Optimizing Task Schedules Using An Artificial Immune System Approach

Han Yu
Physical and Digital Realization Research Center, Morotola Labs
1301 East Algonquin Road Room 1014
Schaumburg, Illinois 60196, USA
a37377@motorola.com

ABSTRACT

Multiprocessor task scheduling is a widely studied optimization problem in the field of parallel computing. Many heuristic-based approaches have been applied to finding schedules that minimize the execution time of computing tasks on parallel processors. In this paper, we design an algorithm based on Artificial Immune Systems (AIS) to scheduling for heterogeneous computing environments. This approach distinguishes itself from many existing approaches in two aspects. First, it restricts the use of AIS to find optimal task-processor mapping, while taking advantage of heuristics used by deterministic scheduling approaches for task sequence assignment. Second, the calculation of the affinity takes into account both the solution quality and the distribution of population in the solution space. Empirical studies on benchmark task graphs show that this algorithm significantly outperforms HEFT, a deterministic algorithm. Further experiments also indicate that the algorithm is able to maintain high quality search even though a wide range of parameter settings are used.

Categories and Subject Descriptors: H.4 [Information Systems Applications]: Miscellaneous

General Terms: Algorithms

Keywords: Artificial Immune Systems, Parallel Computing, Task Scheduling

1. MULTIPROCESSOR TASK SCHEDULING

Scheduling a group of computing tasks on parallel processors is an intensively studied problem in parallel computing. By decomposing a computation into smaller tasks and then executing these tasks on multiple processors, we can potentially reduce the total execution time of the computation.

The problem of multiprocessor task scheduling is typically given by two inputs: a group of dependent computing tasks and a group of interconnected processors. Tasks are dependent if the execution of some tasks relies on the execution of other tasks within the group. The data dependency and the execution procedure among tasks can be described with a directed acyclic graph (DAG). The goal of scheduling is to minimize the total execution time of tasks, also known as *makespan*, by assigning the execution of the tasks to the processors. Many variations of scheduling problems exist. They vary by different assumptions on either the interconnection and processing ability of processors or the permission on task duplication. Traditional scheduling problems assume a homogeneous computing environment in which all processors have the same processing abilities and they are fully connected. Recent studies have been diverted to scheduling for heterogeneous computing environments in which the execution time of a task may vary among different processors, not all processors are directly connected, and the bandwidth of communication links connecting each pair of processors may also be different. In addition, some scheduling problems allow the same task to be executed on multiple processors (as doing so may reduce the makespan in some cases), while other problems restrict a task to be executed on only one processor.

In this paper, we focus on the study of scheduling for heterogeneous computing environments. We assume that processors are fully connected with the same communication links (i.e., with the same bandwidths), but they have different processing abilities. In addition, a task are not allowed to be executed on more than one processor. As tasks may be dependent, the communication time between two dependent tasks (i.e., the data transfer time between tasks) should be taken into account if they are assigned to different processors. We also assume a static computing model in which the dependence relations and the execution times of tasks are known a priori and do not change over the course of scheduling and task execution. In addition, all processors are fully available to the computation on the time slots they are assigned.

Figure 1 shows an example DAG that contains ten tasks, t_1 to t_{10}. The arrows represent data dependencies among tasks. The numbers represent the communication times needed to transfer data between two dependent tasks. Table 1 lists the execution times of each task on each of three processors, P_1, P_2, and P_3. Figure 2 shows an execution schedule of tasks on these three processors with a makespan of 153.

For a pair of dependent tasks, t_i and t_j, if the execution of t_j depends on the output from the execution of t_i, then t_i

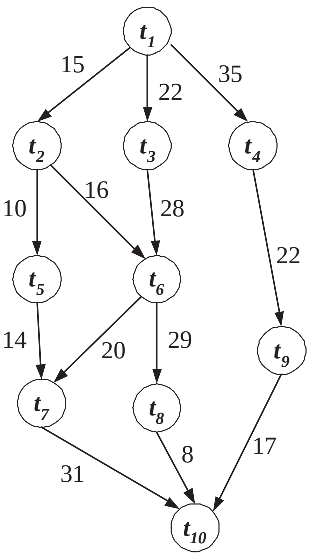

Figure 1: An example DAG containing ten tasks.

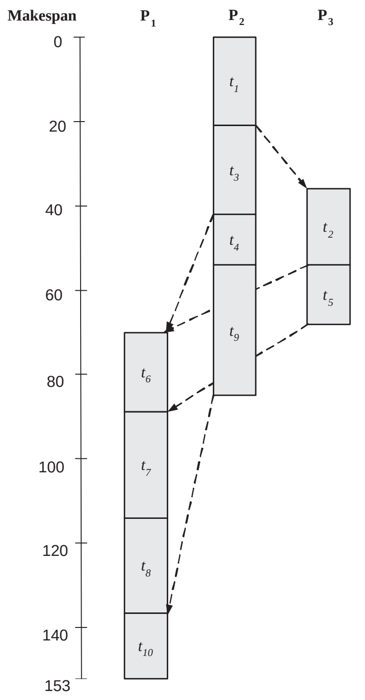

Figure 2: A schedule for the task graph in Figure 1 on three processors. The makespan of the schedule is 153.

Table 1: The computation times of ten tasks in Figure 1 on three processors, P_1, P_2, and P_3.

	P_1	P_2	P_3
t_1	27	21	30
t_2	19	15	18
t_3	35	21	27
t_4	17	12	26
t_5	11	18	14
t_6	19	32	28
t_7	26	15	31
t_8	22	22	20
t_9	22	31	22
t_{10}	16	19	24

is the predecessor of t_j, and t_j is the successor of t_i. We use $pred(t)$ and $succ(t)$ to denote the set of predecessor tasks and successor tasks of task t, respectively.

2. RELATED WORK

The search for an optimal solution to the problem of multiprocessor scheduling has been proven to be NP-hard except for some special cases [6]. Numerous approaches have been developed to solve the problem. These approaches can be mainly classified into two categories: deterministic approaches and non-deterministic approaches.

Deterministic approaches attempt to exploit the heuristics extracted from the nature of the problem in guiding the search for a solution. They are efficient algorithms as the search is narrowed down to a very small portion of the solution space; however, the performance of these algorithms is heavily dependent on the effectiveness of the heuristics. Therefore, they are not likely to produce consistent results on a wide range of problems.

Of all the deterministic approaches, many of them belong to *list scheduling algorithms*. The search in list scheduling algorithms is divided into two phases: in the first phase, a priority value is given to each task according to some criteria; in the second phase, tasks are assigned to processors in decreasing order of their priorities. ISH [11], DSH [11], MCP [22], and CPFD [2] are typical list scheduling approaches to homogeneous computing systems, while HEFT [17] and CPOP [17] are list scheduling algorithms designed for heterogeneous computing systems. A drawback of list scheduling algorithms is that the static assignment of task priority is not able to capture the dynamics in task execution because less important tasks may be scheduled earlier, causing the delay of execution of more important tasks. Dynamic approaches, such as DCP [12], attempt to overcome this problem by overlapping the phases of task order assignment and task scheduling.

Another group of deterministic algorithms is *clustering algorithms* [10, 23]. These algorithms assume that there are an unlimited number of processors available to task execution. Clustering algorithms will use as many processors as possible in order to reduce the makespan of the schedule. If, however, the number of processors used by a schedule is more than the number actually available in a given problem, a mapping process is required to merge the tasks in the proposed schedule onto the actual number of available processors.

Contrary to deterministic algorithms, non-deterministic algorithms incorporate a combinatoric process in the search for solutions. Non-deterministic algorithms typically require sufficient sampling of candidate solutions in the search space and have shown robust performance on a variety of scheduling problems. Genetic algorithms [8, 13, 19, 18, 21], simulated annealing [4, 9, 15], tabu search [16], and artificial immune systems [14, 7, 5, 20] have been successfully applied to various scheduling problems. Non-deterministic algorithms, however, are less efficient and have much higher computational cost than deterministic algorithms.

There are also reported studies on incorporating both non-deterministic and deterministic algorithms for multiprocessor scheduling. Ahmad and Dhodhi's approach [1] uses genetic algorithms for evolving priorities of tasks and a list scheduling algorithm for mapping tasks to processors in a homogeneous computing environment. Boeres et al. [3] use a similar hybrid approach for finding optimal schedules for heterogeneous computing environment. Genetic algorithms are used for task priority determination, while various heuristics are used to produce schedules based on priorities.

3. ALGORITHM DESIGN

The design goal of our algorithm is to exploit the advantages of both non-deterministic algorithms and deterministic algorithms while avoiding their drawbacks. Our algorithm differs from previous hybrid approaches in that we use an AIS-based algorithm to perform task-processor mappings rather than evolving task priorities. The search for an optimal solution, therefore, is divided into two phases. In the first phase, AIS is used to map each task to one of the processors. After a mapping is found, a heuristic approach is then employed to determine the order of tasks assigned to the same processor.

3.1 An AIS Algorithm for Task Mapping

An algorithm based on AIS is used to perform task mapping, i.e., to assign the execution of each task to one of the available processors. The following sections gives detail description of the AIS algorithm.

3.1.1 Encoding of Antibodies

The population consists of a group of antibodies and each antibody represents a candidate solution to a given scheduling problem. Solutions are encoded as strings of integers, and each integer represents the processor to which a task is assigned. Suppose there are n tasks and m available processors. Each solution contains n integers, and the value of each integer in the solution ranges between 1 and m. Let us assume that each task is assigned a unique id (from task t_1 to t_n). The ith integer represents the processor to which task t_i is assigned. The search space of the algorithm, therefore, is m^n. Figure 3 shows the encoding of a solution corresponding to the schedule in Figure 2.

2	3	2	2	3	1	1	1	2	1

Figure 3: The string of integers that encodes the schedule shown in Figure 2. Each integer in the string represents the processor to which a task is assigned.

3.1.2 Calculating the Affinity of Antibodies

Maintaining a delicate tradeoff between the exploration and exploitation of search space is always an important consideration in designing any evolutionary algorithms. In our algorithm, we attempt to address the issue in the calculation of the affinity of antibodies, which consists of two components. The first component, called *fitness affinity* or F_a, evaluates the quality of encoded solutions. As in a typical optimization problem, we do not have an optimal solution that we can refer to as antigen in calculating the affinity. Instead, we compare the makespan of a solution with the makespan of the best solution in the current population. A solution that generates shorter makespan is preferred and given high fitness affinity. We calculate the fitness affinity of an antibody i with Equation 1.

$$F_a(i) = \frac{1}{(makspan(i) - b_{makespan} + 1)} \tag{1}$$

, where $makespan(i)$ is the makespan of the schedule that antibody i encodes, and $b_{makespan}$ is the makespan of the best solution found in the population. The values of fitness affinity ranges between 0 and 1. The shorter the makespan, the higher the fitness affinity. The best solution in the population receives a fitness affinity of 1.

The second component of the affinity, called *adjacency affinity* or A_a, encourages the algorithm to explore new solution spaces and maintain population diversity. To calculate the adjacency affinity of an antibody, the schedule encoded in the antibody is compared with those encoded in the other antibodies in the population. We calculate the average distance between the schedule and the rest of the population in the solution space. A longer average distance results in higher adjacency affinity. To save the computational cost on pairwise distance calculation, we randomly select a portion of the population to compute the adjacency affinity instead of using the whole population. The proportion of random antibodies to be selected from the population is given by a parameter called *sampling rate*. Therefore, the adjacency affinity of an antibody i can be calculated using Equation 2.

$$A_a(i) = \frac{\sum_{j=1}^{N_s} dist(i,j)}{N_s \times N_t} \quad (2)$$

, where N_s and N_t represent the number of sampling antibodies and the number of tasks in a solution, respectively, and $dist(i,j)$ denotes the distance between antibody i and the jth selected antibody for comparison. The distance between two antibodies is given by the number of tasks that are assigned to different processors in the two correspondingly encoded schedules.

The overall affinity of an antibody, then, is the weighted sum of its fitness affinity and adjacency affinity. We assign the weight of each component based on the average fitness and adjacency affinities in a population, using Equation 3.

$$Affinity(i) = F_a(i) \times Average(A_a) + A_a(i) \times Average(F_a) \quad (3)$$

There are two advantages of using this method to combine the two components of affinity. First, it avoids the human factors in determining the weight of each components, a process in many cases resulting in arbitrary decisions. Second, as the method takes into account the relative values of fitness and adjacency affinities, it is applicable to all scheduling problems without reassigning weights for different scheduling problems. Although the value of adjacency affinity is relatively stable, the fitness affinity that an antibody receives is largely dependent on the optimal makespan that a solution can reach. For instance, a scheduling problem whose solution has longer makespan typically receives lower fitness affinity. In this case, our method, observing the relatively lowered fitness affinity, is able to assign a higher weight on the fitness affinity and automatically rebalance these two components.

3.1.3 Selection

Selection is performed in every generation based on the affinity of antibodies. The expected number of times an antibody being selected is proportional to its affinity. The selected antibodies replace the existing population and form the next generation of population.

3.1.4 Mutation

Mutation is performed on all selected antibodies. All tasks in a solution have equal probability of being mutated. When a task is selected for mutation, we reassign the task to a random processor.

3.2 A Heuristic-based Algorithm for Task Order Assignment

After tasks are mapped to processors, we need to determine the order of tasks to be executed on each processor and calculate the makespan of the schedule. We apply a heuristic-based algorithm for task order assignment. Tasks are assigned in the execution queue one by one, and in each step, we choose one of the tasks that are ready for execution. A task is *ready* if it has no predecessor or all its predecessor tasks are already scheduled. Among all ready tasks, the priority is given to the one whose execution is critical to potentially reduce the makespan of a schedule.

Determining the makespan of partially scheduled tasks is important when we choose a ready task to be executed in each step. The calculation of makspan, if a given task is chosen to be executed, should take into account both the completion time of the task (i.e., the duration of time needed from the beginning of the computation until the task finishes execution) and the execution time of all tasks that depend on the execution of this task. The completion time of a task can be calculated, but the latter aspect cannot be accurately determined as the execution order of successor tasks is not known yet (although we have already known to which processors they are mapped). We use the notion of upward rank to give an estimation. The upward rank of a task is used in HEFT algorithm to determine the priority of task assignment [17]. While our purpose of using upward rank is the same as in HEFT, we use a different method in calculating the upward rank. First, in HEFT, the average execution times of tasks are used in the calculation. As in our algorithm the task-processor mapping is performed before the task sequence assignment, we can use the execution time of the tasks on their mapped processors in the calculation. Second, the communication times between dependent tasks are always counted in HEFT, even though some of them may potentially be executed on the same processor. We, instead, can use the task mapping results and ignore the communication times for dependent tasks executed on the same processor. Therefore, our method is able to produce more accurate estimation on the execution time of the unassigned tasks. The upward rank of a task t, $u_r(t)$ is calculated recursively using Equation 4.

$$u_r(t) = \begin{cases} 0 & \text{if } succ(t) = \phi \\ max(comp(t_j, proc(t_j))+ & \\ real\_comm(t,t_j)), \forall t_j \in succ(t) & \text{otherwise} \end{cases} \quad (4)$$

where $comp(t,p)$ denotes the computation time of task t on processor p, $proc(t)$ denotes the processor to which task t is mapped, and $real\_comm(t_1,t_2)$ denotes the communication time between two tasks during task execution and can be calculated using Equation 5:

$$real\_comm(t_1,t_2) = \begin{cases} 0 & \text{if } proc(t_1) = proc(t_2) \\ comm(t_1,t_2) & \text{otherwise} \end{cases} \quad (5)$$

Using the above method for estimating the total execution time of partially scheduled tasks, we always choose a task whose sum of completion time and upward rank is the largest among all ready tasks, as we believe that assigning the execution of this task earlier will help to reduce the makespan. The selected task is then appended to the execution task queue of the processor and removed from S_{ready}. Any tasks that become ready due to the execution of the task are added to S_{ready}. We repeat the above steps until all tasks are scheduled. After that, we calculate the makespan of the schedule which in turn determines the fitness affinity of the antibody.

3.3 Procedure of Algorithm

Combining both the AIS algorithm and heuristic-based algorithm, we show the procedure of the our algorithm as follows:

```
a) initialize a population of antibodies
b) for each generation, do
   b.1) for each antibody, do
      b.1.1) map tasks to processors according to the
```

```
                encoded solution
       b.1.2) determine the order of task execution
       b.1.3) calculate the makespan of the schedule
       b.1.4) calculate the fitness affinity based on the
                makespan
       b.1.5) calculate the adjacency affinity and
                overall affinity
    b.2) perform selection to produce antibodies for the
         next generation
    b.3) perform mutation on selected antibodies
c) select the best solution in the final generation as
   the solution of the algorithm
```

The detailed procedure of determining the order of task execution (step b.1.2) is:

```
a) calculate the upward rank of each task based on the
   task mapping result encoded in the antibody
b) initialize the set of ready tasks that includes tasks
   with no predecessors
c) while the set of ready tasks is not empty, do
   c.1) for each ready task, do
        c.1.1) calculate the completion time of the task
               on the assigned processor
        c.1.2) calculate the sum of its completion time
               and upward rank
   c.2) select the task whose sum is the largest among
        all ready tasks and assign the task to the
        processor
   c.3) update the set of ready tasks by removing the
        scheduled task and adding new ready tasks
```

4. EXPERIMENTS

4.1 Test Bed

We use the 21-task and 28-task Gauss-Jordan graphs to evaluate the performance of our algorithm. Figure 4 shows the DAG for the 21-task Gauss-Jordan Graph. The DAG for the 28-task graph has additional seven tasks on above of the top layer of the 21-task graph. The average computation time of each task in both graphs is 40. For each graph, we test cases with different communication to computation ratios, ranging between 0.25 and 3. For each case, the communication time between each pair of dependent tasks are the same. For instance, if the communication to computation ratio is 0.25, the data transfer time between dependent tasks, if they are assigned to different processors, is 10.

4.2 Parameter Settings and Metrics for Performance Evaluation

For each test case, we randomly generate ten task graphs. The computation time of each task on each processor varies among the ten graphs and it follows Poisson distributions. We run the algorithm fifty times for each task graph. In each run, we calculate the speedup of the schedule using the following equation:

$$speedup = \frac{serial\_execution\_time}{makespan} \tag{6}$$

where *serial_execution_time* is the shortest makespan that can be achieved if we schedule the execution of all tasks on any single processor. The higher the speedup, the more effective the distribution of task execution on parallel processors.

For each task graph, we calculate both the average speedup of solutions over fifty runs with 95% confidence interval (CI)

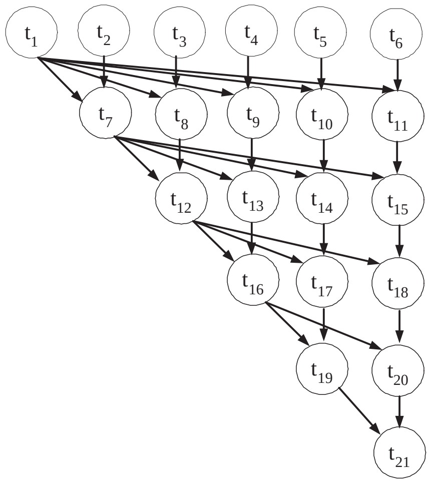

Figure 4: The DAG for 21-task Gauss-Jordan graph. The average communication time of tasks is 40. The communication to computation ratios of task graphs used in our experiments range between 0.25 and 3.

and the highest speedup achieved in fifty runs. Then we calculate the average value of these results for the ten task graphs in each test case.

Unless otherwise specified, we use the parameter settings shown in Table 2 in all runs.

Table 2: Parameter settings used in the experiment.

Parameter	Value
Population Size	200
Number of Generations	200
Sampling Rate	10%
Mutation Rate	0.1
Selection	Proportional to Affinity

The experimental results are compared with the performance of HEFT on the same task graphs. HEFT is a list scheduling algorithm and the priority of tasks is give by their upward ranks. As a deterministic algorithm, HEFT is run only once for each task graph. We calculate the average speedup of the solutions to the ten task graphs in each test case.

4.3 Experimental Results and Analysis

Figures 5 and 6 show the comparison of the results between our algorithm and HEFT on 21-task and 28-task Gauss-Jordan graphs. Our algorithm consistently outperforms HEFT, producing schedules with average speedups higher than those produced by HEFT in all test cases. The experiment also shows that as the communication to computation ratio of task graphs increases, the speedup of execution on parallel processors drops due to longer data transfer times among dependent tasks. We also observe that the gap in the speedups reached by the two algorithms are more noticeable in task graphs with higher communication to computation ratios (e.g., ratio of 3.0). We believe that to schedule a task graph with a high communication to computation ratio, proper task-to-processor mapping is essential to avoid or reduce long data transfer time. The use of AIS for task mapping enables the our algorithm to explore a larger solution space than HEFT, so it is more likely to find better mappings.

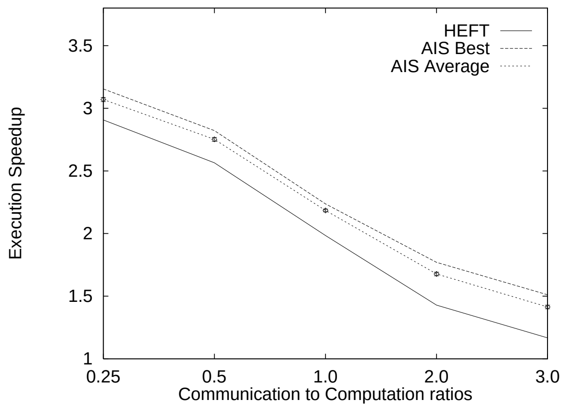

Figure 5: The performance comparison between the AIS algorithm and HEFT on 21-task Gauss-Jordan graphs with varying communication to computation ratios. The results for AIS runs are given by the best speedup and the average speedup with 95% confidence intervals from 50 runs in each task graph.

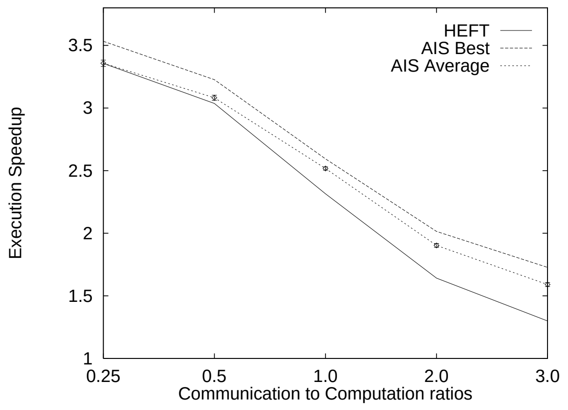

Figure 6: The performance comparison between the AIS algorithm and HEFT on 28-task Gauss-Jordan graphs with varying communication to computation ratios. The results for AIS runs are given by the best speedup and the average speedup with 95% confidence intervals from 50 runs in each task graph.

One of the unique features of our algorithm is that we separate the phases of task-processor mapping and task sequence determination. As the task mapping results are available before task sequence determination begins, we are able to provide more accurate estimation on the execution time needed for the unassigned tasks using a revised method of calculating the upward rank. We performed additional experiments using the traditional method for calculating the upward rank (i.e., the method used in HEFT). The comparison of results between the two methods are shown in Figures 7 and Figure 8.

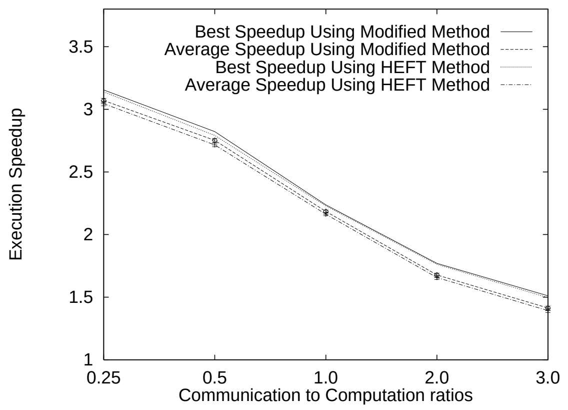

Figure 7: The performance comparison of our algorithm using two methods of calculating the upward ranks of tasks: the modified method that takes into account the task mapping results and the method used in HEFT. Results are given by the execution speedups achieved on 21-task Gauss-Jordan graphs with varying communication to computation ratios.

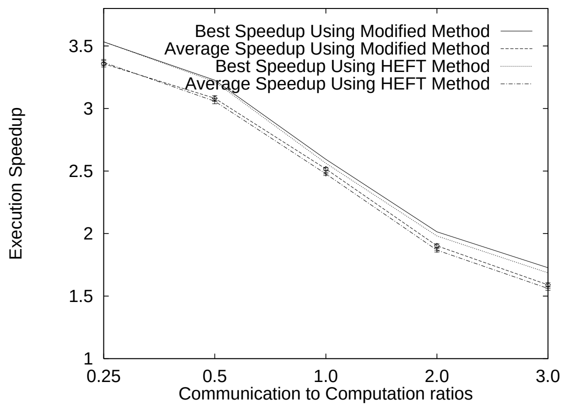

Figure 8: The performance comparison of our algorithm using two methods of calculating the upward ranks of tasks: the modified method that takes into account the task mapping results and the method used in HEFT. Results are given by the execution speedups achieved on 28-task Gauss-Jordan graphs with varying communication to computation ratios.

Figures 7 and 8 show that the revised method helps the algorithm find better solutions in all test cases. The improved accuracy in estimating the task execution times provided by the revised method enables the algorithm to produce better task execution sequences that take into account the task mapping results, so it can potentially further reduce the makespan of task execution.

We also study the effect of two parameters, the mutation rate and sampling rate, on the performance of the algorithm. We rerun the algorithm on all task graphs with varying mutation rates from 0.01 to 0.20, and sampling rates from 10% to 100%. The results show very consistent behaviors in all test cases. Due to page limitation, we only report the results on the 21-task graphs with communication to computation ratio of 1. Figures 9 and 10 show these results.

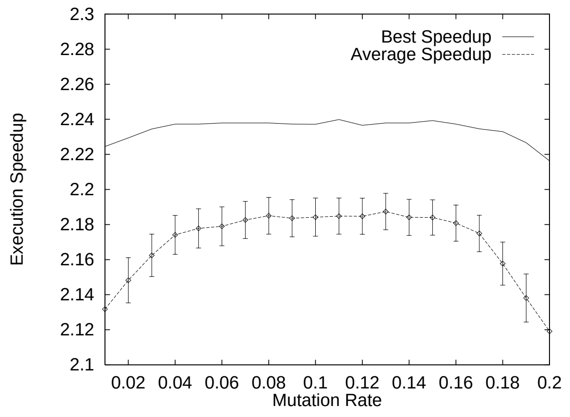

Figure 9: The performance comparison of our algorithm using varying mutation rates from 0.01 to 0.20. Results are collected from runs on 21-task Gauss-Jordan graphs with communication to computation ratio of 1.

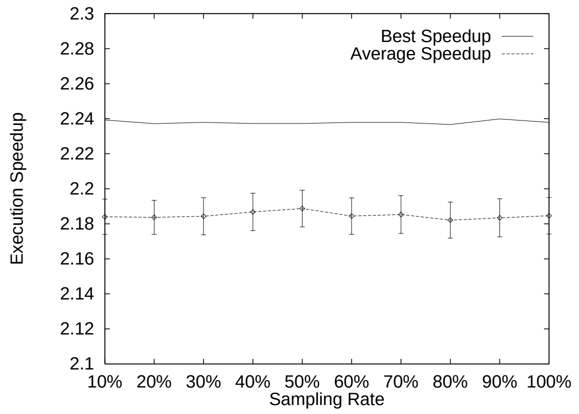

Figure 10: The performance comparison of our algorithm using varying sampling rates from 10% to 100%. Results are collected from runs on 21-task Gauss-Jordan graphs with communication to computation ratio of 1.

Figures 9 indicates that the algorithm with mutation rate between 0.07 and 0.15 produces the best search performance, with the average speedup reaching above 2.18. The search performance gradually degrades with mutation rate lower than 0.06 or higher than 0.15 but not significantly. A mutation rate of 2.0, which exhibits the lowest performance within the range of mutation rates in our experiment, still produces an average speedup of around 2.12, much better than HEFT (with speedup less than 2.0).

Figure 10 shows that the sampling rate does not have much impact on the search quality of the algorithm. The execution speedups fluctuate very slightly with varying sampling rates. The computation time of the algorithm, however, increases significantly when a higher sampling rate is used. Figure 11 shows the increase in the average computation times in runs with higher sampling rates over runs with a 10% sampling rate. The computation time increases linearly with the sampling rate. Running the algorithm with full sampling (i.e., 100% sampling rate) requires more than three times of computation cost than running with a 10% sampling rate, without significant performance improvement. Therefore, a partial population sampling (as low as 10%) appears to be sufficient to guarantee search quality.

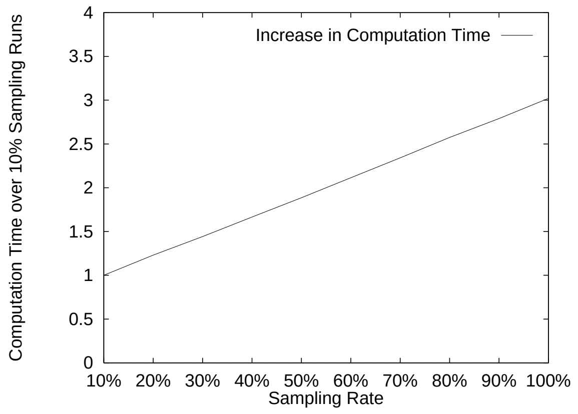

Figure 11: The increase in average computation times in runs with higher sampling rates over runs with 10% sampling rate.

5. CONCLUSIONS

This paper presents an AIS-based algorithm for finding optimal execution schedules for tasks running on heterogeneous computing processors. Two unique features distinguish our algorithm from many other scheduling algorithms. First, we separate the phases of task mapping and task sequence determination. We restrict the use of AIS to task mapping while using a heuristic from deterministic approaches to assigning the order of task execution. As a result, this algorithm effectively reduces the solution space to allow AIS a more sufficient sampling during the search. Also, the separation of the two phases enables the algorithm to find more efficient schedules by assigning task execution order based on the task mapping result. Second, the affinity of an antibody in AIS is determined by both the quality of encoded solution and its phenotype distance to the rest of the population. This affinity function enables the AIS to maintain a dynamic balance between exploration and exploitation during the search for an optimal solution.

The experiment shows that our algorithm consistently outperforms HEFT, a deterministic algorithm, finding schedules with higher execution speedup. Additional experiment also reveals that the use of task mapping results for task sequence assignment is able to improve the search result. The study on the parameters indicates that the algorithm is able to maintain high quality search within a wide range of mutation rates and sampling rates. A low sampling rate of 10% used in the calculation of affinity is sufficient to guarantee the search quality without the need for a complete population sampling.

6. REFERENCES

[1] I. Ahmad and M. K. Dhodhi. Multiprocessor scheduling in a genetic paradigm. *Parallel Computing*, 22:395–406, 1996.

[2] I. Ahmad and Y. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, 1998.

[3] C. Boeres, E. Rios, and L. S. Ochi. Hybrid evolutionary static scheduling for heterogeneous systems. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1929–1935, 2005.

[4] S. W. Bollinger and S. F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *Proceedings of the International Conference on Parallel Processing*, pages 1–7, 1988.

[5] A. Costa, P. Vargas, F. V. Zuben, and P. Franca. Makespan minimisation on parallel processors: An immune based approach. In *Proceedings of the Congress on Evolutionary Computation*, pages 920–926, 2002.

[6] M. R. Garey and D. S. Johnson. *Computers and intractability, a guide to the theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[7] E. Hart and P. Ross. An immune system approach to scheduling in changing environments. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 1559–1565, 1999.

[8] E. S. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, 1994.

[9] K. Hwang and J. Xu. Mapping partitioned program modules onto multicomputer nodes using simulated annealing. In *Proceedings of the International Conference on Parallel Processing*, pages 292–293, 1990.

[10] S. J. Kim and J. C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *International Conference on Parallel Processing*, volume 2, pages 1–8, 1988.

[11] B. Kruatrachue and T. G. Lewis. Duplication Scheduling Heuristic, a new precedence task scheduler for parallel systems. Technical Report 87-60-3, Oregon State University, 1987.

[12] Y. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel & Distributed Systems*, 7(5):506–521, 1996.

[13] Y. Kwok and I. Ahmad. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 47(1):58–77, 1997.

[14] M. Mori, M. Tsukiyama, and T. Fukuda. Adapative scheduling system inspired by the immune system. In *Proceedings of the IEEE Conference on Systems, Man and Cybernetics*, pages 3833–3837, 1998.

[15] A. K. Nanda, D. DeGroot, and D. Stenger. Scheduling directed task graphs on multiprocessors using simulated annealing algorithms. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992.

[16] S. C. S. Porto and C. C. Ribeiro. A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. *International Journal of High-Speed Computing*, 7(2), 1995.

[17] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel & Distributed Systems*, 13(3):260–274, 2002.

[18] T. Tsuchiya, T. Osada, and T. Kikuno. Genetic-based multiprocessor scheduling using task duplication. *Microprocessors and Microsystems*, 22:197–207, 1998.

[19] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogenous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, 1997.

[20] G. Wojtyla, K. Rzadca, and F. Seredynski. Artificial immune systems applied to multiprocessor scheduling. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, pages 904–911, 2005.

[21] A. S. Wu, H. Yu, S. Jin, G. Schiavone, and K.-C. Lin. An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on Parallel & Distributed Systems*, 15(9):824–834, 2004.

[22] M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel & Distributed Systems*, 1(3):330–343, 1990.

[23] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel & Distributed Systems*, 5(9):951–967, 1994.