# Adaptive Scheduling of Computations and Communications on Distributed Memory Systems

Mayez Al-Mouhamed and Homam Najjari
Computer Engineering Department
CCSE, King Fahd University
Dhahran 31261, Saudi Arabia.
mayez,homam@ccse.kfupm.edu.sa

## Abstract

Compile-time scheduling is one approach to extract parallelism which proved to be effective when the execution behavior is predictable. Unfortunately, the performance of most priority-based scheduling algorithms is computation dependent. Scheduling by using *earliest-task-first* (ETF) produces reasonably short schedules only when available parallelism is large enough to cover the communications. A priority-based decision is much more effective when parallelism is low. We propose a scheduling in which the decision function combines: (1) task-level as global priority, and (2) *earliest-task-first* as local priority. The degree of dominance of one of the above concepts is controlled by the available the computation profile such as task parallelism and communication. An iterative scheduler (forward and backward) is proposed for tuning the solution. In each iteration, the new schedule is used to sharpen the task-levels which contribute in finding shorter schedules in next iteration. Evaluation is carried out for a wide category of computations with communications for which optimum schedules are known. It is found that pure local scheduling (like ETF) and static priority-based scheduling significantly deviate from optimum under specific problem instances. Our approach to adapting the scheduling decision to computation profile was able to produce near-optimum solutions through much less number of iterations than other approaches.

## 1 Introduction

Deterministic scheduling can be profitable when the execution behavior can be made predictable at compile-time. The compiler determines the dependencies and estimates the computation and communication requirements which are used to produce a schedule that better matches the underlying parallel hardware.

The problem of minimizing schedule finish time of computations and communications is one NP-complete problem [1]. Different approaches have been proposed which can be classified into the following categories: (1) *search-based*, (2) *task-duplication*, (3) *clustering*, and (4) *priority-based* scheduling.

Search-based methods like branch-and-bound [7], simulated annealing [9], and genetic [2] were proposed for finding good mapping and partitioning of computations. Scheduling based on *task duplication* over idle processors was proposed [5] to reduce the communication without excessively increasing overhead in managing duplicated data.

*Clustering* over unbounded number of processors [8] 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) [11] is a low complexity clustering that accepts merging of a task to a cluster only if the length of the dominant chain to which the task belongs to decreases.

The *dynamic critical path* (DCP) was proposed in [6]. The DCP is a chain of immediate tasks having zero mobility. For each processor having a predecessor of a DCP task $T$, the algorithm searches a vacant slot to fit $T$ while allowing other tasks to move within the limit of their mobilities. Processor selection is based on looking for the potential start times of remaining tasks on each processor which guarantees some processor reservation for the most critical successor.

Example of priority-based scheduling that operate over bounded number of processors are the *dynamic level scheduling* (DLS) [10] and *earliest-task-first* (ETF) [3]. In ETF task and processor selection are based on finding the earliest startable task and its best suited processor. Its main strategy is the knowledge of local task starting times which are used to minimize processor idle times by trying to maximize the overlap between computation and communication. In DLS, the largest sum of computations from a task to exit is considered as *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. Unfortunately, the evaluation of static task-level 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.

Our investigation reveals that ETF is capable of producing acceptable solutions with reasonable complexity when there is enough task parallelism in the computation to hide the communication. In other words, the performance of ETF is sensitive to the ratio of available task parallelism over effective communication. ETF and DLS produce excessively long schedules when task parallelism is not sufficient to hide the communications. There is no performance guarantee for these schedulings.

Our objective is to learn from the above algorithms and to design a scheduling algorithm that adapt its decision by using some profiling information from the computation. This method will prove to be effective when performance must be made insensitive to variation in parallelism and communication. In other words, we propose an iterative scheduling that smoothly adapts its decision function to some profiling information. Two objectives are targeted in each iteration which are: (1) producing a schedule based on so far accumulated knowledge of computation and communication, and (2) instantaneously exploits the task assignment to sharpen the knowledge. This approach primarily applies to static scheduling but also provides useful information for dynamic situations where no global knowledge of computation is available. Analysis and evaluation will show that adaptive scheduling produces near-optimum solutions and contributes to a better understanding of the scheduling problem. Its iterative nature is useful as a compiler optimization approach.

The organization of this paper is as follows. Section 2 presents some background. Section 3 presents a model on performance degradation. Section 4 presents the evaluation of task-level. Section 5 presents the proposed scheduling. Section 6 presents the evaluation. We conclude this work in Section 7.

## 2 Background

A set of $\Gamma(T_1, \ldots, T_n)$ of $n$ tasks ($T$) with their precedence constraints and communication costs are to be scheduled on $p$ identical processors so that their overall execution time is held to a minimum. The computation can be modeled [3] 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 size of message sent from $T$ to its successor $T'$, respectively. The multiprocessor is denoted by $S(P, R)$ where $p \in P$ is processor and $r(p, p') \in R$ is the bandwidth of data path between $p$ and $p'$ which is bound by $r_{max}$. The reference time to the transfer of message $c(T, T')$ is $c(T, T') \times r(p, p')$, where $p$ and $p'$ are the processors running $T$ and $T'$, respectively. Local message transfer has zero cost ($r(p, p) = 0$).

Let $T$ be a task and denote by $Pred(T)$ the set of predecessors of $T$. The *earliest-starting-time* ($est(T, p)$) of $T$ on $p$ is the earliest time the latest message from the predecessors arrives to $p(T)$:

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

where $ct(T', p')$ is the completion time of $T'$ on $p'$. Note that $est(T, p)$ is nil for each $p$ if $T$ has no predecessors.

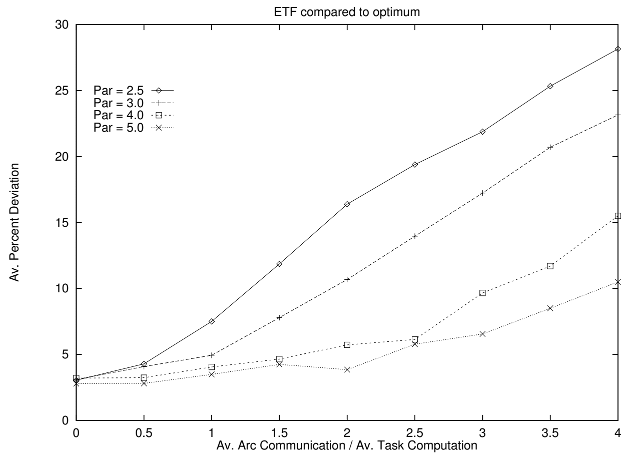


Figure 1: Percentage deviation of ETF schedules from optimum solutions

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:

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

The $est(T, p)$ provides an objective function for minimizing processor idle times by selecting tasks and processors according to *earliest-task-first* (ETF) [3]. ETF uses Graham's list-scheduling [1] 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 enables finding a worst-case bound [3] on schedule length.

Evaluating static task priority for computation with communication times is difficult because the sum of communication along directed paths cannot be determined without the knowledge of task-processor mapping. Thus, on-line task and processor selection in ETF is one advantage because it eliminates the need for any static pre-processing like off-line computation of task priority. ETF schedules gave acceptable deviation from some known solutions as reported in [10, 6]. Some anomalies were also reported. For example increasing parallelism or reducing communication always improves ETF schedule length. The average running time of ETF appears to be reasonable [6] when compared to other well known algorithms.

To minimize overall schedule finish time ETF locally minimizes processor idle time by searching opportunities to overlap computations with communications. However, the efficiency of this strategy drops when: (1) there is not enough inherent task parallelism to hyde communications, or (2) there is excessive communication that cannot be hidden anyway. Task parallelism is the average number of parallel tasks per processor.

ETF schedules suffer from an important flaw. Figure 1 shows the percentage deviation of ETF schedules from optimum solutions versus the communication granularity and the inherent parallelism. Different performance levels are obtained depending on computation profile. Specifically, ETF is capable of generating reasonably good solutions only when there is sufficiently large parallelism to hyde the communication. In this case, no guaranteed performance can be advised. Our objective is to find a strategy that smooths out the variations in performance by adapting the scheduling decision to the amount of communication and parallelism.

In the next section we present analytical explanation for the performance degradation of ETF which will be used latter as the basis for the design of an adaptive scheduling algorithm. In the next section we establish a relation that can be used to predict the deviation from optimum of an ETF schedule as function of communication and parallelism.

## 3 Modelling the algorithm degradation

In his study of multiprocessor anomalies, Graham [1] showed that the sum of all idle times in a schedule with no communications is bounded by the sum of all computations along some chain of tasks. The bound was adapted later in [3] to incorporate non-zero communication times. The main result is that the sum of all idle times in a schedule generated by ETF is bounded by the sum of enough computations and communications along one specific chain of tasks. We use this idea to show how the schedule length can be affected by the communication and parallelism. For this we first shortly review the derivation of the bound on the idle times and then introduce the effect of average communication and parallelism.

Assume a schedule generated by applying *earliest-task-first* as scheduling heuristic and let $\omega$ be the finish time of the schedule. The set of time points in $(0, \omega)$ can be partitioned into two subsets $A$ and $B$ that consist of all the time points for which: (1) all processors are busy ($A$), and (2) at least one processor is idle ($B$). $B$ is the disjoint union of $q$ open intervals $B = \cup_{1 \le i \le q}(b_{l_i}, b_{r_i})$ and $b_{l_1} < b_{r_1} < \ldots < b_{l_i} < b_{r_i} < \ldots < b_{l_q} < b_{r_q}$. The earliest-task-first scheduling allows finding a chain of tasks $X : T_L \rightarrow T_{L-1} \ldots T_2 \rightarrow T_1$ that entirely covers $B$, where $T_1$ is among the tasks that complete last in the schedule. The principle of earliest-task-first enforces the starting times of successively scheduled tasks be a non-decreasing sequence in time. This in turn allows finding a bound on the sum of all idle time intervals ($\mu(\phi)$) in the schedule:

$$\sum_{\phi_i \in \Phi} \mu(\phi_i) \le (p-1) \sum_{j=1}^{L} \mu(T_j) + p r_{max} \sum_{j=1}^{L-1} c(T_{j+1}, T_j) \quad (3)$$

where the left-hand sum covers all idle times ($\mu(\phi_i)$). Consider a chain of immediate tasks ($X_{large}$) whose sum of computations accumulates the largest value among all other chains. Since $\sum_{j=1}^{L} \mu(T_j) \le \sum_{T \in X_{large}} \mu(T)$, then idle times $\sum_{\phi_i \in \Phi} \mu(\phi_i)$ are bounded by:

$$\frac{p}{k} \left( 1 + r_{max} \frac{\frac{1}{L-1} \sum_{j=1}^{L-1} c(T_{j+1}, T_j)}{\frac{1}{L} \sum_{j=1}^{L} \mu(T_j)} \right) \sum_{T \in X_{large}} \mu(T) \quad (4)$$

where $k$ is the largest integer satisfying $k \times \sum_{j=1}^{L} \mu(T_j) \le \sum_{T \in X_{large}} \mu(T)$. To identify some general features of ETF we need to consider a class of computations $\mathcal{G}$ for which $\mu(T)$ and $c(T, T')$ are uniformly distributed within some specified ranges. A given computation graph $G \in \mathcal{G}$ can be characterized by means of two parameters: 1) the *communication granularity* ($\alpha$), and 2) the *degree of parallelism* ($\beta$). Parameter $\alpha$ is defined as the ratio of average communication ($c$) to average computation ($\mu$) that is:

$$\alpha = \frac{\frac{1}{n_{edge}} \sum_{T \rightarrow T' \in G} c(T, T')}{\frac{1}{n} \sum_{T \in \Gamma} \mu(T)} \quad (5)$$

where $n$ and $n_{edge}$ are the number of tasks and the number of non-zero communication edges, respectively. 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 computations over the sum of computations along the longest chain that is $(\sum_{T \in \Gamma} \mu(T))/(\sum_{T \in X_{large}} \mu(T))$. We define the degree of parallelism $\beta$ as the graph parallelism over the number of processors:

$$\beta = \frac{\sum_{T \in \Gamma} \mu(T)}{p \sum_{T \in X_{large}} \mu(T)} \quad (6)$$

In other term, $\beta$ is an indicator of the average number of tasks that compete for scheduling on each processor. Let's $\alpha_{max}$ be an upper bound on the communication granularity for a given computation. Since the communications are uniformly distributed then average communication along partial chains is also bounded by $\alpha_{max}$. Then the bound given in Equation 4 becomes:

$$\sum_{\phi_i \in \Phi} \mu(\phi_i) \le \left( \frac{1 + r_{max}\alpha_{max}}{k\beta} \right) \sum_{T \in \Gamma} \mu(T) \quad (7)$$

This indicates that $(1 + r_{max}\alpha_{max})/k\beta$ is an upper bound on the percentage of idle time in the schedule. The schedule finish time $\omega$ is the sum of all computations and all idle times $\omega \times p = \sum_{T \in \Gamma} \mu(T) + \sum_{\phi_i \in \Phi} \mu(\phi_i)$, then the schedule finish time is bound by:

$$\omega \le \frac{1}{p} \left( 1 + \frac{1 + r_{max}\alpha_{max}}{k\beta} \right) \sum_{T \in \Gamma} \mu(T) \quad (8)$$

The length of optimum solution $\omega_{opt}$ always satisfies $\sum_{T \in \Gamma} \mu(T) \le p\omega_{opt}$, then the schedule length becomes $\omega \le \omega_{opt} \left( 1 + \frac{1 + r_{max}\alpha_{max}}{k\beta} \right)$. This allows finding a bound on the relative deviation of schedule length from length of optimum solution:

$$\frac{\omega - \omega_{opt}}{\omega_{opt}} \le \frac{1 + r_{max}\alpha_{max}}{k\beta} \quad (9)$$

The sum of idle times due to precedence relationships represents a fraction of schedule time that is bounded by $1/k\beta$. Thus large parallelism may completely hide the effects of task precedence. The sum of the idle times due to non-zero communication edges represents a fraction of schedule time that is bounded by $r_{max}\alpha_{max}/k\beta$. The finish time $\omega$ increases at most linearly with increase in $\alpha_{max}$ and $r_{max}$. This explain why the schedule finish time is near optimum only when there is large amount of parallelism to hide the communications. It is clear that a pure *earliest-task-first* decision will cause degradation when $r_{max}\alpha_{max}/k\beta$ is large. In this case, only a small fraction of communication can be hidden by task execution and a better decision is to select tasks that are followed by longer chain of computation and communications, i.e. tasks with higher task-level. Thus task and processor selection must include provision for both *earliest-task-first* and *highest-level-first*.

The above bound and the experimental testing (Figure 1) shows that the above bound can be used to model the deviation $(1+r_{max}\alpha_{max})/\beta)$ from optimum. The model allows to predict possible schedule degradation when the parallelism and communication are known.

## 4 The task level

We estimate the task-level based on how the scheduler maps task computations and communications in a schedule. The computed task-levels can then be used in the task and processor selection of the same scheduler in the next scheduling iteration. The task-level of a given task is computed by using the assignment of predecessors and the implied communication.

Consider a chain of immediate tasks $Y : T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T$, where $T_1$ is an entry task in $G$ and $T$ is any task that is linked to $T_1$ by a dependence chain. The assignment of tasks of path $Y$ in a given schedule depends on the number of available processors, the inherent parallelism in $G$, and the scheduling heuristic used. Given a schedule, we define the *longest activity path* ($lap(T)$) of $T$ as the largest sum of non-overlapped time intervals during which some computation and communication are carried out for some (immediate or not) predecessors of $T$.

Intuitively, we must have $lap(T) = 0$ whenever $T$ is an entry node. A time interval $\Delta t$ should not affect $lap(T)$ if $\Delta t$ occurs: (1) prior to the start of any predecessor of $T$, and (2) following the completion of $T$. $lap(T)$ may be affected by $\Delta t$ that occurs between the starting of some $T_i \in Y$ and that of $T$. In the following we present a recursive evaluation of task-level $lap(T)$ which requires: (1) $lap(T')$ for every predecessor $T'$ of $T$, (2) starting time $st(T')$ and $p'$, and (3) processor $p$ on which $T$ is assigned.

**Definition 1** *The activity interval $act(T', T)$ from task $T'$ to its successor $T$ is the sum of all non-overlapped time intervals $\Delta t \in (st(T'), st(T))$ during which some computation $\mu(T)$ or communication $c(T', T)$ are carried out by arbitrary predecessors of $T$.*

The activity interval accounts for all time points in $[st(T'), st(T,p)]$ during which there is at least: one processor computing a predecessor of $T$, or one communication link transferring data to enable starting of $T$. The decision of scheduling $T$ could have been delayed beyond time point $est(T,p)$ because some other tasks were more prior according to the scheduling decision. Intuitively, we see that $lap(T)$ should not account for the delay $st(T) - est(T, p(T))$ which indicates that $lap(T)$ must not incorporate time points beyond: 1) the latest predecessor completion time or, 2) the latest predecessor message time. The $lap(T)$ depends on the largest sum of $lap$ values that is carried by a predecessor $T_i$ and its activity interval $act(T_i, T)$:

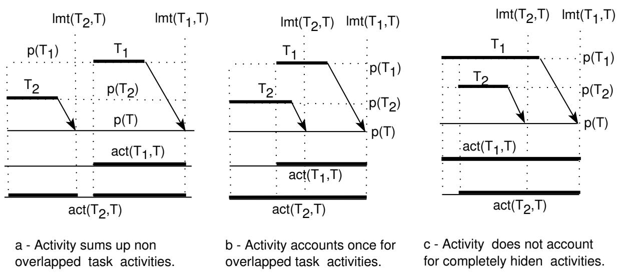

lmt(T2,T)
lmt(T1,T)
p(T1)
T1
T2
p(T2)
p(T)
act(T1,T)
act(T2,T)
a - Activity sums up non overlapped task activities.
lmt(T2,T)
lmt(T1,T)
T1
p(T1)
T2
p(T2)
p(T)
act(T1,T)
act(T2,T)
b - Activity accounts once for overlapped task activities.
lmt(T2,T)
lmt(T1,T)
T1
p(T1)
T2
p(T2)
p(T)
act(T1,T)
act(T2,T)
c - Activity does not account for completely hiden activities.

Figure 2: Task level as the sum of predecessors' activities

$$lap(T) = Max_{T_i \in pred(T)}\{lap(T_i) + act(T_i, T)\} \qquad (10)$$

Denote by $T'$ an immediate predecessor of $T$ that is started at the earliest time among all other predecessors of $T$. Denote by $lmt(T_j, T) = st(T_j) + \mu(T_j) + c(T_j, T) \times r(p_j, p)$ the earliest completion time of $T_j$ if $p_j = p$, or the earliest time all messages $c(T_j, T)$ from $T_j$ reach processor $p$ if $p_j \neq p$. The earliest time $T$ can start on $p$ is $est(T,p) = \max_{T_j \in Pred(T)}\{lmt(T_j, T)\}$. Notice that the actual starting time of $T$ always satisfies $st(T,p) \geq est(T,p)$. To evaluate $lap(T)$ we need to sort the predecessors $T_j \in Pred(T)$ in the decreasing order of their $lmt(T_j, T)$ which facilitates evaluation of activity intervals. Figure 2 shows the evaluation of activity assuming a task $T$ having $T_1$ and $T_2$ as predecessors. We have $lap(T) = max\{lap(T_1) + act(T_1, T), lap(T_2) + act(T_2, T)\}$. Figure 2-a shows that $act(T_1, T)$ must be entirely included in $act(T_2, T)$ due to the non-overlap. Figure 2-b shows that $act(T_2, T)$ includes only a fraction of the activities of $T_1$. Figure 2-c shows that the activity induced by $T_2$ is completely covered by that of $T_1$.

An $O(n)$ algorithm to evaluate the *Longest Activity Path* (LAP) of $T$ is given below, where $n$ is number of predecessors of $T$. LAP assumes the predecessors $Pred(T)$ are sorted in decreasing order of $lmt(T_i, T)$ and stored into a heap $H$. It starts by evaluating the current $lap$ ($current\_lap$) of $T$ and current activity ($current\_act$) induced by the predecessor with the largest $lmt$. At each step, it removes the next predecessor $T'$ from $H$, if $lmt(T') \leq st(prev\_task)$ then there is no overlap between the previously accumulated activity and that induced by $T'$ which is $\Delta t = lmt(T') - st(T')$. $current\_lap$ should then be increased by $\Delta t$ as shown in Figure 2-a. Otherwise, there is some overlap between the previously accumulated activity and that of $T'$. Here the activity induced by $T'$ is $\Delta t = st(prev\_task) - st(T')$ which can be positive (Figure 2-b) or negative (Figure 2-b). In either cases, $current\_lap = current\_lap + lap(T') + \Delta t$ but the accumulated activity is incremented only by positive activity. Finally $lap(T)$ is updated to $current\_lap$ only when $current\_lap$ is the largest among all previous predecessors of $T$.

**Algorithm Longest-Activity-Path LAP**

```
Input: pred. sorted in decreasing $lmt$s and stored in $H$
Output: longest-activity-path $lap(T)$
(1) Initialize: $current\_act = 0,\ current\_lap = 0,$
      $prev\_task = \{\}$, $st(prev\_task) = \infty$, $lap(T) = 0$;
(2) While $(H \neq \emptyset)$ Do
      Begin
        $T' \leftarrow get\_top(H)$;
        If $lmt(T') \leq st(prev\_task)$ Then
                                $\Delta t = lmt(T') - st(T')$;
        Else  $\Delta t = st(prev\_task) - st(T')$;
        $current\_lap = current\_act + lap(T') + \Delta t$;
        If $\Delta t > 0$ Then   $current\_act = current\_act + \Delta t$,
                                 $prev\_task = \{T'\}$;
        If $lap(T) < current\_lap$ Then $lap(T) = current\_lap$;
        $H \leftarrow H - \{T\}$;
      End.
```

## 5 The scheduling algorithm

In the following we present algorithm *ADAPT* which uses set *Ready* and *Assign* to store ready-to-run tasks and assigned tasks, respectively. Initially, *Ready* contains all tasks without predecessors. The processors' free time $t\_free(p)$ is set to 0. For each ready $T$ we set $est(T,p) = 0$ and let $p_0$ be the processor on which $T$ can start at the earliest.

*ADAPT* schedules one task in each iteration of statement 2. In the first block, a tentative task $T_t$ is lexicographically selected among ready tasks. $T_t$ can start at the earliest on processor $p_{min}(T_t)$. Next, all ready tasks which can start at the earliest on $p_{min}(T_t)$ are stored into a set $H$. Tasks of $H$ may compete with $T_t$ for being assigned first on $p_{min}(T_t)$. All tasks of $H$ are examined to find out possible conflict.

If the running of some task $T \in H$ does not overlap with that of $T_t$, then $T$ may precedes $T_t$ or follow it without conflicts. Figure 3-(a) and (c) show that though $T_t$ and $T$ have their earliest startable times on the same processor but they do not conflict. In Figure 3-(a) $T$ can start and complete prior to starting of $T_t$ which means that $T$ becomes the tentative task. In this case, postponing the assignment of $T_t$ does not cause any degradation in the schedule, therefore $T$ can be assigned earlier than $T_t$. This significantly improves processor utilization that would otherwise be left idle if $T_t$ were assigned directly. In Figure 3-(c) $T_t$ can start and complete prior to starting of $T$ which means that $T$ will be assigned at a later decision without causing any delay over its earliest startable time.

If running some task $T \in H$ at its earliest startable time does conflict with the running of $T_t$, then the assignment of one of these tasks causes the task other to be delayed over its earliest startable time. In this case, the conflict is resolved by assigning the most prior task and delaying the other. Figure 3-b shows $T_t$ and $T$ have overlapped preferred execution time (earliest starting time). In this case *ADAPT* evaluates a decision function $d(T) = lap(T) - \kappa \times est(T,p)$ and selects the task that has the highest $d(T)$ among the two conflicting tasks, were $\kappa$ is evaluated as $\kappa = \frac{k\beta}{1+r_{max}\alpha_{max}}$. The scheduler decision is then adapted to task parallelism and communication. If parallelism is large enough to cover

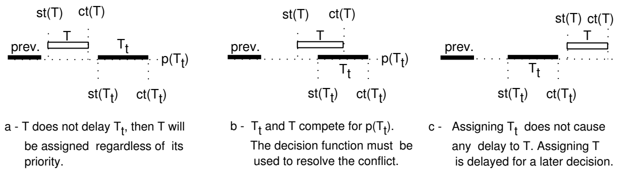

st(T)
ct(T)
T
prev.
$T_t$
$p(T_t)$
$st(T_t)$
$ct(T_t)$
a - T does not delay $T_t$, then T will be assigned regardless of its priority.
st(T)
ct(T)
T
prev.
$T_t$
$p(T_t)$
$st(T_t)$
$ct(T_t)$
b - $T_t$ and T compete for $p(T_t)$. The decision function must be used to resolve the conflict.
st(T)
ct(T)
T
prev.
$T_t$
$st(T_t)$
$ct(T_t)$
c - Assigning $T_t$ does not cause any delay to T. Assigning T is delayed for a later decision.

Figure 3: Resolution of conflicts for the best processor

communication, then $\kappa$ is large and an *earliest-task-first* will be fine. Otherwise, $\kappa$ is small and we use the most updated value of task-level that is $lap(T)$. The important thing is to note that *ADAPT* will select $T$ as long as $d(T) \geq d(T')$ which means that:

$$est(T,p) \leq est(T',p) + \frac{1 + r_{max}\alpha_{max}}{k\beta}(lap(T) - lap(T'))$$

In the second block, the final $T_t$ is assigned on processor $p^* = p_{min}(T_t)$ for which the earliest processor free time $t\_free$ becomes $least\_ct(T_t)$. Next, ADAPT updates the starting times of all ready tasks with respect to $p^*$. It also finds a new least starting time for every ready task $T$ that has $p^*$ as the processor on which it could start at the earliest. For any such a task $T$ the earliest startable processor is stored in $p_{min}$. Every time *least_st* is modified the priority function must be updated. Notice that the updated decision function value is non-decreasing.

In the third block, we use an integer $\lambda_{pred}(T)$ that is initially set to the number of unscheduled predecessors of $T$. Since now $T_t$ is assigned, then $\lambda_{pred}(T)$ must be decremented for every successor $T$ of $T_t$, i.e. $T_t \in Succ(T)$. Such a successor $T$ may become ready to run if $\lambda_{pred}(T) = 0$ which means that all predecessors of $T$ have already been assigned. For every newly ready task $T$, ADAPT evaluates its earliest starting time for every processor and finds the least startable time $least\_st(T)$ and its processor $p_{min}(T)$. Finally, the decision function $d(T)$ is initialized.

```
Algorithm: ADAPT
(1) Initialize: $Ready \leftarrow \{T : Pred(T) = \emptyset\}$, $Comp \leftarrow \emptyset$,
      For each $T \in Ready$: $est(T,p) = 0, p_{min}(T) = p_0$;
      For each $p \in P$: t_free(p)=0;

(2) While $|Comp| < n$ Do
      Begin
        Select task $T_t$ from $Ready$ in lexicogra. order
        Let $H = \{T \in Ready\ :\ p\_min(T) = p\_min(T_t)\}$
        While $(H \neq \emptyset)$ do
          Begin
            Pick $T$ from $H$ in lexicographic order;
            If $(least\_st(T) < least\_ct(T_t))$ Then
              If $(least\_ct(T) \leq least\_st(T_t))$ Then $T_t = T$;
              Else if $(d(T) > d(T_t))$ Then $T_t = T$;
            Remove $T$ from $H$;
          End

        Assign $T_t$ on $p^* = p_{min}(T_t)$,
        update $t\_free(p^*) = least\_ct(T_t)$;
```

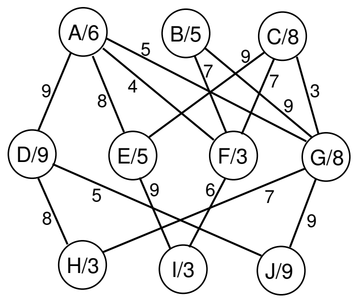
```
Remove $T_t$ from $Ready$, add $T_t$ to $Comp$;
Repeat for each $T \in Ready$:
 $est(T,p^*) = max\{est(T,p^*), t\_free(p^*)\}$,
 If $p\_min(T) = p^*$ Then
   Begin
    Find $least\_st(T) = est(T,p^+)$, $p\_min(T) = p^+$,
    $least\_ct(T) = est(T,p^+) + \mu(T)$;
    $d(T) = lap(T) - \kappa \times least\_st(T)$;
   End

Repeat for each task $T \in Succ(T_t)$ :
 $N_{pred}(T) = N_{pred}(T) - 1$
 If $N_{pred}(T) = 0$ Then
   Begin
    Add $T$ to $Ready$, $least\_st(T) = \infty$
    Repeat for each $p \in P$:
     $est(T,p) = \max_{T' \in Pred(T)}\{ct(T')+$
                 $c(T',T) \times c(T',T)r(p(T'),p)\}$,
     $est(T,p) = \max\{est(T,p), t\_free(p)\}$,
     If $est(T,p) < least\_st(T)$ Then
      $least\_st(T) = est(T,p)$, $least\_ct(T) =$
      $est(T,p) + \mu(T)$, $p\_min(T) = p$;
     $d(T) = lap(T) - \kappa \times least\_st(T)$
   End
End
```

The main loop of ADAPT is statement 2 that executes $n$ times because one task is scheduled in each iteration. Statement 2.1 executes at most $n$ times in order to select one ready task. In statement 2.2, we perform the following three operations. First, we update the parameters of the algorithm. Second we evaluate the largest activity path of the newly assigned task at a cost of $O(n)$. A cost of $O(n)$ is needed for finding new $least\_st$ and updating the priority of some ready tasks. Overall cost of statement 2.2 is $O(n)$. Finally, in statement 2.3 we visit all successors of $T_t$ but the condition $\lambda_{pred}(T) = 0$ occurs only once for each task and for each occurrence we evaluate $est$ for all processors. The global cost of statement 2.3 is $O(pn^2)$ which is also the time complexity of ADAPT.

## 5.1 Iterative scheduling

ADAPT is used in an iterative scheduling which alternatively operates on the forward and backward computation graphs. It produces valid solution in each iteration. In iteration $i$, *ADAPT* evaluates $lap_i(T)$ for each newly scheduled task $T$. In iteration $i$, $lap_i(T)$ is meant to estimate the achieved distance from entry to $T$. In iteration $i+1$, *ADAPT* uses a decision function $d_{i+1}(T) = lap_i(T) - \kappa \times est_i(T,p)$ and after scheduling $T$ it evaluates $lap_{i+1}(T)$ to be uses in iteration $i+2$, etc. The use of $lap_i(T)$ in iteration $i+1$ is meant to provide the scheduler with a measure of distance from $T$ to exit. In the evaluation we will study the number of iterations needed to find the best solution which is function of the size of search space.

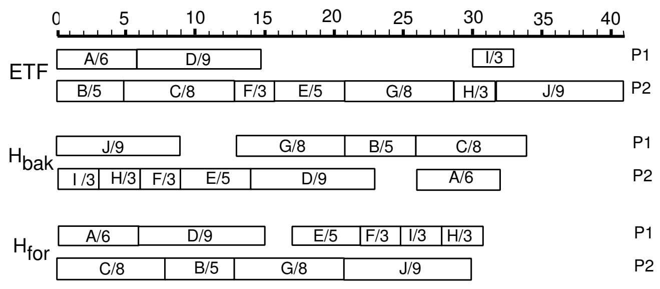
Figure 4 shows: (a) a DAG, (b) scheduling steps of ADAPT, and (c) ETF and ADAPT Gantt charts. Figure 4-(b) shows ADADT steps in scheduling the second forward iteration for which the *lap*s are obtained from the first backward iteration (Gantt chart is shown last on Figure 4-(c)). Figure 4-(c) shows: ETF finish time (41 units), ADAPT first scheduling iteration (backward) (34 units), and ADADT second scheduling iteration (forward) (30 units).



a - Example of Directed acyclic graphs

| Ready | P1( T / est(T,P1)) | P2(T / est(T,P2)) | T( lap(T), est (T)) | Scheduled |
|---|---|---|---|---|
| RDY(A, B, C) | P1(A/0, B/0 ,C/0) | P2(A/0, B/0, C/0) | A(32-0) B(23-0) C(28-0) | ---> A |
| RDY(B, C, D) | P1(D/6) | P2(B/0, C/0) | B(23-0) C(28-0) | ---> C |
| RDY(B, D, E) | P1(D/6) | P2(B/8, E/14) | B(23-8) E(8-14) | ---> B |
| RDY(D, E, F, G) | P1(D/6) | P2(E/14, F/13, G/13) | D(23-6) | ---> D |
| RDY(E, F, G) | P1( ) | P2(E/14, F/13, G/13) | E(8-14) F(6-13) G(18-13) | --->G |
| RDY(E, F, H, J) | P1(E/17, F/20) | P2(H/23, J/20) | E(8-17) F(6-20) | ---> E |
| RDY(F, H, J ) | p1( ) | P2(F/21, H/23, J/20) | F(6-21) H(3-23) J(9-20) | ---> J |
| RDY (F,H) | p1(F/22, H/28) | P2=( ) | F(6-22) H:(3- 28) | ---> F |
| RDY(H ,I) | P1(H/28, I/25) | p2=( ) | H(3-28) I(3-25) | ---> I |
| RDY(H) | P1(H/28) | P2 =( ) | H(3-28) | ---> H |

b - Scheduling steps



c - Gantt chart for ETF, first bakward eteration, first forward iteration.

Figure 4: (a) DAG, (b) ADAPT steps, and (c) Gantt chart

The iterative scheduling can be seen as a deterministic evolutionary process [4] that has hereditary variation and differential production. It changes through iterations such that each new state (solution) is similar to previous state and yet different. The similarity is present because the task-level does not always significantly change, from one iteration to another, to trigger a change in the decision function $d(T)$. Each state is evaluated through mapping of $lap$, local task starting time, and computation profile. Inferior task assignment are discarded because "excessive" task delay in current state means that $lap$ value associated to a task must increase proportionally to the task delay in next iteration, thus partially improves the local state (task). This process continues until finding some balancing which corresponds to some steady local minima.

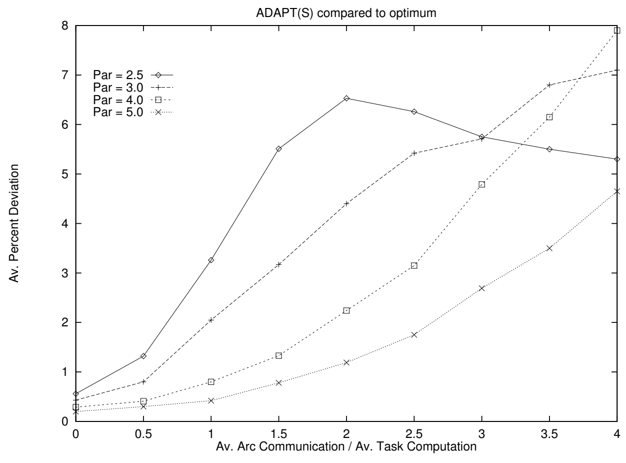


Figure 5: Percentage deviation of $ADAPT_S$ from optimum

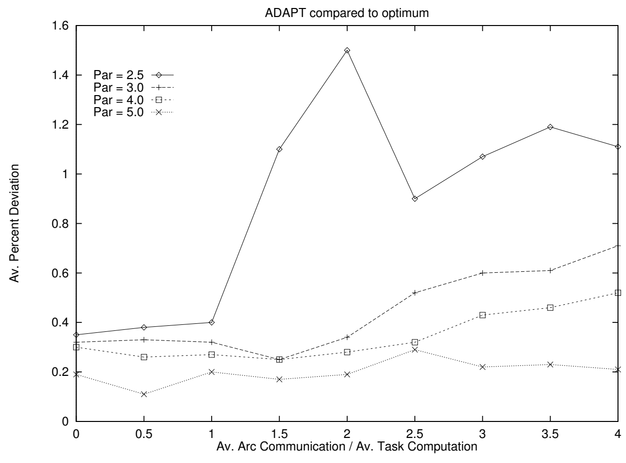


Figure 7: Percentage deviation of $ADAPT$ from optimum

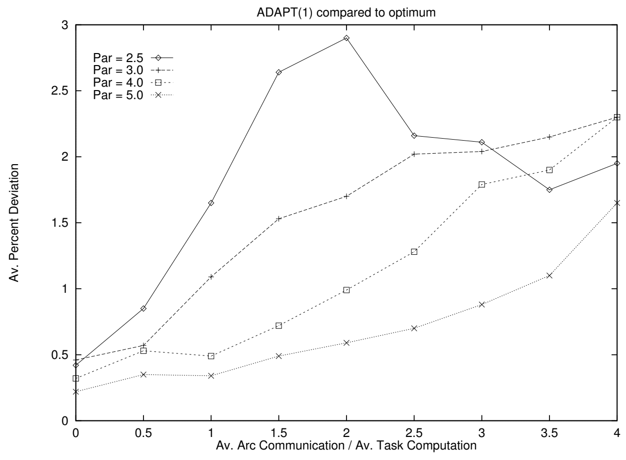


Figure 6: Percentage deviation of $ADAPT_1$ from optimum

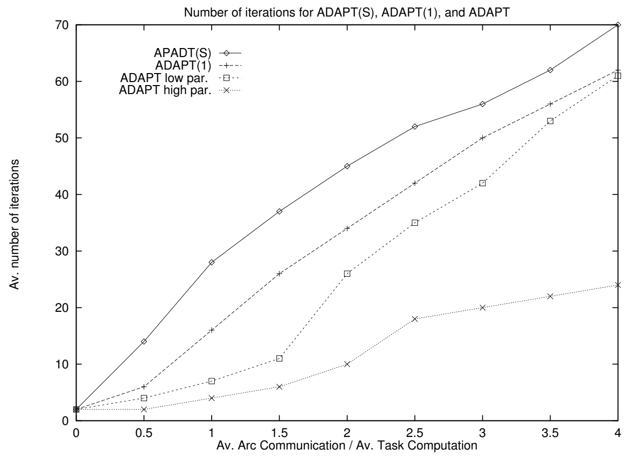


Figure 8: Typical number of iterations for best solutions

## 6 Performance evaluation

A *random problem generator* (RPG) is used to generate computation graphs with few hundred tasks with uniform distribution of computation and communication. The previously defined *communication granularity* $\alpha$ and *degree of parallelism* $\beta$ are used for setting the generated problems and number of processors. We use the RPG to generate computation graphs for each instance of $\alpha$ and $\beta$. Each generated problem is scheduled by randomly selecting a task randomly assigning it to some free processor. The random task and processor selection are meant ot eliminate any possible correlation between the above heuristics and the random schedule. All idle times in the random schedule are filled with additional tasks for which new dependence edges are created so that to preserve as much as possible the original setting of $\alpha$ and $\beta$. However, if the previous setting cannot be maintained the graph and its optimum schedule are rejected. The result is a new computation graph for which we know an optimum solution as its schedule has no idle times.

The studied ranges of $\alpha$ and $\beta$ are $[0-4]$ with a step of 0.5 and $[1, 2, 2.5, 3, 4]$, respectively. For each instances of $\alpha$ and $\beta$ we use a uniform distribution to generate 30 computation problems.

We study performance of: (1) $ETF$, (2) $ADAPT(S)$ which uses $d(T) = ct_{prev}(T) - est(T, p)$, (3) $ADAPT(1)$ which uses $d(T) = lap(T) - est(T, p)$, and (4) $ADAPT$ which uses $d(T) = lap(T) - k\beta \times est(T, p)/(1 + r_{max}\alpha_{max})$. $ADAPT(S)$ is intended to study the effect of using the task completion time ($ct_{prev}(T)$) achieved in the previous scheduling iteration as task-level. By comparing performance of $ADAPT(1)$ to that of $ADAPT$ we can study the effects of weight $k\beta/(1 + r_{max}\alpha_{max})$.

Each generated problem is scheduled by each of the above heuristics. The length of optimum solution is denoted by ($\omega_{opt}$). We plot the relative percentage deviation for each heuristic $h$ which is $(\omega_h/\omega_{opt} - 1)100$. Each plotted point results from averaging the heuristic finish times for 30 generated problems. Figures 1, 5, 7, and 6 show the results.

ETF (Figures 1) can perform well when there is enough task parallelism to cover available communication then ETF provides good management of processor idle time which effectively minimize finish time. This effect is depicted in the bound $(\omega - \omega_{opt})/\omega_{opt} \leq (\frac{1}{k\beta} + \frac{r_{max}\alpha_{max}}{k\beta})$. which predicts the degradation when the available parallelism is relatively low (term $1/k\beta$) or the amount of communication is relatively large (term $r_{max}\alpha_{max}/k\beta$).

$ADAPT(S)$ uses a simple but more balanced decision function ($d(T) = ct_{prev}(T) - est(T,p)$) than ETF which is greatly rewarded by noticeable improvement in performance especially when the parallelism is low. $ADAPT(S)$ deviates on the average by at most 7% from optimum for all the studied cases. It was shown to be much less sensitive to computation profile than ETF. However, the number of iterations needed for $ADAPT(S)$ to achieve the above performance is linear with the communication granularity (Figure 8).

$ADAPT(1)$ differs from $ADAPT(S)$ only by the task-level. The use of $lap(T)$ instead of simple task completion time $ct(T)$ as task-level enhanced the solution generated $ADAPT(S)$ by about 5%. Accurate evaluation of task-level is also rewarded by noticeable improvement in performance as near optimal solutions were generated by $ADAPT(1)$ for all the studied cases. Also $ADAPT(1)$ needs much less number of iterations to find its best solution than that needed for $ADAPT(S)$ (Figure 8).

Finally, $ADAPT$ with a decision function that combines task-level with the concept of earliest-task-first in a manner that is adapted to the computation profile. A task $T$ is selected if for all $t'$ that compete for the same processor we have:

$$est(T,p) \leq est(T',p) + (\frac{1}{k\beta} + \frac{r_{max}\alpha_{max}}{k\beta})(lap(T) - lap(T'))$$

Low ratio of communication to parallelism ($r_{max}\alpha_{max}/k\beta$) leads to earliest-task-first decision. Large ratio of communication to parallelism brings task-priority to decide whether earliest-task-first should be taken or not. Thus, the use of task-priority occasionally breaks the earliest-startable-task order. Thus, successively scheduled tasks do not form non-decreasing sequence in time which means that Graham's bound does not hold any more. In fact this happens only where assigning prior tasks is more important towards minimizing schedule length than earliest-task-first discipline.

$ADAPT$ achieves on the average about 2% deviation from optimum solution. Comparing $ADAPT$ and $ADAPT(1)$ we can see that setting up the weight $k\beta \times /(1 + r_{max}\alpha_{max})$ in $d(T)$ enables adapting the scheduler to the computation profile specified by $\alpha$ and $\beta$. This was rewarded at two levels: (1) producing near optimum solutions (Figure 7), and (2) shortening the number of iterations for finding the best solution (see Figure 8 for low and high parallelism).

## 7 Conclusion

We presented a compile-time adaptive scheduling that incorporates two fundamental concepts: global task priority and minimization of processor idle time. Task and processor selection uses the above concepts in a weighted manner depending on task parallelism and communications. An iterative scheduling was proposed as a deterministic evolutionary process that has hereditary variation and differential production. It changes through iterations such that each new schedule is similar to previous schedule and yet different. Our approach to adapting the iterative scheduling decision to computation profile was able to produce near-optimum schedules with reasonable number of iterations.

## 8 Acknowledgments

The authors would like to acknowledge support from the Research Committee at King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia.

## References

[1] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. on Applied Mathematics*, 17:416–429, 1969.

[2] 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.

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

[4] Ralph Michael Kling and Prithviraj Banerjee. ESP: Placement by simulated evolution. *IEEE Trans. on Computer-Aided Design*, 8, No 3:245–256, Mar 1989.

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

[6] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical path scheduling: an effective technique for scheduling task graphs onto multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7, No 5:506–521, 1996.

[7] 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.

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

[9] J. Sheild. Partitioning concurrent vlsi simulated programs onto multiprocessor by simulated anealling. *IEEE proceedings*, 134:24–30, Jan 1987.

[10] 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.

[11] 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.