Evolution-Based Scheduling of Computations and Communications on Distributed-Memory Multicomputers

Mayez Al-Mouhamed *

Abstract

We present a compiler optimization approach that uses the *simulated evolution* (SE) paradigm to enhance the finish time of heuristically scheduled computations with communication times. This is specially beneficial to the class of *Synchronous dataflow* computations which are generally compiled once and run many times over different data sets. Unlike genetic approaches which generally use task swapping to create differential variation our approach consists of *adding pseudo-edges* to the task graph to guide the scheduler in the alignment and clustering of dominant tasks. Added edges alter only the task graph without modifying the scheduler which provides useful flexibility in the implementation of compiler optimization option. The intelligence of iterative methods is used by SE to reduce the run-time and to avoid local minima by using the hill climbing property of search-based methods. Evaluation is carried out for a wide category of computation graphs with communication times which are studied for different levels of communication granularities and task parallelisms. Statistical analysis of results shows that *Edge Addition SE* is capable of finding *near-optimum schedules* as well as outperforming other known heuristics like ETF, DLS, and GLS. Moreover, this approach is useful to complement heuristics whose solution finish time cannot be guaranteed for arbitrary communication and parallelism. Since the performance of most scheduling heuristics is profile-sensitive, optimizing the heuristic solutions through edge addition SE provides increased confidence on the quality of the solution.

Keywords: Distributed memories, heuristics, message-passing, performance, scheduling, simulated evolution

1 Introduction

Compile-time scheduling of coarse-grained computations and communications [17, 4] is one approach to exploit useful parallelism in distributed-memory systems. When the execution behavior can be predicted by the compiler, scheduling can be made effective in adapting the code to the underlying computation and communication subsystems. The compiler estimates the computation and communication requirements and use the knowledge to produce a schedule in which the communication overheads are hidden to some

*Department of Computer Engineering, King Fahd University of Petroleum and Minerals, P.O. Box 787, Dhahran 31261, Saudi Arabia (Email: mayez@ccse.kfupm.edu.sa)

degree by computations. Optimizing code and extracting parallelism out of large scale scientific computations is especially useful when the programs are compiled once and repeatedly executed over different data sets. Compile-time scheduling [16] can dramatically reduce execution time of a class of computations known as *Synchronous Dataflow* [18] such as those found in circuit behavioral description, digital signal processing, robot dynamics, etc. Unfortunately scheduling is one NP-complete problem [6] which explains the benefit of searching efficient scheduling heuristics.

It is well known that NP-hard complexity problems occur very frequently in many fields of science and engineering such as travelling salesman, bipartite matching, placement of cells, etc. For all these problems we have to content ourselves with heuristic solutions which might significantly deviate from optimal solution in many cases. The search space is very large and exhaustive searches can only be done for very small problems. For this reason heuristics are used to prune the solution space by searching in a directed manner and move towards the optimum. Because of the tremendous computational effort required in reaching the optimum these heuristics stop if the solution satisfies some constraints. Also it is known that greedy heuristics easily get stuck in a local minima because of their very nature. The advantage with constructive or constructive-iterative techniques is that they can produce a solution quickly. But generally there is no way to improve the quality of the solution. Because of this, hill climbing heuristics like *Simulated Annealing* (SA) [11] and *Genetic Algorithm* (GA) [7] produce much better solutions though they may take more computational time to generate them.

Simulated Evolution (SE) is one proposed approach for solving combinatorial optimization problems. SE algorithm mimics the natural evolution of biological species. It is known that species continue to evolve under various constraints and become fitter and fitter with respect to their environment. During this process of *Natural Selection* only fitter individuals of a population are allowed to pass on their favorable characteristics to the next generation. The individuals with unfavorable characteristics die without passing their disadvantageous characteristics to future generations. This process now results in better and better populations as time progresses. Some times a trait which was not present in any of the individuals of the parent population suddenly appears in the offspring. This process of introduction of new traits in a population is called *mutation*. Some times mutations lead to a completely new species much better adapted to the environment then its predecessor species. The predecessor species may die out altogether.

All the three discussed techniques have been used extensively to solve NP-Hard problems in the field of engineering and computer science. They have been used for computer aided design of VLSI systems for placement of standard cells, routing, circuit partitioning etc, [5], [1], [13], [12]. GA and SA were also used for scheduling and high level synthesis of digital systems.

The choice between them is governed by the ease of implementation of each and the run time needed. The SA algorithm is the easiest to implement because it does not need analyzing a solution and identifying badly placed elements. This means that the problem and the relationships between the elements of the solution need not be assessed at all in SA. The solution is only characterized by its cost. The only requirement from the designer is to find a suitable neighbor function which is required to traverse the search space. The drawback is that more run time is needed to generate acceptable solutions. A lot of experimentation and time is also needed to tune its parameters. Overall, SA is profitable when the problem is intractable to analysis and the development of heuristics

as well as run time are not a major concern.

The GA gives acceptable solutions much more quickly than SA because of the parallel nature of GA search in which few solutions are present in each generation. This though might requires a lot of memory. The problem need not be analyzed in any detail. The issue instead is how to map the problem to a series of strings as genes and chromosomes such that after applying the genetic operators we can easily get the resulting solution and find its cost. The placement of each and every element of the solution needs not be assessed and a solution is only characterized by its cost. The GA implementation is more difficult than SA because it needs the solution to be represented in a particular way. Generally, GA leads to lesser run time than SA for finding acceptable solutions.

The Simulated Evolution appears to be more intelligent than the above iterative methods. It takes much less time than both of them. The memory requirements are lower than genetic algorithm but exceed that required by SA. Though SE uses only one solution, as in SA, the goodness values of all elements must be saved. It requires an in depth analysis of the problem to discriminate between good and bad placement of elements together with the ability to evaluate the goodness of each element's placement in the solution. It permits the designer to decide about solution generation in an intelligent manner. Therefore, implementing SE is more difficult than the others. It is a must if lower run-time is needed in certain applications. Overall it is an improvement in both the time requirement and the quality of the final solution.

In this paper we present a Simulated Evolution search-based scheduling approach. This is in contrast to genetic algorithm and simulated annealing where the solutions are generated totally randomly. Its advantage lies in the fact that it does not create totally random solutions and then select from among them. Instead it does a directed search and searches only among better solutions. Each solution is generated by using a priority-based scheduling heuristic. Movement in the search space is controlled by simulated evolution. This just increases the run-time of the algorithms. By using simulated evolution we find a good schedule in an acceptable number of iterations.

The organization of this paper is as follows. Section 2 presents some background. Section 3 presents scheduling of computations with communication times. Section 4 presents our proposed evolution-based scheduling. Section 5 presents the implementation of our evolution-based scheduling. Section 6 presents the performance evaluation. We conclude this work in Section 7.

2 Background

Genetic Algorithm (GA) was developed by Holland [7]. In GA there are a number of solutions and each represents one solution of the problem. GA starts with random solutions and the set of these solutions is called the *population*. Each solution is represented in the form of a string of symbols, called *genes*. The string made up of genes is called a *chromosome*.

The solutions in a population interact among each other via chromosomes through crossover operators. New characteristics are introduced via mutation operator. The solutions on which these operators are applied are called parents and are chosen from amongst the population probabilistically depending on their fitness. This results in the development of a new set of solutions called the *offspring*. Now a population of the initial size is again selected from the combined population. This selection is again probabilisetic and depends on fitness. Measurement of fitness is usually based on the objective function. So

in the scheduling context it can be the finish time of a schedule. Those individuals (solutions) of the population which have low fitness value are given lesser chance of moving to the next generation. The important point to note is that the selection for next generation is not deterministic, rather it is probabilisetic. This means that solutions with low fitness can also be selected but the probability of their selection is small and is proportional to their low fitness. Another point to note is that new solutions are generated by only crossover and mutation operators. Both of them operate in a totally random manner and we do not invest any effort in building up a solution using any intelligence of our own. The disadvantage is that this increases the search space even to those regions which are unlikely to yield a good solution.

The Genetic algorithm has been used for many NP-hard problems successfully. It has also been used for multiprocessor scheduling in [8].

Simulated Annealing was first proposed by Kirk-Patrick, Gelatt and Vecchi in 1983 [11]. This algorithm imitates the process of annealing metals which targets good crystal structure. The movement through the search space is done via the *neighbor function*. The neighbor function operates by creating a random change in the current solution. This is called neighbor function because we cannot jump from one solution to an entirely different solution. Rather we make only a slight change from the present solution. This slight change though may result in a large change in the objective function of the solution. The objective function for a solution is referred to as its cost.

The main problem with SA is that the parameters are difficult to control. Also the run-time of the algorithm is high. Again we see that when making moves via the neighbor function we do not use any intelligence or heuristic. This is done totally randomly.

The *Simulated Evolution* (SE) algorithm was proposed by Kling and Banerjee [12], [13] to the problem of standard cell placement in VLSI. This heuristic is based on an analogy between the process of natural selection in natural environments. The Simulated Evolution algorithm allows the use of some intelligence while generating new solutions. SE attempts to combine the best of pure constructive-iterative and pure search-based techniques because it heuristically prunes the search tree and as a result new solutions are not randomly generated. SE combines the intelligence of iterative methods to reduce the run-time and uses the hill climbing properties of search-based methods to avoid getting stuck at local minima. The generalized simulated evolution algorithm is shown on Figure 1.

An initial seed solution is provided with the objective of refining and modifying it by evolution. The seed is generated by some constructive heuristics or randomly. A hill climbing parameter cp_0 is initialized. Generally, SE converges faster with increasing seed goodness but seeds with lower goodness do not affect the quality of the final solution. Mainly, SE has three phases which are: (1) evolution, (2) mutation, and (3) evaluation. We briefly present each of the above phases.

In the *Evolutionary phase* the elements (tasks) which constitute the solution are associated some goodness values which tell how far or how near an element is to its optimum assignment. The normalized goodness associated to some element is the survival probability of that element. Elements with low goodness become extinct which means they must be assigned new positions by using constructive techniques based on some local cost function. The optimum assignment results in optimum solution.

In the *Mutation phase* some unpredictable alteration of the design of some elements is done to avoid getting stuck at local minima. For this new features are introduced in some

```
Simulated Evolution;
  Begin
    S_0 := GenerateInitialSolution; /* may be constructive or random */
    S := S_0; /* solution S is initialized to S_0 */
    S_best := S_0; /* best solution is initialized to S_0 */
    cp := cp_0; /* control parameter initialized cp_0 */
        For i := 1 to MaxIter do
        Begin
        C_pre := Cost(S);
        FindGoodnes(S); /* evaluate goodness of all elements of
        the solution S */
        S_t := ConstructiveAssignment(S); /* Generate temporary
        solution and store it in S_t*/
        S_t := DoMutation(S_t); /* do mutation with small probability */
        S_t := ReGenerate(S_t); /* make sure S_t is valid a solution */
        Gain := Cost(S_best) - Cost(S_t); /* find gain */
        If (Cost(S_t)=C_pre) then cp := f(cp) /* increase hill climbing parameter */
        else cp := cp_0; /* keep cp at a minimum */
        If (Gain > 0) then S_best := S_t;
        If (Gain >Random(-cp, 0)) then S := S_t;
        End;
    Return(S_best);
  End.
```

Figure 1: Generalized simulated evolution algorithm

elements through mutations which are made with low probability (below 5%) to keep the search directed. The mutation rate is much lower than the evolution rate. This allows SE to jump out local optimum and hopefully traverse the search space to reach better and better solutions. The search space constitutes all the possible valid solutions.

In the *Evaluation phase*, SE examines the following cases. First, SE gets stuck in a local minima if the cost of two successive solutions is the same which requires increasing the value of the hill climbing parameter cp through $f(cp)$ to escape. The control parameter is reset to cp_0 in the next iteration. Thus the basic strategy is to keep cp at a minimum value and increase it only when it is necessary. Second, the new solution becomes best solution if the gain is positive. Third, SE also accepts the new solution even with non-positive gain if the new solution goodness is worse than current best by at most q, where q is obtained by generating a random integer between 0 and $-cp$. Finally, SE checks for the termination conditions. The stopping conditions can be monitored by measures such as (1) the number of iterations, (2) the relative improvement over several steps or (3) the number of steps since the best solution was found.

The SE algorithm is much faster than the previous SA and GA and gives much better results too. Mainly, SE does not need the careful tuning of parameters as for SA and the large memory space required for GA. Moreover, SE provides good results using lesser CPU time. The SE algorithm has given better quality solutions in lesser run time than SA and GA for VLSI cell placement, travelling salesman [5], and high level synthesis [1].

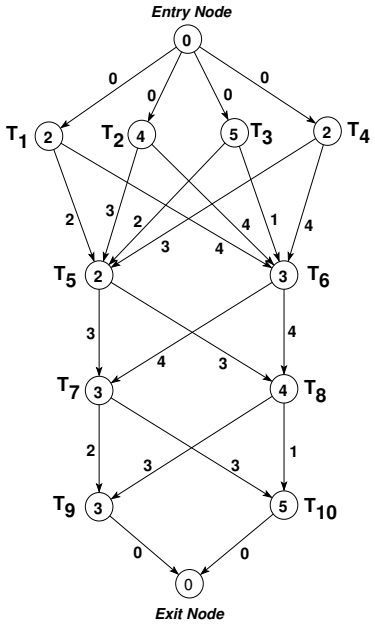


Figure 2: Example of task graph

3 Scheduling with communication costs

A set $\Gamma(T_1, \ldots, T_n)$ of n tasks (T) with their precedence constraints and communication costs are to be non-preemptively scheduled on a set of identical processors. The computation can be modeled [9] by using a directed acyclic task graph $G(\Gamma, \rightarrow, \mu, C)$ where $\rightarrow$, $\mu(T)$, and $c(T, T') \in C$ denote the precedence constraints, the task execution time, and number of messages to be sent from T to its successor T', respectively.

The multiprocessor is denoted by $S(P, R)$ where $p, p' \in P$ are two processors and $r(p, p') \in R$ is the time to transfer one unit of messages from p to p' through the interconnection network. Thus the communication model is based on the latency factors between the processors which depend on the network topology. Assuming that the communication media is *contention-free*, the transfer time of $c(T, T')$ messages is $c(T, T') \times r(p(T), p(T'))$, where $p(T)$ and $p(T')$ are the processors running T and T', respectively. Local message transfer has zero cost ($r(p, p) = 0$).

The problem addressed in this paper consists of scheduling computations represented by $G(\Gamma, \rightarrow, \mu, C)$ over multiprocessor $S(P, R)$ so that *overall finish time is held to a minimum*. As example consider the task graph shown on Figure 2 for which the set of tasks is $\Gamma(T_1, \ldots, T_{10})$. The circled values are the task execution times ($\mu(T)$) and the precedence edges are labeled with the number of messages ($c(T, T')$). For example, edge $T_5 \rightarrow T_8$ indicates that T_8 cannot start before T_5 completes and $C(T_5, T_8) = 3$ data messages are transferred between $p(T_5)$ and $p(T_6)$. Given a set of processors S, the problem is to schedule $\Gamma(T_1, \ldots, T_{10})$ over S so that overall finish time is held to a minimum.

Since the scheduling problem is NP-complete it is useful to design scheduling heuristics that are capable of delivering near-optimal solutions. In the following we review three scheduling heuristics which are ETF, DLS, and GLS.

3.1 Earliest-task-first

Selecting tasks and processors according to the principle of *earliest-task-first* is the strategy used by ETF [9]. The objective function is to minimize processor idle times through selection of a task T and a processor p for which the earliest-starting-time ($est(T, p(T))$) of T on p is the least among all ready tasks and processors. Let T be a ready task and denote by $Pred(T)$ its set of predecessors. The *earliest-starting-time* $est(T, p(T))$ is the earliest time at which the latest message from the predecessors arrives to $p(T)$:

$$est(T, p(T)) = \max_{T' \in Pred(T)} \{ct(T', p(T')) + c(T', T) \times r(p(T), p(T'))\} \quad (1)$$

where $ct(T', p(T'))$ is the completion time of predecessor T'. Note that $est(T, p(T))$ is nil if T has no predecessors. There exists at least one processor p^*, that is free at time $t(p^*)$, for which T can start at the earliest $est(T, p^*)$ among all the available processors:

$$est(T, p^*) = min_p\{\max\{est(T, p(T)), t(p)\}\} \quad (2)$$

In ETF task and processor selection are based on finding the earliest startable task and its best suited processor. Its main strategy is to use the knowledge of local task starting times to minimize processor idle times by trying to maximize the overlap between computation and communication. Consider the task graph of Figure 2 and assume the following completion times $ct(T_1, p_1) = 2$, $ct(T_2, p_2) = 4$, $ct(T_3, p_3) = 5$, and $ct(T_4, p_4) = 2$. The ready tasks are T_5 and T_6. Using Equation 2, ETF evaluates $est(T_5, (p = 1, 2, 3, 4)) = (7, 7, 7, 7)$ and $est(T_6, (p = 1, 2, 3, 4)) = (8, 6, 8, 8)$. Thus ETF selects T_6 first and assigns it to p_2 and then T_5 which is assigned to p_1.

ETF is based on Graham's list-scheduling [6] in which the scheduler tracks the increasing sequence of processors' completion times by using a *global time*. Thus the starting times of successively scheduled tasks form a non-decreasing sequence in time. This enabled finding a worst-case bound [9] on the schedule length.

3.2 Dynamic level scheduling

Another priority-based scheduling approach is *dynamic level scheduling* (DLS) [18]. In DLS, the largest sum of computations along a path going from a task to exit node is considered as the *static task-level*. DLS evaluates a *dynamic task-level* for each ready task as a function of static task-level and task starting time. Task and processor selections are based on selecting the task and processor for which the dynamic task-level is the largest. For the task graph of Figure 2, DLS evaluates the static task-levels of T_5 and T_6 as $level(T_5) = \mu(T_5) + \mu(T8) + \mu(T_{10}) = 11$ and $level(T_6) = \mu(T_6) + \mu(T8) + \mu(T_{10}) = 12$. The decision function is based on selecting T and p for which $DLS(T, p) = level(T) - est(T, p)$ is the highest. Using the values of $est(T, p)$ found in Sub-Section 3.1 we obtain $DLS(T_5, (p = 1, 2, 3, \textit{or } 4)) = 4$ and $DLS(T_5, p = 2) = 6$. Thus DLS assigns T_6 to p_2 and T_5 to p_1.

Unfortunately, the evaluation of static task-levels for computations with communication times does not provide effective task priority because the task-level strongly depends on mapping tasks to processors and their implied communications.

3.3 Generalized list scheduling

Generalized list scheduling (GLS) [2, 3] uses a decision function based on *Highest-Level-Earliest-Task-First* $HLETF$ which augments the ETF discipline with a task-level priority function to improve its performance. GLS iteratively schedules the *forward* and *backward* computation graphs. The task completion time achieved in some scheduling iteration is used as task-level in the next scheduling iteration. In each forward or backward scheduling iteration, GLS selects a task T^* and a processor p^* such that $ct_{i-1}(T^*) - est_i(T^*, p^*)$ is the highest among all ready-to-run tasks, where $ct_{i-1}(T^*)$ is the achieved completion time of (T^*) in the $(i-1)$th iteration and $est_i(T^*, p^*)$ is the earliest-starting time of the same task found in current scheduling iteration (i). The task completion time $ct_{i-1}(T^*)$ is used as an approximation of the longest distance from entry node to T^* in iteration $i-1$. But in the ith iteration $ct_{i-1}(T^*)$ represents the longest distance from T^* to exit node which is similar to the task-level used in list scheduling [6].

Using the task graph of Figure 2, GLS evaluates its decision function for T_5 and T_6 in the ith iteration (forward or backward) as $d(T, p) = ct_{i-1}(T) - est(T, p)$. This is similar to DLS but with the difference that $level(T)$ in DLS is replaced here by $ct_{i-1}(T)$ to account for some communication on a path from T to exit node. This approach enables searching and optimizing solutions as the result of using more refined task-level in each scheduling iteration.

Though [2] GLS was originally proposed for two-iteration scheduling we use it here as an iterative scheduling approach with arbitrary number of iterations to allow exploring a space of solutions. The main advantage of GLS is its ability to find local solution with *probable refinement* of the solution finish time through the iterations.

More generally search-based methods like branch-and-bound [15], simulated annealing [19, 11], and genetic algorithm [8] were proposed for finding good mapping and partitioning of computations. Scheduling based on *task duplication* over idle processors was proposed [14] to reduce the communication without excessively increasing overhead in managing duplicated data. Linear clustering [10] consists of iteratively clustering the tasks along the most communicating chains on one processor. *Clustering* over unbounded number of processors [17] consists of partitioning the set of tasks into clusters of sequential tasks and reducing the number of clusters to the number of processors by merging clusters. The *dominant sequence clustering* (DSC) [20] is a low complexity clustering that accepts merging of a task to a cluster only if it decreases the length of the dominant chain to which the task belongs to decreases.

In the next section we present our proposed approach to formulate and implement the simulated evolution algorithm on the top of arbitrary scheduling heuristics.

4 Evolution-based scheduling

The simulated evolution is based on two fundamental features which are the *hereditary variation* and *differential reproduction*. Hereditary variation refers to an evolutionary process that changes in time by transiting into a series of states and each next state is similar to the previous one in some aspects and yet different. Differential reproduction implies that transfer from one state to the other is subject to an evaluation process that probabilistically discards inferior elements of current state and retains superior elements for regeneration of next state.

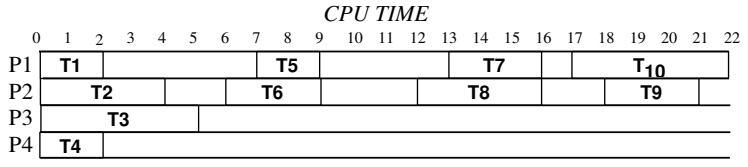
Here we consider the problem of scheduling precedence-constrained computations

with communication times with the objective of minimizing overall schedule time over a bounded number of processors. Constructive approaches use greedy heuristics in gradually building a solution while meeting a set of constraints in scheduling each task. The precedence constraints and communication costs are among the most important constraints to be considered by the scheduler in order to generate valid solution. In our case, maintaining similarity in transiting from one solution to another is done by using one single constructive scheduling heuristic, i.e. the logic used to constructively generate all solutions is the same. Therefore, SE is meant to complement constructive scheduling approaches which may produce good and bad assignments because of its greedy nature and the limitation of its logic. In this case, transfer from one solution to another is subject to evaluation of current solution so that we probabilistically retain or discards elements of current solution depending on how well they may contribute in the optimum solution. Discarding some poor elements of the current solution requires detecting them, evaluation of their goodness, and *modifying some of their constraints* so as to direct the heuristic not to poorly perform with respect to these elements in subsequent solution. Retaining some elements of current solution means keeping unchanged their local constraints and relying on the heuristic in generating similar good assignments for these elements.

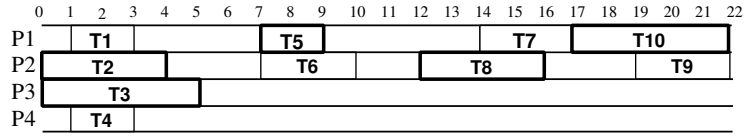
The heuristic always applies the same logic in constructing the solution but overall finish time depends on many factors, among them the local performance of dominant tasks. The ideal assignment is quite complex. A *dominant task* is one that has zero slack in a given schedule. Our effort will be concentrated on dominant tasks and dominant chain of tasks with no explicit action on the other tasks. The quality of the solution is strongly dependent on how efficient is the performance of the heuristic in assigning the chain of dominant tasks which directly controls the finish time. Therefore, the way dominant tasks are assigned with respect to each other must be considered as an element of the solution. An *element* of the solution is the assignment of some immediate dominant tasks that is done by the heuristic. For example three immediate dominant tasks $T_1 \rightarrow T_2 \rightarrow T_3$ provides two elements: $elm(T_1 \rightarrow T_2)$ and $elm(T_2 \rightarrow T_3)$. The goodness of $elm(T_1 \rightarrow T_2)$ is function of: (1) whether T_1 and T_2 must communicate (different processors), and (2) whether the starting of T_2 is tight by the completion of T_1 (or its communication). Different elements may also be defined in the case of two dominant predecessors $T_1 \rightarrow T$ and $T_2 \rightarrow T$ which require defining new element $elm(T_1, T_2 \rightarrow T)$. Two dominant successors $T \rightarrow T_1$ and $T \rightarrow T_2$ represents the dual case and require defining element $elm(T \rightarrow T_1, T_2)$.

4.1 Dominant tasks

Task mobility is one quantifier of task priority. To define task mobility consider the task-processor mapping as generated by some deterministic heuristic. Each processor receives a set of *ordered tasks*. While honoring all precedence and communication constraints some tasks can be pulled up to some later time without increasing the schedule time. By this way, each task T can be associated a non-negative time slack called *mobility* that is the *maximum task mobility* defined by $m(T) = lst(T) - est(T)$, where $est(T)$ and $lst(T)$ are the earliest and latest starting times of T in a given schedule. Clearly, a schedule can be used to find out the relative importance of the tasks based on the schedule mobility values. For example, the schedule finish time is necessarily constrained by any chain of immediate tasks from entry node to exit node which all have zero mobility. Such chain of



(a) A schedule in which the tasks are set to their earliest starting times



(b) A schedule in which the tasks are set to their latest starting times

Figure 3: The forward (a) and reverse (b) schedules for the example

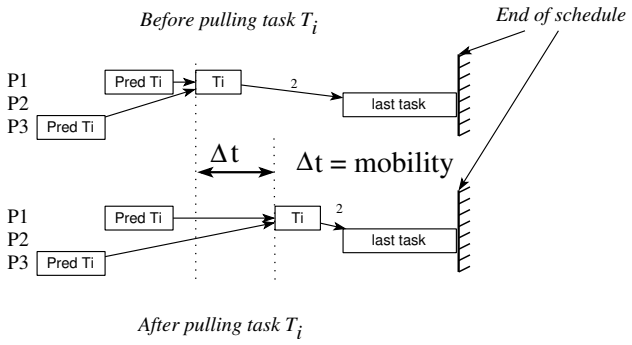


Figure 4: The operation of finding mobility of task T_i

tasks must be considered as *dominant* if the average mobilities of its individual tasks are nil or relatively very low when obtained from different schedules. Delaying a dominant task (zero mobility) beyond its assigned time is likely to cause the schedule overall time to increase. A *non-dominant* task may occasionally get zero or low mobility value out of one schedule but its average mobility is likely to be relatively large. Dominant chains are among the most important contributors to overall schedule length.

Therefore, one way to find shorter schedule time is to assign dominant tasks to shorten the sum of computations and communications along dominant chains. This can be done by increasing the priority of dominant tasks in the next scheduling pass which might shorten dominant chains. It is also possible that some secondary chains become dominant in the newly generated schedule.

As example recall the task graph of 10 tasks (Figure 2) and its schedules on four fully connected processors that are shown on Figure 3-(a) and (b). The schedule finish time is 22 (Figure 3-(a)) and the starting time of task T is denoted by $st_e(T)$. The task-processor assignments remain unchanged. Starting from exit nodes we build up a schedule by pulling the tasks towards the end of schedule mark as much as possible while preserving: (1) the schedule length, and (2) all precedence and communication constraints. Denote by $st_l(T)$ the starting time of T in the new schedule which is shown on Figure 3-(b). The tasks having zero mobility are shown in bold.

4.2 Evaluation of task mobilities

Generalized list scheduling (GLS) [2] is used here for searching local schedules from where to evaluate the average mobility. GLS generates a solution in each iteration. We try keeping a tab on the mobility of each task by running GLS for N iterations and averaging the mobility $m_i(T)$ of T out of the schedule generated from each iteration. The average mobility will be $am(T) = (\sum_{1 \leq i \leq N} m_i(T))/N$. The tasks having low $am(T)$ values were on dominant chain (zero-mobility chain) in the majority of the generated schedules. Tasks having higher values of $am(T)$ almost always were lying on secondary chains. Therefore, we are entitled to modify the scheduling process in such a way that we get better solutions using $am(T)$.

4.3 The edge addition approach

The strategy is to keep the scheduling heuristic unchanged but slightly modify the original task graph to enforce some scheduling decision to be taken by GLS iterative scheduling. Adding a pseudo-edge to a pair of tasks in the task graph allows a lot of control over the scheduling process. A pair of tasks linked by pseudo edges become constrained to carry out some dummy communication of some weight. The scheduler does not distinguish pseudo-edges from precedence edges and therefore assigns two tasks linked by an infinitely weighted pseudo-edge on the same processor while preserving all precedence constraints. Formally, a pseudo-edge $(T \rightarrow T')$ is an edge that carries infinite communication weight which forces some scheduling heuristic to assign both tasks T and T' to the same processor. Thus adding a pseudo-edge between any pair of tasks lead these tasks to run on the same processor provided that no cycles are introduced by the pseudo-edge.

The advantage is that some local desired action can be injected for a subset of dominant tasks while leaving the remaining tasks under competition as dictated by the scheduling heuristic. The evolution process can associate a performance measure for each pseudo edge and then try to mix and match between the available set of individual edges. Adding edges can result in a sort of chains through the graph and these pseudo edges will enforce the tasks of these chains or part of chains to be on one processor in a definite precedence order. There can be more than one chain and all the tasks of a given chain will be scheduled on the same processor.

The edge addition approach is useful to enforce some dominant tasks to be assigned on the same processor, while leaving all the flexibility to the scheduler for finding to which processor they could be better assigned. Thus implied inter-task communication is eliminated. Some genetic approaches swap task-processor assignments in an attempt to introduce variation in the schedule. In addition to limiting the number swapped tasks it also requires propagating the effect of task swapping into the entire schedule. The edge addition approach is much more flexible than task swapping because a constructive scheduler can be run again on the computation graph with added edges and then attempt to find a new solution.

The problem is to find a systematic method for improving the schedule finish time through analysis of mobility confidence and adding pseudo-edges. For this we use GLS iterative scheduling that can be run for a number of iterations which allows collecting the $am(T)$ values for all the tasks and identification of dominant tasks.

In the next sub-sections we explain our approach to enforce some pseudo-edges between

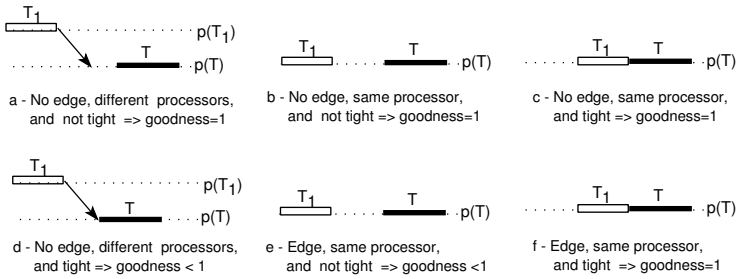


Figure 5: Assignment of tasks for one dominant predecessor

immediate dominant tasks to guide the GLS iterative scheduling towards the optimum solution.

4.3.1 Goodness of one dominant predecessor

Rule 1 is to evaluate the goodness associated to an element $elm(T_1, T)$ defined by two immediate dominant tasks T_1 and T. Element $elm(T_1, T)$ can be assigned by the scheduler according to one of the cases shown in Figure 5. In the following we analyze the above cases. First, we need to distinguish the case where both tasks are assigned to one single processor (Figure 5(b), (c), (e), and (f)) or to two distinct processors (Figure 5(a) and (d)). Second, $elm(T_1, T)$ can be found with a pseudo-edge already in place or not. Third, the starting of successor task T can be bound by the completion of T_1 (Figure 5(c), (d), and (f)) or not (Figure 5(a), (b), and (e)). Assigning immediate dominant tasks to distinct processors is likely is to increase the schedule length because of the need for communication.

One strategy to cancel the need for communication is to enforce these immediate tasks to be scheduled on the same processor in the next schedule generation. Our approach is based on modifying the problem constraints to promote clustering these tasks in a probabilistic manner. This can be done by enforcing an *infinite weight* to edge $(T_1 \rightarrow T)$ so that the scheduler will necessarily schedule T_1 and T on the same processor to avoid infinite communication. The enforcement rule consists of assigning a *large weight pseudo-edge* to element $elm(T_1, T)$. Under this condition, the scheduler still have the liberty to insert tasks between T_1 and T as required. The only constraint introduced is that T must be assigned to the processor that was assigned T_1. The enforcement of pseudo-edge must be reversible because the need of some edges at some solution state might be offset by other newly assigned edges at other states. The idea is that a pseudo-edge that is becoming useless must be removed in order not to block the search for optimum solution. The goodness must be low for some element only when the scheduler is poorly performing with respect to this element. The enforcement rule is one local strategy to eliminate the dominant communication but one may also find instances for which the effect of single processor assignment is of no benefit. In the following we analyse a number of cases in order to evaluate the goodness of the constructive scheduler with respect to elements of the solution.

In the case no pseudo-edge is initially set for $elm(T_1, T)$, the goodness of the scheduler can be considered as satisfactory with respect to $elm(T_1, T)$ as long as the starting time of T is not tight by the completion of T_1 or its communication. Formally, element $elm(T_1, T)$

Procedure Goodness_one_pred(elm,$Edge$,$Gooodness$);
Begin /* Dominant predecessor-successor */
If ($Edge$) **then** /*There is already an edge */
$Goodness := \mu(T_1)/(\mu(T_1) + st(T) - ct(T_1))$;
Else /*No pseudo-edge */
$Goodness := \mu(T_1)/(\mu(T_1) + c(T_1, T) \times r(p(T_1), p(T))$;
End

Figure 6: Evaluation of Goodness for one dominant predecessor

is said to be tight if $st(T) = ct(T_1) + c(T_1, T) \times r(p(T_1), p(T))$. The cases where element $elm(T_1, T)$ is not tight are shown in Figure 5-(a) and -(b). The reason the goodness must be high in this case is that T_1 by its computation and its communication is not directly responsible of delaying T but there must be another reason for which T is delayed. Therefore, the goodness must be high (survive) whenever the element is not tight and there is no pseudo-edge which is already set. It is clear that adopting this approach means that the reason for the delay of a dominant task which is not tight is to be found elsewhere.

In the case the starting of T is found to be tight and there is no pseudo-edge, then the survival of element $elm(T_1, T)$ depends on the scheduler's goodness in assigning the tasks of this element. If the goodness is high (Figure 5-(c)), then the scheduler is performing well with respect to this element and nothing need to be done. On the other hand, finding low goodness (Figure 5-(d)) is an indicator of poor performance of the scheduler at this precise element which requires some action. Clearly, as far as no pseudo-edge is set the goodness can be expressed as $\mu(T_1)/(\mu(T_1) + c(T_1, T) \times r(p(T_1), p(T))$.

In the case a pseudo-edge is already set for $elm(T_1, T)$ the scheduler will necessarily assign T_1 and T on the same processor and no communication will be needed for these tasks. If $elm(T_1, T)$ is tight then we are achieving our local objective because this is the local optimum for this element. In this case, the goodness must be high and the pseudo-edge is to be kept. On the other hand, finding $elm(T_1, T)$ not tight means that there is no benefit from setting the pseudo-edge because T is delayed anyway by some other reason. The penalty for this case is the delay from completion of T_1 to starting of T and that the associated goodness must be low to enable removal of the unnecessary pseudo-edge. Therefore, when there is one already set pseudo-edge the goodness can be evaluated as $\mu(T_1)/(\mu(T_1) + st(T) - ct(T_1))$. Evaluation of the goodness for the case of one dominant predecessor is shown in Figure 6. It is assumed that elm is a data structure that contains: 1) pointers to immediate dominant tasks, 2) task time and communication, 3) task starting time and assignment as per current solution. $Edge$ is a Boolean that is true only when there is pseudo-edge which is already set for the present element elm.

The state diagram of element's evolution is shown in Figure 7 and the code of the corresponding procedure ($Element\_Evolution$) is shown in Figure 8. An element that is tight (Boolean $Tight = True$) but without pseudo-edge (Boolean $Edge = False$) would require no pseudo-edge with a probability that is its goodness. On the other hand, an element that is not tight ($Tight = false$) but with a pseudo-edge ($Edge = True$) would

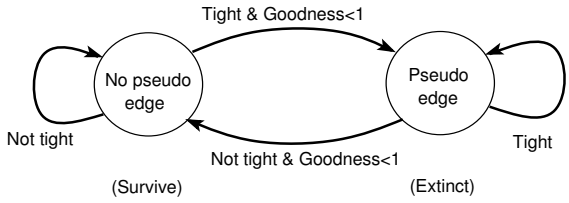


Figure 7: Evolution of element of solution

Procedure $Element\_Evolution(Tight, Edge, Goodness)$;
Begin
If (Not $Edge$) and ($Tight$) and ($Rand(0,1) \leq Goodness$) **then**
$Edge = True$; /* Set posted pseudo-edge (extinct) */
If ($Edge$) and (Not $Tight$) and ($Rand(0,1) \leq Goodness$) **then**
$Edge = False$; /* Remove pseudo-edge (re-generate) */
End

Figure 8: Evolution of element as function of state and goodness

require keeping its pseudo-edge with a probability that is its goodness.

4.4 Goodness of two dominant predecessors

Another important element is the assignment of two dominant predecessors T_1 and T_2 with respect to their dominant successor T. This defines element $elm(T_1, T_2, T)$ for which the possible assignments are shown on Figure 9. The objective is to find evolutionary conditions for this type of elements in order to promote the performance of the scheduler through the possibility of creating pseudo-edges $T_1 \rightarrow T_2$. For this, we analyse a number of cases in order to evaluate the goodness of the scheduler with respect to element $elm(T_1, T_2, T)$ of a given solution.

Assuming no pseudo-edge is initially set for $elm(T_1, T_2, T)$, the scheduler can be considered as performing well as far as the starting of successor T is not tight by its predecessors T_1 nor T_2. Element $elm(T_1, T_2, T)$ is said to be tight if $st(T) = \max_{T_t \in \{T_1, T_2\}}\{ct(T_t) + c(T_t, T) \times r(p(T_t), p(T))\}$ which corresponds to the cases shown in 9(c) and (d). In the case the element is not tight, the goodness must be high because there must be other reasons, than the assignment of T_1 or T_2, for which T was delayed. If the starting of T is found to be tight (Figure 9(c) and (d)) and there is no pseudo-edge, then again the survival of this element without pseudo-edge depends on the scheduler's goodness in assigning element $elm(T_1, T_2, T)$.

In the case a pseudo-edge is already set among predecessors T_1 and T_2 the scheduler will necessarily assign these tasks to the same processor (serializing them). Once a solution is generated after setting the pseudo-edge, then element $elm(T_1, T)$ can be found as tight (Figure 9(f)) or not (Figure 9(e)). The goodness must be high if we are achieving best local performance which corresponds to finding tight element. Therefore, the pseudo-edge must be kept if the element is tight and pseudo-edge is already set. However, if this element

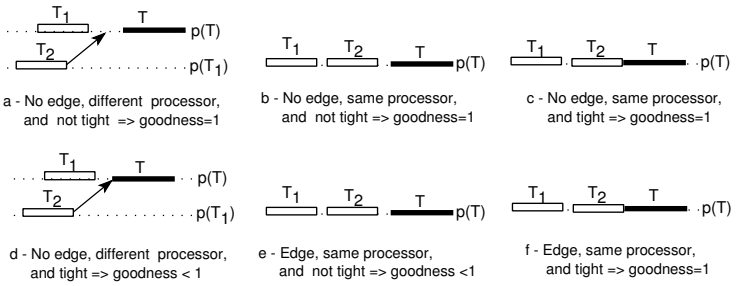


Figure 9: Assignment of tasks for two dominant predecessors

is not, then there is no benefit from maintaining the pseudo-edge because T is delayed anyway by some other reason. Therefore, the goodness must be low to enable removal of the unnecessary pseudo-edge. As one can see the evolution of element $elm(T_1, T_2, T)$ is again the one described in Figure 8.

Another important case is the assignment of two dominant successors which is denoted by element $elm(T, T_1, T_2)$. It can be easily shown that element $elm(T, T_1, T_2)$ is the dual of element $elm(T_1, T_2, T)$. One element can be obtained from the other if we reverse the direction of the dependence edges and appropriately rename the tasks. In any case, the evolution mechanism of elements is the same for all the studied type of elements as depicted in Figure 8. The difference among the various elements' types lies in the evaluation of the goodness associated for each type of elements.

The goodness associated to the assignment of two dominant predecessors is the quotient of *time cost in optimum assignment* to *time cost achieved by the scheduler* for the same assignment. Finding time in optimum assignment for two dominant predecessors or two dominant successors present special cases of the problems of optimally scheduling join or a fork task graphs that were studied in [20]. Function *Goodness_two_pred* for evaluation of goodness is displayed in Figure 10.

We define the time achieved by the scheduler in assigning element $elm(T_1, T_2, T)$ as the union of all time points during which there is computation (T_1 and T_2) or communication due to precedence $T_1 \rightarrow T$ and $T_2 \rightarrow T$. Step 1 of function *Goodness_two_pred* allows evaluation of the *union of active times* for both predecessors, which we call *activity*. It first sorts T_1 and T_2 in non-increasing order of their *last-message-time* $lmt(T_t, T) = ct(T_t) + c(T_t, T) \times r(p(T_t), p(T))$, where $ct(T_t)$ is the completion time predecessor T_t, and $c(T_t, T) \times r(p(T_t), p(T))$ is the time to send $c(T_t, T)$ messages from T_t to T.

To evaluate the union of active times, the predecessor with largest lmt, called $last$, allows initializing the activity to $lmt(last, T) - st(last)$ that is sum of all active times for edge $(last \rightarrow T)$. Next we examine whether the other task $(first)$ overlaps with the active time of $last$ or not. If it overlaps $(lmt(first, T) > st(last))$, then *activity* must be incremented by $st(last) - st(first)$ only when task $first$ starts earlier than $last$. If it does not overlap $(lmt(first, T) \leq st(last))$, then *activity* must be incremented by $lmt(first, T) - st(first)$ which accounts for all active times for the edge $(first \rightarrow T)$.

Step 2 of function *Goodness_two_pred* allows the evaluation of the time cost in optimum assignment of $elm(T_1, T_2, T)$. To minimize the starting time of T, predecessors T_1 and T_2 can be ideally arranged in sequence, in which case the above tasks execute one after the other on the same processor and will be called T_{s1} and T_{s2}, or in parallel and will be called T_{s1} and T_p.

Procedure *Bound_two_pred*($elm(T_1, T_2, T)$,*bound*,*edge*);
Begin /* evaluate bound and best pseudo-edge */
$T_{s1} := T_2$; $T_p := T_1$; $T_{s2} := \emptyset$;
If $(act(T_p, T) \geq act(T_{s1}, T)$ **then** $Exchange(T_{s1}, T_p)$;
If $(\mu(T_{s1}) \leq c(T_p, T) \times r(p(T_p), p(T))$ **then** /* T_{s1} and T_{s1} are serial */
$T_{s2} := T_p$; $T_p := \emptyset$; $bound := \mu(T_{s1}) + \mu(T_{s2})$; $edge{:=}(T_1 \rightarrow T_2)$;
Else /* T_{s1} and T_{s2} must run in parallel */
$bound := Max\{\mu(T_{s1}), act(T_{s2}, T)\}$; $edge{:=}(T_1 \rightarrow T)$;
End

Procedure *Goodness_two_pred* ($elm(T_1, T_2, T)$,*bound*,*Goodness*);
Begin /* two dominant predecessors-successor */
/* Step 1: evaluate scheduler time for $elm(T_1, T_2, T)$ */
$last := T_2$;$last := T_1$;
If $(lmt(last, T) < lmt(first, T))$ **then** $Exchange(last, first)$;
$activity = lmt(last, T) - st(last)$;
If $(lmt(first, T) > st(last))$ **then** /* first overlap with last */
If $(st(first) < st(last))$ **then**
$activity = activity + st(last) - st(first)$; /* first starts earlier */
Else /* first and last do not overlap */
$activity = activity + lmt(first, T) - st(first)$;
/* Step 2: evaluate goodness of $elm(T_1, T_2, T)$ */
$Goodness = bound/activity$;
End

Figure 10: Evaluation of goodness for two dominant predecessors

The ideal arrangement is governed by the tasks' execution times and their communication costs and allows the least value of union of active times for the predecessors. Clearly the ideal arrangement allows starting T at the earliest if one assumes that T will be assigned by the scheduler to run on processor $p(T_{s1})$. Denote by $act(T_t, T) = \mu(T_t) + c(T_t, T)$ the worst active computation and communication time for predecessor T_t. It can be easily shown that T_{s1} must be the task with largest value of $act(T_t, T)$. Now we need to decide whether the other task (now called T_p) should run in sequence or in parallel with T_{s1}.

If the communication cost $c(T_p, T)$ covers (larger) computation $\mu(T_{s1})$ then arranging both predecessors in sequence gives shorter overall activity ($T_{s2} = T_p$). Therefore, the least union of activity is $bound = \mu(T_1) + \mu(T_2)$. If there is no edge already set in current state and if for any reason the goodness is low and a pseudo-edge is set then this pseudo-edge must be $edge(T_{s1}, T_{s2})$. Such an edge enforces the scheduler to sequentially assign the above predecessors.

On the other hand, if communication $c(T_p, T)$ covers computation $\mu(T_{s1})$ then assigning the other predecessor in parallel gives the least overall activity. The least union of activity in this case is then $bound = \max\{\mu(T_{s1}), \mu(T_p) + c(T_p, T)\}$.

The goodness of element $elm(T_1, T_2, T)$ is simply $Goodness = bound/activity$ which

can be used by procedure *Element_Evolution* shown in Figure 8. The performance of the scheduler with respect to element $elm(T_1, T_2, T)$ will be probabilistically subject to the following three cases: (1) no change if goodness is high, (2) setting of pseudo-edge if the goodness is low and there is no edge, and (3) removal of currently set pseudo-edge if the goodness is low. Note that the whole processing of two-dominant-predecessors is identical to the case of two-dominant-successors if one takes the dual representation the associated dependence edges.

5 Implementation

In this section we present our simulated evolution scheduling algorithm which we call (*SE-Schedule*). The pseudo-code of *SE-Schedule* is shown in Figure 11. We briefly explain the steps of the algorithm which has two sections: (1) initialization, and (2) evolution.

In the *initialization section*, the process begins by getting the initial set of task priority (task-level) values from the procedure *GetInitialLevels*. This procedure takes as input the reverse graph G_r and the multiprocessor $S(P, R)$. It does one pass of *earliest-task-first* ETF scheduling and returns the set of task completion time ($ct(T)$) as task-priorities which are stored into set L. We then do K runs of GLS iterative scheduling using L for the first run. GLS takes four parameters namely: G, the graph; $S(P, R)$, the multiprocessor; L the set of task-levels; and the number of iterations K. This procedure returns its best solution in S_{best} after exploring 10 ($K = 10$) scheduling iterations because extensive testing showed that the probability of finding noticeably shorter finish time than the best generated out of the first 10 iterations is very small [3]. It also returns the zero mobility confidence set ZMC which is evaluated from all the generated schedules. The set of all possible *dominant elements* ($ElmSet$) is found by using procedure *FindDominant* which takes as argument the current task graph. Procedure *FindBound* uses set $ElmSet$ and G in evaluating the previously defined activity time that corresponds to optimum placement of each dominant element. Set $ElmSet\_Edge$ consists of a set of Boolean that indicates whether a given pseudo-edge $ElmSet\_Edge(elm)$ is set or not. All pseudo-edges are reset ($ElmSet\_Edge(elm) = \emptyset$) at the start of the SE algorithm. Finally, the initial value of the control parameter is set at 1% of the best solution cost.

Now the *simulated evolution section* of the program begins. This section consists of five phases: (1) the *evaluation phase* for finding goodness of new solution, (2) the *evolutionary phase* to do evolution of dominant elements based on their goodness, (3) *mutation phase*, (4) the *regeneration phase* to build new solution, and (5) the *selection phase* to carry out evolutionary selection of the solution.

In the *evaluation phase*, a procedure *FindPerformance* uses a new solution S (if any) and bounds $ElmSet\_Bound$ to evaluate the performance of elements of $ElmSet$ and their goodness ($ElmSet\_Goodness$). Similarly, procedure *FindTight* examines how the successor task for the elements of $ElmSet$ are set in the solution S. A Boolean $ElmSet\_Tight(elm)$ is set by *FindTight* when the starting time of successor task is the completion time of its latest dominant: (1) predecessor, or (2) predecessor communication as previously explained.

In the *evolutionary phase* we examine evolution of each dominant element. Here, the evolution of each element consists of setting or resetting of its pseudo-edge with a probability that is equal to the goodness of the element. This means that an element elm having a good individual performance has a higher chance of selection. This is done by using the procedure *Element_Evolution* which examines each dominant element elm

Program: Evolution-Based Edge Addition Scheduling ($SE - Schedule$);
Begin

(1) **Initialization:**

$n_{edges} := 0$;
$L :=$ **GetInitialLevels**$(G_r, S(P,R))$; /* using constructive ETF on G_r */
$(S_{best}, ZMC) :=$ **GLS**$(G,S(P,R),L,K)$; /*return best solution so far and zero-mobility confidence ZMC */
$S := S_{best}$; $New\_Solution{:=}True$;
$ElmSet :=$ **FindDominant**(G, ZMC); /* find set of dominant elements */
$ElmSet\_Bound :=$ **FindBound**$(ElmSet,G)$; /* find least activity of dominant elements */
$ElmSet\_Edge := \emptyset$; /* initially no pseudo-edges */
$cp_0 :=$ **Cost(**S_{best}**)**/100 ; /* initial value of control parameter */
$cp := cp_0$; /* control parameter initialized to cp_0 */

(2) **Evolution:**

For i := 1 to $MaxIter$ **do**
Begin
$C_{pre} :=$ **Cost(**S**)**;
Evaluation phase:
If $New\_Solution$ **then** /* do evaluation only if new solution is admitted */
Begin
$ElmSet\_Goodness{:=}$**FindPerformance**$(ElmSet,S,ElmSet\_Bound)$;
/* evaluate performance of elements and their goodness */
$ElmSet\_Tight :=$ **FindTight**$(ElmSet,S)$; /* find placement of each element */
$New\_Solution{:=}False$;
End
Evolutionary phase:
For each $elm \in ElmSet$ **do** /* do evolutionary phase for each element */
Element_Evolution $(ElmSet\_Tight(elm),ElmSet\_Edge(elm),ElmSet\_Goodness(elm))$
end
Mutation phase:
Mutation$(ElmSet,ElmSet\_Edge)$ /* do mutation of the set of elements */
Regeneration phase:
$S_{temp} :=$ **GLS**$(G,S(P,R),ElmSet\_Edge)$;
Selection phase:
$Gain :=$ **Cost(**S_{best}**)** - **Cost(**S_{temp}**)**; /* find gain */
If $(Gain > 0)$ **then** $S_{best} := S_{temp}$; /* a better solution is found */
If (**Cost(**S_t**)** $= C_{pre}$) **then** $cp := f(cp)$ /* $f(cp) = 1.2 \times cp$ */
else $cp := cp_0$;
If $(Gain >$ Random$(-cp, 0))$ **then**
Begin
$S := S_{temp}$; /* accept new solution */
$New\_Solution{:=}True$;
End;
Return(S_{best});
End.

Figure 11: Evolution-Based Edge Addition Scheduling

of *ElmSet* together with its goodness value and tightness state and decides whether the pseudo-edge should be: (1) set, (2) removed, or (3) left unchanged. This procedure updates the state of edges *ElmSet_Edge*.

In the *mutation phase*, the edge state of a randomly selected element is switched. For example if the selected element *elm* is found with an already set pseudo-edge, then we delete this pseudo-edge with some probability.

In the *regeneration phase* we use a procedure *GLS* to carry out K iterative scheduling of task graph G over system $S(P, R)$ after augmenting G with the set of pseudo-edges defined in *ElmSet_Edge*. This allows searching for the optimum solution starting from the current solution S and enforcing the scheduling decisions as dictated by the pseudo-edges. Some added pseudo-edges may not be valid because they introduce cycling in the graph. In this case we reject all the edges that were added to current solution and restart again the evolutionary phase with the current solution. To save time, the detection is done within the scheduler when the set of tasks that are ready for next scheduling is empty but there are still unscheduled tasks. A task that is ready for scheduling has all its predecessors already assigned. If there is no cycling procedure *GLS* returns the solution with the shortest finish time (S_{temp}).

In the *Selection phase* we compare the cost of the best solution found S_temp with cost of current best solution S_{best} to find out whether the simulated evolution should accept S_{temp}. If cost of S_{temp} is lesser than that of S_{best}, then we accept the new solution as well as the present set of pseudo-edges. This means that S_{temp} and current edge state values *ElmSet_Edge* become a starting point for the evolutionary iteration. We increase the control parameter if the cost of S_{temp} is equal to cost of previous solution (S_{prev}). Finally, we accept the new solution S_{temp} even with non-positive gain if the new solution goodness is worst than current best solution by at most q, where q is obtained by generating a random integer between 0 and $-cp$. We restart the evolutionary section if a new solution is accepted but we skip the evaluation phase if no solution accepted. If no solution is accepted, we restart the evolutionary phase with the current solution. The solution is returned after we have exhausted $MaxIter = 80$ iterations. We experimentally determined [3] that running beyond $MaxIter = 80$ is unlikely to find solutions with shorter finish times for the set of studied instances of task parallelism and communication granularity.

The complexity of ETF [9], DLS [18], and GLS [2] is $O(pn^2)$, where n and p are the number of tasks and the number of processors, respectively. The initialization of edge addition SE (step 1 of $SE-schedule$) is dominated by $O(pn^2K)$, where K is the number of forward-backward iterations of GLS and $O(pn^2)$ is the complexity of one GLS scheduling iteration. The main loop of edge addition SE (step 2 of $SE - schedule$) is dominated by $O(p \times n^2 \times K \times MaxIter)$, where $MaxIter$ is the allowed number of iterations.

6 Performance evaluation

The objective is to compare performance of local scheduling heuristics and the proposed approach which is based on pre-evaluation of the task-priority and generalized list scheduling. We compare our approach to some scheduling heuristics which use bounded number of processors which are: ETF [9], DLS [18], and GLS [2], and our edge addition SE scheduler.

A *random graph generator* (RGG) is used for generating computation graphs with few hundred tasks and with task computation time ranging from 10 to 190 time units.

The average communication cost, number of levels, and the number of processors are indirectly controlled using three parameters. The average communication to the average computation is denoted by $\alpha = \Sigma_{T,T'} c(T',T) r_{min} \,/\, \Sigma_T \mu(T) = c_{edge}/\mu_T$, where r_{min} is the least time to transfer a unit of data between two processors (set to 1).

The graph parallelism is the average number of tasks that can be made ready to run at the same time. This can be measured by using the ratio of the sum of all computation times in the problem over the sum of computation times along the longest chain ($X_{longest}$). $X_{longest}$ is a chain of immediate tasks starting at entry node and ending at exit node such that the sum of all its task times is the largest among all available chains. In other words, the graph parallelism is $\sum_{T\in\Gamma} \mu(T) / \sum_{T\in X_{longest}} \mu(T)$. We define the degree of parallelism (β) as the task graph parallelism over the number of processors (p):

$$\beta = \frac{\sum_{T\in\Gamma} \mu(T)}{p \times \sum_{T\in X_{longest}} \mu(T)} \tag{3}$$

The degree of parallelism is an indicator of the average number of tasks that can be made ready per processor. It also indicates the average number of tasks that may compete for each processor. The simulator assumes a number of independent processors connected by using a fully connected network ($r(p,p') = 1$). The values of parameters studied are $\alpha \in [0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]$ and $\beta \in [1, 2, 2.5, 3, 4, 5]$. The variance on C_{edge} is set to 50% of the current average of C_{edge}. Each graph has at least 6 levels and 70% of the outgoing edges from one level are incoming edges to the next level and the remaining 30% reach arbitrary forward levels. For each instance of α, β, and topology (126 instances), the RGG uses the uniform distribution to generate 40 random computation graphs that are scheduled by each of the previously defined heuristics.

Using the RGG, the simulator used is schematically described by the flow-chart shown on Figure 12. The simulation uses as input the statistical profiles (communication and parallelism) of the graph problems to be generated. For each profile instance RGG generates 40 graph problems that are scheduled by each of the algorithms ETF, DLS, GLS, and our *SE-Schedule*. The length of shortest finish time solution is denoted by (ω_b). We store the relative percentage deviation from (ω_b) of each heuristic h, that is $(\omega_h/\omega_b - 1)100$, for each studied instance of communication (α) and parallelism (β). Each plotted point results from averaging the heuristic finish times for 40 generated problems. Figures 13, 14, 15, and 16 show the percentage deviation of the above scheduling approaches from best known solutions. Figure 17 shows the average number of iterations needed for our *SE-Schedule* to find its best solution.

ETF (Figure 13) can perform well only when there is enough task parallelism to cover available communication. However, ETF finish time significantly degrades when (1) the available parallelism is relatively low, or (2) the amount of communication is relatively large compared to available task parallelism. The results is that the schedule length of ETF may deviate by more than 20% from best known solution for the studied ranges of α and β.

DLS (Figure 14) uses a simple but more balanced decision function ($d(T) = DL(T) - est(T,p)$) than ETF which is rewarded by noticeable improvement in performance especially when the parallelism is low. DLS may deviate by more than 10% from best known solution especially when communication (α) is relatively large compared to available task parallelism (β). However, the finish time of DLS schedules seem to be much less sensitive to computation profile than in the case of ETF. Like ETF, the heuristic DLS gives its

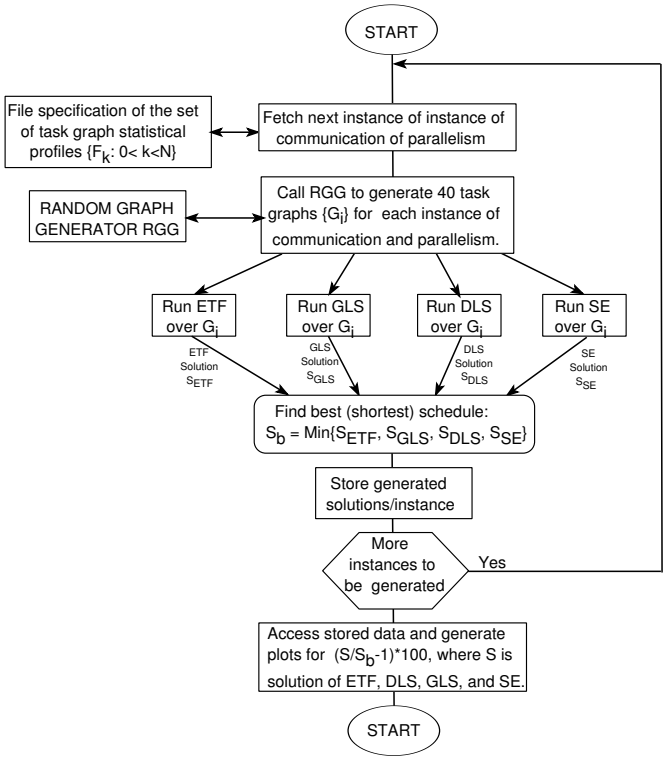


Figure 12: Flowchart of testing ETF, DLS, GLS, and SE by using the RGG

best when the parallelism is large enough to cover the needed communications. The time spent by DLS to find its solution is nearly 2 times the time spent by ETF because DLS needs one pass on the task graph to compute the task-levels prior to scheduling.

GLS (Figure 15) uses has a decision function that incorporates a more effective task-level than DLS in addition to the traditional earliest-startable-task discipline. GLS deviates by about 5% from best known solutions. Here, the number of iterations is 10 for each run of GLS which means that the time spent by GLS to find its solution is 10 times and 5 times the time spent by ETF and DLS, respectively. Its worst deviation is recorded when the communication are relatively large and task parallelism is relatively low. However, it can be considered as near optimum for computation having low communication and large task parallelism. We believe that the reason for this due to: (1) the use of a task-level ($ct(T)$ of previous iteration) that accounts for the computations and communications, and (2) its iterative nature which allows it to sharpen its solution.

The proposed *SE-Schedule* (Figure 11) has the lowest average deviation from best known solutions compare to ETF, DLS, and GLS. It outperforms GLS by a few percents only because GLS already generates near-optimum schedules in most cases. However, GLS does not provide full guarantee of performance because of its slight dependence on the computation profile. For example, GLS may have greater deviation if task-parallelism is limited in the presence of large communication. Thus *SE-Schedule* is one tool to complement GLS and to optimize its solutions especially for difficult instances such as large average communication compared to average computation.

There are a number of observations on the best schedules generated by *SE-Schedule*. A graph giving its best in a forward run would keep doing so irrespective of the iterations we do on it. The same will happen for a graph giving its best in backward scheduling.

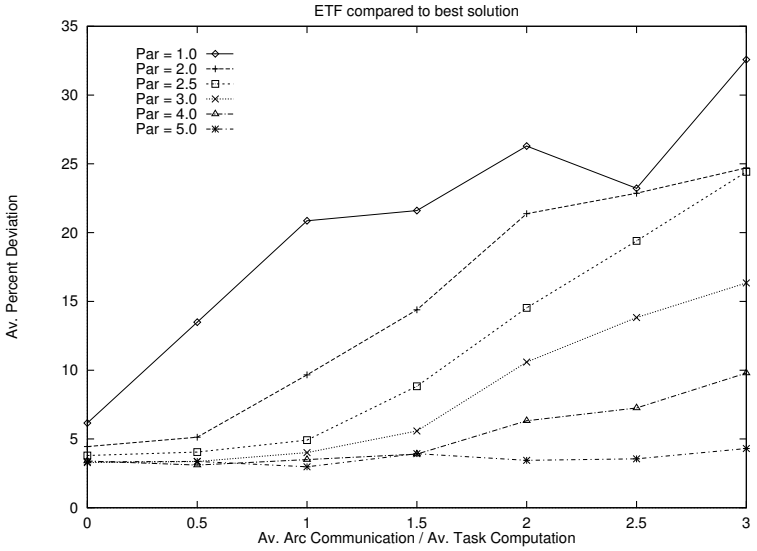


Figure 13: Percentage deviation of ETF schedules from best known solutions

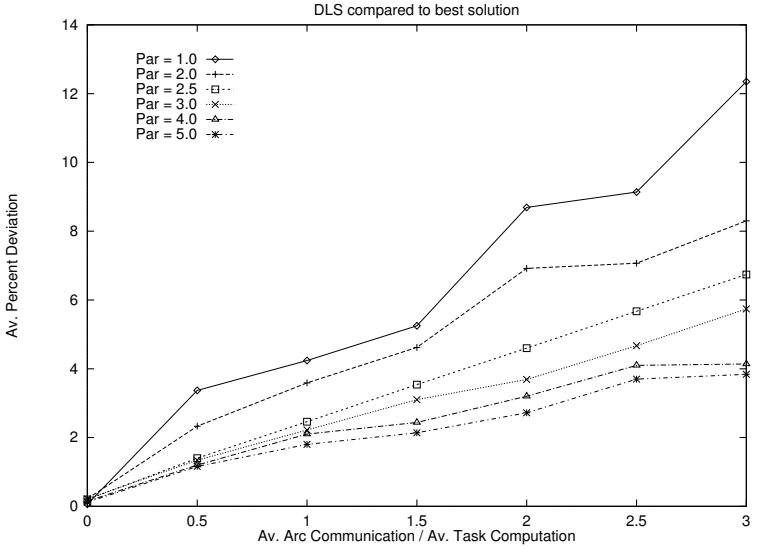


Figure 14: Percentage deviation of DLS schedules from best known solutions

This categorizes our graphs in two groups; ones which give their best solution in forward schedules and the others which do so in reverse schedules. The edge addition operations are found to be different for each group. The addition of pseudo-edges was further subdivided into two categories. It was observed that we should concentrate on tasks which are in the first level for task graph which are of the forward type and on bottom level tasks for the graphs which give their best on the reverse schedule.

The number of iterations $N_{SE-Schedule}$ needed for *SE-Schedule* to generate its the best solution is shown on Figure 17. The function $N_{SE-Schedule}$ depends on the computation profile. Computation with larger communications require more iterations because there are many edge addition, that significantly affect the finish time, to be tried by *SE-Schedule*. On the other hand, increasing task parallelism helps shortening of the number of iterations because it provides more alternatives (more freedom) to cover communications. Figure 18 summarizes overall results across all values of parallelism and communications.

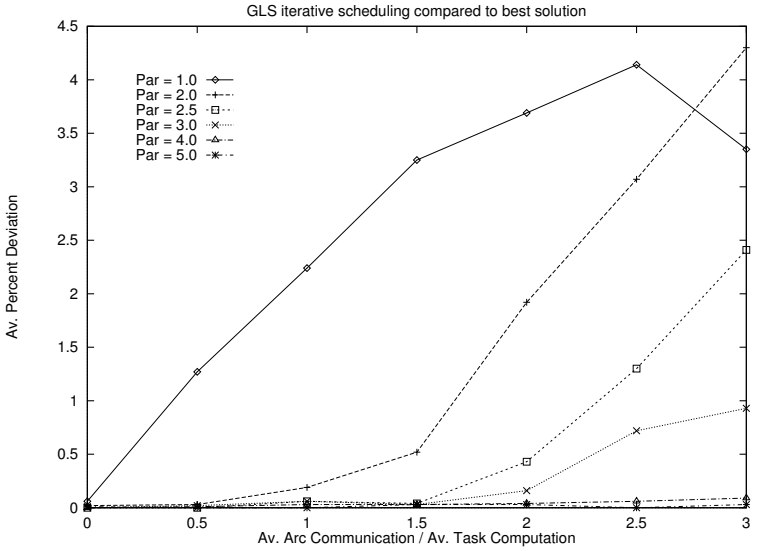


Figure 15: Percentage deviation of GLS schedules from best known solutions

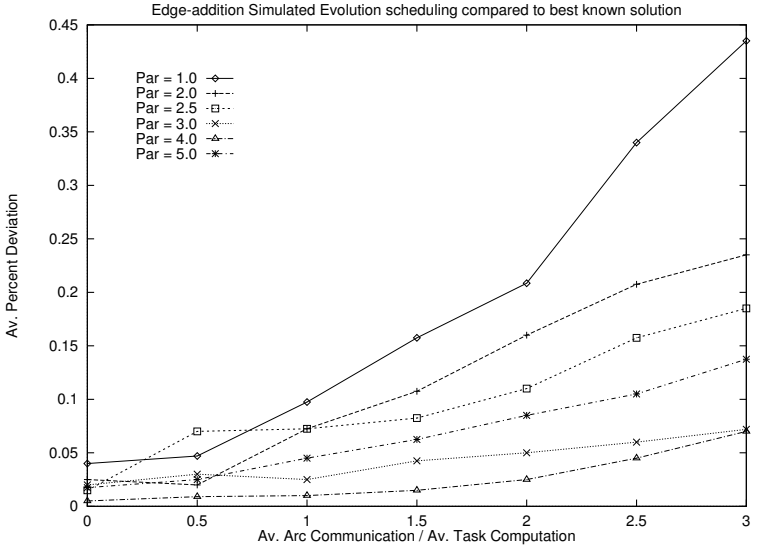


Figure 16: Percentage deviation of Edge Addition SE schedules from best known solutions

We study the time spent in each iteration of *SE-Schedule*. We refer to the main loop of the edge addition SE algorithm shown on Figure 11. Generally, most of the dominant pair of tasks are properly aligned by the GLS and only a small fraction of these dominant pair of tasks need to be handled through SE mechanism (Elm). This explains why the running times of phases like the *evaluation*, *evolution*, and *mutation* are dominated by the running time of the *regeneration* (mainly GLS) because the number of elements (Elm) that are manipulated in the above phases is a small fraction of the total number of tasks. Thus the time spent in each iteration of *SE-Schedule* is largely dominated by the running time of GLS. On the other hand, GLS can be considered as equivalent to running ETF for K iterations. Due to computation of static levels, each run of DLS is about the running of two iterations of ETF. In other words, the time spent by one iteration of *SE-Schedule* is nearly 1:1, 1:5, and 1:10 the time spent by GLS, DLS, and ETF, respectively.

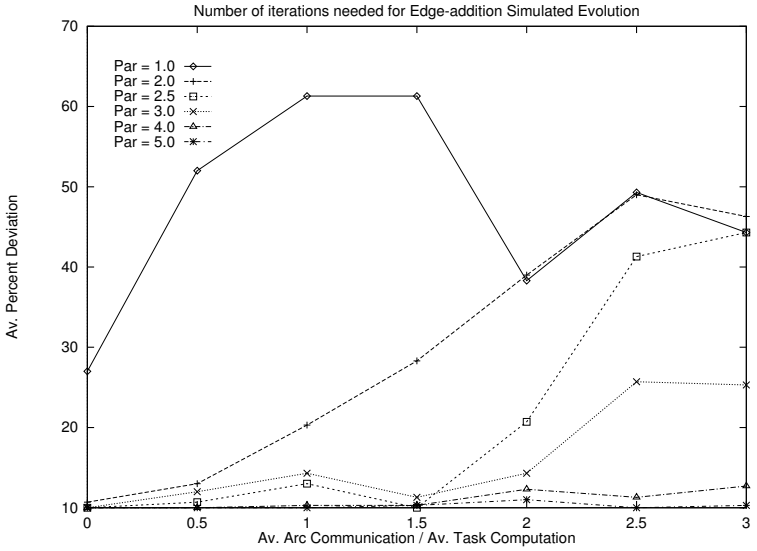


Figure 17: Number of iterations for Edge Addition SE scheduling

7 Conclusion

Synchronous dataflow computations is a class of programs that are generally compiled once and run many times over different data sets. In this paper we presented a compiler optimization approach that uses the *simulated evolution* SE paradigm to enhance the finish time of heuristically scheduled synchronous dataflow computations with communication times. Unlike genetic approaches which generally use task swapping to create differential variations our approach consists of adding pseudo-edges to the task graph to guide the scheduler in the alignment and clustering of dominant tasks. Edge addition SE provides a flexible formulation to the optimization problems of a class of graph-based computations because added edges alter only the task graph without modifying the heuristic that finds guided solutions.

The edge addition SE scheduling is useful to complement scheduling heuristics whose finish time performance cannot be guaranteed for arbitrary communication granularity and parallelism profile. For example the finish time of ETF solutions deviates by more than 20% from optimum when task parallelism is not large enough to cover the communication [3]. In general, heuristic solutions do not provide reliable performance. Implementing the edge addition SE on the top of a scheduling heuristic like GLS enables optimizing the solution finish time as well as statistically guaranteeing that the SE solution outperforms other solutions generated by heuristics like ETF, DLS, and GLS. Although these heuristics are capable of finding good solutions optimizing the solution through edge addition SE search provides more confidence in the quality of the solution.

The edge addition SE scheduler is a flexible compiler optimization approach that is capable of finding *near-optimum schedules* as well as *providing higher confidence* in the performance of profile-sensitive scheduling heuristics.

8 Acknowledgments

The author acknowledges Mr. Nadim Mohsen for his work in the design of a primary version of the Simulated Evolution Algorithm. The author thanks the Department of Computer Engineering, College of Computer Science and Engineering, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia for its computing facilities.

Heuristic	*% of Cases*	*% Deviation*
ETF = DLS	3.1	0.00
ETF better than DLS	17.3	2.23
DLS better than ETF	79.6	8.18
ETF = GLS	1.7	0.00
ETF better than GLS	0.06	0.23
GLS better than ETF	98.24	11.83
ETF = Edge Addition SE	1.70	0.00
ETF better than Edge Addition SE	0.00	0.00
Edge Addition SE better than ETF	98.30	12.40
DLS = GLS	16.76	0.00
DLS better than GLS	1.86	0.29
GLS better than DLS	81.38	7.23
DLS = Edge Addition SE	3.57	0.00
DLS better than Edge Addition SE	0.06	0.04
Edge Addition SE better than DLS	96.37	9.25
GLS = Edge Addition SE	34.49	0.00
GLS better than Edge Addition SE	0.41	1.05
Edge Addition SE better than GLS	65.10	3.72

Figure 18: Pairwise comparison between heuristics on fully-connected topology

References

[1] Tai A. Ly and Jack T. Mowchenko. Applying simulated evolution to high level synthesis. *IEEE Trans. on Computer-Aided Design of Circuits and Systems*, 12, No 3:389–408, Mar 1993.

[2] M. Al-Mouhamed and A. Al-Maasarani. Performance evaluation of scheduling precedence-constrained computation on message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 5, No 12:1317–1322, December 1994.

[3] M. Al-Mouhamed and H. Najjari. Adaptive scheduling of computations and communications on distributed memory systems. *Proceedings of the Inter. Conf. on Parallel Architectures and Compilation Techniques, pp. 366-373*, 1998.

[4] B. Falsafi and D.A. Wood. Scheduling communication on an SMP node parallel machine. *Third Inter. Symp. on High-Performance Computer Architecture, pp. 128-138*, 1997.

[5] Youssef G. Saab and Vasant B. Rao. Combinatorial optimization by stochastic evolution. *IEEE Trans. on Computer-Aided Design*, 10, No 4:525–535, Apr 1991.

[6] R.L. Graham and E. Coffman. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.

[7] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[8] N. Hou, E.S.H. Ansari and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. on Parallel and Distributed Systems*, 5, No 2:113–120, Feb 1994.

[9] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times, pp. 244-257. *SIAM Journal on Computing*, Apr 1989.

[10] S.J. Kim and J.C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. *Proc. of the Inter. Conf. on Parallel Processing*, 3:1–8, Aug 1988.

[11] S. Kirkpatrick, Jr. C. Gelatt, and M Vecchi. Optimization by simulated annealing. *Science*, No 220(4598):498–516, May 1983.

[12] R. M. Kling and P. Banerjee. ESP: Placement by simulated evolution. *IEEE Trans. on Computer-Aided Design*, Vol 8, No 3:245–256, Mar 1989.

[13] R. M. Kling and P. Banerjee. Empirical and theoretical studies of the simulated evolution method applied to standard cell placement. *IEEE Trans. on Computer-Aided Design*, Vol 10, No 10:1303–1315, Oct 1991.

[14] B. Kruatrachue. Static task scheduling and grain packing in parallel processing systems. *Ph.D. Thesis, Department of Computer Science*, 1987. Oregon State University.

[15] P.Y. Richard Ma, E.Y.S. Lee, and T. Masahiro. A task allocation model for distributed computing systems. *IEEE Trans. on Computers*, C-31:41–47, Jan 1982.

[16] E. Rosti, E. Smirni, and L.W. Dowdy. Processor saving scheduling policies for multiprocessor systems. *IEEE Trans. on Computers*, 47, No:178–189, Feb. 1998.

[17] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. *Proc. of the SIGPLAN Symp. on Compiler Construction, pp. 17-26*, Jul 1986.

[18] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. on Parallel and Distributed Systems*, 10, No 2:175–187, Feb 1993.

[19] T. Yamada, B.E. Rosen, and R. Nakano. A simultated annealing approach to job shop scheduling using critical block transition operators. *Inter. Conf. on Neural Network*, 7:4687–4692, 1994.

[20] T. Yang and A. Gerasoulis. DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5, No 3:951–967, Sep 1994.

NOMENCLATURE

Symbol	Meaning
SE	Simulated evolution
GA	Genetic algorithm
SA	Simulated annealing
S	Schedule solution
S_0	Initial solution
S_{best}	Best solution found so far (shortest finish time)
cp	Control parameter
$Gain$	Difference in cost between current and best solutions
$\Gamma(T_1, \ldots, T_n)$	Set of tasks $T_1, \ldots, T_n$
P	Set of processors
$T_1 \rightarrow T_2$	Precedence relationship (T_1 is a predecessor of T_2)
$\mu(T)$	Execution time of task T
$S(P, R)$	Set of processors P and their routing network R
$c(T, T')$	Number of messages produced by T for its successor T'
$r(p, p')$	Time to route a unit of message between processors p and p'
$Pred(T)$	Set of predecessor tasks of T
$p(T)$	Processor running task T
$ct(T, p(T))$	Completion time of T on $p(T)$
$est(T, p(T))$	Earliest starting time of T on $p(T)$
EST	A scheduling heuristic called Earliest-Task-First
DLS	A scheduling heuristic called Dynamic Level Scheduling
GLS	A scheduling heuristic called Generalized List Scheduling
$elm(T_1 \rightarrow T_2)$	Sub-graph formed by $T_1 \rightarrow T_2$
$m(T)$	Maximum mobility of task T
$am(T)$	Average mobility of task T
$st_e(T)$	Earliest time of T without delaying the schedule
$st_e(T)$	Latest time of T without delaying the schedule
$d_i(T)$	Value of decision function applied to task T at the ith iteration
$Edge$	A Boolean used to state whether a directed edge is set or not
$st(T)$	Starting time of T
$lmt(T_1, T_2)$	Time at which the latest message from T_1 reaches T_2