Adaptive Scheduling of Computations and Communications on Distributed-Memory Systems

Mayez Al-Mouhamed and Homam Najjari *

Abstract

Compile-time scheduling is one approach to extract parallelism which has proved effective when the execution behavior is predictable. Unfortunately, the performance of most priority-based scheduling algorithms is computation dependent. Scheduling based on the concept of *earliest-startable-task* produces reasonably short schedules only when available parallelism is large enough to cover the communications. A priority-based decision is more effective when parallelism is low. We propose a scheduling in which the decision function combines two concepts: (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 a computation profile factor that is the ratio of task parallelism to communication. It is shown that the above factor is an upper bound on the deviation of schedule length from optimum. To tune the solution finish time the above scheduler is iteratively applied on the computation graph. In each iteration, the newly generated schedule is used to sharpen the task-levels which contribute in finding shorter schedules in the next iteration. Evaluation is carried out for a wide category of computation graphs with communications for which optimum schedules are known. It is found that pure local scheduling and static priority-based scheduling significantly deviate from the optimum under specific problem instances. Our approach to adapting the scheduling decision to computation profile is able to produce near-optimum solutions via a much reduced number of iterations than other approaches.

1 Introduction

Deterministic scheduling can be profitable when the execution behavior of the computation 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[7, 8, 9, 18].

*Computer Engineering Department, College of Computer Science and Engineering, King Fahd University, Dhahran 31261, Saudi Arabia (mayez/homam@ccse.kfupm.edu.sa).

The problem of minimizing the schedule finish time of computations and communications is one NP-complete problem [10]. Lower bounds [2] and worst case analysis [12, 3] have been proposed for scheduling precedence computations with and without communication costs. The objective is to find efficient non-preemptive scheduling approaches that combine knowledge on the computation structure and multiprocessor in order to minimize overall finish time. 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 [20], simulated annealing [22], and genetic [11] have been proposed for finding good mapping and partitioning of computations. Scheduling based on *task duplication* over idle processors has been proposed [15, 7, 8] to reduce the effects of dominant communication without excessively increasing the overhead in managing duplicated data. The impacts of task and processor selection as well as the profitability of task duplication were experimentally studied in the above papers. The degree of task duplication can also be adjusted [17] depending on the number of available processors or their difference in speeds.

Linear clustering [13] consists of iteratively assigning tasks along the most communicative chains to the same processor. *Clustering* over an unbounded number of processors [21] 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. *Dominant sequence clustering* (DSC) [25] is a low complexity clustering that accepts the 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* method (DCP) was proposed in [16]. 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.

Examples of priority-based scheduling that operate over a bounded number of processors are *dynamic level scheduling* (DLS) [23] and *earliest-task-first* (ETF) [12]. 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 a *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. Therefore the performance of these scheduling heuristics depends on the computation and communication profile.

Our objective is to learn from the above algorithms and to design a scheduling algorithm that adapts 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 currently accumulated knowledge of computation and communication, and (2) instantaneous exploitation of 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 an analysis of the performance problem, Section 4 presents a model on performance degradation. Section 5 presents the evaluation of task-level. Section 6 presents the proposed scheduling. Section 7 presents the evaluation. We conclude this work in Section 8.

2 Background

Wu and Gajsky [24] proposed two algorithms which are the *modified critical path* (MCP) and *mobility directed* (MD). These algorithms use the *as-soon-as-possible* starting time ($t_s(T)$) that is the length of the longest path including computation and communication from entry node to T. The *as-late-as-possible* completion time ($t_l(T)$) is the difference between the longest path in the graph and the length of the longest path from T to some exit node. MCP selects the free task with the largest ($t_l(T)$) and assigns it to the processor that can start it at the earliest time among all processors. MD selects the free task T with least *relative mobility* ($rm(T) = (t_l(T) - t_s(T))/\mu(T)$) and assigns it to a processor so as not to increase the current critical path, i.e. none of the scheduled tasks is delayed beyond its latest time. MD uses an unbounded number of processors and its complexity is $O(pn(n+e))$.

Linear clustering [13] consists of iteratively merging tasks along the most communicative chains in an attempt to minimize overall schedule time. Sarkar and Hennessy [21] proposed a scheduling that consists of 1) clustering the tasks on an unbounded number of processors and 2) merging some clusters by incorporating the network latency in order to match the number of clusters to the number of processors. To cluster the tasks the edges are sorted in non-increasing order of the number of messages. At each step, the edge with the highest weight is "zeroed" if the the parallel

time $PT(T)$ does not increase. PT(T) is the sum of $tlevel(T)$ and $blevel(T)$ which are the lengths of the longest path from an entry node to T and from T to some exit node, respectively. Finally, tasks in each cluster are sorted in decreasing order of their *blevel*. The complexity of clustering is $O(e(e+n))$, where e is the number of edges.

Yang and Gerasoulis[25] improved Sarkar's scheduling over an unbounded number of processors. The *dominant sequence* (DS) consists of the tasks that lie on the longest path from entry to exit in a clustered graph. DS is formed by a chain of tasks with maximal $tlevel(T) + blevel(T)$ which is also taken as task-priority. A task whose predecessors have all been examined is called free. A sequence of edge zeroing is performed with the aim of reducing DS after considering the free tasks in decreasing order of priority. The selected edge zeroing is done so that the current task T starts at the earliest time and this clustering does not increase $tlevel(T)$. In other terms, none of the unexamined DS tasks can be delayed. Updating the values of $tlevel(T)$ and $blevel(T)$ is done only when one of the free tasks' child is clustered. This clustering strategy reduces complexity to $O((n+e)logn)$.

Kwok and Ahmad proposed a scheduling called *Dynamic Critical Path* (DCP) with the aim of minimizing the number of processors. DCP minimizes the starting times of tasks after estimating the effect on the successors. Only processors holding predecessors or successors of the current task are examined. The notation uses the *absolute earliest starting time* to start a task T on processors p which is denoted by $(AEST(T,p))$. Similarly, the *absolute latest starting time* is denoted by $(ALST(T,p))$. The critical tasks have identical $AEST(T,p)$ and $ALST(T,p)$, i.e. zero mobility. A DCP task T is selected when all its DCP predecessors have already been scheduled. Only processors holding the predecessors of T are examined. For each such processor, the algorithm searches a vacant slot for fit T with the possibility of starting earlier or delaying previously assigned tasks within the limit of their $AEST$ and $ALST$ values. Processor selection is based on minimizing the sum $AEST(T,p) + AEST(T',p)$, where T' is the successor of T that has the least difference between its $AEST$ and $ALST$. This condition guarantees some processor reservation for the most critical successor. Finally, all the tasks set at their $AEST$ values. The complexity of DCP scheduling is $O(n^3)$.

Hwang, Chow, Anger, and Lee proposed the *Earliest-Task-First* (ETF) scheduling for a bounded number of processors. ETF is a variant of *Graham's List-scheduling* with the difference that task selection is based on the principle of *earliest-startable-task-first*. This requires evaluation of the earliest time for all ready tasks and for all processors. The complexity of the ETF algorithm is $O(n^2P)$. Sih and Lee proposed the *Dynamic Level Scheduling* (DLS) which also uses a bounded number of processors. In DLS, the largest sum of computations (only) from T to exit is called the *static task level* $SL(T)$. The ready task that has the highest *dynamic level* $DL(T,p) = SL(T) - ST(T,p)$ is selected first, where $ST(T,p)$ denotes the earliest time at which all incoming data transfers are complete for T. This requires an evaluation of $ST(T,p)$ for all ready tasks and all processors. The complexity of DLS scheduling is $O(n^2P)$.

3 The computation dependent performance problem

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 [12] 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.

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) [12]. ETF uses Graham's list-scheduling [10] in which the scheduler tracks the increasing sequence of the processor 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 of a worst-case bound [12] on the 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 such as off-line computation of task priority. ETF schedules gave an acceptable deviation from some known solutions as reported in [23, 16]. Some anomalies were also reported. For example increasing parallelism or reducing communication always improves ETF schedule length [4]. The average running time of ETF appears to be reasonable [16] 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 hide communications, or (2) there is excessive communication that cannot be

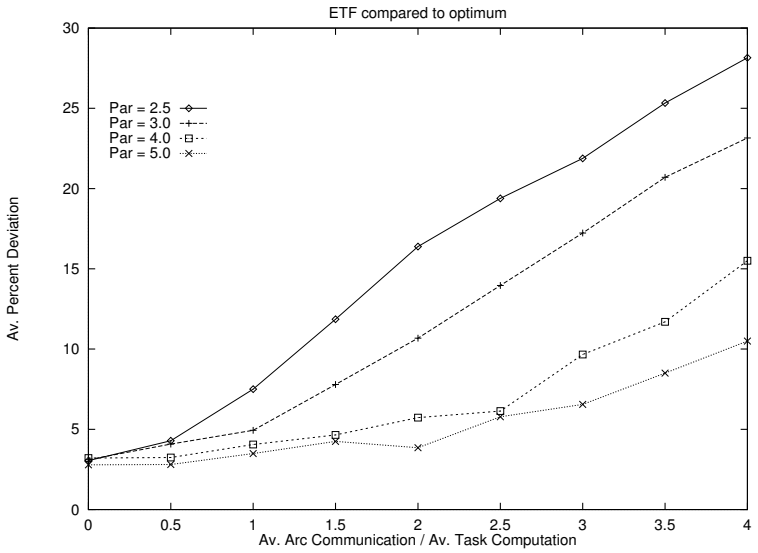


Figure 1: Percentage deviation of ETF schedules from optimum solutions

hidden anyway. Task parallelism is the average number of parallel tasks per processor. Therefore, these heuristics require increasing the parallelism, or decreasing the number of processors, in order to achieve acceptable deviations as shown in Figure 1.

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 hide the communication. In this case, there is no guaranteed performance. 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 an analytical explanation for the performance degradation of ETF which will be used later as the basis for a design of an adaptive scheduling algorithm. In the next section we establish a relation that can be used to predict the deviation from the optimum of an ETF schedule as a function of communication and parallelism.

4 Modelling the algorithm degradation

In his study of multiprocessor anomalies, Graham [6] 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 [12] 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 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 a scheduling heuristic and let ω 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 $B = \cup_{1 \leq i \leq q}(b_{l_i}, b_{r_i})$ 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 the finding of 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 have completed 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 the finding of a bound on the sum of all idle time intervals ($\mu(\phi)$) in the schedule:

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

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 $\sum_{j=1}^{L} \mu(T_j) \leq \sum_{T \in X_{large}} \mu(T)$, then idle times $\sum_{\phi_i \in \Phi} \mu(\phi_i)$ are bounded by:

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

where k is the largest integer satisfying $k \times \sum_{j=1}^{L} \mu(T_j) \leq \sum_{T \in X_{large}} \mu(T)$. 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* (α), and 2) the *degree of parallelism* (β). Parameter α is defined as the ratio of average communication (c) to average computation (μ), that is:

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

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 β as the graph parallelism over the number of processors:

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

In other terms, β is an indicator of the average number of tasks that compete for scheduling on each processor. Let α_{max} be an upper bound on the communication granularity for a given computation. Since the communications are uniformly distributed

then the average communication along partial chains is also bounded by α_{max}. Then the bound given in Equation 4 becomes:

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

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 ω 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:

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

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

$$\frac{\omega - \omega_{opt}}{\omega_{opt}} \leq \frac{1 + r_{max}\alpha_{max}}{k\beta} \tag{9}$$

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 ω increases at most linearly with and increase in α_{max} and r_{max}. This explains 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 chains of computation and communications, i.e. tasks with a higher task-level. Thus task and processor selection must include provision for both *earliest-task-first* and *highest-level-first* strategies.

The above bound and the experimental testing (Figure 1) shows that it can be used to model the deviation $((1 + r_{max}\alpha_{max})/k\beta)$ from optimum. When task parallelism and communication are known, the model allows the prediction of possible degradation in the finish time of a schedule obtained by using the concept of *earliest-task-first*.

In the next section we present an approach to estimate the task-level in order to account for computations and communications along chains of immediate tasks.

5 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 computations and communications 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 Δt should not affect $lap(T)$ if Δt occurs: (1) prior to the start of any predecessor of T, or (2) following the completion of T. $lap(T)$ may be affected by Δt 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 $c(T', T)$ is 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 the starting of T. The decision of scheduling T could have been delayed beyond time point $est(T, p)$ because some other tasks were executed earlier 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 $act(T_i, T)$:

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

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 the 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.

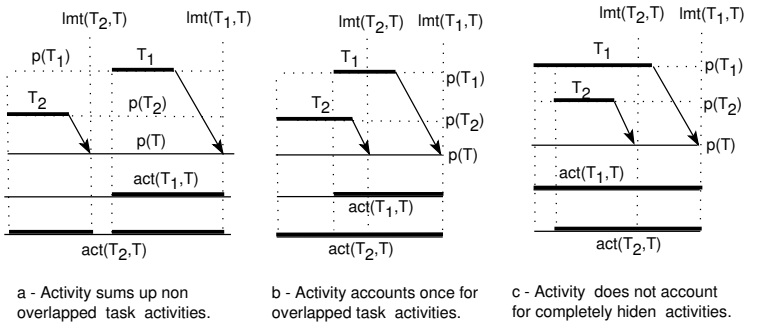


Figure 2: Task level: (a) non-overlapped, (b) overlapped, and (c) coverage

An $O(n)$ algorithm to evaluate the *Longest Activity Path* (LAP) of T is given below, where n is the number of predecessors of T. LAP assumes that 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 Δt 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 case, $current\_lap = current\_lap + lap(T') + \Delta t$ 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 of 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.

Now we are well equipped to define a scheduling decision function ($d(T)$) that combines two fundamental concepts which are: (1) earliest-task-first (etf), and (2) highest-level-first (hlf) for scheduling computations with communication times. The knowledge of task parallelism and communication allow the finding of an upper bound on the relative deviation of earliest-task-first schedule length from optimum ($\kappa = (1 + r_{max}\alpha_{max})/k\beta$). This can be used as a weighting factor in the decision function to direct task and processor selection so that: (1) a large value of κ (low task parallelism and/or large communication) leads to an etf decision, and (2) a low value of κ leads to an hlf decision (large task parallelism and/or low communication). In the next section we present a scheduling algorithm that implements the above proposal through the use of a decision function $d(T)$ that is a linear function of task-level, κ, and task starting time.

6 The scheduling algorithm

In this section we present our proposed algorithm which is called $ADAPT$. Its strategy is simple and consists of selecting a tentative task and the processor for which this task can run at the earliest time. Next it selects all the ready tasks that have their earliest starting times on the above processor. The previously defined decision function $d(.)$ is used to find a new tentative task only when the execution time of a ready task conflicts (overlaps) with a current tentative task. Otherwise, the first assigned task is one that can start and finish earlier. In the following we present in some detail our proposed algorithm $ADAPT$ which is shown on Figure 4.

In statement 2.1, $ADAPT$ uses set *Ready* and *Assign* to store ready-to-run tasks and assigned tasks, respectively. Initially, *Ready* contains all tasks without predecessors. The processor's free time $t\_free(p)$ is the earliest time a processor can be assigned a new task. $t\_free(p)$ is set to 0 for every p. 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 statement 2.1, a tentative task T_t is lexicographically selected among ready tasks. T_t can start at the earliest on some 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. Thus set H contains all the tasks that may compete with T_t for being assigned first on the $p_{min}(T_t)$. All tasks of H are examined to find out possible conflicts.

If the running of some task $T \in H$ does not overlap with that of T_t when it is set at its earliest starting time, then T may precede T_t or follows it without conflict. Figure 3-(a) and (c) show that though T_t and T have their earliest startable times on the same processor, they do not conflict. In Figure 3-(a) T can start and complete prior to the start 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 because $p_{min}(T_t)$ would otherwise be left idle if T_t were assigned directly. In Figure 3-(c) T_t can start and complete prior to the start of T, which means that T will be assigned to a later decision order without causing any delay over its earliest startable time.

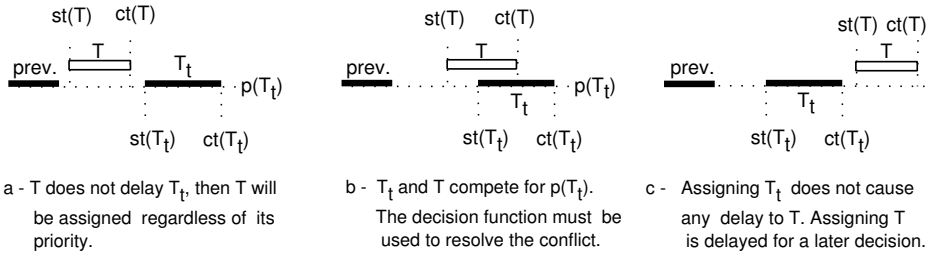


Figure 3: Resolution of conflicts for the best processor

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 T_t to be delayed over its earliest startable time. In this case, the conflict is resolved by assigning the earliest task and delaying the other. Figure 3-b shows that T_t and T will have an overlapped (conflicting) execution time if both must be assigned at their 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 $d(T)$ of the two conflicting tasks, where κ is evaluated as $\kappa = \frac{k\beta}{1+r_{max}\alpha_{max}}$ and k is some constant (≥ 1). The scheduler decision is then adapted to task parallelism and communication. If the parallelism is large enough to cover communication, then κ is large and an *earliest-task-first* will be fine. Otherwise, κ is small and we use the most recently updated value of the task-level that is $lap(T)$. The important thing to note is that $ADAPT$ selects T as long as $d(T) \geq d(T')$ which means that:

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

Clearly ETF discipline is not strictly enforced as T can still be selected first even if $est(T,p) \geq est(T',p)$ provided that the difference $est(T,p) - est(T',p)$, which is the additional idle time, is no more than the weighted difference in task-level which is $\frac{1+r_{max}\alpha_{max}}{k\beta}(lap(T) - lap(T'))$.

In statement 2.2, 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 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 statement 2.3, we use an integer $\lambda_{pred}(T)$ that is initially set to the number of unscheduled predecessors of T. Since T_t is now 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
(2.1) Select task T_t from *Ready* in lexicographical 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

(2.2) 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

(2.3) 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**
$least\_st(T) = est(T, p)$, $p\_min(T) = p$,
$least\_ct(T) = est(T, p) + \mu(T)$;
$d(T) = lap(T) - \kappa \times least\_st(T)$;
End
End

Figure 4: Scheduling algorithm *ADAPT*

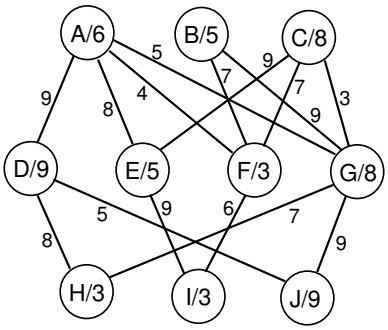
The main loop of *ADAPT* is statement 2 that executes n times because one task is scheduled in each iteration. Statement 2.1 executes at most n times in order to select one ready task. In statement 2.2, we perform the following three operations. First, we update the parameters of the algorithm. Second 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 the new *least_st* and updating the priority of some ready tasks. The 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.

6.1 Iterative scheduling

ADAPT is used in an iterative scheduling framework which consists of alternatively scheduling the forward and backward task graphs associated to a given computation. It produces one valid solution in each iteration. In the ith iteration, *ADAPT* evaluates $lap_i(T)$ for each newly scheduled task T. $lap_i(T)$ represents an estimate of the achieved longest path from entry node to T as achieved in the ith schedule. In iteration $i+1$, *ADAPT* uses a decision function $d_{i+1}(T) = lap_i(T) - \kappa \times est_{i+1}(T, p)$ and after scheduling T it evaluates $lap_{i+1}(T)$ to be used in the $i+2$th scheduling iteration. The use of $lap_i(T)$ in the $i+1$th iteration is meant to provide the scheduler with a measure of the longest path from T to exit node. To start the first scheduling iteration we set $lap(T) = 0$ and use $d_1(T) = -est_1(T, p)$ in backward scheduling of the computation graph. In the evaluation we will study the number of iterations needed to find the best solution which is a function of the size of the search space.

Figure 5 shows: (a) a DAG, (b) scheduling steps of ADAPT, and (c) ETF and *ADAPT* Gantt charts. The Gantt chart of the schedule obtained from the first iteration of *ADAPT* over the backward task graph is shown (H_{bak}) in Figure 5-(c). The decision function during this first iteration is $d_1(T) = -est_1(T)$ because $lap(T) = 0$ for all Ts. During this iteration, we evaluate $lap(T)$ after scheduling each task which will be used in the second iteration. The scheduling of steps of *ADAPT* in the second forward iteration are shown in Figure 5-(b). The values of evaluated $lap(T)$ are shown in the fourth column of Figure 5-(b). Here we use the *lap* values which are obtained from the first backward iteration (Figure 5-(c) H_{bak}) which are used in the second forward scheduling iteration (Figure 5-(c) H_{for}). The second iteration uses $d(T) = lap(T) - \kappa \times est(T)$, where *kappa* is set to 1 for simplicity. The tasks that are ready to run (*Ready*) are listed in first column of Figure 5-(b). The second and third columns show the tasks (with their starting times *est*s) that can start at the earliest on $p1$ and on $p2$, respectively. The fourth and fifth columns show the $lap(T) - est(T, p)$ (assuming $\kappa = 1$) and scheduled tasks, respectively. Figure 5-(c) shows: (1) ETF finish time (41 units), ADAPT's first scheduling iteration (backward) (34 units), and ADAPT's second scheduling iteration (forward) (30 units).

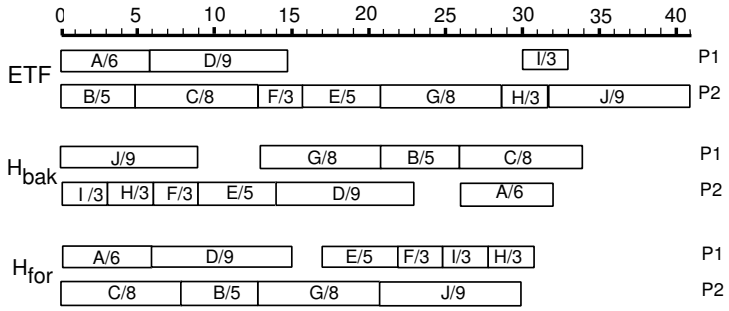
Generally, a few iterations of *ADAPT* are sufficient to find solutions whose finish time are much shorter than those produced by ETF scheduling. Each new iteration is likely to retain some similarity with the previous solution and introduces variation



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 5: (a) DAG, (b) *ADAPT* steps, and (c) Gantt chart

at the same time. This process continues until a balance is found (task-level) which corresponds to some steady local minima. This corrective process has been shown in practice to be a process that explores a space of "good" solutions which allows optimization of the solution finish time.

Iterative scheduling can be seen as a deterministic evolutionary process [14] that has hereditary variation and differential production. Change is introduced via each iteration in such a way that each new state (solution) is similar to the previous state and yet different. The similarity is present because the task-level does not always change enough, from one iteration to another, to trigger a change in the decision function $d(T)$. Each state is evaluated through the mapping of *lap*, local task starting time, and computation profile. Inferior task assignments are discarded because "excessive" task delay in the current state leads the *lap* value associated with a task to increase proportionally to the task delay. This partially improves the local state (task) in

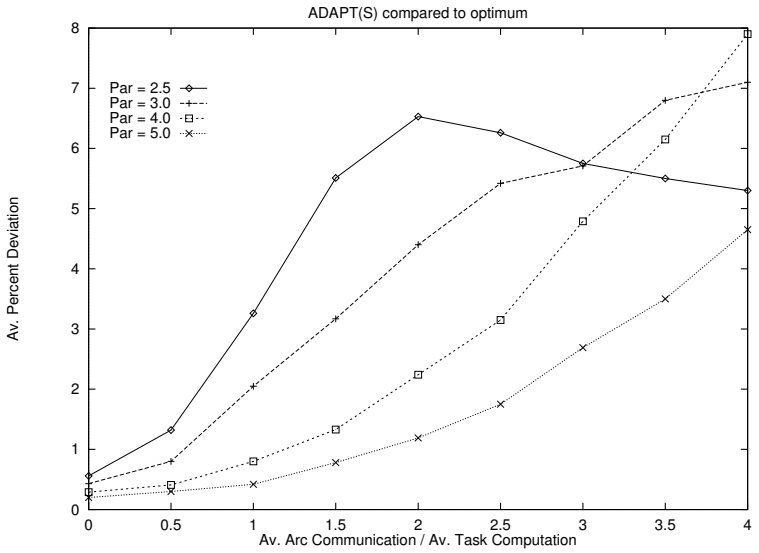


Figure 6: Percentage deviation of $ADAPT(S)$ from optimum

the next scheduling iteration. This process continues until a balance is found which corresponds to some steady local minima.

7 Performance evaluation

A *random problem generator* (RPG) is used to generate computation graphs with a few hundred tasks and a uniform distribution of computation and communication. The previously defined *communication granularity* α and *degree of parallelism* β are used for setting the generated problems and the number of processors. We use the RPG to generate computation graphs for some instances of α and β. Each generated computation problem is scheduled by randomly selecting a task and randomly assigning it to some free processor. The random task and processor selections are meant to eliminate possible correlations 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 in order to preserve as much as possible the original settings of α and β. However, if the previous setting cannot be maintained the graph and its schedule are rejected. The result is a new computation graph for which we know an optimum solution over a given number of processors because the schedule has no idle times.

The studied ranges of α and β are $[0-4]$ with a step of 0.5 and $[1, 2, 2.5, 3, 4]$, respectively. For each instance of α and β we use a uniform distribution to generate 30 computation problems. The variance on the number of edges is set to 50% of the average needed number of edges. 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% of the edges reach arbitrary forward levels.

We study the 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)

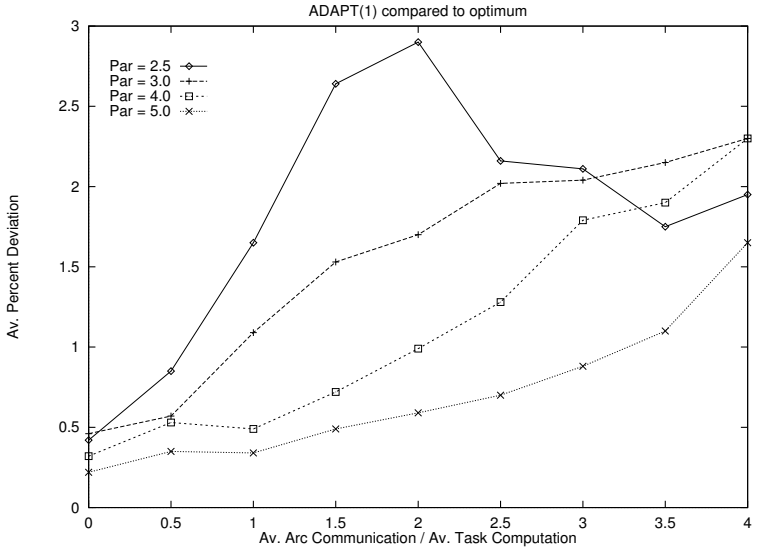


Figure 7: Percentage deviation of $ADAPT(1)$ from optimum

$ADAPT$ 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 the task-level. $ct(T)$ represents a primary measure of longest path from entry node to T. By comparing the 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 the optimum solution is denoted by (ω_{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, 6, 7, and 8 show the results.

ETF (Figures 1) can perform well when there is enough task parallelism to cover available communication. In this case the ETF strategy provides good management of processor idle time which effectively minimizes the schedule 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 of the heuristic finish time when: (1) the available parallelism is relatively low (term $1/k\beta$), or (2) the amount of communication is relatively large compared to the available task parallelism (term $r_{max}\alpha_{max}/k\beta$). The result is that the schedule length of ETF may deviate by more than 20% from optimum for the studied ranges of α and β.

$ADAPT(S)$ uses a simple but more balanced decision function ($d(T) = ct_{prev}(T) - est(T,p)$) than ETF which is greatly rewarded by a noticeable improvement in performance especially when the parallelism is low. $ADAPT(S)$ deviates on 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 9).

$ADAPT(1)$ differs from $ADAPT(S)$ by the use of a more accurate task-level. The use of $lap(T)$ instead of simple task completion time $ct(T)$ as task-level enhanced the solution generated by $ADAPT(S)$ by about 5%. Accurate evaluation of task-level

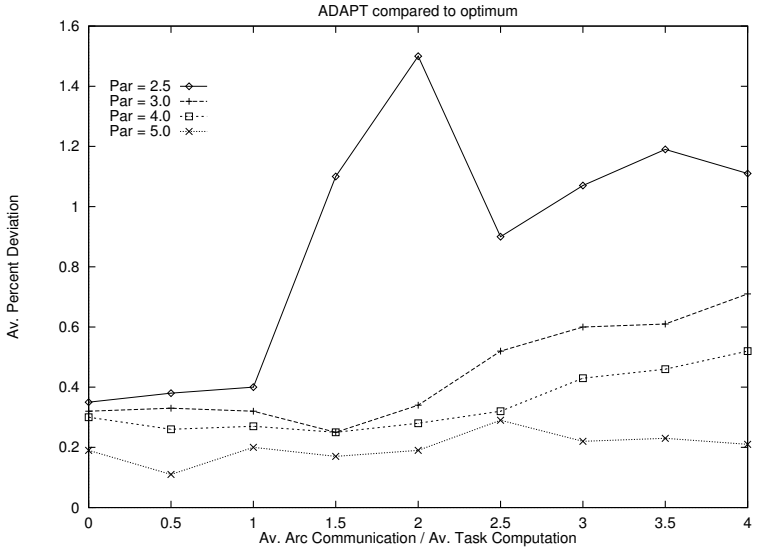


Figure 8: Percentage deviation of $ADAPT$ from optimum

is also rewarded by a noticeable improvement in performance as near optimal solutions were generated by $ADAPT(1)$ for all the studied cases. Also $ADAPT(1)$ needs far fewer iterations to find its best solution than that those needed for $ADAPT(S)$ (Figure 9).

$ADAPT$ is designed using 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 each ready task T' that competes for the same processor we have:

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

A low ratio of communication to parallelism ($r_{max}\alpha_{max}/k\beta$) leads to the earliest-task-first decision. A 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. Therefore, successively scheduled tasks do not form non-decreasing sequence in time which means that Graham's bound no longer holds. In fact this happens only where assigning more prior tasks has a greater importance in minimizing schedule length than a simple application of the earliest-task-first discipline.

$ADAPT$ achieves on average about a 2% deviation from the optimum solution. Comparing $ADAPT$ and $ADAPT(1)$ we can see that setting up the weight $(1 + r_{max}\alpha_{max})/k\beta$ in $d(T)$ enables the adaptation of the scheduler to the computation profile specified by α and β. This was rewarded at two levels: (1) producing near optimum solutions (Figure 8), and (2) shortening the number of iterations for finding the best solution (see Figure 9 for low and high parallelism).

Graham's list-scheduling [10] (LS) applies only to zero communication problems. LS generates optimum solutions for tree-computations with equal task times but as experimental evaluation of LS [1] showed its ability to generate near-optimum solutions for both deterministic and stochastic computations as it deviates from the optimum

by less than 5% in 90% of the cases. For zero communication problems $ADAPT$ does not deviate by more than 1.5% in 90% of the cases.

7.1 Comparing to best know solutions

We also repeated the above testing by directly scheduling the computation graphs without filling the idle times. In this case no optimum solution is known for the generated problems. Each generated computation graph is scheduled by ETF, $ADAPT(S)$, $ADAPT(1)$, and $ADAPT$. The shortest finish time that is achieved by any of the above heuristics for a given computation graph is denoted by (ω_{best}) and used as a *reference* of the optimum solution. We similarly evaluated the average relative deviation from ω_{best} for each heuristic. The results were similar to those of Figure 1, 6, 8, and 7 but with some difference in the value of deviation from ω_{best}.

ETF deviated by less that 6% for $\frac{1+\alpha}{\beta} \leq 0.8$ and deviated by up to 20% from ω_{best} when $0.8 \leq \frac{1+\alpha}{\beta} \leq 1.6$. For $ADAPT(S)$, the deviation from ω_{best} was generally below 5%. However, for large task parallelism the deviation of $ADAPT(S)$ did not exceed 2%. The deviation of $ADAPT(1)$ was below 3% and was mostly close to 1% for large parallelism. Finally, $ADAPT$ did not produce schedules deviating by more than 1% from ω_{best}. This provides another level of confidence in the proposed approach.

7.2 Optimization of the schedule

Using $ADAPT$ to carry out iterative scheduling on the forward and backward computation graphs allows the exploration of a space of solutions. Although the finish time of the solutions found fluctuates, there is a good chance of finding new solutions with shorter finish times compared to the solution generated out of the first iteration. The iterative solutions can be classified into four categories: a) converges to its local minima, b) converges to a solution other than its local minima, c) cyclic, and d) does not converge. The average number of required iterations $N_{\alpha,\beta}$ needed to generate the best solution for the first three categories strongly depends on the computation profile such as task parallelism (β) and communications (α). The last category may converge if the iterative process is continued beyond $N_{\alpha,\beta}$.

Figure 9 shows the average number of iterations $N_{\alpha,\beta}$ at which $ADAPT(S)$, $ADAPT$ (1), and $ADAPT$ found their best solution. The number of iterations is plotted versus task parallelism. Two plots are shown for each algorithm: one for low communications ($0 \leq \alpha \leq 0.5$) and another for high communications ($3 \leq \alpha \leq 4$). For a given computation, increasing task parallelism leads to a decrease in the number of possible task-processor mapping as a result of decreasing the number of processors. Thus higher values of β mean lower number of alternatives for task-processor mapping. This effect is shown for all of the above three decision functions.

Increasing the communication requirements of a problem instance leads to an increase in the number of possible scheduling decisions that can be attempted in searching to optimize the solution. Therefore, increasing α leads to an increase in $N_{\alpha,\beta}$. Coarse-grain computation (low α) requires far fewer iterations for finding the best solution

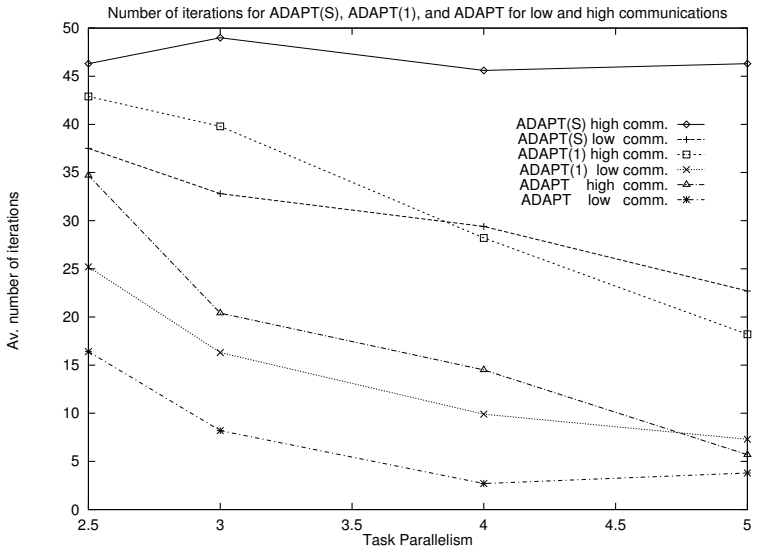


Figure 9: Typical number of iterations to find best solutions for $ADAPT(S)$, $ADAPT(1)$, and $ADAPT$

than fine grain computations (large α). Ranking the above algorithms in increasing order of $N_{\alpha,\beta}$ is $ADAPT$, $ADAPT(1)$, and $ADAPT(S)$ which is also the order of least deviation from the optimum solution. $ADAPT$ requires the least number of iterations to achieve the least deviation from optimum. The difference in the number of iterations needed in the case of low and high communications is bounded by 25 iterations for $ADAPT(S)$ and $ADAPT(1)$ and bounded by only 16 iterations for $ADAPT$. This shows the benefits of adapting the decision function of $ADAPT$ to the computation profile to speed convergence to the best solution.

7.3 Comparison to other approaches

Pase [19] experimentally studied 12 scheduling heuristics ($S_1 - S_{12}$) including the *task duplication* technique ($O(pn^2)$) [15] and ETF [12] (S_2). His heuristic (S_1) assigns priority from the graph bottom and selects the task that is closest to top. The priority function [19] is evaluated based on task computation times but neither account for the communication edges nor the network latency. He found that S_1 and ETF are among the best heuristics and both outperform the TD scheduler of [15]. Our study indicates that the use of $ADAPT$ with iterative scheduling significantly outperforms ETF versus change in communication and task parallelism.

Table 1 compares the average percentage improvement of the schedule finish time generated by some scheduling heuristics. The plotted deviations are evaluated as $(1 - \omega_X/\omega_Y)100$, where ω_X and ω_Y are the average finish times generated by scheduling heuristics H_X and H_Y, respectively.

The heuristic called *Dominant Sequence Clustering* DSC [25] was proposed for scheduling DAGs on an unbounded number of completely connected (FC) processors. DSC scheduling improves the clustering approach presented in Sarkar and Hennessy [21] but slightly outperform (3.3%) ETF as reported by Yang and Gerasoulis [25].

				IRS		ADAPT	
Comm/Comp	DSC/ETF	DSC/Sarkar	ETF	HLETF	Iter.	ADAPT	Iter.
0.1-0.3	0.06	5.56	3-4	0.3-1.3	22	0.2-0.6	5
0.83-1.25	3.3	20.74	4-12	0.7-6	40	0.4-2.6	20
3.3-10	2.36	19.39	10-28	7-9	50	1-2.6	40

Table 1: Comparison of average deviation of DSC, Sarkar, ETF, IRS, and ADAPT

Heuristic ETF is used as reference in [25] and this work, therefore we can compare our work to that reported in [25, 21]. A schedule generated by using DSC ($O(n \log n)$) is a few percent shorter (Table 1) than one generated by using ETF [25] when communication granularity is in the range of $0.83 - 1.25$ and $3.3 - 10$. Table 2 shows that ETF schedules deviate from optimum by 3% (low β) to 4% (high β), 4% to 12%, and 10% to 28% in the ranges of low, medium, and high communication repectively.

The *dynamic critical path* DCP [16] was tested over a number of known task graphs for which the size was varied, and seven scheduling heuristics were run on each graph instance. DCP schedules were shorter than ETF schedules by at most 7% in all studied cases and problems [16]. However, the running time of ETF was also found to be among the best [16] when compared to DSC [25], MCP [24], EZ [21], DCP [16], MD [24], and DLS [23].

The *iterative refinment scheduling* (IRS) [5] uses the HLETF for each scheduling iteration. Due to its iterative refinment, IRS scheduling largely outperforms ETF over all the studied range of parallelism and communication. Table 2 shows that IRS schedules are near optimum in the range of low to medium communication. However, IRS requires about 50 iterations in the range of high communications to achieve a schedule length that deviates by 7% (low β) to 9% (high β) from optimum. Note that the cost of running IRS is linear with the cost of running ETF scheduling and the number of iterations.

A priority-based scheduling is the *dynamic level scheduling* DLS [23]. No performance data is presented for DLS. The decision function of DLS is similar to $ADAPT(S)$ but with the difference that DLS uses a static task-level and $ADAPT(S)$ uses the completion time $ct(T)$ of T from an iteration as the task level for the next iteration. Thus the task-level for $ADAPT(S)$ is some combination of computation and communication as achieved by the scheduler. However, $ADAPT$ has three improved features compared to DLS which are: (1) a more accurate task-level (lap) that accounts for computation and communication along directed paths, (2) adapting the decision function to a computation profile, and (3) possible refining of the solution through the use of iterative scheduling. Finally, Table 2 compares the time complexities of some well-known scheduling methods.

$ADAPT$ schedules are comparable to those of IRS in the range of low and medium communications. However, $ADAPT$ largely outpeforms ETF and IRS in the case of coarse grain communication and low parallelism as shown on Table 1. $ADAPT$ provides a scheduling performance that is independent of the instance of communication and parallelism and enables a refinment of the solution through iterative scheduling.

Heuristic	*Operability*	*Time complexity*
Sarker (EZ) [21]	Unbounded	$O(e(e+v))$
Yang (DSC) [25]	Unbounded	$O((n+e)logn)$
Wu (MCP) [24]	Bounded	$O(n^2logn)$
Wu (MD) [24]	Unbounded	$O(n^3)$
Hwang (ETF) [12]	Bounded	$O(pn^2)$
Ahmed (DCP) [16]	Unbounded	$O(n^3)$
Kruatrachue (DSH) [15]	Unbounded	$O(n^4)$
Al-Mouhamed (IRS) [5]	Bounded	$O(pn^2 \times iterations)$
($ADAPT$)	Bounded	$O(pn^2 \times iterations)$

Table 2: Comparison of time complexities

8 Conclusion

Minimizing the schedule finish time of precedence-constrained computations with communications times has been addressed by using quite different approaches ranging from genetic algorithms to clustering. Graham's list scheduling requires that each task be associated with its task-level which, due to non-zero communication, cannot be determined without task-processor binding. ETF [12] and DLS [23] were two attempts to redesign the scheduling heuristic based on list scheduling. ETF preserves Garham's global clock and ensures that the starting times of successively scheduled tasks are non-decreasing. This guarantees that ETF schedules satisfy Graham's bound with an additional term for non-overlapped communications. The price of meeting the worst case bound is that the performance of the scheduler depends upon the computation profile. The DLS obtained encouraging results after relaxing the global time and introducing a static priority to assist the earliest-task-first discipline.

Our work is an extension of the above effort. We have presented a compile-time adaptive scheduling having no global time and using a decision function that incorporates: (1) an earliest-task-first discipline, and (2) a dynamic task-level. This ensures good management of processor idle time when there is enough parallelism to hide the communications and provides a means of distinguishing dominant tasks from others when needed. To eliminate the computation dependence aspect a computation profile factor was introduced and used to control the dominance (weight) of the above two disciplines. Due to the dynamic nature of the task-level the scheduling was proposed within an iterative framework which allows the exploration of a variety of solutions corresponding to a variety of task-levels. Iterative scheduling is an evolutionary process (deterministic) that has hereditary variation and differential production.

Evaluation was carried out for a wide category of computation graphs with communications for which optimum schedules are known. It was found that pure local scheduling and static priority-based scheduling significantly deviate from the optimum under specific problem instances. Our approach to adapting the iterative scheduling decision to computation profile was able to produce near-optimum schedules with a reasonable number of iterations within the limits of the studied computation problems.

References

[1] T.L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. *Comm. of the ACM*, 17, No 12:685–690, Dec 1974.

[2] M. Al-Mouhamed. Lower bounds on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Trans. on Software Engineering*, 16, No 12:1390–1401, 1990.

[3] M. Al-Mouhamed. Analysis of macro-dataflow dynamic scheduling on non-uniform memory access architectures. *IEEE Trans. on Parallel and Distributed Systems*, 19, No 3:875–888, Nov 1993.

[4] 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.

[5] M. Al-Mouhamed and A. Al-Maasarani. Scheduling optimization through iterative refinement. In *Inter. Conference on Parallel Architecture and Compilation Techniques (PACT'95)*, pages 178–184, Cyprus, 1995.

[6] E.G. Coffman et al. *Computer and Job-Shop Scheduling Theory.* John Willey and Sons, 1976.

[7] S. Darbha and S. S. Pande. Effect of imprecise compile time costs on scheduling tasks on distributed memory systems. *Frontiers '96: The 6th IEEE/ACM Symposium on the Frontiers of Massively Parallel Computation*, pages 134–141, 1996.

[8] S. Darbha and S. S. Pande. A robust compile time method for scheduling task parallelism on distributed memory systems. *The 1996 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques (PACT'96)*, pages 156–162, 1996.

[9] S. Darbha and K. Psarris. Program repartitioning on varying communication cost parallel architectures. *Journal of Parallel and Distributed Computing*, 33:205–213, March 1996.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] Yu-Kwong Kwok and Ishfaq Ahmad. On using task duplication in parallel program scheduling. *Under review with IEEE Trans. on Parallel and Distributed Systems*, 1996.

[18] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19, No. 2:322–328, April 1990.

[19] D.M. Pase. A comparative analysis of static parallel schedulers where communication costs are significant. *Ph.D. Thesis, Oregon*, Jul 1989.

[20] 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.

[21] 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.

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

[23] 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.

[24] M.-Y. Wu and D.D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1, No 3:330–343, Jul 1990.

[25] 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.