GRAPH MATCHING ALGORITHM FOR TASK ASSIGNMENT PROBLEM

R.Mohan[1] and Amitava Gupta[2]

[1]Dept. of Computer Science and Engineering, National Institute of Technology
Tiruchirapalli Tamil Nadu 620015, India
rmohan@nitt.edu
[2]Dept. of Power Engineering Jadavapur University,Kolkata, West Bengal 700098, India
amitg@pe.jusl.ac.in

ABSTRACT

Task assignment is one of the most challenging problems in distributed computing environment. An optimal task assignment guarantees minimum turnaround time for a given architecture. Several approaches of optimal task assignment have been proposed by various researchers ranging from graph partitioning based tools to heuristic graph matching. Using heuristic graph matching, it is often impossible to get optimal task assignment for practical test cases within an acceptable time limit. In this paper, we have parallelized the basic heuristic graph-matching algorithm of task assignment which is suitable only for cases where processors and inter processor links are homogeneous. This proposal is a derivative of the basic task assignment methodology using heuristic graph matching. The results show that near optimal assignments are obtained much faster than the sequential program in all the cases with reasonable speed-up.

KEYWORDS

task assignment, load balancing, graph partition, Heuristic graph matching, Symmetrical Multi-processors.

INTRODUCTION

Load balancing is the process by which task modules constituting an application are assigned to processors, with the goals of maximizing the processor utilization and minimizing the total turnaround time. This can be viewed as a task assignment problem by which task modules are assigned to available processors in order to achieve the aforesaid two goals. Achievement of the above proposal is possible by a number of techniques viz. graph partitioning, graph matching (the 2 most widely used ones), hybrid methodology and mathematical programming. First a survey of task assignment strategies is presented. Next, the heuristic graph matching based task assignment by methodology of Shen and Tsai [3] is explained. Finally the parallel algorithm is presented and its performance is analyzed using several representative test cases.

Graph Partitioning based Methodologies

Graph partitioning techniques view the task as a task graph where the vertices represent the task modules and edges represent the communication between those tasks. In load balancing, the graph partitioning methodologies are used to produce equal partitions of the task graph with the inter-node communication or the volume of such communication minimized. The number of

DOI : 10.5121/ijcsea.2011.1609

partitions depends on the number of processing elements and their topology in the processor graph. Some factors are:

•*Load balance:* The computational work of each processor should be balanced, so that no processor will be waiting for others to complete.

•*Communication cost:* On a parallel computer, accumulating the contributions from nodes that are not on the current processor will incur communication cost which must be minimized.

The graph bisection problem has been studied in the past by many authors (see, e.g., [4], [5], [6]) in the context of graph theory as well as VLSI circuit layout. Advances in parallel computing hardware and software have renewed interest in the problem. The graph bisection problem is a NP hard problem, so there are no known algorithms that can find the exact solution to the problem in polynomial time. Most of the graph bisection methods, therefore, seek a good approximation to the optimal partitioning that can be calculated efficiently. Some already proposed algorithms are

• Recursive graph bisection (RGB): algorithm [6] attempts to partition the task graph recursively. The RGB algorithm first finds the two vertices that are the furthest apart (their distance is called the diameter of the graph). Then, starting from one of them (the root), the half of the vertices that are closer to the root from one partition, the rest from other. This process is then recursively executed on each of the partitions.

• Greedy algorithm starts with a vertex with the smallest degree, and marks its neighbours, and then the neighbours' neighbours. The first n/p marked vertices (assuming a task graph with n vertices to be partitioned amongst p processors) are taken to form one partition and the procedure is applied to the remaining graph until all of the vertices are marked. This algorithm [7] is similar to the RGB algorithm, although it is not a bisection algorithm. Like RGB it has a low complexity of $O(n)$.

• K-L (Kernighan-Lin) algorithm ([4] [8] [9]) is an iterative algorithm. Starting from a load balanced initial bisection, it first calculates for each vertex the gain in the reduction of edge-cut that may result if that vertex is moved from one partition of the graph to the other. In each of inner iteration, it moves the unlocked vertex that has the highest gain, from the partition in surplus to the partition in deficit. This vertex is then locked and the gains updated. The procedure stops if the gain of the move is negative, and this negative move is undone which results with bisection with the smallest edge-cut in this iteration. Other iterations will continue until that time. If one iteration fails to result in reduction of edge-cut, the problem terminates.

Parallel partitioning algorithms

Although the multilevel approach of Kernighan-Lin Algorithm reduces the computing time significantly, it can prove to be memory intensive for a very large task graph - often exceeding the limits of single CPU memories. Furthermore as the ultimate purpose of partitioning is for the subsequent implementation of the application code on parallel machines, it makes sense to have a parallel partitioning code. Besides, a fast parallel partitioning algorithm can also be used for the purpose of dynamic load balancing. There have been a number of efforts in this area.

In [10] [11], the multilevel spectral bisection was parallelized specifically for the Cray architecture using the Cray SHMEM facility. The linear algebra algorithms are used to parallelize the tasks. Difficulties arose in the parallel graph coarsening, in particular, in the parallel

generation of the maximal independent set. These were tackled by using a number of parallel graph theory algorithms. On a 256 processor Cray T3D, the resulting algorithm PMRSB (Parallel multilevel recursive bisection algorithm) is able to partition a graph of 1/4 million vertices into 256 sub domains, which is 140 times faster than a workstation using an equivalent serial algorithm.

In [12], a parallel multilevel *p* way-partitioning scheme was developed. This is based on work in the sequential *p* way partitioning algorithm METIS [13], but with some interesting modification to facilitate parallelization. In the previous PMRSB algorithm, for finding the maximal independent set was employed. However unlike MRSB, where the maximal independent set is used directly to form the coarse graph, here the maximal independent set is used for a different purpose. By considering one independent set at one time, it is possible to avoid conflicts during the coarsening stage when the vertices are matched, as well as during the uncoarsening stage when the boundaries are refined. One way of avoiding this conflict is for the two processors to communicate with each other and to collaborate on a migration strategy. When there are more than two processors, this may necessitate a colouring of the processor graphs so that processors are grouped in pairs and refinement is carried out on boundaries of the paired processors.

The parallel partitioning algorithms use a different refinement strategy. This algorithm is designed also for dynamic load balancing, the initial partitioning is assumed to be unbalanced. For any two neighbouring sub domains p and q, the flow (the amount of load to be migrated to achieve global load balance) is first calculated. The flow from p to q is denoted as f(pq). Let g(pq) denote the total weight of the vertices on the boundary of p which have a preference to migrate to q. Let d=max (g(pq)-f(pq)+g(qp)-f(qp), 0), which represents the total weight of all boundary vertices with a positive gain after the flow is satisfied. Then the load has to be migrated from p to q. This allows the flow to be satisfied and at the same time an additional load of equal amount is exchanged between the two processors to optimize the edge-cut.

In other parallel graph partitioning algorithms, the parallel single level algorithm, combining inertia bisection with K-L refinement was implemented. Possible conflict during the refinement stage was avoided by the pairing of processors based on the edge colouring of the processor graph. The quality of the partition was not as good as multilevel algorithms.

In other parallel graph partitioning algorithms, the parallel single level algorithm, combining inertia bisection with K-L refinement was implemented. Possible conflict during the refinement stage was avoided by the pairing of processors based on the edge coloring of the processor graph. The quality of the partition was not as good as multilevel algorithms.

Then a spectral inertia bisection algorithm was introduced. The spectral set of eigenvectors for the coarsest graph was first calculated which serves as the spectral coordinates of the vertices. The graph was partitioned with the inertia bisection algorithm, based on these spectral coordinates. Part of the algorithm was parallelized. This algorithm is also suitable for dynamic load balancing on applications where the mesh is enriched by the refinement of the individual elements.

In such a case, the refinement can be captured by updating the vertex weights of the dual graph of the mesh, without changing the graph itself. The mesh is repartitioned quickly after refinement, using the spectral information originally calculated for the top-level coarse mesh. Since the size of the dual graph does not change, the repartitioning time does not change with the increase of the mesh size, as the refinement steps are carried out.

Existing partitioning software

A number of packages are publicly available. Many of them are constantly being updated.

- CHACO(http://www.cs.sandia.gov/CRF/chac.html)
  A package has been available. The latest version CHACO 2.0 uses multilevel combined with a number of available methods such as K-L refinement, spectral bisection. Available by request.
- JOSTLE(http://www.gre.ac.uk/jostle/)
  Multilevel graph partitioning using *p*-way refinement. Both the sequential and parallel version is available free for academic use as an executable, after signing an agreement.
- METIS(http://www.cs.umn.edu/~karypis/metis/metis.html)
  Multilevel graph partitioning using *p*-way refinement. Both the sequential and the parallel version are available free for down-load.
- PARTY(http://www.unipaderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html)
  Sits on top of existing packages, offers greedy, multilevel spectral bisection, coordinate bisection, inertia bisection and other algorithms. Version 1.1 is available after signing a license.
- PMRSB
  parallel multilevel spectral bisection. Available only on CRAY platforms.
- S-HARP [41] (http://www.cs.njit.edu/sohn/sharp/)
  Multilevel graph partitioning using multiple eigenvectors and (spectral) inertia bisection on the coarsest level.
- SCOTCH (http://www.labri.ubordeaux.fr/Equipe/ALiENor/membre/pelegrin/scotch/)
  Multilevel implementation of a number of algorithms including K-L. The code is free for down-load for academic use.
- TOP/DOMDEC
  Greedy algorithm, recursive coordinate and graph bisection, recursive inertia bisection, multilevel spectral bisection algorithm with Tabu search, simulated annealing and stochastic evolution as smoothers**.**
- WGPP
  Multilevel graph partitioning. Similar to METIS with some new features.

Heuristic Graph Matching

This is a graph matching based method, which uses a *task-graph* and a *processor-graph*. While the task-graph denotes the dependency amongst the task modules, the *processor-graph* defines the topology of interconnection amongst the processors. A classical example of this is the work by Shen and Tsai [3] which uses the well-known A^* algorithm to find the optimal task assignment.

A *mapping* implies assignment of any one or more of the *n* task modules to any one or more of the *p* processors with no task module assigned to more than one processor. This *branch and bound heuristics* based methods starts by an initial mapping and expands the state-space by generating other permissible *mappings.* Each mapping or state-space entry has a cost function associated which is denoted by *f.* In [3], this cost function is expressed in terms of a single entity *viz.* time and may be thought to be composed of two parts *viz. g* which may be viewed as the cost of generation of the state-space entry and *h,* which may be viewed as the cost to generate the

goal-state from the present state-space entry and is the heuristic weight associated with the state-space entry. Thus, for each mapping or state-space entry,

$$f = g + h \qquad 1.1$$

As long as there is an upper bound h_{max} for h, i.e. $h \leq h_{max}$, the A* algorithm guarantees that an optimal solution is found [14]. Thus with $g = 0$, the search becomes a purely heuristic search and with $h = 0$ it becomes a *best-first* search.

If n task modules are assigned to m processors, there can be m^n assignments theoretically possible. The method proposed by Shen and Tsai has a typical complexity of $O(n^2)$ for $n \leq 20$ and this complexity approach $O(m^n n^2)$ as n becomes large. Therefore this algorithm is not suitable for large task graphs.

1.1. Hybrid load balancing methodology for a cluster of SMPs

A hybrid methodology to obtain an optimal task assignment across a cluster of SMPs was proposed by Gao et. al. in [1][2]. Each processing element of a cluster is a node comprising a number of tightly coupled processors. The hybrid methodology graph partitioning first assigns the task modules across all nodes of the cluster so as to have equal load on all nodes with inter-node communication optimized. Next, modules constituting each of these sub-tasks are assigned to processors constituting respective nodes using this algorithm of heuristic graph matching. This algorithm works for a moderate number of modules (approximately 20) per node, but fails for large numbers. The intra-node task assignment algorithm proposed in [1] has been further modified by Gao et. al. in [2] where multi step algorithm has been proposed.

Load distribution for parallel application on a cluster of *Symmetrical Multiprocessors* (SMPs) posses a challenging problem. A cluster of SMPs essentially comprising of a finite number of computing nodes, each comprising of 2 or more identical, tightly coupled processing elements, the nodes being connected over a network. While the approximate method of load balancing using standard methods like graph partitioning, for example, can produce acceptable task assignment across the nodes, they cannot be applied to obtain optimal task assignment on the processor constituting a node. This is because of the complex optimization involved when one considers the fact that all the processing element in a node have only one network interface and that the node turn-around is the minimum when the computation and communication activities of the processing elements can be interleaved optimally.

Motivation for the work done

Though parallel graph partitioning algorithms exist, parallel graph matching algorithms do not. Graph matching algorithms like the one proposed by Shen and Tsai are extremely useful considering fact that they can tackle processor heterogeneity; they cannot be used for practical test cases with a large number of task modules. This is primarily because of the exponential complexity of the resource hungry A* algorithm. A parallel graph matching methodology is likely to reduce the size of the state-space handled by each parallel graph-matching task, thus producing acceptable mapping within acceptable time limits.

For example, if a parallel graph-matching algorithm can substitute the sequential graph matching part used with the hybrid methodology proposed by Gao *et. al.* in [1], the processor utilization is bound to increase and the execution time for finding optimal task assignment on the processor constituting the SMPs node is bound to reduce. This is further justifiable when one considers the fact that in case of application integrated load balancing, all processor would be involved in

finding the optimal task assignment which makes faster restart possible in case of a node failure, for example.

In this dissertation it is attempted to parallelize the basic heuristic graph-matching algorithm of task assignment put forwarded by Shen and Tsai. This is particularly suitable for cases where processors and inter processor links are homogeneous. A typical example is a node of a SMP cluster comprising a finite number of identical processors.

This methodology is a derivative of the basic task assignment methodology using heuristic graph matching proposed in [3] by Shen and Tsai. The methodology is tested on a few representative test cases Message Passing Interface (MPI) is used as the platforms. The results show that near optimal assignments are obtained much faster than the sequential program in all the cases.

PARALLEL GRAPH MATCHING ALGORITHM

In this section, the original sequential algorithm proposed by Shen and Tsai in [3] is first presented. This algorithm is then analyzed and the portions which can be parallelized are identified. Finally, the parallel graph-matching algorithm is presented and explained with an illustrative example.

Parallel graph-matching algorithm

The basic methodology proposed by Shen and Tsai is based on a generate and test mechanism. Generation of state space nodes expands the state space and incurs computation time as successive nodes are generated and the associated costs are computed. The parallel graph matching algorithm parallizes the generate mechanism, thus dividing the state space into several smaller state-spaces. The basic steps involved are as follows.

Let N be the number of parallel graph matching tasks.

Let T = (VT, ET) represent the task graph

P = (VP, EP) represent the processor graph

Let Pi = (Vpi, Epi) be a sub graph of P, which is used by the ith task for mapping.

The number of sub graphs of P is assumed to be equal to the number of tasks. Each parallel graph-matching task is assumed to follow the steps 1 to 5 followed by the sequential algorithm, the only difference being the fact that the node for expansion is the one with minimum value of f computed across all the parallel tasks. For this purpose, it is further assumed that the tasks send to each other the mapping corresponding to the entry with minimum value of f once a fixed number of new entries are added to the state-space. This variable is defined as node_count.
Each parallel graph-matching task proceeds as follows: -

Step 1

Put Ki = ø on a list OPEN and set f (Ki) = 0 when f is the evaluation function defined by equation 2.1. If Mglobal represents the global optimal mapping, i.e. the mapping with smallest value of f found by all graph matching tasks, then initialize this to Ki.

Step 2

Set n = Mglobal

Step 3

If n represents the state with no unmapped task, with the smallest value of f among all OPEN nodes, or the number of new additions to the state-space equals node_count then send this optimal mapping (Mlocal) to all parallel graph-matching tasks. Also wait for others to send in their Mlocal. Find the mapping with minimum value of Mlocal and set it to Mglobal. If Mglobal has no unmapped tasks, this is the desired optimal task. Otherwise set n = Mglobal and continue.

Step 4

Remove from OPEN the node n with the smallest f value and put it on a list called CLOSED.

Step 5

Expand the node n, using operators applicable to n and compute f (n') = g (n') + h (n') for each successor n' of n. it is to be noted that the ith graph matching task generates the successors by expanding the mapping corresponding to n by adding successive task modules to processors represented by the set Vpi only. Go back to step 3.

It is clear that the value of node_count determines how frequently the parallel graph matching tasks communicate. If this value is small, the tasks communicate too often and this increases the turnaround time so far the task assignment problem is concerned. If this is too large, then the solution cannot find optimal solution, as many possibilities remain unexplored. The method is very useful in cases where processors are homogeneous and so are the links as the optimal mapping is not unique in this case, and the state space contains many replicated entries.

RESULTS AND DISCUSSIONS

Two representative test cases are presented in Fig 1 and Fig 2 representing task graphs. In Fig 1 and Fig 2, the vertices vi, vj represent task modules and the edge (eij) represent the connection between the vertices vi and vj. The number on the vertices represents the computation time of task module of that particular vertex. Similarly, the numbers on the edges represent the communication time involved in data transfer between two vertices vi and vj through edge eij. The computation and communication time are represented in m sec associated with the vertices and edges.

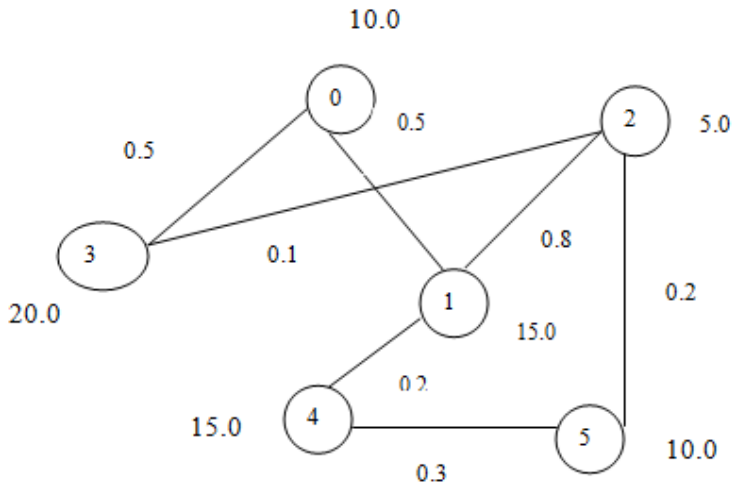

Figure 1 A representative task graph with 6 nodes

In Fig 1, the number of nodes in the task graph is 6, which means that there are 6 modules defined by ***T*** = *{0, 1, 2, 3….5}* which need to be mapped. The computation time associated with these modules defined by the set ***TP*** = *{10.0, 15.0, 5.0, 20.0, 15.0, 10.0}*. The inter module communication is defined by the matrix **C**.

$$\mathbf{C} = \begin{matrix} 0.0 & 0.5 & 0.0 & 0.5 & 0.0 & 0.0 \\ 0.5 & 0.0 & 0.8 & 0.0 & 0.2 & 0.0 \\ 0.0 & 0.8 & 0.0 & 0.1 & 0.0 & 0.2 \\ 0.5 & 0.0 & 0.1 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.2 & 0.0 & 0.0 & 0.0 & 0.3 \\ 0.0 & 0.0 & 0.2 & 0.0 & 0.3 & 0.0 \end{matrix}$$

Similarly, In Fig 2, the number of nodes in the task graph is 12, defined by ***T*** = *{0, 1, 2, 3…11}* which need to be mapped. The computation time associated with these modules defined by the set ***TP*** = *{10.0, 15.0, 5.0, 20.0, 15.0, 10.0, 10.0, 5.0, 2.0, 1.0, 5.0, 10.0}*.

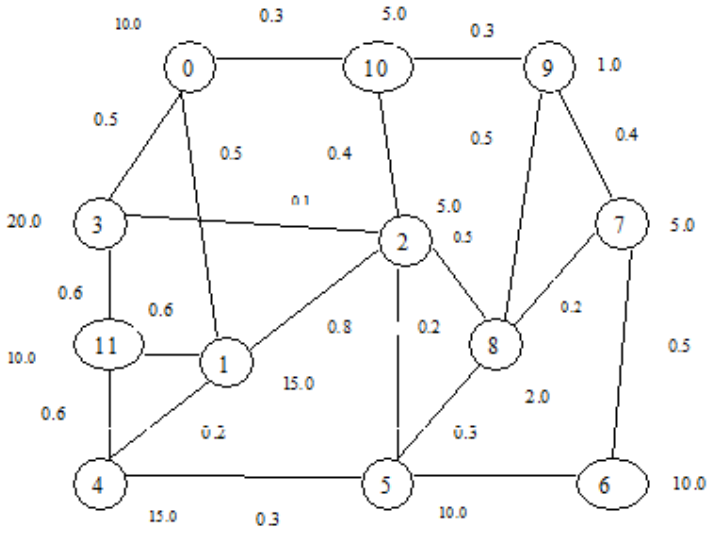

Figure 2 A representative task graph with 12 nodes

$$C = \begin{bmatrix} 0.0 & 0.5 & 0.0 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.3 & 0.0 \\ 0.5 & 0.0 & 0.8 & 0.0 & 0.2 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.6 \\ 0.0 & 0.8 & 0.0 & 0.1 & 0.0 & 0.2 & 0.0 & 0.0 & 0.5 & 0.0 & 0.4 & 0.0 \\ 0.5 & 0.0 & 0.1 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.6 \\ 0.0 & 0.2 & 0.0 & 0.0 & 0.0 & 0.3 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.6 \\ 0.0 & 0.0 & 0.2 & 0.0 & 0.3 & 0.0 & 0.6 & 0.0 & 0.3 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.6 & 0.0 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.5 & 0.0 & 0.2 & 0.4 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.5 & 0.0 & 0.0 & 0.3 & 0.0 & 0.2 & 0.0 & 0.5 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.4 & 0.5 & 0.0 & 0.3 & 0.0 \\ 0.3 & 0.0 & 0.4 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.3 & 0.0 & 0.0 \\ 0.0 & 0.6 & 0.0 & 0.6 & 0.6 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

The number of homogenous processors involved is assumed to be two to start with and the communication link speed is same between two modules of the application it maps.

First the heuristic function *h (n)* is assumed to be 0 for the state space based search algorithm. Next, the test is repeated with a non-zero *h (n)*. Each case is compared to corresponding sequential implementation.

An index called the *optimality index* denoted by μ is introduced to quantify the result.

$$\mu = \frac{\text{Optimal turnaround time (Parrallel)}}{\text{Optimal turnaround time (Sequential)}} \qquad 3.1$$

Similarly, an index η is defined an

$$\eta = \frac{\text{Nos. nodes generated (Parrallel)}}{\text{Nos. nodes generated (Sequential)}} \qquad 3.2$$

Table-1 Results for a *task graph* with 6 task modules setting *h(n) = 0*

node_ count	Optimal mapping	Turnaround time *(m sec)*	No. of nodes generated	No. of nodes (sequential)	Optimality index
2	*0A1B2B3A4B5A*	42.09	32	32	1.0
3	*0A1A2B3B4B5A*	42.20	9	32	0.99
4	*0B1B2B3A4A5A*	46.59	6	32	0.90
6	*0B1B2B3B4B5A*	69.69	6	32	0.60

Table-2 Results for a *task graph* with 12 task modules setting *h(n) = 0*

node_count	Optimal mapping	Turn around time *(m sec)*	No. of nodes generated	No. of nodes (sequential)	Optimality index
2	*0B 1A 2B 3B 4A 5A 6B 7B 8B 9B 10A 11A*	74.199	1292	1292	0.841
3	*0B 1B 2A 3A 4B 5A 6A 7B 8B 9B 10B 11A*	62.500	84	1292	0.99
4	*0B 1B 2B 3A 4A 5A 6B 7B 8B 9B 10B 11A*	64.300	20	1292	0.970
6	*0B 1B 2B 3B 4B 5A 6A 7A 8A 9A 10A 11A*	72.699	13	1292	0.858
8	*0B 1B 2B 3B 4B 5B 6B 7A 8A 9A 10A 11A*	95.19	12	1292	0.655
10	*0B 1B 2B 3B 4B 5B 6B 7B 8B 9A 10A 11A*	104.80	12	1292	0.595

Table-3 Test case for task graph of Fig. 1 with 6 task modules with *h (n)≠0*

node_count	Optimal mapping	Turnaround time *(m sec)*	No. of nodes generated	No. of nodes (sequential)	Optimality Index
2	*0B 1A 2A 3B 4A 5A*	48.59	32	32	0.86
3	*0A 1A 2B 3B 4B 5A*	42.20	9	32	0.99
4	*0B 1B 2B 3A 4A 5A*	46.59	6	32	0.90
6	*0B 1B 2B 3B 4B 5A*	69.69	6	32	0.60

Table-4 Test case for task graph of Fig. 2 with 12 task modules with *h (n)≠0*

node_count	Optimal mapping	Turnaround time *(m sec)*	No. Of nodes generated	No. of nodes for sequential	Optimality index
2	*0B 1B 2B 3A 4A 5A 6B 7B 8B 9A 10B 11A*	62.70	1476	1052	1.0
3	*0B 1B 2A 3A 4B 5B 6A 7A 8B 9B 10A 11A*	63.200	100	1052	0.992

4	*0B 1B 2B 3A 4A 5A 6B 7B 8B 9B 10B 11A*	64.300	36	1052	0.975
6	*0B 1B 2B 3B 4B 5A 6A 7A 8A 9A 10A 11A*	72.69	12	1052	0.862
8	*0B 1B 2B 3B 4B 5B 6B 7A 8A 9A 10A 11A*	95.19	12	1052	0.658
10	*0B 1B 2B 3B 4B 5B 6B7B 8B 9A 10A 11A*	104.80	12	1052	0.592

3.2 Inferences from the tables

The results presented in Tables 1 through 4 show that following:-

a) As the value of *node_count* increases, the size of the search state-space reduces.
b) As the value of *node_count* is varied, optimality index also varies. It is maximum at a certain value of the ratio α, where

$$\alpha = \frac{\text{node\_count}}{\text{nos\_ nodes}} \qquad 3.3$$

The variable *nos_nodes* represents the number of task modules. While μ defines the quality of solution reported by the parallel implementation, η defines the efficiency of the parallel implementation in terms of the time required to find optimal solution. From results, it is clear that higher the value of μ, lower the value of η because of the fact that to achieve a higher value of μ, the parallel graph matching tasks must communicate more often, thus reducing the value of η. For each case, the variation of indices μ and η with the ratio α is plotted and the plots are represented in Fig 3a, Fig 3b (test case of Fig 1 with $h(n) = 0$ and $h(n) \neq 0$) , Fig 4a and Fig 4b (test case of Fig 2 with $h(n) = 0$ and $h(n) \neq 0$) . The plots in solid line represent η and plots in dashed lines represent μ . The plots indicate that the variation of μ and η with α follows the same pattern for all the 4 cases. From the plots it is seen that a mapping which is 90% optimal ($\mu \geq 0.9$) is obtained for $\alpha \leq 0.5$ in all cases. The corresponding values of η lies between 0.1 and 0.3. This means that a 90% optimal solution is obtainable at roughly one-third time by the parallel implementation when compared to sequential implementation.

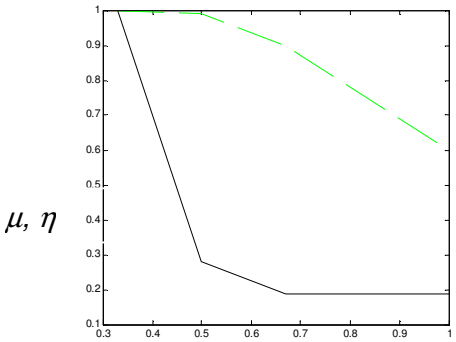

Figure 3a. Variation of μ, η with α, *h(n)=0* with α, *h(n)≠0* for test case 1

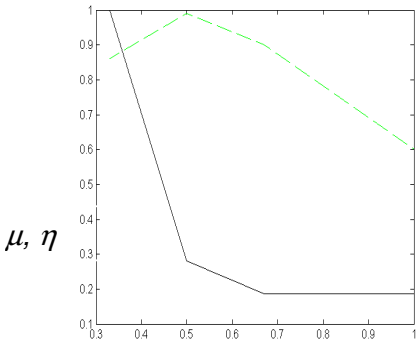

Figure 3b. Variation of μ, η for test case 1

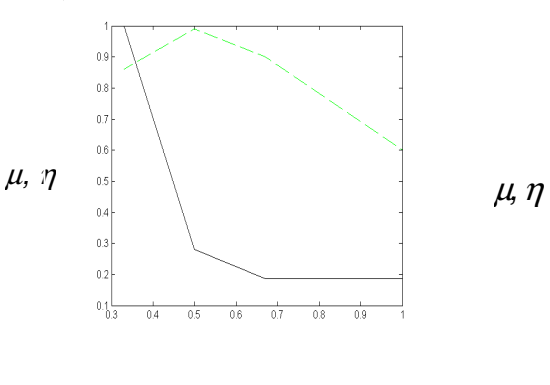

Figure 4a. Variation of μ, η with α, *h(n) ≠0 h(n)≠0* for test case 2

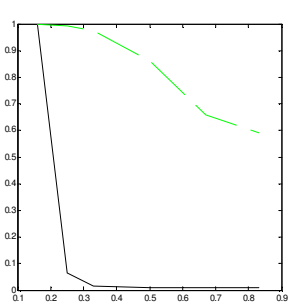

Figure 4b. Variation of μ, η with α, for test case 2

It is further seen that with $h(n) \neq 0$, the value of η reduces much faster as α is increased which means that heuristic search further increases the efficiency of parallel graph matching algorithm. The theoretical value of α has to be matched against the actual value of α supported by the computation and communication speeds.

$$\beta = \frac{\text{actual time in msec for sequential implementation}}{\text{actual time in msec for parallel implementation}} \qquad 3.4$$

Then the actual value of β will depend upon the computation and communication speeds associated with parallel graph-matching tasks. The value of β is determined with $\alpha=0.4$ for graphs represented by Fig 1(test case 1) and Fig 2 (test case 2) with $h(n) = 0$ and $h(n) \neq 0$ and the results are presented in Table5.

Table-5 Actual speed up obtained

Test Case	β
Case 1, Sequential	1.0
Case 1, $h(n) = 0$	26.0
Case 1, $h(n) \neq 0$	19.0
Case 2, Sequential	1.0
Case 2, $h(n) = 0$	1.4
Case 2, $h(n) \neq 0$	1.86

This represents the actual time taken from launch of the parallel job to its completion. Though the size of the state-space reduces drastically, the overheads incurred in communication over the network and *mpirun* to launch and terminate processes actually increase the turnaround time. This implies that the methodology shall be effective for cases with large number of task modules where the speed-up due to reduction is state-space size makes up for these overheads.

4. CONCLUSION

This paper establishes a methodology by which the original heuristic graph matching algorithm can be parallelized so that large practical test case can be handled. The parallel implementation actually uses divide and conquer policy by which the size of the state-space is reduced and hence the complexity. This is because of the fact that each task actually tries to map the tasks on a selected number of processors but not all at the same time. The methodology is especially effective in cases where all processors and inter processor communication links are identical. Thus, this is ideal for further increasing the effectiveness of the methodology proposed by Gao et. al. in [2] to find optimal task mapping on a cluster of SMPs.

5. FURTHER WORK

The following points are identified for further research

- To investigate for multiprocessor (>2) cases.
- To investigate behavior of the parallel implementation for large test cases.
- To investigate the use of a heuristic bound to eliminate expansion of 'non-promising' nodes in the state-space.

6. REFERENCES

[1] Gao, H. Schmidt, A., Gupta, A. and Luksch, P., Load balancing for Spatial-Grid-Based Parallel Numeric Simulation on Clusters of SMPs, Proceeding of the Euromicro PDP2003 conference, February 5-7,2003, Genoa, Italy, IEEE Computer Society Publications, pp. 75-82.

[2] Gao, H. Schmidt, A., Gupta, A. and Luksch, P., A Graph-matching based Intra-node Load balancing Methodology for clusters of SMPs, Proceedings of the 7th World Multiconference on systems, Cubernetics and Informatics (SCI 2003), July 2003.

[3] Chien-chung Shen and Wen-Hsiang Tsai, A Graph Matching Approach to Optimal task assignment in Distributed computing systems using a Minimax Criterion, IEEE Transactions on Computers, vol. C-34,No.3, March 1985.

[4] B.W.Kernighan and S.Lin, An efficient Heuristic procedure for partitioning graphs, Bell Systems Tech, J. 49(1970), pp 291-308.

[5] B.Mohar, The Laplacian spectrum of graphs, Technical Report, Department of mathematics, University of Ljubljana, Ljubljana, Yugoslavia, 1988.

[6] T.N.Bui, S.Chaudhuri, F.T.Leighton, M.Sipser, Graph Bisection Algorithms with good average case behavior, Combinatorica, 7 (1987), pp.171-191.

[7] R.D.Williams, Performance of Dynamic load balancing algorithms for unstructured mesh calculations, Concurrency: Practice and Experience, 3 (1991), pp. 457-481.

[8] C.Farhat, A simple and efficient automatic FEM domain decomposer, Computers and Structures, 28(1988), pp.579-602.

[9] C.M Fiduccia and R.M.Mattheyses, A liner-time heuristic for improve network partitions, ACM IEEE Nineteenth Design Automation Conference Proceedings, vol.1982, ch.126, pp 175-181, 1982.

[10] P.Sadayappan, F.Ercal and J.Ramanujam, Cluster Partitioning approach to mapping parallel program onto a hypercube, Parallel Computing, 13(1990), pp. 1-16.

[11] S. T. BARNARD AND H. D. SIMON, A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes, in: SIAM Proceedings Series 195, D.H. Bailey, P. E. Bjorstad, Jr Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon, J. T. Watson, eds., SIAM, Philadelphia, 1995, pp. 627-632.

[12] S. T. BARNARD, PMRSB: parallel multilevel recursive spectral bisection, Manuscript, 1996.

[13] G. KARYPIS AND V. KUMAR, Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs, Technical Report TR 96-036, Department of Computer Science, University of Minnesota, 1996.

[14] Nils J. Nilsson, Artificial intelligence: a new synthesis, ISBN: 1-55860-467-7, March 1998.