A Task Duplication Based Scheduling Algorithm for Heterogeneous Systems

Samantha Ranaweera and Dharma P. Agrawal
University of Cincinnati, Department of Electrical, Computer Engineering and Computer Science, Cincinnati, Ohio, 45221.
({aranawee|dpa}@ececs.uc.edu)

Abstract

Optimal scheduling of tasks of a directed acyclic graph (DAG) onto a set of processors is a strong NP-hard problem. In this paper we present a scheduling scheme called TDS to schedule tasks of a DAG onto a heterogeneous system. This models a network of workstations, with processors of varying computing power. The primary objective of this scheme is to minimize schedule length and scheduling time itself. The existing task duplication based scheduling scheme is primarily done for totally homogeneous systems. We compare the performance of this algorithm with an existing scheduling scheme for heterogeneous processors called BIL. In initial simulations TDS has been observed to generate scheduling lengths shorter than that of BIL, for communication-to-computation cost ratios (CCR) of 0.2 to 1. Moreover TDS is far more superior than BIL as far as scheduling time is concerned.

1. Introduction

With the advent of high speed communication technologies and the concept of NOW, network of workstations, distributed processing has been investigated extensively for parallel computing purposes. The computing power of a local network, can even exceed that of a super computer. Not only will this be a cost effective measure, where you only have to invest money on a few moderately powerful computers, applications can be partitioned in such a manner so as to extract the maximum benefits of using a particular architecture. Some applications which can use such computing environments are fluid flow, weather modeling, database systems and image processing. To achieve full benefits, a parallel application is first transformed into a task DAG by a task partitioning algorithm. Then these tasks are mapped onto processors using a scheduling algorithm.

The problem of mapping DAG's onto processors is proven to be NP-hard [4], in it's pure form. There are large number of static heuristic scheduling schemes *for homogeneous processors*. These can be classified into three broad categories, namely, priority based scheduling, cluster based scheduling, task duplication based scheduling.

Priority based scheduling assigns priorities to tasks and use priorities to map tasks onto processors. These methods are fairly simple to implement, but, their solutions are sub optimal. Some examples of such schemes can be found in [5],[6]. The algorithms we use to compare the performance of our proposed algorithm, BIL [7], too, falls into this category but it is far superior.

Then, there are number of algorithms based on clustering. The essence of these methods is to cluster tasks that communicate among themselves onto the same processor. If the available number of processors is less than the number of clusters, their solutions may not be very efficient. Some examples can be found in [8-13].

The main idea behind duplication based scheduling is to utilize processor idling time to duplicate predecessor tasks. This may avoid transfer of results from a predecessor, through a communication channel which bring IPC in to the picture. This may eliminate waiting slots on other processors. Some of these schemes are found in [1-3], [14-16]. There are genetic based variants of this algorithm as well [18].

In *heterogeneous processor* environments, one of the earliest algorithms resulted from the works of G.C. Sih and E.A. Lee [6]. The techniques discussed in this paper gave rise to a scheme called 'General Dynamic Level Scheduling (GDL)'. However a more recent work by Hyunok Oh and Soohoi Ha [7], generated another scheme called 'Best Imaginary level scheduling (BIL)', which the authors claim is about 20% faster than the GDL scheme. BIL is proven to generate optimal results if the topology of the DAG is *linear*. We have used the BIL scheme for comparison purposes, and have seen

This work has been supported by the Ohio Board of Regents Enhancement Funds and the National Science Foundation under grant No. CCR-9902748.

that our method is superior than this scheme in several aspects.

Another related piece of work can be found in [17]. The basis of their research is the best-first search technique known as the A* algorithm, from the area of artificial intelligence. The athors claim that "*The algorithm, in its traditional form, guarantees optimal solutions, but does not work for large problems, due to its high time and space complexity*". Since we are interested in a more general algorithm to suit any type of a DAG, we have not used this method for comparison purposes.

Another interesting piece of work has been carried out by C.C. Hui and S.T. Chanson [19]. However their representation of the parallel application is a task interaction graph, which does not take into account the *precedence* among tasks. Therefore their work is not applicable in our context.

This paper introduces a task duplication based scheduling scheme (TDS), which initially generates a set of clusters similar to linear clusters. Then duplication is carried out until system resources are exhausted.

The rest of the paper is organized as follows. In Section 2, we present the new algorithm. We shall demonstrate the way the algorithm works using an example in Section 3. We present our results in Section 4. Section 5 will provide conclusions and plans for future work.

2. TDS Algorithm.

What we planed to do is to minimizing the schedule length, *also called makespan*, and to minimize the complexity, which ensures reasonable runtime. Roots of this algorithm is found in [1], [2] and [3].

The input to the algorithm consist of the a directed acyclic graph, DAG, which is defined by the tuple (V,E,P,τ,c). V is a set of task nodes, E is a set of communication edges, P is a set of processors, $\tau=\tau(j,p)$ where $j\in V$ and $p\in P$ and $\tau(j,p)$ indicates the running time of j^{th} node on P^{th} processor, $c=c(j,k)$ where $j,k\in V$ and $c(j,k)$indicates the communication cost between tasks j and k. The actual figure may depend on the processor to which tasks j and k are ultimately assigned, but that can be taken care of by slightly modifying the equations in table 1, but to receive it as an input means a very large input. Therefore we assume that communication cost to be equal between all processors.

A task is assumed to be an indivisible unit of work which is non preemptive. The underlying architecture is assumed to be heterogeneous. We also assume that there are dedicated communication channels available, i.e. any number of message passing can take place at any given time.

The crux of the algorithm lies in a few computed quantities. Namely these are, Earliest Start Time of a node, *est*, Earliest Completion Time of a node, *ect*, Latest Allowable Start Time of a node, *last*, Latest Allowable Completion Time, *lact*, *fpred* which stands for the favorite predecessor task of a given task, *fproc* which stands for the favorite processor. *'fproc'*, is the favorite processor of a given task, to which when the said task is assigned, ensures that the task has the smallest completion time. It must be emphasized here that *fproc* of a given task may not be the processor on which the said task will have the shortest running time.

The algorithm runs in four steps and they are as follows. In the first step, the DAG is traversed in a top-down fashion to compute the *est, ect, fpred, fproc1 to fproc*n *and level* for each task. The *level* is defined to be the highest value of the summation of computational costs along different paths to the exit node from the node under consideration. The elements in the array *queue* are the nodes sorted in the ascending order of *level*. In the second step *lact* and *last* for each node are computed in a bottom-up fashion.

The third step will generate an initial set of task clusters, using a reasonably small number of processors. In the fourth step task duplication and message forwarding is carried out. Since the first two steps involve traversing all tasks and edges, in the homogeneous case, the worst case complexity will be O(E). But in the heterogeneous case, since at each task, to compute the favorite processors, P steps are taken, the worst case complexity will be O(PE).

The algorithm for the third step is given in figure 2. Since this is similar to a depth first search the order will be O(V+E). For a connected graph it will be O(E) in the worst case.

In the fourth step, duplication is carried out. We assume that there are enough number of processors to allocate the initial set of clusters. We check clusters to see whether the preceding node of a given node is it's *fpred*. If this is not the case, we replace the tail of that cluster starting with the *fpred* of the given cluster. This is done recursively until the entry node is reached. The initial tail is assigned to a new processor, which is the next available *fproc*, of the last node of that tail. Sometimes duplication increases *makespan*, if this is the case, we nullify that change and start from the next cluster and proceed with duplication.

Table 1. Mathematical equations used.

PRED(j) : Set of predecessors of any task j.
SUCC(j) : Set of successors of any task j.
P : Set of processors available.
est(j/p) : Earliest start time of task j given that it will execute in processor p.
τ(j,p) : Execution time of the j^{th} task on p^{th} processor.
c(j,k) : Communication cast between tasks j and k

est(j) = 0 , for the entry node.

fproc(j) = $p \in P$ s.t. (*est*(j/p) + τ(j,p) < *est*(j/q) + τ(j,q) where $q \in P$).
(In other words, assigning task *j* to processor *p* will yield a minimum completion time for task *j*).

est(j/p) = $\max_{k \in PRED(J)}$ ($ect(k)_{fproc(k)=p}$ || $ect(k)+c(k,j)_{fproc(k)\neq p}$).

(In other words, if task *j* is assigned to a processor which was the favorite processor of a predecessor *k* of *j* take *ect*(*k*); else take communication cost into account)

est(j) = *est*(j/*fproc*(j)).

ect(j) = *est*(j) + τ(j,*fproc*(j)).

fpred(j) = $k \in$ PRED(j) s.t. *ect*(k) + c(k,j) > *ect*(l) +c(l,j) where $l \in$ PRED(j) && $k \neq l$.

lact(j) = *ect*(j), for the exit node.

lact(j) = min($\min_{k \in SUCC(j),\ j \neq fpred(k)}$ (*last*(k) – c(j,k), $\min_{k \in SUCC(j),\ j = fpred(k)}$ (*last*(k))).

last(j) = *lact*(j) - τ(j/*fproc*(j)).

An additional check is carried out to see if the completion time of a task is greater than the time it takes (completion plus communication time) to get the results of the same task executed in another processor. If so the task is removed from the current processor and the results from the other processor is *forwarded.* This is a result of each task having different run times on different processors. This is what we mean by message forwarding. Following is an example.

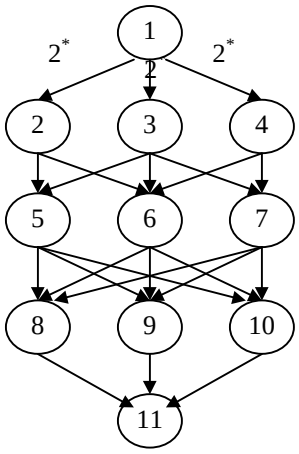

Figure 1. Sample DAG.
(* signifies that all communication costs are 2units)

```
Input     : DAG(V,E,P,τ,c)
Output    : Set of initial clusters
Uses      : PRED(i) and SUCC(i)
Queue     : Array[1..|v|]; /* Has tasks in ascending order of level */
Begin{
  x=First element of queue;
  current processor = fproc1 of x;
  assign x to current processor;
  While(NOT all tasks assigned){
    y=fpred(x);
    if x has more than one predecessors{
      if ((last(x)-lact(y))>c(x,y)) /* y is not critical */{
          y = predecessor of x which has the lowest
          running time on this processor ;}
      else{
          if y is already assigned{
              y = predecessor of x which has the lowest
              running time on this processor;  }}}
    assign y to current processor;
    x = y;
    if x is the entry node{
      assign x to current processor;
      x = next unassigned element in queue;
      current processor = next available fprocm of x; }
    }
```

Figure 2. Algorithm for step 3 of TDS.

3. Running trace of the algorithm.

Consider the DAG in Figure 1. The runtimes of tasks in different processors is tabulated in table 1 below. We have taken communication costs for all edges to be equal but it is not a necessary condition.

Table 2. Run times of tasks.

Processor ⇒ Task ⇓	1	2	3	4	5
1	5	6	7	5	5
2	3	2	2	2	3
3	4	3	3	2	3
4	9	8	8	8	9
5	3	3	3	3	3
6	7	6	6	7	6
7	4	3	3	4	3
8	8	7	7	6	8
9	3	2	3	2	3
10	3	4	3	4	4
11	5	5	5	5	5

3.1. Step 1 & 2.

Using the mathematical equations defined in table 1, the computed values are tabulated in table 3.

Table 3. Start and completion times.

Node	*Level*	*est*	*fproc 1*	*fproc 2*	*fproc 3*	*fproc 4*	*ect*	*fpred*	*last*	*lact*
1	36	0	4	5	1	2	5	-	0	5
2	23	5	4	5	1	2	7	1	9	11
3	24	5	4	2	3	5	7	1	9	11
4	29	5	4	1	5	3	13	1	5	13
5	16	9	4	5	1	2	12	2	15	18
6	20	13	4	2	3	5	20	4	13	20
7	17	13	4	2	3	5	17	4	14	18
8	13	20	4	2	3	5	26	6	20	26
9	8	20	4	2	3	5	22	6	22	24
10	9	20	4	2	3	5	24	6	20	24
11	5	26	4	2	3	5	31	8	26	31

The *queue* will be as follows.

Queue = {11,9,10,8,5,7,6,2,3,4,1}

3.2. Step 3.

The first cluster will begin from task 11. It's *fpred* is task 8. Task 8's *fpred* is task 6. This selection process will follow until the cluster 11-8-6-4-1 is generated. This cluster is assigned to processor 4 which is *fproc*1 of task 11. The next set of linear clusters start at task 9. But since *fproc1* of task 9 is already selected, *fproc2* of task 9, which is processor 2 is selected to carry the second cluster. Task 9's *fpred* is task 6, but since task 6 has been allocated to processor 4 already, we can choose either task 5 or 7. On processor 2, both these have a runtime of 3 units. Therefore since it is a tie, we make a random choice between them. Hence task 7 is selected or comes the entry node this cluster. Task 7's *fpred* is task 4, which is already assigned. So we

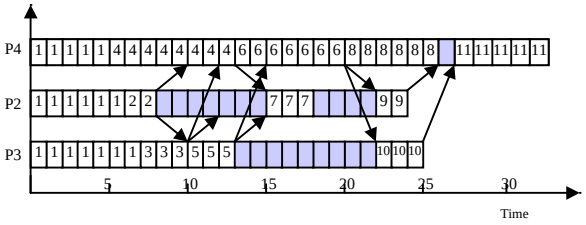

Figure 3 Initial clusters.

choose task 2. Next So the cluster will be 9-7-2-1. This process will continue and the third cluster will be 10-5-3-1. Now all tasks have been assigned, using just three processors. This completes the third step. The generated schedule will be as in Figure 3 and the *makespan* is 32.

3.3. Step 4.

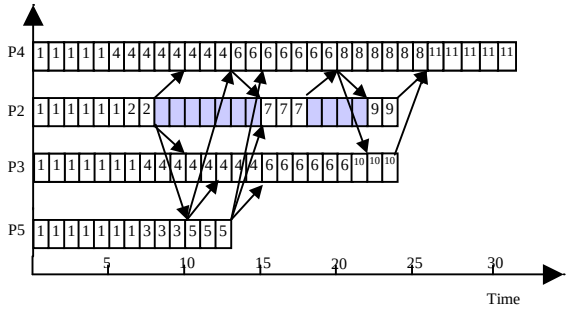

Figure 4. *fpred* of task 10 duplicated.

Duplication is not required for the first cluster. Since the third cluster is the one which forces an idle slot in the critical cluster. So we try duplicating the *fpred* tail of task 10. The schedule after the duplication of *fpred* of task 10 is depicted in Figure 4. Time for completion is now 31 units. In duplicating, task 10 had retained processor 3. The remaining chain starts from task 5 and since *fproc*1 of task 5, which is processor 4, is not available, that cluster is assigned to processor 5, which is *fproc 2* of task 5.

Figure 5 depicts the schedule after the *fpred* of task 9 has been duplicated. Now, however, the time taken to complete the whole set of task has gone back to 32. This is even after cutting down the actual completion time of task 7 *by not*

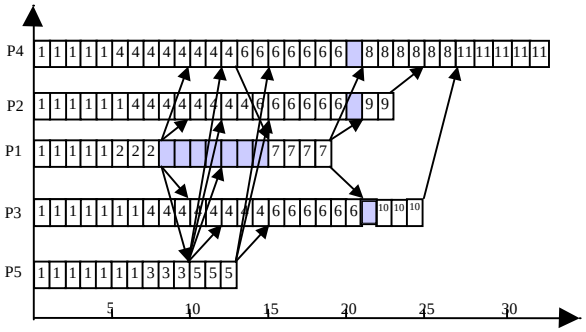

Figure 5- *fpred* of task 9 duplicated.

duplicating task 4 on processor 1 because it will have 9 time units as run time which will further delay the completion time of task 7. Instead use the results of task 4 from processor 1 to get task 7 going. Since this is more time consuming than the previous one, we stick to the previous one. Therefore the ultimate *makespan* will be 31 time units.

4. Results.

We first observe an absolute performance indicator, which is the ratio of *actual completion time* (ACT) to *earliest possible completion time* (ECT) for varying values of *diversity* and CCR. ECT is an absolute minimum. Therefore closeness of ACT to ECT is an indication of how good the schedule is. The quantity *diversity* is defined as follows.

Diversity =Average of
[(Max node cost –Min node cost)/2)/ Average node cost]

Similarly to compute CCR, we used the following formula.

CCR = (Average edge cost)/(Average Node cost).

We used the original communication times but generated random runtimes to introduce heterogeneity. Therefore *diversity* is an indicator of how diverse the system is.

We used two inputs in this simulation. One is the DAG from Cholesky decomposition algorithm, which has 2925 nodes and 5699 edges. The other is a Diamond DAG. It has 2502 nodes and 4902 edges. Each experiment was repeated 10 times with different random seeds.

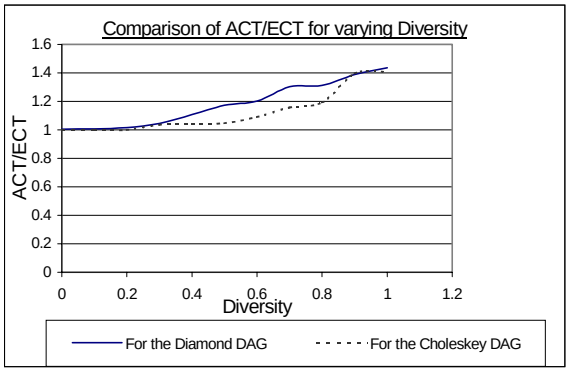

Figure 6. (ACT/ECT) Vs *diversity*.

The closeness of ACT/ECT to 1 for small values of *diversity* can be attributed to the fact that the system gets closer to a *homogeneous* system and because the mother algorithm is proven to be optimal for homogeneous system. The variation of ACT/ECT for varying CCR is given below. Once again the closeness of the algorithm to optimality can be seen for small CCR values.

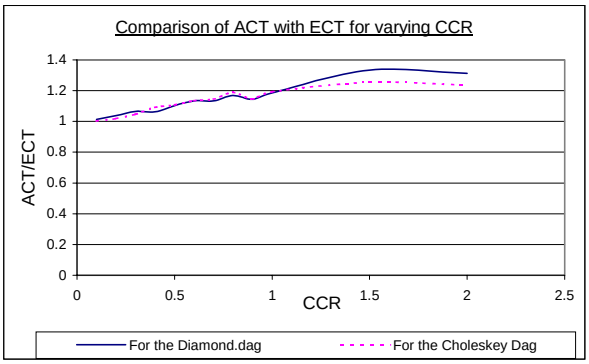

Figure 7. (ACT/ECT) Vs CCR.

Next we look at a relative performance indicator by comparing our method and the scheduling scheme BIL, for heterogeneous processors presented in [7]. We used the same input mentioned above. We looked at *makespan* and *scheduling time* (time taken to generate the schedule) for varying number of processors. The results of the experiments carried out are depicted in figure 6. It is quite evident that TDS is very much faster than BIL for large inputs. This was true even for smaller inputs with number of node from 10 to 100's even. Most of the time TDS generated shorter schedules than those generated by BIL when CCR was more than 0.2. However these improvements diminished for CCR greater than 1.0. Performance of the two algorithms were comparable when CCR was smaller than 0.2 and larger than 1. However the running time of BIL heavily depended on the number of processors as it's time complexity was $O(v^2 p \log p)$.

5. Conclusions and future work.

In this paper we presented a task duplication based scheduling scheme for heterogeneous systems called TDS. Performance of this algorithm has been compared with a similar scheme called BIL. Inputs used for comparison were Cholesky decomposition DAG, Diamond DAG and the DAG for Gaussian elimination code. It was observed that TDS much faster than BIL for large inputs. For CCR of 0.2 to 1.0, TDS outperformed BIL also when *makespan* is concerned. We plan to improve this algorithm by selectively carrying out the duplication process and by making this algorithm completely scalable.

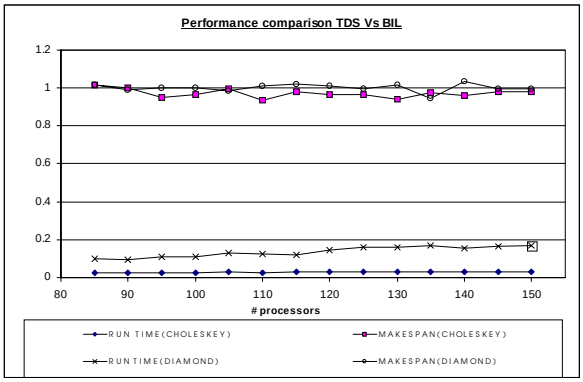

Figure 8. Performance comparison, Task Duplication based Scheduling Scheme (TDS) and Best Imaginary Level Scheduling Scheme (BIL).

References

1. S. Darbha and D. P. Agrawal, "*A task duplication based scalable scheduling algorithm for distributed memory systems*", Journal of parallel and Distributed Computing, Vol. 46, No. 1, October 1997, pp. 15-27.

2. S. Darbha and D.P. Agrawal, "*Optimal Scheduling algorithm for distributed memory machines*", IEEE transactions on parallel and distributed systems, Vol. 9, No. 1, January 1998, pp. 87-95.

3. S. Darbha, "*Task scheduling algorithm for distributed memory machines*", PhD thesis, North Carolina State University, 1995.

4. R. L. Graham, L. E. Lawler, J. K. Lenstra and A. H. Kan, "*Optimization and Approximation in deterministic sequencing and scheduling: a survey"*, In annuls of discrete Mathematics, 1979, pp. 287-326.

5. J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "*Scheduling precedence graphs in systems with inter processor communication times*", SIAM Journal of Computing, Vol. 18, No. 2, April 1989, pp. 244-257.

6. G.C. Sih and E. A. Lee, "*A compile time scheduling heuristic for interconnection-constrained heterogeneous processors architectures*", IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 2, February 1993, pp. 175-187.

7. Hyunok Oh and Soonhoi Ha, "*A static heuristic for heterogeneous processors*", Euro-Par'96, Lyon, France, August 1996.

8. H. El-Rewini and T. G. Lewis, "*Scheduling parallel programs onto arbitrary target architectures*", Journal of Parallel and Distributed Computing, Vol. 9, No. 2, June 1990, pp. 138-153.

9. A. Gerasoulis and T. Yang, "*A comparison of clustering heuristics for scheduling directed acyclic graphs onto multiprocessors"*, Journal of Parallel and Distributed Computing, Vol. 16, No. 4, December 1992, pp. 276-291.

10. S. J. Kim and J. C. Brown, "*A general approach to mapping of parallel computations upon multiprocessor architectures"*, proceedings of International Conference on Parallel Processing, August 1988, Vol. 3, pp. 1-8.

11. S. S. Pande, D. P. Agrawal and J. Mauney, "*A new threshold scheduling strategy for Sisal programs on distributed memory systems*", Journal of Parallel and Distributed Computing, Vol. 21, No. 2, May 1994, pp. 223-236.

12. S. S. Pande, D. P. Agrawal and J. Mauney, "*A scalable scheduling method for functional parallelism on distributed memory multiprocessors*", IEEE transactions on parallel and distributed systems, Vol. 6, No. 4, April 1995, pp. 388-399.

13. V. Sarkar, "*Partitioning and scheduling programs for execution on multiprocessors"*, MIT press, Cambridge, MA, 1989.

14. I. Ahmed and Y. K. Kwok, "*Anew approach to scheduling parallel programs using task duplication"*, International Conference on Parallel Processing, August 1994, Vol. 2, pp 47-51.

15. H. B. Chen, B. Shirazi, K. Kavi, and A. R. Hurson, "*Static scheduling using linear clustering and task duplication"*, In proceedings of the ISCA International Conference on Parallel and Distributed Computing and systems, 1993, pp. 285-290.

16. J. Y. Colin and P. Chritienne, "*C.P.M. Scheduling with small Communicational delays and task duplication"*, Operational research, Vol. 39, No. 4, July 1991, pp. 680-684.

17. Muhammad Kafil and Ishfaq Ahmed, "*Optimal Task Assignment in Heterogeneous Distributed Computing Systems"*, IEEE Concurrency, Vol. 6, No. 3, July-September 1998, pp. 42-51.

18. T. Tsuchiya, T. Osada, T. Kikuno, "*A new heuristic algorithm based on GA's for multiprocessor scheduling with task duplication"*, Third International Conference on Algorithms and Architectures for Parallel Processing, 1997, pp. 295-308.

19. C. C. Hui and S. T. Chanson, "*Allocating task interaction graphs to processors in heterogeneous networks"*, IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 9, September 1997, pp. 908-926.

20. M. Cosnard and E. Jeannot, "*Compact DAG representation and it's dynamic Scheduling"*, Journal of Parallel and Distributed Computing, Vol. 58, No. 3, September 1999, pp. 487-514.