Implementation of Genetic Algorithm on Distributed Memory

Afaf Badie* **Dr. Hilal M. Yousif****

Abstract

This paper focuses on implementing genetic algorithm on distributed memory. This parallel computer can be programmed using MPI standard message passing interface. We use a DAG to model parallel computation. The work is based on priority task graph representation of jobs that contain sequential segments with varying dependencies. We consider compile-time static scheduling when communication overhead is not negligible. DSC is used to cluster tasks and then use a load balancing and physical mapping heuristic to map the clusters onto processors. The main optimization issues are balancing computation among processors, reducing inter-processor communication and overlapping communication with computation. Theoretical and experimental results are presented to verify the performance of these algorithms.

Keywords: Clustering, Directed Acyclic Graph, DSC algorithm, parallel processing, Genetic Algorithm.

* Al-Rafidain College University
** * Al-Rafidain College University

1. Introduction

The multiple instruction stream, multiple data stream (MIMD) architecture is the most popular architecture employed by parallel computers. In MIMD architectures, each processor has its own control unit and local memory unit. They execute independently and transfer information via interconnection networks by using message passing paradigm[10]. Data communication is carried out through messages. Each message consists of a number of fixed-size packets but the length of each message can be varied. Our work builds on the two-phased decomposition of multiprocessor scheduling. In this decomposition, the application graph is first mapped to a fully connected multiprocessor architecture that has an unbounded number of processors. The objective in the mapping onto fully connected processors is the same as the overall objective … minimization of network execution time. In the second phase of two-phase process, called merging, the schedule derived from the first phase is mapped onto the given resource-constrained architecture.

2. Clustering and Scheduling

Clustering is a method to collect small tasks together and put them in the same cluster. The problem of finding the optimal clustering is NP-complete when the objective function is to minimize the parallel time, with unbounded number of processors [2]. Clustering can be divided into two types, linear and nonlinear[16]. In nonlinear clustering, there are 2 or more independent tasks in a cluster. As shown in figure (1.a), nodes n1 and n2 are independent nodes and collected into the same cluster. Thus by using nonlinear clustering, the parallelism of the program is reduced. In linear clustering, dependent tasks are gathered into a cluster, as shown in figure (1.b). Dependent nodes that are in a precedence path of the task graph are gathered into a cluster. In figure (1.b) nodes n0, n1 and n2 are executed sequentially and collected into a cluster.

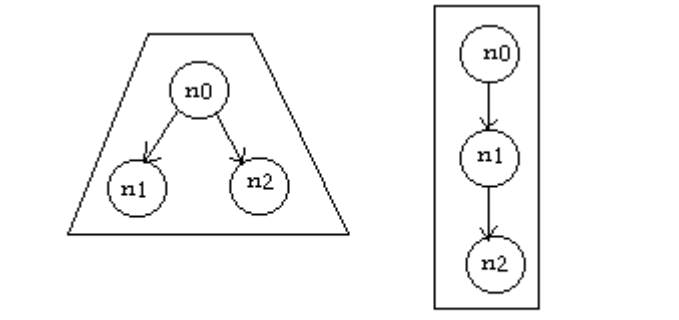

Figure 1: Linear and nonlinear clustering

Tasks are optimally placed on processors of a particular parallel machine by using a scheduling mechanism. As shown in figure (2), given a task graph consisting of 4 tasks and a target machine consisting of 2 processors, a scheduler attempts to obtain minimum total execution time [4]. A scheduling algorithm determines node allocation to processors. Suppose that the scheduler allocates nodes of the critical path, which consists of n0, n1 and n3 onto a processor PE0 and n2 onto PE1. Gantt chart [2] is used to present the time each task spends in executions as well as the processor on which it executes as shown in figure (2).

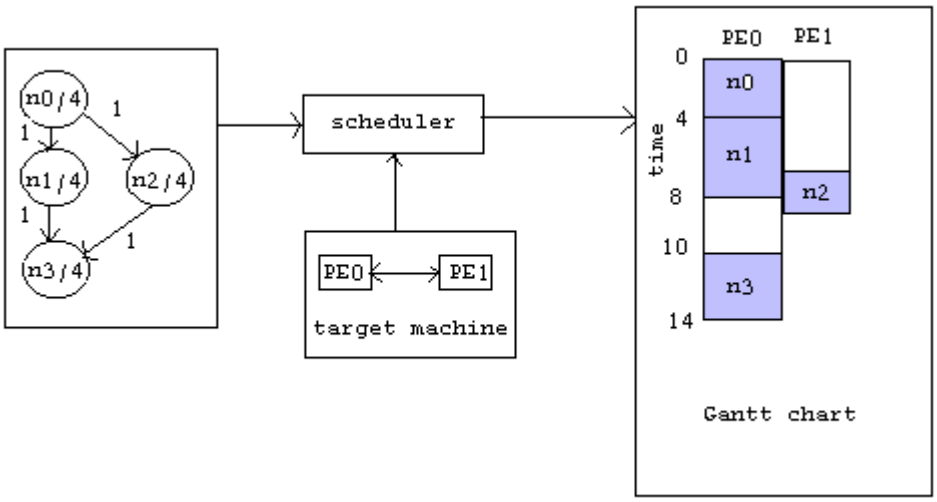

Figure 2: Scheduling process

The execution time of an application consists of computation and communication times. The computation time of an application running on a processor can be reduced by distributing tasks to many processors for parallel execution. The communication time increases when tasks are distributed to many processors because of communication costs. To achieve low execution time, communication cost should be minimized. The parallel execution time is minimal if communication costs are minimized. The communication can be reduced by collecting nodes using linear clustering.

3. Performance

Analyzing performance of multiprocessor system executing an application program is very complex [12,14,16]. Many factors, for instance, grain size of execution or cluster size, the processor speed and the communication bandwidth of the system, affect system performance. Parallelism inherent in an application problem is one of the factors affecting speedup and efficiency. After an application problem is divided into small tasks, they are distributed to be executed on many processors. In clustering schemes, small tasks are collected into clusters and allocated processors. If the size of each cluster is small, clusters are distributed to many processors

with high degree of parallelism. However, overall execution time may increase due to overheads associated with context switching, scheduling time, and communication delays [1,7,9,15]. The total communication overhead among clusters can be reduced by increasing the cluster size by gathering many tasks into a cluster. However, this reduces the degree of parallelism. The overall execution time increases because less number of processors is used and each processor executes many tasks. When tasks are distributed to many processors, the communication overheads are less than those in systems with low communication bandwidth. Then, the number of tasks in each cluster can be less and many clusters can be distributed to many processors. When tasks are distributed to many processors in a network with low communication bandwidth, computation time may be dominated by communication overheads. The computation speed of processors and communication bandwidth hereafter referred to as system parameters, vary from system to system.

4. The DAG Model

A parallel program can be represented by a directed acyclic graph (DAG) *G(V, E)*, where *V* is a set of *v* nodes and *E* is a set of *e* directed edges. A node in the DAG represents a *task*, which in turn is a set of instructions, which must be executed sequentially without preemption (must allow a task to execute until completion on a single processor) in the same processor. The weight of a node n_i is called the *computation cost* and is denoted by $\tau(n_i)$. The edges in the DAG, each of which is denoted by (n_i, n_j), correspond to the communication messages and precedence constraints among the nodes. The weight of an edge is called the *communication cost* of the edge and is denoted by $c(n_i, n_j)$. The source node of an edge is called the *parent node* while the sink node is called the *child* node. A node with no parent is called an *entry* node and a node with no child is called an *exit* node.

The objective of DAG scheduling is to minimize the overall program finish-time by proper allocation of the tasks to the processors and arrangement of execution sequencing of the tasks. Scheduling is done in such a manner that the precedence constraints among the program tasks are preserved. A *critical path (CP)* is defined as a set of nodes and edges, forming the longest path that includes a root node and a leaf node in that graph. The final schedule length is the length of the CP of the scheduled graph (is composed of both clustering and task execution ordering information) [8]. Sometimes this is called the dominant sequence (DS) of the clustered DAG [5]. The *parallel time (PT)* of executing a clustered DAG is not determined by the CP of this DAG but by the CP of the scheduled DAG. A *tlevel* of a node *n* is the length of the longest path from an entry node to *n* excluding the weight of *n*, and *blevel* of *n* is the length of the longest path from node *n* to an exit node.

5. Dominant Sequence Clustering (DSC) Algorithm

DSC by Yang and Gerasoulis [4] is also an edge-zeroing algorithm. The two major ideas behind DSC are to directly attempt to reduce the Dominant Sequence of the graph, and to create an algorithm with low computational complexity, this algorithm keeps track of the dominant sequence to reduce parallel time. The order of complexity of the algorithm is also reduced to $O((v+e)\log v)$ by decreasing the focus range of the steps in the execution. To describe DSC we need to specify certain constraints and definitions. Node types: *scheduled*: A node is scheduled if it has been assigned to a processor. *Free*: A node is free if it is unscheduled and all its predecessors are scheduled. *Partial free*: A node is partial free if it is unscheduled and at least one of its predecessors is unscheduled.

The reduction in the focus range is achieved by maintaining lists of *free* and *partial free* nodes the lists are ordered according to priority as defined below figure (3). At any one iteration in the algorithm the focus is on the nodes on the first elements in these lists and the outgoing edges of the free node in question. This reduces the complexity by confining the range of edges to be zeroed.

1- *EG = 0. UEG = V*. FL = All free entry nodes.
2- Compute *blevel* for each node and set *tlevel = 0* for each free node, compute priority for free and partial free nodes.
3- Every task is marked unexamined and assumed to constitute one unit cluster.
4- **WHILE** $UEG \neq 0$ **DO**
5- In descending order of priority, sort list of free nodes
6- In descending order of priority, sort list of partial free nodes
7- $n_x = head(FL)$; /* the free task with the highest priority *PRIO* */
8- $n_y = head(PFL)$; /*the partial free task with the highest priority ${}_PPRIO$*/
9- **IF** $(PRIO(n_x) \geq {}_PPRIO(n_y))$ **THEN**
10- call the minimization procedure to reduce $tlevel(n_x)$ *and merge* n_x *with the cluster of one of its predecessors.*
11- if no zeroing is accepted(increase $tlevel(n_x)$), n_x remains in a unit cluster
12- **ELSE**
13- call the minimization procedure to reduce $tlevel(n_x)$ under the constraint DSRW *and merge* n_x *with the cluster of one of its predecessors.*
14- if no zeroing is accepted (increase $tlevel(n_x)$), n_x remains in a unit cluster
15- **ENDIF**
16- recalculate the priority values of n_x's successors and find new free and partial free node.
17- Mark n_x as scheduled
18- $UEG = UEG - \{n_x\}$; $EG = EG + \{n_x\}$
19- **ENDWHILE**

Figure 3: DSC algorithm

1- Sort the predecessors of n_x such that

$$tlevel(n_j) + \tau_j + c_{j,x} \geq tlevel(n_{j+1}) + \tau_{j+1} + c_{j+1,x}, \quad j=1: m\text{-}1$$

2- Let *h* be the maximum number of children, node n_t (predecessor) satisfies the following constraint:

If n_t in CLUST(n1), then n_t has one child only
else
begin
if n_t in CLUST(n1), then n_t has more than one child and all children in the same cluster.
else
n_t is not in CLUST(n1)
end
Execlude all children from CLUST(n_1).

3- Find the best stopping point *k* between 1 and *h*. Zero (n_1,n_x) up to (n_k,n_x) so that $tlevel(n_x)$ is minimum

Figure 4: The minimization procedure for DSC

6. Genetic Search Algorithm (GA)

GA is one of a new class of stochastic search algorithms. Stochastic algorithms are those that use probability to help guide their search [11]. One of GA's most important qualities is its ability to evaluate many possible solutions simultaneously. This ability, called implicit parallelism. Pseudo code of GA is:

Create an initial population of strings.
Calculate the fitness of each string.
While an acceptable solution is not found,
    Select parents for the next generation.
    Combine the parents to create new offspring.
    Calculate the fitness of each offspring.
End while.

6.1 Knapsack Problem

The Knapsack problem is a classic NP-complete problem. Given a pile of items that vary in weight and value, find that combination of items having the greatest total value but which does not exceed a maximum weight. In other words, the goal is to fill up a hypothetical Knapsack with the most expensive loot it can carry [6].

6.2 Parallel GA

The parallel GA consists of the following steps:

a) An initial pool of mappings is created by the master processor.
b) The master sends out a portion of the pool to each slave processor, Figure 5.
c) Each slave processor performs a certain number of generation calculations (for example 100 generations) on it's sub-population.
d) Each slave processor finds the best of it's new population and sends it back to the master processor, Figure 6.
e) The master chooses the best of all the mappings sent by all slaves quality.
f) If the number of iterations have met the tolerance without change the best mapping is chosen. If not the best mappings are sent back to the slaves for more iterations, Figure 7.

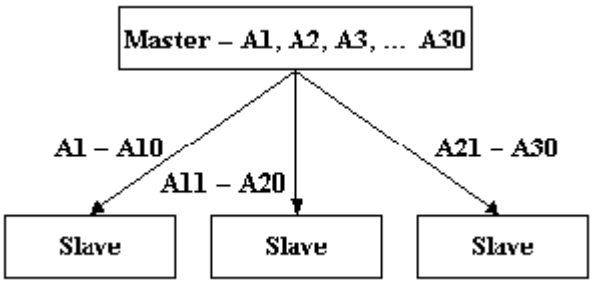

Figure 5: Master broadcasting initial mapping pool

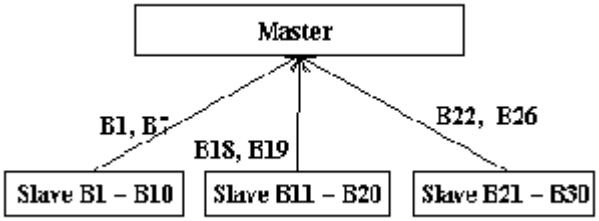

Figure 6: Slaves sending back best of new population

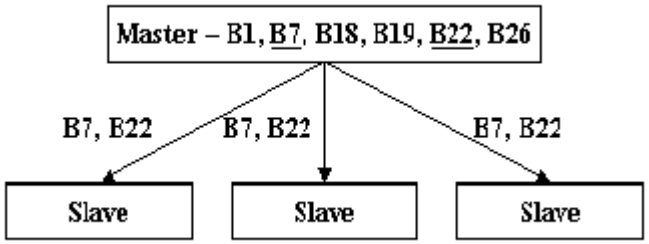

Figure 7: Master chooses best of all slaves and broadcasts

7. Simulation Results

We use standard definitions of speedup and efficiency. *Speedup* is the ratio of the serial and parallel execution times (seconds):

$$Speedup = \frac{serial - elapsed}{parallel - elapsed}$$

Efficiency is the ratio of a given speedup to a linear speedup. A linear speedup occurs when *speedup* = *n*, where *n* is the number of processors:

$$Efficiency = \frac{speedup}{n}$$

The experimental work involves the study of the performance characteristics of problem such as GA. The main objective is to analyze the performance of our algorithms for automatic scheduling to minimize the execution time by overlapping communication with computation and to balance computation among processors.

Table (1) shows the parallel time and efficiency of executing Knapsack problem with string length= 50, population size =200 and number of generation =500. The sequential time is 15.6 seconds.

Table 1: Parallel time of executing Knapsack problem with string length= 50, population size =200 and number of generation =500.

#P	Parallel time	Speedup	Efficiency
2	8.666	1.8	0.9
4	4.457	3.5	0.875
6	3.0	5.2	0.866
8	2.4	6.5	0.812
10	1.95	8	0.80
12	1.642	9.5	0.791
14	1.56	10	0.714
16	1.529	10.2	0.637

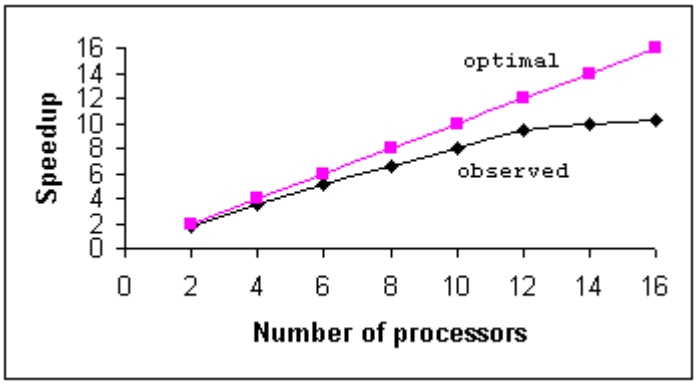

Figure 8: The speedup for knapsack problem with string length= 50, population size =200 and number of generation =500.

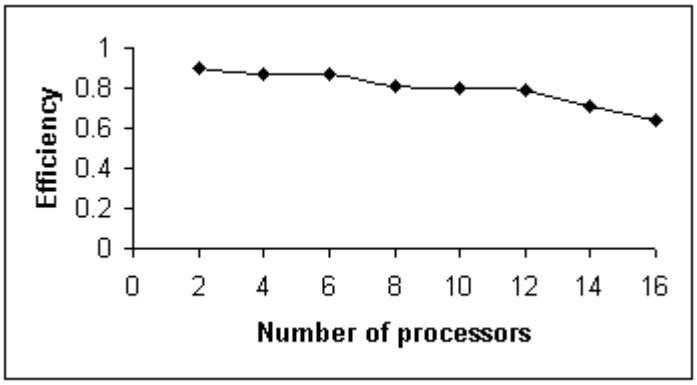

Figure 9: The efficiency for knapsack problem with string length= 50, population size =200 and number of generation =500.

Figs. (8) and (9) show the speedup and efficiency of knapsack problem. We notice that as the number of processors increases, the speedup is increased and the efficiency is decreased. We notice from the Fig. (8) for a very few number of processors, the speedup is nearly linear and close to ideal. With more than 8 processors, the speedup quickly diverges from the ideal. This is due to high communication cost.

8. Implementation Results

The implementation results show that by using the DCS algorithm, tasks are gathered into clusters of different sizes. The application is implemented on a closed network consisting of four processors for experimental studies. Each of them runs Windows 2000 as the operating system and supports the MPICH parallel processing software-using message passing interface (MPI).

In our experiments, the parallel system comprises four PCs running Windows 2000 operating system connected as a dedicated cluster. To support parallel processing, software called MPICH is used. MPICH is an MPI programming environment and development system for heterogeneous computers on a network [17]. It features a full implementation of Message Passing Interface (MPI) programming standard (some commands of MPI are found in appendix A). It is also a parallel processing environment and development system for a network of independent computers, supporting monitoring and debugging. MPICH runs as a windows 2000 daemon on each computer, which is structured as a nano-kernal and hand-threaded virtual process. MPICH commands can be divided into three parts, setting up a system, executing, monitoring. Before running parallel application software, the MPICH daemon is booted by using MPICH command. The system can concurrently execute one or more programs.

8.1 Knapsack Problem

When we execute the Knapsack problem with different problem size (50,100,200) we obtain the following results as shown in table (2):

Table 2: Execution time of knapsack problem.

Problem size	Execution time in seconds		
	P=2	P=3	P=4
50	0.533	0.144	0.079
100	0.829	0.256	0.100
200	2.013	1.722	1.203

We notice from the above results that the execution time is increased as the problem size increases and it is decreased as the number of processors increases.

9. Conclusions

Our algorithm takes advantage of such idle cycles by scheduling all processors as full as possible to reduce parallel time. It is often the case that a job initiated from a workstation can be split into tasks which are sent to idle workstations on the network. Results are then sent back to the starting machine and merged to form the final solution. We require that the first and last tasks be located on the main processor.

The experimental results clearly demonstrate the effectiveness of the two-phase method of DSC. There is never a need to use more processors (even though they are available) than the number of clusters produced by the clustering algorithm doing so would only increase the parallel execution time. The low complexity makes DSC very attractive in practice. The results show parallel execution times using 4 processors. When using 4 processors, the execution time is less than that using 3 processors. When the problem size is large many processors must be used for distributing the large number of clusters. Linear speedup generally does not occur in multicomputer systems because of communication overhead. Whenever the number of processors increases the interprocessor communication cost also increases. In order to achieve greater speedups in such systems, the communication cost should be taken into consideration in making scheduling decisions

10. References

1. Bhujade, R.M. (1995). *Parallel Computing*. Department of Computer Science and Engineering. India Institute of Technology, Bombay.
2. El-Rewini, H. (1994). *Task Scheduling in Parallel and Distributed Systems*. New Jersey: PTR Prentice Hall.
3. El-Rewini, H. (1992). *Introductionask to Parallel Computing*. New Jersey: USA: Prentice Hall.
4. Gerasoulis, A. and Yang, T. (1992). *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors*. Journal or Parallel and Distributed Computing, pp. 276-280.
5. Gerasoulis, A. and Yang, T. (1990). *Clustering Task Graphs for Message Passing Architectures*. Department of Computer Science, Rutgers University.
6. Grant, K. (1995). *An Introduction to Genetic Algorithms*. C/C++ Users Journal.
7. Jeannot, E. and Wanger, F. (2004). *Message Scheduling for Data Redistribution through High Performance Network*. France.
8. Koziris, N. and Romesis, M. (1997). *An Efficient Algorithm for the Physical Mapping of Clustered Task Graphs onto Multiprocessor Architectures*. Department of Electrical and Computer Engineering, University of Athens.
9. Miranda, E.R. (1998). *Parallel Computing for Musicians*. SONY CSL- Paris.
10. Moldvan, D.I. (1993). *Parallel Processing: From Application to Systems*. California:Morgan Kaufmann Publishess.
11. Raouf Kh. and Aliaksei V. (2001). *Algorithm for Optimization of Parallel Computations on the Bases of Genetic Algorithms and Model of a Virtual Network*. Byelorussian State University of Informatics and Radioelectronics.
12. Senar, M.A. and Ripoll, A. (2003). *Clustering and Reassignment-based Mapping Strategy for Message-Passing Architectures*. Department of Informatics, University of Barcelona.
13. Shirazi, B. (1995). *Scheduling and Load Balancing in Parallel and Distributed Systems*. California: IEEE Computer Society Press.
14. Stone, H. (1987). *High-Performance Computer Architectures*. Reading, Mass.:Addison Wesley.
15. Wilson, Gregory. (2000). *History of the Development of Parallel Computing*.
    http://ei.cs.vt.edu/~history/parallel.html
16. Yang, T. and Gerasoulis, A. (1990). *On the Granularity and Clustering of Directed Acyclic Task Graph*. Department of Computer Science, Rutgers University.
17. Argonne National Laboratory.
    www.mcs.anl.gov/mpi

تنفيذ الخوارزمية الجينية على الذاكرات الموزعة

عفاف بديع جميل* أ. د. هلال محمد يوسف**

الخلاصة

يركز هذا البحث على تنفيذ الخوارزمية الجينية على الحواسيب ذات الذاكرات الموزعة. هذه الحواسيب يمكن برمجتها باستخدام واجهة عبور الرسائل (MPI). تم استخدام نموذج DAG في الحسابات المتوازية. هذا العمل مبني على أسبقية تمثيل المهام وقد تم اعتماد جدولة وقت الترجمة الثابت. تم استخدام خوارزمية DSC في عملية تجميع المهام (Clustering) . الغرض الأساسي هو موازنة الحسابات بين المعالجات، تقليل الاتصال الداخلي للمعالجات وتداخل الاتصال مع الحساب. تم عرض نتائج نظرية وعملية للتحقق من كفاءة أداء هذه الخوارزمية.

* كلية الرفدين الجامعة.
* * كلية الرفدين الجامعة.