A Compile-Time Scheduling Heuristic for Multiprocessor Architectures

GORAN LJ. DJORDJEVIĆ AND MILORAD B. TOŠIĆ

Faculty of Electronic Engineering, University of Nis, Beogradska 14, P.O. BOX 73, 18000 Nis, Serbia, Yugoslavia
Email: gdjordjevic@europa.elfak.ni.ac.yu

The multiprocessor scheduling problem can be stated as finding a schedule for a task graph to be executed on a multiprocessor architecture so that the execution time can be minimized. Since this problem is known to be NP-hard, in all but a few very restricted cases, the main research efforts in this area are focused on heuristic methods for obtaining near-optimal solutions in a reasonable amount of time. A new compile-time single-pass multiprocessor scheduling technique, called chaining, has been developed and is presented in this paper. Chaining can be used to schedule task graphs onto multiprocessor architectures that contain an arbitrary number of processors connected in an irregular fashion, taking into account the expected execution and communication requirements of the task graph on the given multiprocessor architecture. This technique can be viewed as a generalization of the list scheduling technique, that does not impose any preconditions about the ordering according to which tasks are selected for scheduling. Varying the selection criteria, implemented in this technique, we have generated a new class of scheduling algorithms. An evaluation of this class was made on 360 randomly generated examples, and the estimated performances were compared with two list scheduling algorithms, the dynamic level scheduler proposed by Sih and Lee, and the earliest task first algorithm proposed by Hwang et al.

Received June 28, 1995; revised July 25, 1996

1. INTRODUCTION

The scheduling of tasks to processors is an essential step in exploiting the capabilities of a multiprocessor system. To achieve an effective mapping of the parallel program tasks onto processors, scheduling strategies that take into account both the interprocessor communication overheads and architectural characteristics are required. The scheduling can be performed either dynamically (i.e. at runtime) or statically (i.e. at compile-time). Static scheduling has the following two benefits: (i) it is easier to realize; and (ii) it eliminates run-time overheads. Also, the compile-time scheduling is widely used especially in digital signal processing applications, where the individual tasks have deterministic processing and communication times [1].

In a static task scheduling approach, a parallel program is represented by a weighted directed acyclic graph (DAG) called task graph (TG). Between the set of nodes and set of tasks one-to-one correspondence exists. The set of directed edges models both the precedence constraints and data paths. Associated with each task is its computation cost, and with each edge its communication cost. The target multiprocessor architecture consists of an arbitrary number of processors, connected via an arbitrary interconnection topology. We adopt a task execution model where each task is executed non-preemptively (i.e. during one continuous time interval) on a single processor. The scheduling consists of: (i) task assignment to processors; and (ii) ordering of tasks for execution in each processor. The precedence constraints specified by the TG limit the amount of parallelism, because any pair of tasks related by a path in the TG cannot execute concurrently. The scheduling goal is to minimize the total execution time of the TG, i.e. the schedule length. This problem is proven to be NP-hard [2]. As a consequence, optimal solution strategies are often sacrificed in favour of fast heuristic approximation techniques, even for small sized problems.

List scheduling (LS) is a well-known single-pass scheduling heuristic, characterized by low (small-exponent polynomial) computational complexity. In the LS, an unscheduled task is said to be ready if all its predecessors are scheduled. At each step, first both a ready task and a processor are selected. Then, the task is scheduled by adding to the top of the sequence of tasks previously scheduled in the processor. Several different heuristics for 'ready task-processor' selection have been suggested for use in the LS [1, 3–8]. Simulations conducted in [4] and [8] indicate that the LS demonstrates near-optimality when inter-task communication is negligible. However, when inter-task communication is non-negligible a trade-off between parallelism and communication overheads have
to be considered. Parallelism dictates assignment of tasks to different processors. On the other hand, communication overhead is zeroed when tasks are assigned to the same processor. To accomplish this trade-off, various extensions of the basic LS principle were proposed. The earliest task first (ETF) strategy, presented in [5], considers the time delay imposed by message transmission, and attempts to schedule some tasks as early as possible on each processor as it becomes free. The dynamic level scheduling algorithm [1] uses dynamically changing priorities to match tasks and processors for scheduling at each scheduling step. Insertion scheduling heuristic (ISH) [6] and algorithm described in [7] allow the task to be scheduled into 'scheduling hole', where 'scheduling hole' represents the time interval during which the processor is idle, occurring before the task last scheduled to the processor is finished. Duplication scheduling discipline (DSH) [6] schedules some tasks on more processors with a goal to reduce communication delays.

Clustering techniques are widely used for scheduling task graphs on an unbounded number of processors in presence of inter-task communication [2, 9, 10]. These schemes try to avoid excessive interprocessor communication cost by grouping heavily communicating tasks into the same clusters, which are then mapped onto separate processors. Most of the existing clustering algorithms can be characterized by using a framework described in [9]. The initial step assumes that each task is mapped in an unit cluster. At each step, the algorithm tries to improve on the previous clustering by merging the appropriate clusters, i.e. by zeroing an edge cost connecting two clusters. In [10] a low-complexity heuristic, the dominant sequence clustering (DSC) algorithm is presented. This algorithm has performances comparable with, or even better, on average, than much higher-complexity clustering heuristics. In [2] a two-step scheduling method that extends clustering technique in a way that allows scheduling on a bounded number of processors is suggested. In the first step, called internalization pre-pass, the nodes are clustered together in an attempt to minimize the execution time of the TG on unbounded number of processors. In the second step, called a processor assignment phase, the finishing clusters are merged and mapped to the physical processors by using a modified list scheduling algorithm. The main disadvantage of this two-step approach is that it often faces significant difficulties when mapping clusters to the processors: namely, inability of the clustering algorithm, used for internalization pre-pass, to take into account the number of processors when choosing the granularity of the clusters, may cause load imbalance (see for example [11]).

In this paper we describe a compile-time scheduling strategy, called chaining. Using chaining we map TGs onto multiprocessor architectures that contain an arbitrary number of processors connected in an irregular fashion. Chaining is a single-pass scheduling heuristic and can be viewed as a generalized concept of the LS. With an aim to represent a partial schedule and a valid (complete) schedule we define both partially scheduled task graph and scheduled task graph. Both graphs are directed acyclic multigraphs (DAMs) and are obtained by extending the original TG with new precedence edges so that tasks scheduled to the same processor are arranged into a linear cluster or chain. Therefore, we look at the scheduling as a process that transforms the original TG into the scheduled task graph. As an outcome of each intermediate step, a new partially scheduled task graph is obtained. Chaining is a scheme that performs this transformation by scheduling one task at each step. A fundamental property of this scheme is that it does not impose any constraints about the order according to which tasks are selected for scheduling. This
scheduling freedom allows us to obtain a new class of efficient scheduling algorithms by specifying different rules by which the task and its insertion location are selected.

The paper is organized as follows. Section 2 gives some basic terminology about DAGs, precisely describes the adopted task execution model and the target multiprocessor architectures, and introduces the notions of scheduled task graph and partially scheduled task graph. Section 3 presents a detailed description of the proposed scheduling technique. Section 4 describes a new class of chaining-based scheduling algorithms. Section 5 presents the results of performance evaluation for the proposed class of algorithms. Finally, Section 6 summarizes this discussion.

2. MULTIPROCESSOR SCHEDULING PROBLEM

A TG is a weighted DAG defined by a tuple \(G = (V, E, W, C)\), where \(V = \{s = n_0, n_1, ..., n_m, q = n_{m+1}\}\) is the set of nodes (tasks), \(E = \{e_{i,j} = (n_i, n_j)\}\) is the set of communication edges, \(W\) is the set of node weights and \(C\) is the set of edge weights. The value \(w_i \in W\) is the execution time of task \(n_i \in V\). The value \(c_{i,j} \in C\) is the amount of data (in bits, bytes or words) that \(n_i\) passes to \(n_j\). Nodes \(s\) and \(q\) are 'dummy' tasks, with zero execution time \((w_0 = w_{m+1} = 0)\), which represent the unique entry and exit point of the TG, respectively. The \(s\) is the initial task of the TG, i.e. it precedes all other tasks, while \(q\) is the terminal task of the TG, i.e. it succeeds all other tasks. The weight of the edges connecting \(s(q)\) with any other node is assumed to be zero \((c_{0,i} = c_{i,m+1} = 0)\). Tasks \(n_1, ..., n_m\) are computation tasks. An example of the TG, with seven computation tasks, is shown in Figure 1a. The numbers inside the nodes correspond to node weights, while the numbers accompanied with the edges correspond to edge weights; if no number is written the node/edge weight is zero.

Given a TG \(G\) and a number of processors \(p\), the multiprocessor scheduling problem is to partition tasks in the \(G\) into \(p\) computation-task-disjoint execution sequences so that the precedence constraints are satisfied and the execution time of the \(G\) is minimized.

2.1. Model

In order to determine a schedule certain assumptions about the task execution model and the multiprocessor architecture need to be specified. We assume:

- A copy of the zero-weighted 'dummy' task \(s(q)\) is 'executed' as the first (last) task on each processor. Each computation task is executed by a single processor (i.e. duplication of the same computation tasks in separate processors is not allowed). The tasks are executed without interruption (i.e. non-preemptively). After the processor has finished an assigned task, it starts the execution of the next assigned task as soon as all input data of that task are stored into the local data buffer (we call this assumption earliest task starting time). After completion, the task sends output data to all successors in parallel.

- The target architecture is a multiprocessor system made up of an arbitrary number \(p\) of homogeneous processors, \(P_k, k = 1, ..., p\). Processors communicate by sending messages through a point-to-point interconnection network. The interconnection topology is arbitrary, while the communication is packet switching. The time required to send a data unit from the \(P_i\) to \(P_j\) is denoted by \(d(i, j)\). The parameter \(d(i, j)\) is the topology factor and is proportional to the number of
hops between the $P_i$ and $P_j$. Moreover, $d(i,j) = 0$ if $i = j$. Communication links are assumed to be full-duplex and contention free. Each processor contains specialized hardware which enables it to perform both the computation and communication in parallel, and to handle simultaneously all its communication channels.

The adopted task execution model is identical to the compile time macro-dataflow model presented in [2]. Some possible multiprocessor architectures onto which the TG can be scheduled comprise: hypercube, mesh, star, ring, for example. Communication overhead due to link contention can be assumed negligible if the communication load is much smaller than the communication bandwidth provided by the interconnection network.

### 2.2. Scheduled task graph

A Gantt chart is a standard form for schedule representation. Figure 1b shows a Gantt chart of a schedule for the TG given in Figure 1a for two-processors system ($p = 2$). We assume that for transfer of $D$ data units, between two processors, $D$ time units are needed. A Gantt chart completely describes the schedule, since it defines both the task execution sequences and the task starting times, but precedence constraints originated from TG are not visible. Alternatively, modeling of schedule which includes all inter-task dependencies, those defined by TG as well as new ones introduced by the schedule, can be done by means of a scheduled task graph (STG). The STG derived from TG $G = (V,E,W,C)$ for the $p$-processors system is a weighted DAM $STG_p = (V,E+R,W,T)$. The STG has identical node set as $G$ and its edge set includes all edges in $G$. The set of additional edges $R$ models the execution order of tasks assigned to each processor. For every pair of tasks $n_i$ and $n_j$ assigned to the same processor such that $n_i$ is scheduled for execution immediately after $n_i$ we add a zero-weighted directed edge $(n_i,n_j) \in R$. We call these edges $p$-edges. The path consisting of $p$-edges, only, we call $p$-path. The $p$-chain is a subgraph of the $STG_p$ induced by a $p$-path which starts in node $z$ and ends in node $q$. Let $\{p_1, p_2, \ldots, p_p\}$ be a set of all $p$-chains in the $STG_p$. We assume that the $P_i$ executes $p_i$. The weight of the $STG_p$'s communication edge corresponds to the number of time units needed by the system to transfer data carried by that edge in the $G$. Namely, the weight of the communication edge $e_{i,j} \in E$ in the $STG_p$, $t_{i,j}$ is calculated as

$$t_{i,j} = \pi(n_i) \cdot d(\pi(n_i), \pi(n_j)),$$

where $\pi(n_i)$ denotes the index of the processor to which $n_i$ is assigned.

For example, a schedule given in a form of Gantt chart (Figure 1b), for TG in Figure 1a, has an equivalent representation form in $STG_2$ shown in Figure 1c. In Figure 1c dashed lines indicate $p$-edges.

### 2.3. Partially scheduled task graph

Generally speaking, scheduling is a process during which a sequence of partial schedules (i.e. incomplete assignments of task sets to processors) is produced. In order to represent the structure of a partial schedule, we define a partially scheduled task graph. An example of such a graph is shown in Figure 2a.

Given TG $G = (V,E,W,C)$, the number of processors $p$, and an integer $0 \leq k \leq v = |V| - 2$, a partially scheduled task graph is a weighted DAM $STG_{p,k} = (V,E+R^k,W,T^k)$ obtained by adding $p + k$ $p$-edges to the $G$, so that they form $p$ computation-task-disjoint $p$-chains. $R^k$ is the set of $p$-edges, and $T^k$ is the set of communication edge weights in the $STG_{p,k}$. Let $\{\rho^1_k, \rho^2_k, \ldots, \rho^p_k\}$ be the set of all $p$-chains in the $STG_{p,k}$, and $\rho^i_k = (R^k_i, V_i)$, where $R^k_i \subset R^k$ is the set $p$-edges, and $V_i \subset V$ is the set of nodes in the $\rho^i_k$. All tasks in the $STG_{p,k}$ can be partitioned into two disjoint sets: the set of scheduled tasks, $ST = \cup_{i=1}^{p} V_i$, and the set of unscheduled tasks, $UT = V \setminus ST$. The total number of scheduled computation tasks in the $STG_{p,k}$ is $k$. Similarly, all communication edges in the $STG_{p,k}$ can be partitioned into two disjoint sets: the set of scheduled edges $SE = \{e_{i,j} | n_i \in ST \land n_j \in ST\}$ and the set of unscheduled edges $UE = E \setminus SE$. Clearly, a communication edge is scheduled if both tasks joined by that edge are scheduled. Otherwise, a communication edge is unscheduled. The weight of a communication edge $e_{i,j}$ in the $STG_{p,k}$, $t_{i,j}^k \in T^k$, depends on whether that edge is scheduled or not, and it is defined as

$$t_{i,j}^k = \begin{cases} l_{i,j} \cdot c_{i,j} & \text{if } e_{i,j} \in SE \\ l_{i,j} & \text{if } e_{i,j} \in UE \end{cases}$$

where $l_{i,j}$ denotes the weight of a scheduled communication edge evaluated according to Equation (1), and $\lambda \in R^+$ is a tunable parameter, representing a presumed time required by the system to transfer one data unit between any two tasks. Before scheduling of the communication edge the real communication time cannot be known. Therefore, $\lambda \cdot c_{i,j}$ can be viewed as an expected time needed by the system to transfer $c_{i,j}$ data units between tasks $n_i$ and $n_j$.

Note that the same partial schedule can be represented by a number of partially scheduled task graphs, which differ with respect to the parameter $\lambda$. Let us note that the scheduled task graph is a special case of the partially scheduled task graph, i.e. $STG_p = STG_{p,0}$. The graph shown in Figure 2a is $STG_{2,0,0}$ derived from TG in Figure 1a for $p = 2$, and $\lambda = 0.5$. For this graph the following is valid: $ST = \{s, n_1, n_2, n_6, n_7, q\}, UT = \{n_3, n_4\}, SE = \{e_{0,1}, e_{1,5}, e_{5,6}, e_{6,7}, e_{7,8}\}$ and $UE = \{e_{0,2}, e_{0,3}, e_{1,4}, e_{2,6}, e_{3,6}, e_{4,7}\}$.

### 2.4. Definitions and notation

For a given $STG_{p,k}$, $(V,E+R^k,W,T^k)$, let $\text{Succ}^k(n_i)(\text{Pred}^k(n_i))$ denote the set of all successors (predecessors) of node $n_i \in V$ (not necessarily immediate). With respect to $n_i$, we say that node $n_j \in V, j \neq i$, is of
the type $P$, $S$ or $I$ if $n_y \in \text{Pred}^t(n_x)$, $n_y \in \text{Succ}^t(n_x)$ or $n_y \notin \text{Pred}^t(n_x) \wedge n_y \notin \text{Succ}^t(n_x)$, respectively. Similarly, we say that an edge $(n_x, n_z) \in E + \mathcal{R}^t$ is of type $AB$ with respect to node $n_z$ if $n_z$ is of type $A$, and $n_x$ is of type $B$, with respect to $n_z$, where $A, B \in \{P, S, I\}$.

The length of a path in the STG$_{P, S, I}$ is the summation of all node and edge weights along the path. The critical path is a path with longest length in the graph. The length of the STG$_{P, S, I}$ (or partial schedule length), denoted as $cp(\text{STG}_{P, S, I})$, is the length of the STG$_{P, S, I}$'s critical path.

The entry distance of node $n_x$ in the STG$_{P, S, I}$, denoted as $l^t(n_x)$, is the length of the longest path connecting node $s$ to node $n_x$, excluding the weight of $n_x$. Symmetrically, the exit distance of the node $n_x$, denoted as $e^t(n_x)$, is the length of the longest path connecting node $n_x$ to node $q$, excluding the weight of the node $n_x$. The entry distance of $s$ and exit distance of $q$ are assumed to be zero.

The following formulae can be used to determine the entry and exit distances of $n_x$ in the STG$_{P, S, I}$:

\begin{equation}
l^t(n_x) = \begin{cases} 0 & \text{if } \text{ImPred}^t(n_x) = \emptyset \\ \max_{n_y \in \text{ImPred}^t(n_x)} [e^t(n_y) + w_y + t^x_{x,y}] & \text{else} \end{cases}
\end{equation}

\begin{equation}
e^t(n_x) = \begin{cases} 0 & \text{if } \text{ImSucc}^t(n_x) = \emptyset \\ \max_{n_y \in \text{ImSucc}^t(n_x)} [e^t(n_x) + w_y + t^x_{x,y}] & \text{else} \end{cases}
\end{equation}

where $\text{ImPred}^t(n_x)$ and $\text{ImSucc}^t(n_x)$ denote the set of all immediate predecessors and immediate successors, respectively.

If the values $l^t(n_x)$ and $e^t(n_x)$ are known, then we can calculate the $lp^t(n_x)$ as the length of the longest path passing through the $n_x$ in the STG$_{P, S, I}$, i.e.

\begin{equation}
lp^t(n_x) = l^t(n_x) + w_x + e^t(n_x).
\end{equation}

To calculate the length of STG$_{P, S, I}$, the following formula can be used

\begin{equation}
cp(\text{STG}_{P, S, I}) = \max_{i} \{lp^t(n_i)\}.
\end{equation}

The relative mobility of a task $n_x$ in the STG$_{P, S, I}$ is defined as a difference between the length of the STG$_{P, S, I}$ and the length of the longest path passing through $n_x$ in the STG$_{P, S, I}$, divided by the weight of $n_x$:

\begin{equation}
rtm^t(n_x) = \frac{cp(\text{STG}_{P, S, I}) - lp^t(n_x)}{w_x}.
\end{equation}

The width of a $p$-edge $(n_i, n_j)$ in the STG$_{P, S, I}$ is defined to be

\begin{equation}
wd^t((n_i, n_j)) = cp(\text{STG}_{P, S, I}) - (l^t(n_i) + w_j + e^t(n_j)).
\end{equation}

Note that if a schedule is represented by the STG$_p$, then task starting times are not explicitly given, but can be
computed by topological traversing of the \( STG_p \). Having in mind the earliest task starting time assumption, the starting time of the task \( n_x \) is equal to the entry distance of the corresponding node in the \( STG_p \), i.e., \( l'(n_x) \). The execution time of the TG is determined by the critical path of the \( STG_p \). It is equal to the length of the \( STG_p \), i.e., \( cp(STG_p) \). Note that the multiprocessor scheduling problem can be stated as a graph-theoretical problem of derivation of the minimal length \( STG_p \) from a given TG.

### 3. Chaining

Our approach concerning the multiprocessor scheduling problem is based on considering a scheduling algorithm which directly constructs \( STG_p \). Having this in mind, we propose a single-pass scheduling technique, called chaining. This technique constructs the \( STG_p \) incrementally, in a step-by-step fashion, by scheduling one task at each step and using the \( STG_p \) to represent the state of the scheduling process during the \( k \)th scheduling step.

Let us consider the scheduling of the TG \( G = (V,E,W,C) \), with \( v = |V| - 2 \) computation tasks, onto \( p \) processor system. The proposed scheduling technique works as follows. During the initial step \( STG_{p,0} \) is created. The algorithm extends the \( G \) with \( p \) new nodes connecting node \( s \) to node \( q \). As a consequence, tasks \( s \) and \( q \) are identified as scheduled tasks, \( ST = \{s,q\} \), and all computation tasks are identified as unscheduled tasks, \( UT = \{n_1, n_2, \ldots, n_v\} \). Also, each communication edge \( e_{ij} \) is marked as unscheduled edge and a new weight [defined by Equation (2)] is assigned.

At the \( k \)th scheduling step, \( k = 1, \ldots, v \), one task in the \( UT \) is scheduled, transforming the \( STG_{p,k-1} \) into the \( STG_{p,k} \). The task is scheduled by its incorporation into a \( p \)-chain, i.e., by inserting it between two already scheduled tasks connected by a \( p \)-edge. Due to the precedence constraints specified by the \( STG_{p,k-1} \), an unscheduled task \( n_s \) can only be inserted between scheduled tasks \( n_s \) and \( n_b \) which satisfy the following condition:

\[
 n_s \notin Succ^{k-1}(n_s) \land n_b \notin Pred^{k-1}(n_s) \tag{9}
\]

For now on, a \( p \)-edge \((n_a, n_b)\) will be considered valid with respect to unscheduled task \( n_s \) if it satisfies the Equation (9). According to the definition of edge type (see Subsection 2.2) a \( p \)-edge is valid with respect to \( n_s \), if that edge is of the type \( PS, IS, PI \) or \( II \). We use \( VR^k(n_s) \) to denote the set of all \( p \)-edges in \( STG_{p,k} \), which are valid with respect to \( n_s \).

To perform the task scheduling operation, the algorithm first selects the task \( n_e \in UT \) which should be scheduled, then determines the set \( VR^{k-1}(n_e) \), after that chooses a \( p \)-edge \((n_a, n_b) \in VR^{k-1}(n_e) \) and finally, inserts the \( n_e \) between tasks \( n_a \) and \( n_b \). Task insertion is performed as follows. The \( p \)-edge \((n_a, n_b) \) is destroyed and two new \( p \)-edges, \((n_a, n_e) \) and \((n_e, n_b) \), are created. Then all communication edges connecting \( n_e \) with scheduled tasks are marked as scheduled edges; and for each such edge a new weight, which corresponds to the real communication time [defined by Equation (1)], is calculated. Since the task insertion is completed we say that the \( n_e \) is placed into the \( p \)-edge \((n_a, n_b) \). In this way, the \( n_e \) is transferred from the \( UT \) to the \( ST \). The \( STG_{p,k} \) created during the \( k \)th scheduling step represents a starting point for the next step. This process is repeated until the \( UT \) is empty.

The essential problem concerning the correctness of the proposed scheduling procedure is based on the following fact. During each step for the selected unscheduled task, does a valid \( p \)-edge exist? The answer is given by the following lemma.

**Lemma 3.1.** In the \( STG_{p,k-1}, k = 1, \ldots, v \), for arbitrary \( n_x \in UT \), there exists at least one \( p \)-edge, in every \( p \)-chain, which is valid with respect to \( n_x \).

**Proof.** Let consider the \( STG_{p,k-1} \) and \( n_x \in UT \). As \( s \) is the predecessor and \( q \) is the successor of every computation task, the sequence of \( p \)-edge types with respect to \( n_x \), along any \( p \)-chain in the \( STG_{p,k-1} \), has the following form: \( PX_1, X_1X_2, \ldots, X_s, S \), for \( X_i \in \{P, S, I, I\} \). It is quite apparent that for any permutation of \( (X_1, X_2, \ldots, X_s) \), there exists at least one \( p \)-edge of a valid type (i.e. \( PS, IS, PI \) or \( II \)).

Lemma 3.1 ensures that the scheduling process will not be prematurely stopped, because the set \( VR^{k-1}(n_x) \) is empty. Let us note that no constraints about the ordering according to which tasks are selected for scheduling are imposed by the Lemma 3.1.

### 3.1. Complexity

The chaining procedure takes \( v \) steps to schedule the TG with \( v \) computation tasks. Therefore, its complexity can be expressed as \( O(v \cdot f) \), where \( f \) is the complexity of one scheduling step. The \( f \) is a three term sum: \( f = K_1 + K_2 + K_3 \), where \( K_1 \) is the complexity of task selection, \( K_2 \) is the complexity of determining the set of valid \( p \)-edges and \( K_3 \) is the complexity of \( p \)-edge selection. The terms \( K_1 \) and \( K_2 \) depend on the heuristic which is used to make corresponding selections. In order to identify valid \( p \)-edges, all nodes in the partially scheduled task graph need to be marked with an appropriate type (e.g., \( P, S \) or \( I \)) with respect to the selected task \( n_x \). To do this, since all nodes are marked as \( I \), starting from \( n_x \) topological traversing in upward and downward directions of the partially scheduled task graph is needed. All nodes visited during upward (downward) traversing are marked as \( P \) (\( S \)). The complexity of the topological traversing of the DAG is \( O(\epsilon + v) \), where \( v \) is the number of nodes, and \( \epsilon \) is the number of edges in the DAG. Thus, the complexity of the chaining procedure is \( O(\epsilon (K_1 + K_3) + \epsilon (v + \epsilon)) \). Obviously, the worst case complexity of the chaining procedure cannot be less than \( O(\epsilon (v + \epsilon)) \).

**Example 1.** We demonstrate one scheduling step of the chaining procedure by using the \( STG_{1,4,0.5} \) shown in Figure 2a. Suppose this partially scheduled task graph is created at the 4th step of the chaining procedure that schedules...
the TG in Figure 1a on two-processor system. Let $n_k$ be the task selected for scheduling at the 5th scheduling step. The node marking with respect to $n_k$ is shown in Figure 2b. Accordingly, the set of $\rho$-edges valid with respect to $n_k$ is $V R^4(n_k) = \{(s, n_k), (n_1, n_k), (n_5, n_k), (n_7, n_k), (n_9, q), (n_9, q)\}$. Each $\rho$-edge in the $V R^4(n_k)$ represents a valid scheduling option. Suppose the algorithm chooses the $\rho$-edge $(n_9, q)$. The $STG_{2,5,0.5}$ which comes out as a result of this decision is shown in Figure 2c.

A similar approach that we use in chaining can be found in DSC algorithm [10]. Like the chaining, the DSC is based on a graph transformation technique that transforms the TG into a some form of $STG$. However, while doing this transformation, the DSC strictly follows the topological order of task selection. Moreover, the DSC is intended for scheduling on an unbounded number of completely connected processors, and can only be used for the first step of Sarkar's two step scheduling approach. On the other hand, the chaining performs scheduling in the single pass, simultaneously taking into account both the task graph and architectural characteristics.

The chaining represents a scheduling framework that allows us to obtain a number of multiprocessor scheduling algorithms by incorporating different heuristics for task and $\rho$-edge selections. For example, it is easily to see that most of LS variants can be characterized as chaining algorithms, where during each step the unscheduled task with all its predecessors already scheduled is selected and placed into a terminal $\rho$-edge (i.e. $\rho$-edge ending in node $q$). However, with chaining we are able to schedule tasks in any order and to place the selected task into any valid $\rho$-edge. An interesting question is: can the additional scheduling freedom be used to obtain scheduling algorithms with better performance with respect to the previous single-pass scheduling algorithms? This question has motivated us to make an extensive research on the effectiveness of different selection heuristics which can be incorporated into the proposed scheduling framework. The outcome of this research is a new class of multiprocessor scheduling algorithms. Detailed description of this class of algorithms and performance simulation results of different algorithms are given in the following two sections.

4. SELECTION RULES

When we schedule TGs using chaining technique the following issues have to be dealt with during each scheduling step: (i) Which task to schedule? And (ii) Where to place the selected task? Since the chaining procedure addresses both issues in separate phases and utilizes heuristic, a task selection rule and a $\rho$-edge selection rule need to be specified. The chosen 'unscheduled task–$\rho$-edge' pair should be one which appears to be the most capable of leading to a minimal length schedule. Clearly, effective and correct identification of such a pair during each scheduling step is difficult to perform, but is important for the goodness of the algorithm. Since the chaining is a single-pass algorithm it is reasonable to apply greedy heuristics, i.e. those that perform selections in a local optimal manner. Intuitively, we need selection rules which are capable of finding effectively a trade off between the maximal total workload (computation + communication) scheduled during each scheduling step, and the minimal length extension of the partial schedule. However, we confine our analysis only to such selection rules that will not increase the worst case complexity of the chaining procedure over $O(\nu(v + e))$. Consequently, the selection rules with complexity greater than $O(v + e)$ are not taken into consideration.

4.1. Task selection rules

For the chaining procedure let us consider task selection during the $k$th step, when the $STG_{p_k, \lambda}$ is transformed into the $STG_{p_k, \lambda}$. In this study we consider the following four criteria for task selection:

- $WT$: choose the unscheduled task with the largest weight. This criterion considers the computation time of individual tasks and does not include communication effects.
- $SW$: calculate the scheduled workload for each unscheduled task. Choose the unscheduled task with the maximal scheduled workload. The scheduled workload of the unscheduled task $n_k$ is defined as the summation of $n_k$'s weight and weights of all unscheduled communication edges which connect the $n_k$ with already scheduled tasks. This criterion simultaneously incorporates both the execution and communication aspects. It gives higher priority to the tasks which are heavy and/or intensively communicate with already scheduled tasks. By doing this, the clustering of tasks which extensively communicate can more easily be attained.

The scheduled workload is a dynamic task parameter and monotonically increases as scheduling process progresses. Initially, a task's scheduled workload is equal to a task's weight. During each scheduling step,
the scheduled workload of each unscheduled task is increased by the weight of the communication edge connecting that task with the task scheduled during this step. Hence, the complexity of recalculating the scheduled workloads for each of all unscheduled tasks is $O(v)$.

- **RTM**: for each unscheduled task calculate the relative mobility [defined by Equation (7)]. Choose the unscheduled task with minimal relative mobility. This criterion gives scheduling preference to the heavy unscheduled tasks which lie on critical or near-critical paths of the partially scheduled task graph. It assumes that these tasks have more effect on the overall execution time than tasks of larger relative mobility. It should be noted that the RTM, as the task selection criterion, was originally proposed in [12].

According to Equation (7) relative mobility calculation of all unscheduled tasks in the $STG_{p,k-1,\lambda}$ has the same complexity as the calculation of $I_p^{k-1}(n_\lambda)$ for all tasks in the $STG_{p,k-1,\lambda}$. The values $I_p^{k-1}(n_\lambda)$ are calculated according to Equations (3–5). Since application of Equations (3) and (4) leads to downward and upward topological traversing of the $STG_{p,k-1,\lambda}$, the complexity of the RTM is $O(v + e)$.

- **RND**: randomly choose an unscheduled task.

The task selection rule is specified as an ordered sequence of task selection criteria $Y_1 - Y_2 - ... - Y_r$, where $Y_i \in \{WT, SW, RTM, RND\}$. Criterion $Y_{i+1}$ is applied only to those unscheduled tasks that satisfy criterion $Y_i$. The following four task selection rules are suggested:

- $T_1 = WT\cdot RND$
- $T_2 = SW\cdot RND$
- $T_3 = RTM\cdot WT\cdot RND$
- $T_4 = RTM\cdot SW\cdot RND$

In order to be always sure that only one task will be selected, the RND as the last criterion in each of the suggested task selection rules is used. Note that all
unscheduled tasks lying on the critical path of the partially scheduled task graph have the same, zero relative mobility, and consequently all of them pass the criterion $RTM$. Thus, in rules $T_3$ and $T_4$, where $RTM$ is used as primary selection criterion, the additional selection is provided by $WT$ and $SW$, respectively.

4.2. $\rho$-edge selection rules

Let us consider $\rho$-edge selection at the $k$th step of the chaining procedure. Assume $n_k$ to be the task selected for scheduling during this step. The following criteria are considered for $\rho$-edge selection:

- $LP$: for each $\rho$-edge in $VR^{k-1}(n_x)$ calculate the $lp^k(n_x)$ assuming that $n_x$ is placed into that $\rho$-edge. The $\rho$-edge in $VR^{k-1}(n_x)$ which yields the minimal $lp^k(n_x)$ is chosen. This criterion exposes a global perspective on how each task placement affects the total structure of the partially scheduled task graph. It is also a greedy heuristic that builds a schedule in a fashion to maintain as small as possible the partial schedule length for as long as possible a period.

The $LP$ is implemented in the following manner. First, the values $l^{k-1}(n_x)$ and $e^{k-1}(n_x)$ are calculated for all tasks $n_x$ in $STG_{p,k-1}$. Then, the value $lp^k(n_x)$ for each $\rho$-edge $(n_x, n_y)$ in $VR^{k-1}(n_x)$ is determined as follows. For the $STG_{p,k}$ which will be created if $n_x$ is placed into the $\rho$-edge $(n_x, n_y)$, the following is valid:

$$ImPred^k(n_x) = ImPred^{k-1}(n_x) \cup \{n_x\}$$
$$ImSucc^k(n_x) = ImSucc^{k-1}(n_x) \cup \{n_x\}$$

if $n_y \in Pred^k(n_x)$ then $l^k(n_x) = l^{k-1}(n_x)$

if $n_x \in Succ^k(n_y)$ then $e^k(n_x) = e^{k-1}(n_x)$

Since the values $l^{k-1}(n_x)$ and $e^{k-1}(n_x)$ are known, values $l^k(n_x)$ and $e^k(n_x)$ can be directly obtained from Equations (3) and (4), without constructing and traversing the $STG_{p,k-1}$. Finally, $lp^k(n_x)$ is calculated by substituting $l^k(n_x)$ and $e^k(n_x)$ into Equation (5). Thus, the complexity of $LP$ is determined by the complexity of calculating the entry and exit distances for each task in the $STG_{p,k-1}$, i.e. $O(u + e)$.

- $COM$: determine the scheduled communication times for each of $\rho$-edges in $VR^{k-1}(n_x)$. Choose the $\rho$-edge in $VR^{k-1}(n_x)$ which yields the minimal scheduled communication time. The scheduled communication time of the $\rho$-edge $(n_x, n_y)$ is defined as the summation of real communication times [defined by Equation (1)] that appears on the communication edges that become scheduled by placing $n_x$ into $(n_x, n_y)$. This criterion penalizes the $\rho$-chains that incur a large amount of interprocessor communication.

For a given $\rho$-edge and a given $n_x$, calculation of scheduled communication time has the complexity of $O(|ImPred^{k-1}(n_x)| + |ImSucc^{k-1}(n_x)|)$ since all edges connected to $n_x$ have to be examined. Let us note that all $\rho$-edges in the same $\rho$-chain have the same scheduled communication time. Hence the complexity of $COM$ is $O(|ImPred(n_x)| + |ImSucc(n_x)|)$, and its upper bound is $O(pu)$.

- $MAX.W$: for each $\rho$-edge in $VR^{k-1}(n_x)$ calculate the width (defined by Equation (8)). The valid $\rho$-edge with maximal width is then chosen. The criterion according to which we place the task into the 'widest' $\rho$-edge uniformly distributes the load among the processors.

- $MIN.W$: for each $\rho$-edge in $VR^{k-1}(n_x)$ calculate the width. The valid $\rho$-edge with minimal width is then chosen. Contrary to the $MAX.W$, this criterion avoids using new processors. Clearly, this criterion could be effective only in a combination with some other criteria. According to Equation (8) the complexity of both $MAX.W$ and $MIN.W$ is $O(u + e)$.

Only the $\rho$-edge selection rules that use the $LP$ as the primary selection criterion are investigated in the sequel, since in preliminary simulations (not reported here) the performance of other criteria was consistently inferior. However, since there are usually several $\rho$-edges satisfying the $LP$ criterion, the final selection is provided by the other criteria. We expect that the additional selection will give an additional flexibility to the algorithm, needed to handle TGs efficiently with different inherent parallelism and communication demands. Four $\rho$-edge selection rules are suggested:

- $R_1 = LP-MAX.W-RND$
- $R_2 = LP-MIN.W-RND$
- $R_3 = LP-COM-MAX.W-RND$
- $R_4 = LP-COM-MIN.W-RND$

4.3. Class of chaining algorithms

By combining the proposed task and $\rho$-edge selection rules and varying the parameter $\lambda$, a number of chaining algorithms can be obtained. All these algorithms constitute a class of chaining algorithms, $C$. Each algorithm in $C$ can be represented by tuple $(TSR, RSR, \lambda)$, where $TSR \in \{T_i | i = 1, \ldots, 4\}$, $RSR \in \{R_i | i = 1, \ldots, 4\}$ and $\lambda \in \mathbb{R}^+$. The complexity of each algorithm in $C$ is $O(u(v + e))$.

5. SIMULATION STUDIES

5.1. Simulation model and method

To evaluate the performance of the proposed class of chaining algorithms we have developed a simulation model in Unix environment on Silicon Graphics Indigo workstation. The program is written in C++. Performance of different algorithms has been estimated with respect to a set of randomly generated TGs. We have used a TG generation procedure which is similar to the method for parameterized generation of random DAGs outlined in [10]. First, the number of layers in DAG is generated. Then, a random number of independent tasks is placed in each layer. Next, the edges between tasks at different layers are randomly linked. Finally, random values are assigned to the task and edge weights. We have generated the 360 random graphs by
using the following ranges for the above random parameters: (i) the number of layers is between 6 and 30, (ii) the number of tasks in each layer is between 2 and 10 and (iii) the task and edge weights are between 10 and 200. Also, about 60% of outgoing edges from one layer are incoming to the next layer, and the remaining 40% reach arbitrary succeeding layers. Graph parallelism of the TG $G$ is defined to be $g_{p}(G) = \frac{1}{cp(G)} \sum_{i=1}^{n} w_i$, where $cp(G)$ is the length of the graph $G$ when the edge weights are not included. For the generated set of random TGs the range of graph parallelism is between 2 and 20.

Generated random TGs have been scheduled onto a simulated eight-processor hypercube multiprocessor architecture, by using the scheduling algorithm and the amount of inter-task communication as the parameters. Each simulation is characterized by a 3-tuple $SIM = < G, A, \alpha >$, which means that the algorithm $\alpha$ is used to schedule the TG $G$ whose edge weights are multiplied by $\alpha \in R^+$. We have introduced the parameter $\alpha$ in order to analyse the algorithms' performances under different communication ratios. Our simulations have been conducted for tree $\alpha$ values: $\alpha = 0.5$ (typical for computation-intensive applications), $\alpha = 1$ (characteristic for balanced communication and computation applications) and $\alpha = 2$ (typical for communication-intensive applications).

In order to estimate the performance of some heuristic algorithm it is necessary to examine how close its solutions are to the optimal ones. Reaching an optimal schedule, even for small-sized TGs (e.g. with several tenths of nodes), requires a prohibitively large amount of computation time. Due to this fact in this study we have taken a different approach. First, we have defined a restricted class of chaining algorithms $F = \{TSR, RSR, \lambda\}$. TSR $\in \{T, i = 1, \ldots, 4\}$, RSR $\in \{R, i = 1, \ldots, 4\}$, $\lambda \in \{1, 1.2, 1.3, 1.4\}$. Then, we have scheduled each generated random TG by every of 64 algorithms in F. The shortest completion time that is achieved by some algorithm for a given TG is denoted by $(\alpha_{\text{best}})$ and is used as a reference of the optimal solution. We have qualified the performance of a particular algorithm using its average percent deviation from $\alpha_{\text{best}}$ (APD). Notice that a similar approach of estimating the performance of heuristic scheduling algorithms was used in [3].

5.2. Simulation results

To get a quality indication for the best solutions obtained by the algorithms in F, we compared them with the solutions generated by the ETF algorithm [4], and DLS algorithm [1]. Both ETF and DLS are $O(n^2 \cdot \rho)$ list scheduling algorithms. Figure 3a-c plot APD versus graph parallelism for ETF and DLS, and three $\alpha$ values. For both algorithms the APD increases with the increasing of the communication. For the studied levels of communication and graph parallelism, DLS deviates from 4% (for low communication and low graph parallelism) up to 15% (for high communication and high graph parallelism). On the other hand, deviation of ETF ranges from 40% (for high communication and low graph parallelism) down to 3% (for low communication and high graph parallelism).

The influence of the parameter $\lambda$ on performance of the chaining algorithms has been analysed in the following manner: We have considered the chaining algorithm that uses the task selection rule $T_1$ and $\rho$-edge selection rule $R_2$, with $\lambda$ varying from 0 to 2.5 in steps of 0.1. We have applied the algorithm for all these $\lambda$ values to schedule the subset of generated random TGs whose graph parallelism is within the range from 7.5 to 8.5 (i.e. approximately the same as the number of processors). Figure 4 shows the APD as a function of $\lambda$. From this figure we see that when the communication is medium ($\alpha = 1$) and high ($\alpha = 2$), the algorithm's performance is highly sensitive to parameter $\lambda$. For each of analysed $\alpha$ values, the best results are obtained when the $\lambda$ is within the range from 1.2 to 1.5. Similar observations regarding the algorithm's sensitivity to $\lambda$ have been made studying other groups of random TGs.

The simulation analysis concerning the performance of the suggested task selection rules have been carried out with respect to the following set of chaining algorithms: $T = \{(T_i, R_3, 1.2) | i = 1, \ldots, 4\}$. Note that the four algorithms in T differ in the adopted task selection rule. The simulation results are presented in Figure 5a-c. Each figure is sketched for different $\alpha$ values and plots the APD versus graph parallelism for each of four algorithms in T. From these figures we see that the algorithms which employ $RTM$ as the primary task selection criterion [i.e. $(T_i, R_3, 1.2)$ and $(T_4, R_3, 1.2)$] demonstrate better average scheduling performance over the two other algorithms which use $WT$ and $SW$. Inefficiency of criteria $WT$ and $SW$ with respect to the $RTM$ can be attributed to the fact that they consider each task individually. On the other hand, $RTM$ gains improvement by exposing a global perspective on task importance. Comparing algorithms $(T_1, R_3, 1.2)$ and $(T_3, R_3, 1.2)$ we conclude that $SW$ used in $(T_2, R_3, 1.2)$ is more effective than $WT$ used in $(T_1, R_3, 1.2)$, when communication is high and graph parallelism is low. In the cases when the communication is low and/or graph parallelism is high, the importance of the communication is reduced and consequently the $WT$ becomes better choice.

To analyse the effectiveness of suggested $\rho$-edge selection rules we have considered the following set of chaining algorithms $R = \{(T_i, R_1, 1.2) | i = 1, \ldots, 4\}$. The simulation results are presented in Figure 6a-c. Let us recall that all suggested $\rho$-edge selection rules use the $LP$ as the primary selection criterion. Thus, performance differences among the algorithms in R are a consequence of different additional selection criteria employed in $\rho$-edge selection rules. Effects of different additional selections are most visible in boundary regions of the communication and graph parallelism. The $MAX.W.$, used as the secondary $\rho$-edge selection criterion in $(T_i, R_1, 1.2)$, forces load balancing, i.e. tends to distribute uniformly the computation among the processors. Drop in performance of $(T_4, R_1, 1.2)$ with the increasing of $\alpha$, and the improvement in performance with the increasing of the graph parallelism can be observed from Figure 6. Obviously, $MAX.W.$ is effective when the
FIGURE 6. Average percent deviation from $\omega_{best}$ as achieved by algorithms in $R = \{(T_i, R_i, 1.2) | i = 1, \ldots, 4\}$ versus graph parallelism for: (a) $\alpha = 0.5$; (b) $\alpha = 1.0$; (c) $\alpha = 2.0$.

communication is low or the graph parallelism is sufficiently high to cover the communication only. When the additional selection is provided by criteria $COM$ or $MIN.W$, the algorithm tends to cluster communicating tasks onto the same processor. These heuristics become effective when the communication is high and the graph parallelism is low.

The simulation results, presented so far, indicate the following:

(i) Although the chaining algorithms that are based on the $RTM$, as the primary task selection criterion, work more efficiently than others, there is no single chaining algorithm that is superior to all others in all circumstances.

(ii) The accurate prediction of communication times, through proper setting of parameter $\lambda$, is important for the success of the algorithm, especially when the communication is medium or high. Fortunately, there exists a relatively wide range of $\lambda$ values (from 1.2 to 1.5) in which the algorithm performs close to its maximum. We believe that this range is a topology dependent parameter which should be determined empirically for each concrete interconnection topology.

(iii) The majority of the analysed chaining algorithms show better performance with respect to two competitive list scheduling algorithms, ETF and DLS. The performance difference becomes significant when the communication and/or graph parallelism are high. However, this performance advantage is partially attained at the price of added computational complexity.

The observation (i) indicates that algorithm redundancy could be important in chaining. To investigate a needed amount of redundancy, we have considered the scheduler which first performs scheduling for each algorithm in a particular set of chaining algorithms, and secondly takes the best result. To limit the size of investigation we have only examined algorithms in set $R$, which the previous simulation study shows have better performance with respect to other analysed sets. We start with the scheduler that uses only the algorithm $S_1 = (T_4, R_3, 1.2)$, and then incrementally extend the number of used algorithms until all algorithms in $R$ are included. A summary of the simulations for the 360 graphs for all three $\alpha$ values is given in Table 1. We list the percentage of cases in which the results obtained by the scheduler are equal to the best known, and deviate for less than 2%, 5% and 10% from $\omega_{best}$. In the last column of the Table 1, the overall APD is given. When only one algorithm is used (see row $S_1$ in Table 1), the APD is 3.8%, the percentage of cases in which the best results are attained is 8.3%, while the deviation < 5% is observed in 71.1%
for all cases. As the number of used algorithms increases, the performance of the scheduler increases, too. When all algorithms in \( R \) are used (see row \( S_4 \) in Table 1), the APD is reduced to 1.8%, the best results are obtained in 28% and the deviation < 5% is attained in 89.9% for all cases. This investigation also shows that the average performance of chaining-based schedulers is better than that of DLS and ETF.

6. CONCLUSION

We have presented a new compile-time scheduling technique called chaining which takes into account the interprocessor communication overheads when mapping task graphs onto multiprocessor architectures. Chaining is a graph-transformation technique that takes \( v \) steps to transform a task graph into the scheduled task graph, where \( v \) is the number of tasks. A fundamental property of this technique is that it does not impose any constraints about the ordering according to which tasks are selected for scheduling. This scheduling freedom has allowed us to obtain a class of new scheduling algorithms by specifying different rules by which the task and insertion location within the current partial schedule are selected. Simulation study has shown that the majority of algorithms in this class have better performance with respect to two list scheduling algorithms, ETF and DLS. Effectiveness of the scheduler that uses several chaining algorithms to determine alternative schedules, and then selects the best one, was investigated. Using a carefully chosen small subset of only four chaining algorithms, a further performance improvement was obtained.

Chaining is a flexible technique that can be adapted for applications which characterize complicated scheduling models, e.g. heterogeneous processors, and models that account for contentions in communication channels. We are currently focusing on the generalization of the chaining technique, which will perform scheduling and routing simultaneously to enable the scheduling of all communications as well as all computations.

**REFERENCES**