A CLUSTERING HEURISTIC FOR MULTIPROCESSOR ENVIRONMENTS USING COMPUTATION AND COMMUNICATION LOADS OF MODULES

Pramod Kumar Mishra[1], Kamal Sheel Mishra[2], and Abhishek Mishra[3]

[1]Department of Computer Science, Banaras Hindu University, Varanasi, India
mishra@bhu.ac.in
[2]Department of Computer Science, School of Management Sciences, Varanasi, India
ks_mishra@yahoo.com
[3]Department of Computer Engineering, Institute of Technology, Banaras Hindu University, Varanasi, India
abhishek.rs.cse@itbhu.ac.in

ABSTRACT

In this paper, we have developed a heuristic for the task allocation problem on a fully connected homogeneous multiprocessor environment. Our heuristic is based on a value associated with the modules called the Computation-Communication-Load (CCLoad). This value is dependent on the computation and the communication times associated with the module. Using the concept of CCLoad, we propose a clustering algorithm of complexity $O(|V|^2(|V|+|E|)log(|V|+|E|))$, and demonstrate its superiority over a generic version of Sarkar's algorithm.

KEYWORDS

Clustering, Distributed Computing, Homogeneous Systems, Task Allocation

1. INTRODUCTION

A multiprocessor environment consists of a number of processors that are connected through a communication network [1]. The processors are generally identical and are fully connected with identical links. Multiprocessor environments are used to solve large-scale software applications. The availability of multiple processors helps in concurrent execution of processes.

Task allocation algorithms try to exploit parallelism by dividing the software into modules, and allocating them to processors, so that the parallel execution time can be minimized. Modules may have data dependencies between them. Interdependencies between modules are represented as a weighted directed acyclic graph (DAG) called the task graph. Vertices represent the modules having weight as the running time. Edges represent the dependency between modules having weight as the communication time.

Let the two modules M_i and M_j be connected by a directed edge from M_i to M_j, having weight as w_{ij}. Then this means that the module M_j can start its execution after getting some data from the module M_i. Let the module M_i be allocated to the processor P_k, and the module M_j be allocated to the processor P_l. Let the time taken for communication between the module M_i and the module M_j be denoted by c_{ij}. The time for communication of data is given by:

$$c_{ij} = w_{ij}, \tag{1}$$

when the two modules are allocated to different processors $(k \neq l)$, otherwise $(k = l)$ it is given by:

$$c_{ij} = 0. \quad (2)$$

We are using the same computational model as used by Kadamuddi and Tsai [2].

The parallel execution time of a software system may depend on the way in which its modules are allocated to the processors. Task allocation algorithms try to minimize this time. When the multiprocessor environment is homogeneous and fully connected and having an unlimited supply of processors (as in our case), the task allocation problem is also called the clustering problem [2]. A cluster is a set of modules allocated to a processor. In this paper we are trying to solve a clustering problem for a homogeneous and fully connected multiprocessor environment, having an unlimited supply of processors. For a heterogeneous multiprocessor environment that may not be fully connected, and having a limited supply of processors (the real case), we may solve the task allocation problem as a two-step process. In the first step, we solve the clustering problem for the given task graph assuming a homogeneous and fully connected multiprocessor environment, having an unlimited supply of processors. In the second step, we can use some heuristic to allocate the clusters generated in the first step to the given heterogeneous multiprocessor environment that can reduce the parallel execution time.

Given a task graph of modules, the problem of finding an optimal clustering of modules is an NP-Complete problem (Sarkar [3], Papadimitriou [4]). Therefore, heuristics are used for solving the clustering problem in polynomial time.

In our heuristic, we use a value associated with the modules called the *CCLoad* (Computation-Communication-Load). *CCLoad* tries to measure whether a module is computation intensive, or communication intensive. Using the concept of *CCLoad*, we have proposed a clustering algorithm of complexity $O(|V|^2(|V|+|E|)log(|V|+|E|))$.

The remainder of this paper is organized in the following manner. Section 2 discusses some heuristics for solving the clustering problem. Section 3 explains the concept of *CCLoad*. Section 4 presents the *CCLoadClustering* algorithm. In section 5, this algorithm is explained with the help of a simple example. Some experimental results are presented in section 6. And finally in section 7, we conclude our work.

2. CURRENT APPROACHES

The Two modules M_1 and M_2 are called independent if they do not have any data dependency between them. A clustering in which independent modules are kept on separate clusters is called a linear clustering. Whereas a clustering in which independent modules can be clustered together is called a nonlinear clustering [5].

Edge zeroing is a concept in which the two clusters that are connected by a large weight edge are clustered together to avoid the large communication time between them [5]. Sarkar's algorithm [3] uses the concept of edge zeroing. This algorithm first sorts the edges of the task graph in decreasing order of edge costs. It then merges the clusters connected by the highest cost edge, if on doing so the parallel execution time does not increase. This step is repeated until all the edges are examined. Sarkar's algorithm [3] uses level information to determine the parallel execution time, and these levels are computed for each step. The complexity of Sarkar's algorithm [3] is $O(|E|(|V|+|E|))$. The clusters generated are nonlinear.

The Dominant Sequence Clustering (DSC) algorithm of Yang and Gerasoulis [6] finds the critical path of the graph. The critical path is called the Dominant Sequence (DS). An edge from the DS is used to merge its adjacent nodes, provided the parallel execution time reduces. After merging, a new DS is computed and the clustering is tried again until all the modules are

scheduled. Yang and Gerasoulis [6] have shown the complexity of DSC algorithm as $O((|V| + |E|)log(|V|))$. The Greedy Dominant Sequence (GDS) algorithm, of Dikaiakos et al. [7] is a greedy version of DSC algorithm of Yang and Gerasoulis [6]. In GDS, an edge from DS that reduces the parallel execution time the most is selected and the nodes belonging to the edge are merged. The algorithm stops when there is no edge in DS that is able to decrease the parallel execution time. The complexity of GDS algorithm is $O(|V|(|V| + |E|))$ [7].

The Clustering Algorithm for Synchronous Communication (CASC) by Kadamuddi and Tsai [2] is a four-stage algorithm. Its four stages are Initialize, Forward-Merge, Backward-Merge, and Early-Receive. In addition to achieving the traditional clustering objective of reducing the parallel execution time, the CASC algorithm also reduces the performance degradation caused by synchronizations, and avoids deadlocks during clustering. The complexity of CASC algorithm is $O(|V|(|E|^2 + log(|V|)))$ [2].

3. COMPUTATION-COMMUNICATION-LOAD OF A MODULE

3.1. Notation

Let there be n modules $M_i\ (1 \leq i \leq n)$. Let Mi be in cluster $C_i\ (1 \leq i \leq n)$. Let the set of modules be given by:

$$M = \{M_i \mid 1 \leq i \leq n\}. \tag{3}$$

Then the clusters C_i ($M\ (1 \leq i \leq n)$ are such that for $i \neq j\ (1 \leq i \leq n,\ 1 \leq j \leq n)$

$$C_i \cap C_j = \Phi, \tag{4}$$

and

$$U^n_{i=1} C_i = M. \tag{5}$$

Let the label of the cluster C_i be denoted as an integer $cluster[i]\ (1 \leq i \leq n,\ 1 \leq cluster[i] \leq n)$. Let the set of vertices of the task graph be denoted as:

$$V = \{i \mid 1 \leq i \leq n\}. \tag{6}$$

Let the set of edges of the task graph be denoted as:

$$E = \{(i, j) \mid i \in V, j \in V, \exists \text{ an edge from } M_i \text{ to } M_j\}. \tag{7}$$

Let m_i be the execution time of module M_i. If $(i, j) \in E$, then let w_{ij} be the weight of the directed edge from M_i to M_j. If $(i, j) \notin E$, or if $i = j$, then let w_{ij} be 0. Let T be the adjacency list representation of the task graph.

3.2. CCLoad of a Module

A module is computation intensive, if it spends more time in computation as compared to its time spent in communication. Similarly, a module is communication intensive, if it spends more time in communication as compared to its time spent in computation. To measure the relative *Computation-Communication-Load* of a module, we define a value called *CCLoad* of a module as follows:

$$CCLoad_i = m_i - max\_in_i - max\_out_i, \tag{8}$$

where

$$max\_in_i = MAX(\{w_{ji} \mid 1 \le j \le n\}), \quad (9)$$

and

$$max\_out_i = MAX(\{w_{ik} \mid 1 \le k \le n\}). \quad (10)$$

Computation-Communication-Load of a module M_i *(*$CCLoad_i$*)* is defined as its execution time *(*m_i*)* subtracted by maximum weight of incoming edge *(*max_in_i*)* subtracted by maximum weight of outgoing edge *(*max_out_i*)*.

3.3. An Example of CCLoad

In Fig. 1, *CCLoad* of modules are calculated. As an example, for module M_2, $m_2 = 4$, maximum weight of incoming edge is $w_{12} = 4$, maximum weight of outgoing edge is $w_{25} = 6$. Therefore we have:

$$CCLoad_2 = m_2 - w_{12} - w_{25} = 4 - 4 - 6 = -6. \quad (11)$$

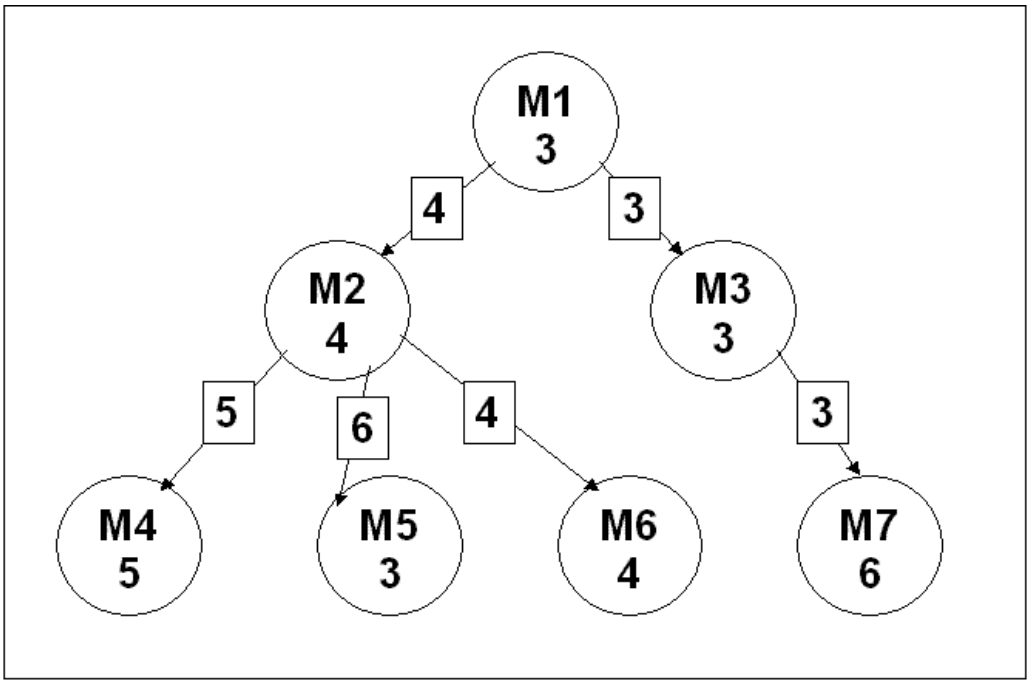

Figure 1. An example task graph for showing the calculation of *CCLoad*. $(CCLoad_i)_{1 \le i \le 7}$ = *(-1, -6, -3, 0, -3, 0, 3)*.

4. THE CCLOAD-CLUSTERING ALGORITHM

4.1. Evaluate-Load

Evaluate-Load (T)
01 **for** $i \leftarrow 1$ to $|V|$
02 **do** $max\_in[i] \leftarrow 0$
03 $max\_out[i] \leftarrow 0$
04 **for** $i \leftarrow 1$ to $|V|$
05 **do** $load[i].index \leftarrow i$
06 **for** each $(i, j) \epsilon E$
07 **do if** $w_{ij} > max\_out[i]$
08 **then** $max\_out[i] \leftarrow w_{ij}$
09 **if** $w_{ij} > max\_in[j]$
10 **then** $max\_in[j] \leftarrow w_{ij}$
11 **for** $i \leftarrow 1$ to $|V|$
12 **do** $load[i].value \leftarrow m_i - max\_in[i] - max\_out[i]$
13 **return** *load*

Given a task graph T, the algorithm *Evaluate-Load* calculates the *CCLoad* for each module in the array *load*. For $(1 \leq j \leq |V|)$, if the *CCLoad* of module M_j is l_j, and if it is stored in $load[i]$, then we have:

$$load[i].value = l_j, \tag{12}$$

and

$$load[i].index = j. \tag{13}$$

In lines 01 to 03, maximum weight of incoming edge $(max\_in[i])$, and maximum weight of outgoing edge $(max\_out[i])$ are initialized to 0. In lines 04 to 10, we consider each edge $(i, j) \epsilon E$, and update the values of $max\_out[i]$ and $max\_in[j]$ accordingly. Finally, in lines 11 to 12, we store the *CCLoad* of module M_i in $load[i]$ for $(1 \leq i \leq |V|)$. Line 13 returns the *load* array.

Lines 01 to 03, and lines 11 to 12 each have complexity $O(|V|)$. Lines 04 to 10 have complexity $O(|E|)$. Line 13 has complexity $O(1)$. Therefore, the algorithm *Evaluate-Load* has complexity $O(|V| + |E|)$.

4.2. Evaluate-Time

Given a task graph T, and a clustering *cluster*, the algorithm *Evaluate-Time* calculates the parallel execution time of the clustering. It is based on the event queue model. There are two types of events: computation completion event, and communication completion event. Events are denoted as 3-tuples (i, j, t). For example, a computation completion event of module M_i, that finishes its computation at time t_i will be denoted as (i, i, t_i), and a communication completion event of a communication from M_i to M_j, that is completed at time t_{ij} will be denoted as (i, j, t_{ij}).

Evaluate-Time (T, cluster)

Step 0: Let ***E*** be the event queue. It is an ascending priority queue based on the value of t. Each processor can be in two states: *IDLE* when it is not executing any module. Initially all processors will be in *IDLE* state. A processor is in *BUSY* state, when it is executing a module. For a given allocation, we also maintain a *ready queue* of ready to run modules for each processor. A module M_i is ready to run, if it is not having any incoming edges or if all of its incoming edges have finished their communication. Initially at least one module is ready to run for which there is no incoming edge. For each processor, one ready to run module will be added

to the event queue (if one exists), setting the *t* value as the time for execution. Corresponding processor states will be set to *BUSY*.

Step 1: One node is deleted from the event queue ***E***. There is a *TIME* variable initialized to *0*. *TIME* denotes the current time. Whenever any node is deleted from ***E***, *TIME* is updated to its timestamp. If the deleted node is a computation completion event, then **go to** *Step 2*. If the deleted node is a communication completion event, then **go to** *Step 5*.

Step 2: When a module finishes its execution, the corresponding processor's state is changed to *IDLE*.

Step 3: If there is any other ready to run module on that processor, then its *t* value is set to *TIME + computation time of module*, and will be added to ***E***. That processor state is again changed from *IDLE* to *BUSY*.

Step 4: For each outgoing edge from the module, a communication completion event is added to ***E***. For setting the value of ***t***, we have two possibilities (for a communication from M_i to M_j):

Step 4A: M_i and M_j are on the same processor. Then *t* is set to *TIME*, since there will not be any communication delay.

Step 4B: M_i and M_j are on different processors. Then *t* is set to *TIME + edge weight*.

Step 5: Let *(i, j, t)* be the deleted event. Check for M_j if all of its incoming edges have completed their communication. If this is so, add M_j to the *ready queue* of its allocated processor. If that processor is currently *IDLE*, then change its state to *BUSY*, and add a computation completion Event of M_j to ***E***.

Step 6: **Repeat** *Step 1* to *Step 6* until a total of $|V| + |E|$ events are added to, and deleted from ***E***.

Step 7: **Return** *TIME*.

There are a total of $(|V| + |E|)$ events out of which $|V|$ events are computation completion events corresponding to each module, and $|E|$ events are communication completion events corresponding to each edge.

Step 1 to *Step 6* are repeated a total of $(|V| + |E|)$ times. In each repetition, the complexity is dominated by the addition and deletion from the event queue ***E*** that has a complexity of $O(log(|V| + |E|))$ (Tenenbaum [8], Cormen and Rivest [9]) if the priority queue is implemented as a min-heap. Therefore the complexity of the *Evaluate-Time* algorithm is $O((|V| + |E|)log(|V| + |E|))$.

4.3. CCLoad-Clustering

```
CCLoad-Clustering (T)
01 load ← Evaluate-Load (T)
02 Sort-Load (load)
03 c_max ← 2
04 for j ← 1 to |V|
05     do cluster[j] ← 1
06 for j ← 1 to |V|
07     do i ← 1
08     t_min ← Evaluate-Time (T, cluster)
09     for k ← 2 to c_max
```

```
10      do cluster[load[j].index] ← k
11      time ← Evaluate-Time (T, cluster)
12      if time < t_min
13         then t_min ← time
14         i ← k
15   cluster[load[j].index] ← i
16   if i = c_max
17      then c_max ← c_max + 1
18 return (t_min, cluster)
```

Our heuristic is based on the following two observations:

(1) Computational intensive tasks can be kept on separate clusters because they mainly involve computation. These tasks will heavily load the cluster. By keeping these tasks separated, we can evenly balance the computational load.

(2) Communication intensive tasks can be kept on same clusters because they mainly involve communication. By keeping these tasks on the same cluster, we may reduce the communication delays through edge zeroing.

The *CCLoad-Clustering* algorithm implements this heuristic. Given a task graph *T*, line 01 evaluates the *CCLoad* of modules. Line 02 sorts the *load* array in decreasing order. Initially all modules are kept in the same cluster (cluster 1, also called the *initial cluster*, lines 04 to 05). c_{max} (line 03) is the number of possible clusters that can result, if one module is removed from the *initial cluster*, and put on a different cluster (including the *initial cluster*).

Modules are taken out from the *initial cluster* one-by-one (line 06) in decreasing order of *CCLoad* (line 10). We calculate the parallel execution time, when it is put on all possible different clusters (lines 09 to 11). t_{min} records the minimum parallel execution time, and *i* records the corresponding cluster (lines 12 to 14). We put the module on the cluster that gives the minimum parallel execution time (line 15).

It may happen that the parallel execution time was minimum when the module was put alone on a new cluster. In this case we have to increment c_{max} by 1 (lines 16 to 17). Line 18 returns the parallel execution time, and the corresponding clustering.

Line 01 has complexity $O(|V| + |E|)$. Line 02 has complexity $O(|V|^2)$ if bubble sort is used (Tenenbaum [8]). Lines 03 and 18 each have complexity $O(1)$. Lines 04 to 05 have complexity $O(|V|)$. Lines 08 and 11 have complexity $O((|V| + |E|)log(|V| + |E|))$. For each iteration of the **for** loop in line 06, *Evaluate-Time* (lines 08 and 11) is called a maximum of $|V|$ times (c_{max} can have a maximum value of $|V|$, when all the modules are on separate clusters). The complexity of the **for** loop of lines 06 to 17 is dominated by *Evaluate-Time* that is called a maximum of $|V|^2$ times. Therefore, the *for* loop has complexity $O(|V|^2(|V| + |E|)log(|V| + |E|))$ that is also the complexity of *CCLoad-Clustering* algorithm.

5. A SIMPLE EXAMPLE

Consider the task graph in Fig. 2. Initially all modules are clustered in the *initial cluster* as $(cluster[i])_{1 \le i \le 4} = (1, 1, 1, 1)$. Parallel execution time is *8*. Modules are sorted according to *CCLoad* in decreasing order as (M_1, M_2, M_3, M_4).

Initially module M_1 is taken out to form the clustering *(2, 1, 1, 1)*. Parallel execution time for this clustering comes out to be *9*. This is not less than *8*. Therefore, module M_1 is kept back in the *initial cluster* as *(1, 1, 1, 1)*.

The next module to be taken out is M_2 to form the clustering *(1, 2, 1, 1)*. Parallel execution time for this clustering comes out to be *8*. This is not less than *8*. Therefore, module M_2 is kept back in the *initial cluster* as *(1, 1, 1, 1)*.

The next module to be taken out is M_3 to form the clustering *(1, 1, 2, 1)*. Parallel execution time for this clustering comes out to be *7*. This is less than *8*. Therefore, module M_3 is kept in a *separate cluster* as *(1, 1, 2, 1)*.

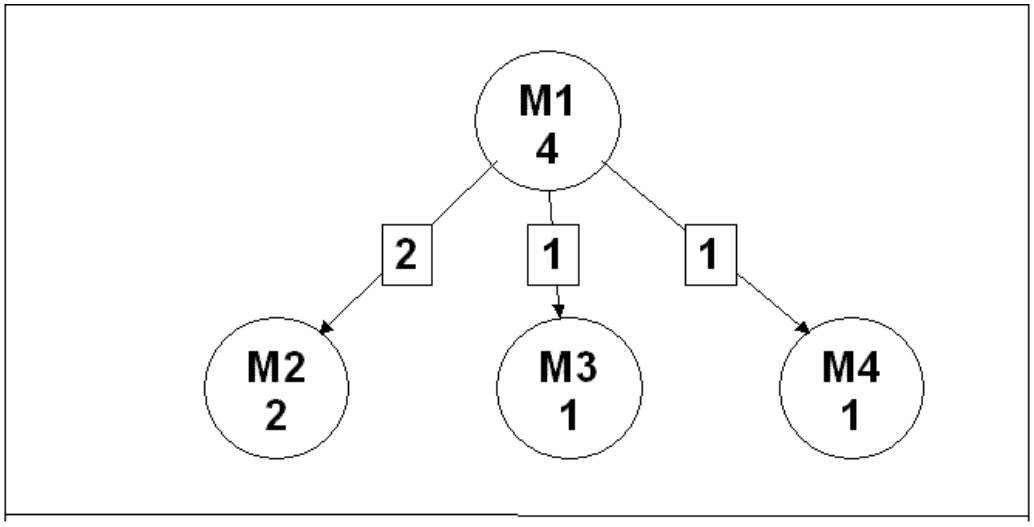

Figure 2. An example task graph for explaining the *CCLoad-Clustering* algorithm. $(CCLoad_i)_{1 \le i \le 4} = (2, 0, 0, 0)$. The *CCLoad-Clustering* algorithm clusters the modules as $(M_1, M_2)(M_3)(M_4)$, giving a parallel execution time of *6*.

The next module to be taken out is M_4. The two possible clustering are *(1, 1, 2, 2)*, and *(1, 1, 2, 3)*. Parallel execution time for the clustering *(1, 1, 2, 2)* comes out to be *7*. Parallel execution time for the clustering *(1, 1, 2, 3)* comes out to be *6*. The minimum parallel execution time is *6* for the clustering *(1, 1, 2, 3)* that is also less than *7*. Therefore, module M_4 is also kept in a *separate cluster* as *(1, 1, 2, 3)*.

This gives a clustering of modules as $(M_1, M_2)(M_3)(M_4)$ in which the modules M_1 and M_2 are clustered together, while the modules M_3 and M_4 are put on separate clusters. This clustering gives a parallel execution time of *6*.

6. EXPERIMENTAL RESULTS

The *CCLoad-Clustering* algorithm is compared with a generic version of Sarkar's algorithm [3]. It is the algorithm 17 in Sinnen [5]. We will call it the *Generic-Sarkar* algorithm. In this algorithm, we set the priority of each edge as its weight (greater weight means higher priority) and *Evaluate-Time* is used for evaluating the parallel execution time of clusterings. This implementation of the *Generic-Sarkar* algorithm has a complexity of $O(|E|(|V| + |E|)log(|V| + |E|))$.

We have tested the two algorithms on the benchmark task graphs of Tatjana and Gabriel [10], [11]. *120* task graphs having number of nodes as *50*, *100*, *200*, and *300* respectively are selected

for testing. Each task graph is labelled as *tn_i_j.td*, where *n* is the number of nodes. The variable *i* is a parameter depending on the edge density having possible values as *20*, *40*, *50*, *60*, and *80* respectively. The variable *j* can have *6* possible values ranging from *1* to *6* for each combination of *n* and *i*, representing *6* task graphs each. This makes a total of *30* task graphs for each *n*.

Figures from Fig. 3 to Fig. 6 show a comparison of parallel execution times between the *CCLoad-Clustering* and the *Generic-Sarkar* algorithms for *n* having values *50*, *100*, *200*, and *300* respectively. From the figures it is clear that the average improvement of *CCLoad-Clustering* over *Generic-Sarkar* ranges from *6.18%* (Fig. 3) to *7.69%* (Fig. 6). We also observed that the *CCLoad-Clustering* algorithm was practically very fast as compared to the *Generic-Sarkar* algorithm. This was due to the fact that our algorithm was not taking its worst-case time, since the number of clusters generated was small. We define the *Speedup Ratio (SR)* as the ratio between the running time of *Generic-Sarkar* ($T_{GenericSarkar}$) and the running time of *CCLoad-Clustering* ($T_{CCLoadClustering}$):

$$SR = T_{GenericSarkar} / T_{CCLoadClustering}. \quad (14)$$

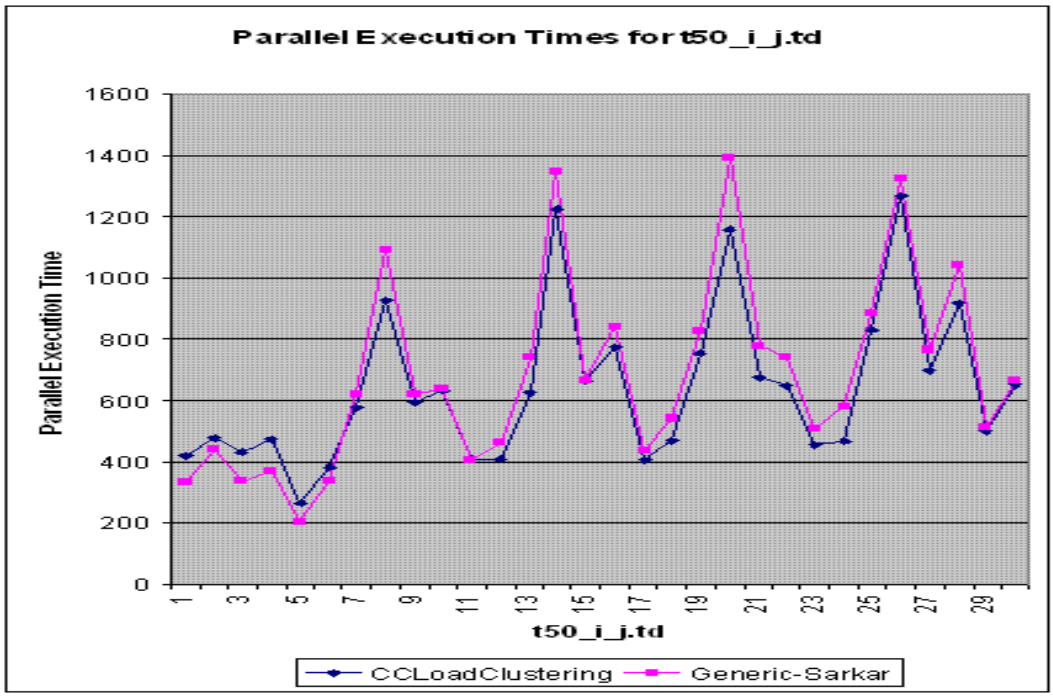

Figure 3. Parallel execution times for *t50_i_j.td*. Average improvement = *6.18%*.

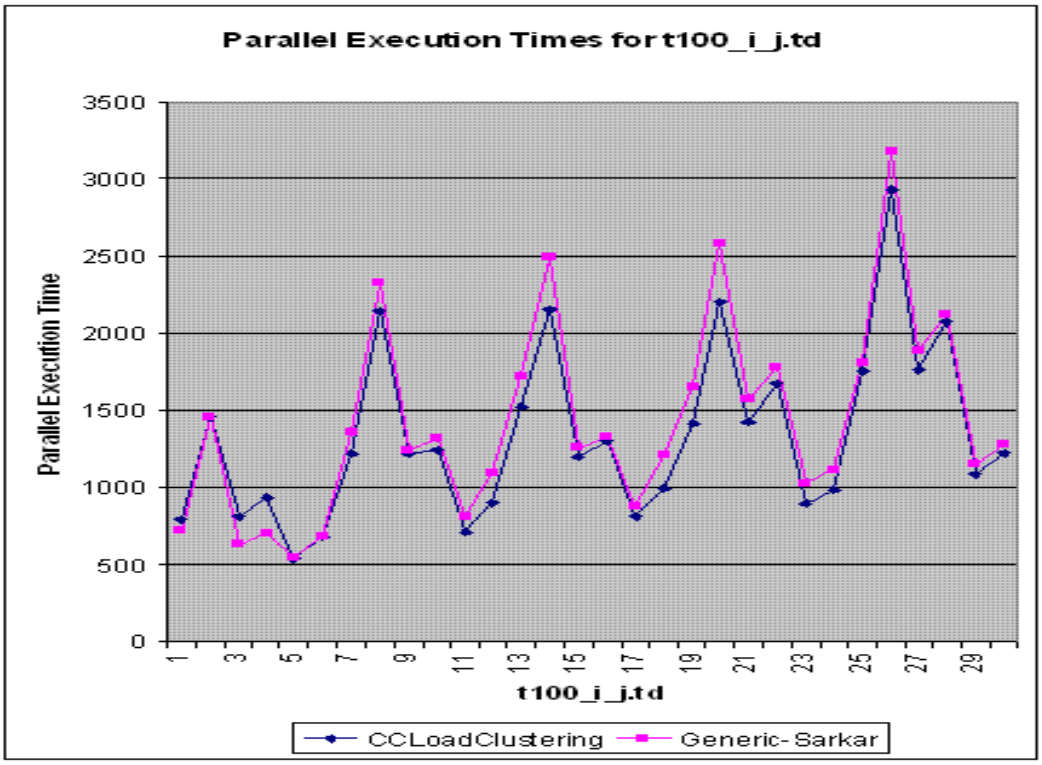

Figure 4. Parallel execution times for *t100_i_j.td*. Average improvement = *6.72%*.

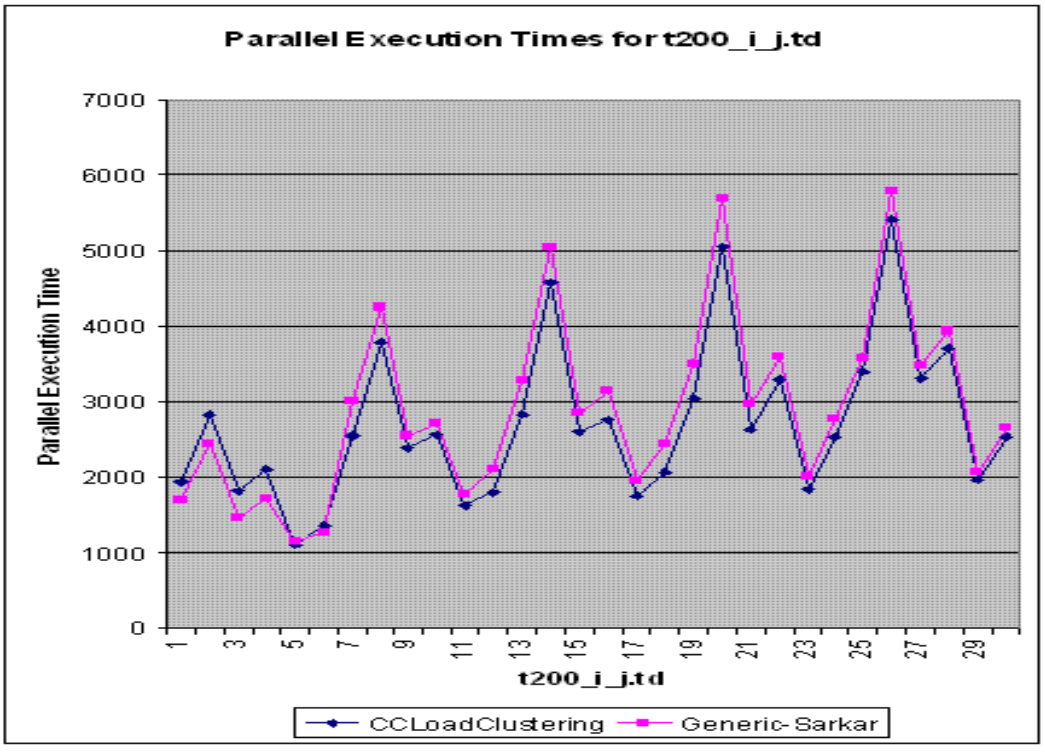

Figure 5. Parallel execution times for *t200_i_j.td*. Average improvement = *6.52%*.

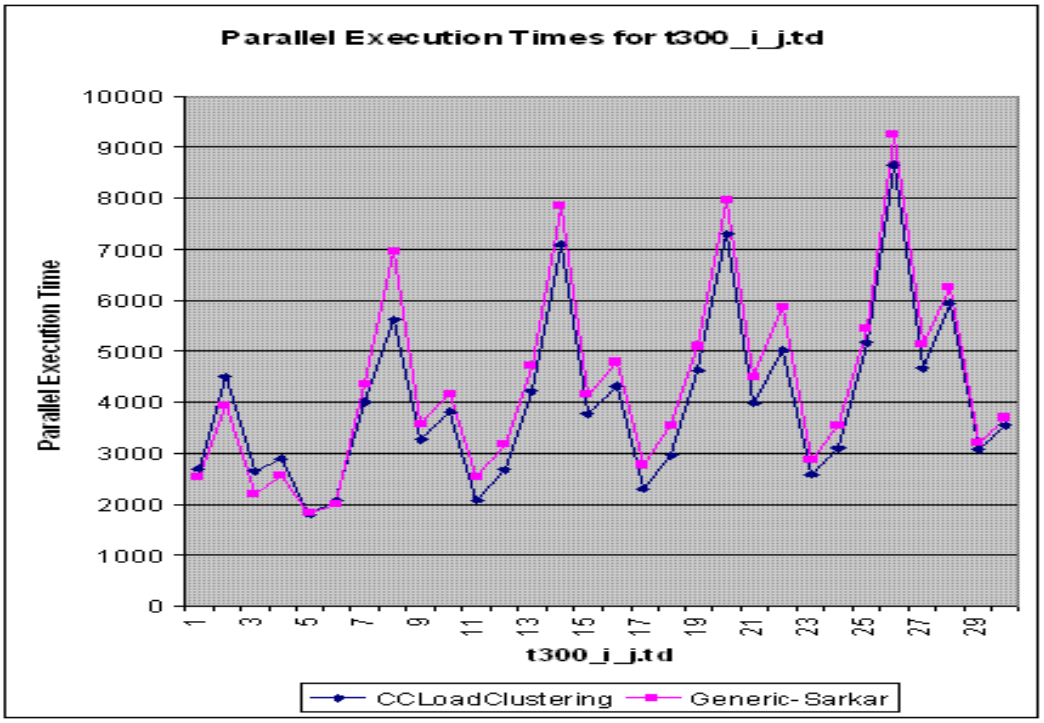

Figure 6. Parallel execution times for *t300_i_j.td*. Average improvement = *7.69%*.

From Fig. 7 it is clear that the *CCLoad-Clustering* algorithm is up to *6.87* times faster than the *Generic-Sarkar* algorithm.

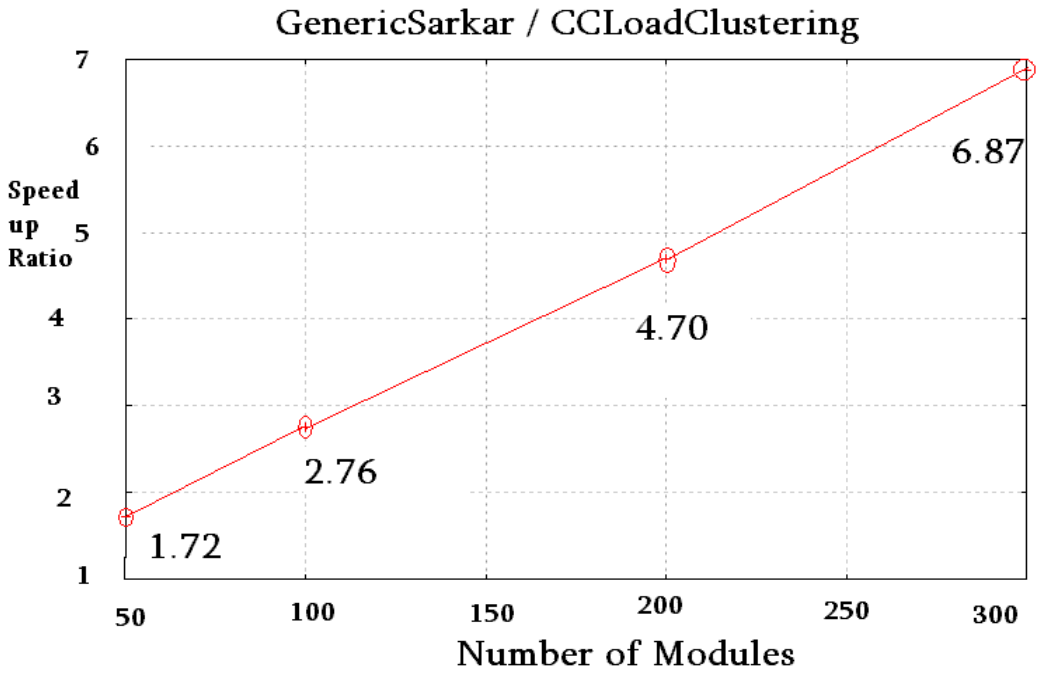

Figure 7. Average Speedup Ratios.

7. CONCLUSION

We developed the idea of *CCLoad* (Computation-Communication-Load) of a module that is dependent on the computation and communication requirements of the modules. We used a heuristic based on it to develop the *CCLoad-Clustering* algorithm to solve the task allocation problem on a fully connected homogeneous multiprocessor environment. We proved the time complexity of the algorithm to be $O(|V|^2 (|V| + |E|) log(|V| + |E|))$. We compared our algorithm with a generic version of Sarkar's algorithm [3] called the *Generic-Sarkar* algorithm (Sinnen [5]). We demonstrated its superiority over the *Generic-Sarkar* algorithm in terms of performance as well as speed.

A possible future work is to decrease the complexity of the *CCLoad-Clustering* algorithm. We have used the event queue model to calculate the parallel execution time of the clusterings. The resulting time complexity was $O((|V| + |E|) log (|V| + |E|))$ that resulted in a time complexity of the *CCLoad-Clustering* algorithm to be $O(|V|^2 (|V| + |E|) log(|V| + |E|))$. However if we use the concept of scheduled DAG's (Sarkar [3], Sinnen [5], Yang and Gerasoulis [6]) the resulting time complexity for determining the parallel execution time of clusterings will decrease to $O((|V| + |E|))$ reducing the time complexity of the *CCLoad-Clustering* algorithm to be $O(|V|^2 (|V| + |E|))$.

Another possible future work is to use the concept of *Dynamic Computation Communication Load* of a module. The *CCLoad* that we defined in this paper is a static quantity that only depends on the computation and communication requirements of the modules. It is independent of the current allocation of the modules. We can extend this idea and can include other parameters that are dynamically dependent on the current allocation of the modules. It will be interesting to compare the *dynamic* version of the *CCLoad-Clustering* algorithm with the *static* version of the *CCLoad-Clustering* algorithm that we developed in this paper.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referee for providing helpful suggestions for improving the quality of the paper.

REFERENCES

[1] K. Hwang (1992) *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, First Edition, McGraw-Hill, ISBN: 978-0070316225.

[2] D. Kadamuddi, & J.J.P. Tsai, (2000) "Clustering Algorithm for Parallelizing Software Systems in Multiprocessors Environment", *IEEE Transactions on Software Engineering*, Vol. 26, No. 4, pp. 340-361.

[3] V. Sarkar (1989) *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Research Monographs in Parallel and Distributed Computing, MIT Press.

[4] C. Papadimitriou, & M. Yannakakis, (1990) "Towards an Architecture Independent Analysis of Parallel Algorithms", *SIAM Journal on Computing*, Vol. 19, pp. 322-328.

[5] O. Sinnen (2007) *Task Scheduling for Parallel Systems*, Wiley Interscience. ISBN: 978-0-471-73576-2.

[6] T. Yang and A. Gerasoulis, (1991) "A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors", *Proc. Fifth Int'l Conf. Supercomputing*, pp. 633-642.

[7] M.D. Dikaiakos, A. Rogers, & K. Steiglitz (1994) *A Comparison of Techniques Used for Mapping Parallel Algorithms to Message-Passing Multiprocessors*, Technical Report, Princeton Univ.

[8] Y. Langsam, M.J. Augenstein, & A.M. Tenenbaum (1996) *Data Structures Using C and C++*, 2nd edition, 1996, Prentice Hall, ISBN: 9780130369970.

[9] T. H. Cormen, C.E. Leiserson, R.L. Rivest, & C. Stein (2001) *Introduction to Algorithms*, Second Edition, MIT Press, ISBN: 978-0-262-03293-3.

[10] T. Davidovic, & T.G. Crainic, (2006) "Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems", *Computers & Operations Research*, Vol. 33, No. 8, pp. 2155-2177.

[11] Benchmark task graphs DOI: http://www.mi.sanu.ac.rs/~tanjad/sched_results.htm.

Authors

Pramod Kumar Mishra is Associate Professor in Banaras Hindu University, Varanasi. His research interests include Parallel and Distributed Computation, Computational Complexity, Parallel and Clustered Data Mining, High Performance Computing, and VLSI Algorithms.

Kamal Sheel Mishra is Reader of Computer Science at the School of Management Sciences, Varanasi. He is the Head of the Department of Computer Science. His research interests include Parallel and Distributed Computing, and Software Engineering.

Abhishek Mishra is pursuing his Ph.D. degree in Computer Engineering at Institute of Technology, Banaras Hindu University, Varanasi. He received his Bachelor degree in Computer Engineering from the same institute in 2003. His current research interests include Approximation Algorithms, and Combinatorial Optimization.