the third round.

4. Experiments and Evaluation

The heuristic algorithm is evaluated using 1,400 randomly generated task interaction graphs, in which the number of threads varies from 16 to 1024, and threads belong to 16—64 processes. The weights of edges are random values ranging from 0 to 100.

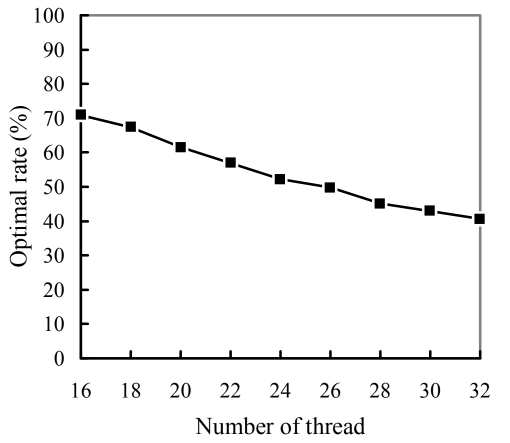
The heuristic, exhaustive searching and genetic algorithms are implemented and their solutions are compared by allocating tasks to different platforms with number of processing nodes = 8, 16, 24, 32 and number of cores inside of a processor = 2, 4 and 8, using randomly generated task interaction graphs. In the evaluation, α in expression (3-1) is set to 0.1, and maximum number of backtracking is limited to 2.

For graphs with $N_{thread} \leq 32$, optimal solutions are exhaustively searched and compared with heuristic solutions; for graphs with $N_{thread} > 32$, optimal solutions are difficult to search due to its high computation cost. As a substitution, we use genetic algorithm to evaluate our heuristic.

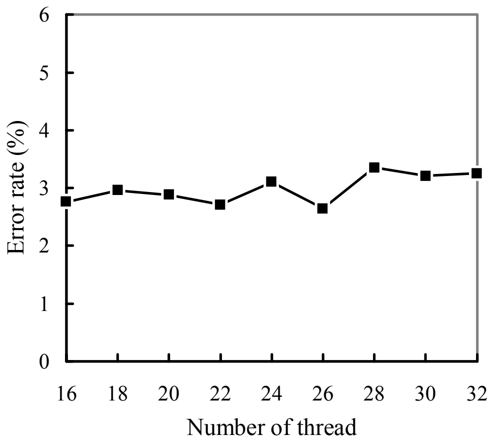
Figure.2 shows the optimal rate and error rate of our heuristic, in which the error rate is calculated as follows according to the sum of weights between threads assigned to different processing nodes (noted as LSW: Left Sum of Weights):

$$ErrorRate = \frac{\sum Heuristic\_LSW - \sum Optimal\_LSW}{\sum Optimal\_LSW} \times 100\% \tag{4-1}$$

As Figure.2 shows, although the percentage of optimal solutions decreases with increasing of thread number, error rates are always lower than 5%. This means that the heuristic algorithm can always find near-optimal solutions.



(a) Optimal rate



(b) Error rate

Figure.2 Optimal Rate and Error Rate of the Heuristic Algorithm

The comparison results between heuristic and genetic algorithms are shown in Figure.3 and Table 1. Figure.3 gives the error rate between the two algorithms, in which the error rate is calculated by replacing the *Optimal-LSW* in expression (4-1) with *Genetic-LSW*. Therefore, a positive value of error rate means that genetic solutions are better than heuristic solutions, and vice versa.

Tabe.1 gives the computation cost of heuristic and genetic algorithms. The data shows that the heuristic can find solutions in much less time than genetic algorithm with increasing of thread number.

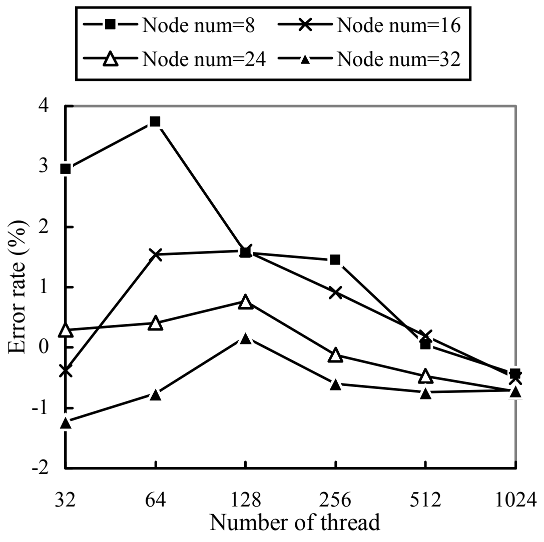


Figure. 3. Error Rate between Heuristic and Genetic Algorithms

5. Related Works

The problem of task scheduling has been studied for many years. Most of the proposed algorithms are based on a task precedence graph (TPG) or a task interaction graph (TIG). TPG is a directed graph where the nodes and directed edges represent the tasks and task precedence constraints respectively. On the other hand, TIG is an undirected graph where two tasks communicate if there is an edge between the nodes. It has been proved in [1] that the problem of finding an optimal schedule for a set of tasks is NP-complete in the general case.

So most researches are focused on finding near-optimal solutions. One kind of methods is to propose dedicated heuristic algorithm, and another is based on AI techniques such as genetic algorithm, simulated annealing, etc. Many algorithms have been proposed in past years, and they are based on different system models, e.g. message-passing or shared memory, homogeneous or hetegeneous systems; aiming for different targets and policies, e.g. load balancing, minimum execution-time, using minimum resources; and suited for different levels of parallelism, e.g. process/thread level, iteration-level or instruction-level.

Table 1. Average computation cost of heuristic and genetic algorithm

Number of thread	32	64	128	256	512	1024
Heuristic algorithm (ms)	38	48	581	605	760	1119
Genetic algorithm (ms)	2362	19365	34427	43561	79847	3107907

Tested on PC-compatible with Intel P4 3.2GHz processor running Windows XP

In multi-core parallel systems, there will be much more processes or threads than ever, and it requires the task allocating and scheduling algorithms to be efficient and scalable. Research works focused on task allocation for multi-core based systems include: [2] allocates and schedules tasks using a hybrid of genetic algorithm (GA) and Ant Colony Optimization (ACO) on multi-core system with both global and local memory; and there are some research works on task allocation for multi-core DSPs or network processors[3].

6. Conclusion

This paper discusses the task allocation problem in multi-core processor based massively parallel systems, builds the task allocation model, and proposes a heuristic algorithm for task allocation. The algorithm is composed of two rounds of operations which assign processes to processing nodes and threads to cores respectively. Each round of operations is composed of multiple iterations with backtracking, and finally a partition of the task interaction graph is obtained. The algorithm is evaluated with exhaustive searching and genetic algorithms. The results show that the algorithm can find near-optimal solutions in reasonable time, and behaves better than genetic algorithm when the number of threads increases, since it can find solutions in much less time than genetic algorithm.

References

[1] Ullman J. D, NP-Complete scheduling problems, Journal of Computer and System Sciences, 1975, 10(3): 384--393

[2] Min Li, Hui Wang and Ping Li, Tasks mapping in multi-core based system: hybrid ACO&GA approach, In Proceedings of 5th International Conference on ASIC, Oct 2003.

[3] Ennals Robert, Sharp, Richard and Mycroft Alan, Task partitioning for multi-core network processors, In Proceedings of 14th International Conference on Compiler Construction, April 2005.

[4] Di Nan, Wang Tao and Li Xiaoming, Static task distribute algorithm based on task relation in LilyTask, Journal of Computers, 2005, 28(5): 892--899

[5] Zhang Hongli, Fang Binxing and Hu Mingzeng, Algorithm on task scheduling in structural parallel control mechanism, Journal of Software, 2001, 12(5): 706--710

[6] Bansal Savina, Kumar Padam and Singh Kuldip, An improved two-step algorithm for task and data parallel scheduling in distributed memory machines, Parallel Computing, 2006, 32(10): 759--774

[7] Kwok Yu-Kwong and Ahmad Ishfaq, On multiprocessor task scheduling using efficient state space search approaches, Journal of Parallel and Distributed Computing, 2005, 65(12): 1515--1532

[8] Berenbrink Petra, Czumaj Artur, Friedetzky Tom, et.al. Infinite parallel job allocation, In Proceedings of 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2000), July 2000.

[9] Achalla Chandrasekhar, De Doncker Elise, Kaugars Karlis, et.al. α-Load balancing for parallel adaptive task partitioning, Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems, Nov 2004.

[10] Liu Zhen and Righter Rhonda, Optimal parallel processing of random task graphs, Journal of Scheduling, No.3, vol 4, 2001.

[11] A. Enrique, L. Guillermo and O. Guillermo, Analyzing the behavior of parallel ant colony systems for large instances of the task scheduling problem, Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005), Apr 2005.