A heuristic for scheduling task graphs with communication delays onto multiprocessors

Goran Lj. Djordjević *, Milorad B. Tošić

Faculty of Electronic Engineering, University of Niš, Beogradska 14, P.O. Box 73, 18000 Niš, Serbia, Yugoslavia

Received 15 April 1995; revised 25 June 1996

Abstract

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 takes into account the communication overhead and can be applied to scheduling task graphs onto fully-connected multiprocessor architectures containing an arbitrary (bounded as well as an unbounded) number of processors. This technique can be viewed as a generalized list scheduling concept, that does not impose any preconditions about the ordering in which tasks are selected for scheduling as well as about the position within the current partial schedule where selected task can be placed. Varying the selection policy, implemented in this technique, we are able to generate a class of scheduling algorithms. As a representative example of this class we present Task Selection First (TSF) scheduler. We compare performances of the TSF scheduler with the dynamic level scheduler proposed by Sih and Lee, the dominant sequence clustering algorithm proposed by Yang and Gerasoulis, and the DSC/MLS algorithm, a modified version of Sarkar’s two-step scheduling technique.

Keywords: Multiprocessor scheduling; Task graph; Interprocessor communication; Chaining; List scheduling; Parallel processing

* Corresponding author. Email: gtjordjevic@europa.elfak.ni.ac.yu.
1. Introduction

We study the problem of scheduling the tasks of precedence constrained task graphs onto the processors of a multiprocessor in a way that minimizes the execution time. Task graph is a weighted Directed Acyclic Graph (DAG) which is used to represent the computation performed in a parallel program. In a task graph every node represents a parallel program task, while an edge in the task graph signifies that data produced by one task is used by another task. A node (task) weight represents the execution time of the task, while a weight associated with each edge denotes the amount of communication between tasks. We adopt a task execution model where each task is executed nonpreemptively (i.e., during one continuous time interval) on a single processor. A valid schedule of a given task graph is any scenario of tasks execution that satisfies precedence constrains. The scheduling objective is to find a valid schedule that minimizes the overall execution time of the task graph. This scheduling problem is usually referred to as the minimum execution time multiprocessor scheduling problem.

The multiprocessor scheduling problem has received considerable attention in recent years. This problem is known to be NP-hard [8]. As a consequence, heuristic algorithms were developed for obtaining satisfactory suboptimal solutions. List Scheduling (LS) is a scheduling principle that encompasses a class of multiprocessor scheduling algorithms. The LS is the single-pass technique that schedules tasks from the top to the bottom of the task graph. In its classic implementation, LS schedules the tasks by building task execution sequences, through adding each new task to the top of one of the previously scheduled sequence. A task is not ready for scheduling until all its predecessors are scheduled. If more than one task is ready, then the one with the highest priority is chosen to be scheduled next. The key idea behind the LS is that it maximizes processors' utilization, forces load balancing, and systematically attacks the critical path of the task graph (through the proper priority assignment). This strategy demonstrates near-optimality when inter-task communication is negligible [2,5,10]. However, in models that account for communication delays its application cannot be fully justified. An attempt to exploit as much parallelism as possible, without regarding the corresponding communication costs, may tend to increase the execution time. Thus, to efficiently schedule task graphs with communication delays, a trade-off between maximizing parallelism and minimizing communication overheads must be concerned. With this aim, various extensions of the classic LS principle were proposed. The Earliest Task First (ETF) strategy, presented in [4], 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. Insertion Scheduling Heuristic (ISH) [6] and the algorithm described in [9] allow the ready task to be scheduled into "scheduling holes", where a "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. The Dynamic Level Scheduling algorithm (DLS) [11] uses dynamically changing priorities to match tasks and processors for scheduling at each scheduling step. A class of scheduling heuristics, called Generalized List Scheduling...
(GLS), introduced in [1], evaluates the task priority as the completion time of the task that results from backward scheduling of the task graph.

Clustering techniques are widely used for scheduling task graphs on an unbounded number of processors in presence of inter-task communication [3,8,13]. These schemes try to avoid excessive interprocessor communication cost by grouping heavily communicating tasks into the same clusters, that are then mapped onto separate processors. Most of the existing clustering algorithms can be characterized by using a framework described in [3]. The initial step assumes that each task is mapped in an unit cluster. At each step, the algorithm tries to improve the previous clustering by merging the appropriate clusters, i.e., by zeroing an edge cost connecting two clusters. Yang and Gerasoulis [13] presented a low-complexity heuristic, the Dominant Sequence Clustering (DSC) algorithm, with performances comparable to, or even better, on average, than much higher-complexity clustering heuristics. Sarkar suggested a two-step scheduling method that extends clustering technique in a way that allows scheduling on a bounded number of processors [8]. In the first step, called internalization pre-pass, the nodes are clustered together in an attempt to minimize the execution time of the task graph 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. For an illustrative example see [12].

In this paper we present a novel compile-time single-pass scheduling scheme, called chaining. Chaining is a general scheduling technique that accounts for communication overheads and can be applied to scheduling task graphs onto fully connected multiprocessor architectures containing an arbitrary (bounded or unbounded) number of processors. Chaining can be viewed as a generalized concept of the LS, that does not impose any prerequisites about the ordering in which tasks are selected for scheduling as well as about the position within the current partial schedule where selected tasks can be placed. For the chaining, any valid schedule of the given task graph is represented by the scheduled task graph. This graph is a Directed Acyclic Multigraph (DAM) obtained by extending the task graph with new precedence edges so that all tasks assigned to the same processor are arranged into a linear cluster, or chain. A partially scheduled task graph is introduced to represent the state of the scheduling process at each scheduling step. Starting from a task graph, the algorithm, at each step, selects one unscheduled task and combines it with previously scheduled tasks into a new partially scheduled task graph. After \( v \) scheduling steps, where \( v \) is the number of tasks, the task graph is transformed into the scheduled task graph. Changing some selection rules in this construction procedure a class of scheduling algorithms can be obtained. Such a scheduling technique has not been proposed in the literature, and is important because it points to some solutions that have not been evaluated, and that may have potential for providing high performances.

The paper is organized as follows: Section 2 gives some basic terminology about DAGs, precisely describes the adopted task execution model and the target multiproces-
2. Multiprocessor scheduling problem

A task graph is a weighted DAG defined by a tuple $TG = (V, E, W, C)$, where $V = \{s = n_0, n_1, \ldots, n_v, q = n_{v+1}\}$ 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 communication cost incurred along the edge $e_{i,j}$, which is zero if both tasks are assigned to the same processor. Nodes $s$ and $q$ are "dummy" tasks, with zero execution times ($w_0 = w_{v+1} = 0$), 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 ($c_{0,i} = c_{i,v+1} = 0$). Tasks $n_1, \ldots, n_v$ are computation tasks. An example of the task graph, with seven computation tasks, is shown in Fig. 1(a). The numbers inside the nodes are node weights, while the numbers on the edges are edge weights; if no number is written the node/edge weight is zero.

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

![Fig. 1](image-url)  
(a) An example of task graph $TG$; (b) A schedule for two processors displayed as Gantt chart; (c) A scheduled task graph $STG$. 
task graph (i.e., the schedule length) is minimized. The execution time of the task graph is defined as
\[
\max_{i=1,\ldots,\nu}\{\text{CT}(n_i)\},
\]
where the CT\(n_i\) is the completion time of task \(n_i\).

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 “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., nonpreemptively). 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 have arrived in the local message buffer (earliest task starting time assumption). 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 homogenous processors, \(P_k\), \(k = 1, \ldots, p\). The time required to send a data unit from \(P_i\) to \(P_j\) is constant, independent of the processors (identical communication paths assumption), and it is zero if \(i = j\). All processors can communicate directly with all others without contention (fully-connected processors). Each processor contains specialized hardware which enables it to perform both computation and communication in parallel, and to handle all its communication channels simultaneously.

The adopted task execution model is the same as the compile time macro-dataflow model presented in [8]. Assuming identical communication paths between identical processors notably simplifies scheduling, because the difficulty of determining “favored combinations” of processors is removed. Multiprocessors that satisfy the identical communication paths assumption are those in which the processors are equidistant for the purpose of inter-processor communication. Examples of such systems are: (a) shared memory multiprocessor in which a set of homogeneous processors with local memory, communicate through global shared memory modules via some type of interconnection network, such as shared-bus, or crossbar; and (b) message passing multiprocessors which are completely connected. 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

Fig. 1(b) shows a Gantt chart of a schedule for the task graph in Fig. 1(a) on \(p = 2\) processors. The Gantt chart completely describes the schedule, since it defines both the task execution sequences and the task starting times. Alternatively, modeling of schedule
can be done by means of a Scheduled Task Graph (STG). The STG is a weighted DAM derived from the task graph using the execution ordering of the tasks imposed by the schedule within each processor. In the STG, tasks assigned to the same processor are linked with zero-weighted precedence edges, so that task ordering for each processor is uniquely determined by the single path of precedence edges. In the following, precedence edges will be called \( \rho \)-edges, and a path consisting only of \( \rho \)-edges will be called \( \rho \)-path. The \( \rho \)-chain is a subgraph of the STG induced by a \( \rho \)-path which starts in node \( s \) and ends in node \( q \). Let \( \{ \rho_1, \rho_2, \ldots, \rho_p \} \) be the set of all \( \rho \)-chains in the STG. We assume that \( P_i \) executes \( \rho_i \). A communication edge weight in the STG becomes zero if the start and end node of this edge are in the same \( \rho \)-chain. The edge weights between \( \rho \)-chains remain unchanged. For example, a schedule for the task graph in Fig. 1(a), which is the same as the one in Fig. 1(b), but represented in a form of the STG is given in Fig. 1(c). Dashed lines indicate \( \rho \)-edges.

2.3. Partially scheduled task graph

A partially scheduled task graph \( \text{STG}(p,k) \) is introduced to represent the structure of a partial schedule, i.e., a partial assignment of the task set to \( p \) processors where \( k \) computation tasks are scheduled. This graph can be viewed as an intermediate form between the task graph and the scheduled task graph. An example of \( \text{STG}(2,4) \) is shown in Fig. 2(a). Given \( TG = (V,E,W,C) \), a number of processors \( p \), and an integer \( 0 \leq k \leq u = |V| - 2 \), partially scheduled task graph \( \text{STG}(p,k) = (V,E + R^k,W,C^k) \), is a weighted DAM obtained by adding \( p + k \) \( \rho \)-edges to the TG, so that they form \( p \) computation nodes − disjoint \( \rho \)-chains. \( R^k \) is the set of \( \rho \)-edges, and \( C^k \) is the set of communication edge weights in the \( \text{STG}(p,k) \). Let \( \{ \rho_{i1}^k, \rho_{i2}^k, \ldots, \rho_{ip}^k \} \) be the set of all \( \rho \)-chains in the \( \text{STG}(p,k) \), and \( V_i = (R_i^k, V_i) \), where \( R_i^k \subset R^k \) is the set \( \rho \)-edges, and \( V_i \subset V \) is the set of nodes in the \( \rho_{ij}^k \). Then, \( R^k = \bigcup_{i=1}^p R_i^k \), and \( V_i \cap V_j = \{ s, q \} \), \( i, j = 1, \ldots, p \), \( i \neq j \). All tasks in the \( \text{STG}(p,k) \) can be partitioned into two disjoint sets: the set of scheduled tasks, \( ST = \bigcup_{i=1}^p V_i \), and the set of unscheduled tasks, \( UT = V \setminus ST \).
The total number of scheduled computation tasks in the STG\((p,k)\) is \(k\). The weight of a communication edge, \(c_{i,j}^k\), in the STG\((p,k)\) is zeroed if both nodes joined by that edge are scheduled in the same \(p\)-chain. Otherwise, \(c_{i,j}^k = c_{i,j}\).

Note that STG\((0,0) = TG\), and STG\((p,0) = STG\). The STG\((p,0)\) is called initial STG. This graph is obtained by extending the task graph with \(p\) empty \(p\)-chains, i.e., by adding \(p\) \(p\)-edges that connect node \(s\) to node \(q\). The STG\((2,4)\) shown in Fig. 2(a) is derived from the TG in Fig. 1(a). Tasks \(\{s,n_1,n_5,n_6,n_7,q\}\) are scheduled, while the tasks \(\{n_2,n_3,n_4\}\) are unscheduled. Fig. 2(b) shows the initial STG, which corresponds to TG in Fig. 1(a) for \(p = 2\).

2.4. Definitions and notation

We use \(\text{Succ}^k(n_x)\) to denote the set of successors of \(n_x\), and \(\text{Pred}^k(n_x)\) the set of predecessors of \(n_x\) in the STG\((p,k)\) (not necessarily immediate in both sets). With respect to a fixed node \(n_x \in V\), we will say that node \(n_y \in V\) \(y \neq x\) is of the type \(P, S\), or \(I\) if \(n_y \in \text{Pred}^k(n_x)\), \(n_y \in \text{Succ}^k(n_x)\), or \(n_y \notin \text{Pred}^k(n_x) \land n_y \notin \text{Succ}^k(n_x)\), respectively. Similarly, we will say that an edge \((n_x,n_y) \in E + R^k\) is of type \(AB\) with respect to node \(n_x\) if \(n_y\) is of type \(A\), and \(n_y\) is of type \(B\), with respect to \(n_x\), where \(A, B \in \{P,S,I\}\).

The length of a path in the STG\((p,k)\) is the summation of all node and edge weights in that path. The critical path is a path of the longest length in the graph. The length of the STG\((p,k)\) (or partial schedule length), \(\text{cp}(\text{STG}(p,k))\), is the length of the STG\((p,k)\)'s critical path. The entry distance of node \(n_x\) in the STG\((p,k)\), \(l^k(n_x)\), 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 \(n_x\), \(e^k(n_x)\), is the length of the longest path connecting node \(n_x\) to node \(q\), excluding the weight of the node \(n_x\). Entry distance of \(s\), and exit distance of \(q\) are assumed to be zero. The following recursive formulae can be used to determine the entry and exit distances of the \(n_x\) in the STG\((p,k)\):

\[
l^k(n_x) = \begin{cases} 
0 & \text{if } \text{ImPred}^k(n_x) = \emptyset, \\
\max_{n_y \in \text{ImPred}^k(n_x)} \{l^k(n_y) + w_y + c^k_{y,x}\} & \text{otherwise},
\end{cases}
\]

\[
e^k(n_x) = \begin{cases} 
0 & \text{if } \text{ImSucc}^k(n_x) = \emptyset, \\
\max_{n_y \in \text{ImSucc}^k(n_x)} \{e^k(n_y) + w_y + c^k_{x,y}\} & \text{otherwise},
\end{cases}
\]

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

If the values \(l^k(n_x)\) and \(e^k(n_x)\) are known, then we can calculate the length of the longest path passing through the \(n_x\) in the STG\((p,k)\), \(l_p^k(n_x)\), as

\[
l_p^k(n_x) = l^k(n_x) + w_x + e^k(n_x).
\]

To calculate the length of the STG\((p,k)\) the following formula can be used

\[
\text{cp}(\text{STG}(p,k)) = \max_{i=1,p} \{l_p^k(n_i)\}.
\]
We define the relative mobility of task $n_x$ in the STG($p,k$) to be the difference between the length of the STG($p,k$) and the length of the longest path passing through the $n_x$ in the STG($p,k$), divided by the weight of the $n_x$:

$$\text{rtm}^k(n_x) = \frac{\text{cp}(\text{STG}(p,k)) - \text{lp}^k(n_x)}{w_x}. \quad (5)$$

The width of the $p$-edge $(n_i,n_j)$ in the STG($p,k$) is defined to be

$$\text{wd}^k((n_i,n_j)) = \text{cp}(\text{STG}(p,k)) - (l^k(n_i) + w_i + w_j + e^k(n_j)). \quad (6)$$

The values $l^k(n_x)$, $e^k(n_x)$, $\text{lp}^k(n_x)$, and $\text{rtm}^k(n_x)$ of the tasks of the partially scheduled task graph STG(2,4) shown in Fig. 2(a) are given in Table 1. The length of this partial schedule is nine.

Note that if a schedule is represented by the STG, then the task starting times are not explicitly given, but can be computed by topological traversing the STG. 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, i.e., $l^k(n_x)$. The execution time of the task graph is determined by the critical path of the STG. It is equal to the length of the STG, i.e., cp(STG). Note that, in this context, the multiprocessor scheduling problem can be stated as a graph-theoretical problem of derivation the minimal length STG (i.e., STG($p,v$)) from a given task graph.

### 3. Chaining

Our approach to the multiprocessor scheduling problem is to consider a scheduling algorithm that constructs STG, directly. Having this in mind, we propose a single-pass scheduling technique, called chaining, that constructs the STG incrementally, in a step-by-step fashion, by scheduling one task at each step, and uses the STG($p,k$) to represent the state of the scheduling process during the $k$-th scheduling step.

Let us consider the scheduling of the task graph $TG = (V,E,W,C)$, with $v = |V| - 2$ computation tasks, onto $p$ processors. At the initial scheduling step STG($p,0$) is
formed. As a consequence, tasks $s$ and $q$ are identified as scheduled tasks, $ST = \{s, q\}$, and all computation tasks are identified as unscheduled tasks, $UT = \{n_1, n_2, \ldots, n_v\}$. At the $k$-th scheduling step, $k = 1, \ldots, v$, one task from UT is scheduled, transforming STG($p, k - 1$) into STG($p, k$). The algorithm schedules a task by inserting it into one of the $p$ $\rho$-chains which already exist in STG($p, k - 1$). Task insertion is performed in the following way: By analyzing the STG($p, k - 1$), the algorithm selects both, an unscheduled task $n_x \in UT$, and a $\rho$-edge $(n_a, n_b) \in R^{k-1}$. Then, the selected $\rho$-edge is destroyed, and two new $\rho$-edges, $(n_a, n_x)$ and $(n_x, n_b)$, are created. Now, the weights of the communication edges between the $n_x$ and tasks previously scheduled into the same $\rho$-chain as $n_x$ are zeroed. After the task insertion is performed, we say that task $n_x$ is placed into $\rho$-edge $(n_a, n_b)$. In this way $n_x$ is transferred form UT to ST, i.e., $UT = UT \setminus \{n_x\}$, and $ST = ST \cup \{n_x\}$. This process is irreversible, i.e., once a task is put into ST it cannot be returned to UT. The STG($p, k$), created during the $k$-th scheduling step, represents the starting point for the next step. This process is repeated until UT becomes empty.

Precondition that STG($p, k$) must be DAM restricts the set of $\rho$-edges of the STG($p, k - 1$) into which the selected unscheduled task can be placed. We say that $\rho$-edge $(n_a, n_b)$ is valid with respect to $n_x \in UT$ if placing $n_x$ into $(n_a, n_b)$ does not create cycles. It is easy to see that a $\rho$-edge $(n_a, n_b) \in R^{k-1}$ is valid with respect to $n_x \in UT$ iff the following condition is satisfied: $n_a \in Succ^{k-1}(n_x) \land n_b \in Pred^{k-1}(n_x)$. According to the edge type definition (see Section 2.4) a $\rho$-edge is valid with respect to task $n_x$ if that edge is of the type PS, IS, PI, or II with respect to $n_x$.

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

**Proof.** Let us consider STG($p, k - 1$), and let $n_x \in UT$. As $s$ is the predecessor, and $q$ is the successor of every computation task, the sequence of $\rho$-edge types with respect to $n_x$, along any $\rho$-chain in the STG($p, k - 1$) is of the following form: $PX_1, X_1, X_2, \ldots, X_S$, where $X_i \in \{P\}$. Clearly, for any permutation of $(X_1, X_2, \ldots, X_S)$, there exists at least one $\rho$-edge of a valid type (i.e., PS, IS, PI or II). $\square$

Lemma 1 proves the correctness of the chaining technique in the sense that it ensures that the scheduling process cannot be prematurely stopped, because a $\rho$-edge valid with respect to an unscheduled task does not exist. Thus, the chaining does not impose any constraints about the ordering according to which tasks are selected for scheduling.

**Example.** Let us consider again the STG(2,4) in Fig. 2(a). Assume that this graph is created in the 4-th step of the chaining algorithm that schedules TG in Fig. 1(a) on $p = 2$ processors. In the 5-th algorithm's step, STG(2,4) is transformed into STG(2,5). Depending on the fact which unscheduled task is selected, and which $\rho$-edge is chosen for task placement, several different STG(2,5) can be derived from the STG(2,4). All valid unscheduled task-$\rho$-edge combinations are represented by the bipartite graph given in Fig. 3(a). Each edge in this graph indicates one valid scheduling option. Since the algorithm is single-pass, it selects only one option. The selected option is determined by
the implemented selection policy. Assume that the selected pair is \((n_s, (n_5, q))\). The STG(2,5) which comes out as a result of this decision is given in Fig. 3(b).

Note that chaining can be used for scheduling task graphs on an unbounded number of processors, also. Although, in this case, the number of available processors is not limited, the number of used processors cannot be greater than the number of computation tasks in the task graph. So, in order to schedule the task graph on an unbounded number of processors, we only have to set \(p = v\).

From the previous discussion follows that the chaining, as a general single-pass scheduling technique, can become the source of a number of new scheduling algorithms. Varying the selection policy, i.e., the rules according to which, at each scheduling step, a pair of the unscheduled task-\(\rho\)-edges is selected, different scheduling algorithms can be obtained. For example, it is obvious that the majority of LS variants can be explained as special cases of the chaining technique, where, at each scheduling step, an unscheduled task with all predecessors scheduled is selected and placed in a terminal \(\rho\)-edge (i.e., \(\rho\)-edge that ends in node \(q\)). However, chaining allows us to schedule the tasks in any order (regardless of the current state of task’s predecessors and successors), and to place the task into any valid \(\rho\)-edge. An important question is: Can this freedom of defining the selection policy give to us a scheduling algorithm with comparable performances, or even with better ones, in respect of the existing scheduling algorithms? In order to find such an algorithm we carried out a number of simulations, by varying the selection policy. It is intuitively clear that the most desirable property of a selection policy is its ability to effectively find a trade-off between maximizing the total workload (computation + communication) that is scheduled at each scheduling step, and minimizing the extension of the partial schedule length. Having this in mind, we focused our attention on searching for parameters of the current partial schedule that will clearly reflect this trade-off and be easy to compute. The outcome of this investigation is the Task Selection First (TSF) scheduler. Its detailed description and simulation results indicating its performances are given in the following two sections.
4. Task selection first scheduler

The Task Selection First (TSF) scheduler is a chaining algorithm that uses the following general selection policy: At each scheduling step the algorithm initially selects the task to be scheduled, and then chooses the $p$-edge where the selected task will be placed. Thus, to implement TSF scheduler, the task selection rule, and the $p$-edge selection rule need to be specified. Since the algorithm is a single-pass, then it is reasonable to apply a greedy strategy, i.e., one that always selects the most “critical” task, and places it into the “best” $p$-edge.

4.1. Task selection rule

After investigating several heuristics for initial task selection, we conclude that the most important parameters, which make the task $n_x \in UT$ to be “critical” in the $k$-th scheduling step, are: (a) the length of the longest path passing through the task in $STG(p,k-1)$, $lp^{k-1}(n_x)$; (b) the task weight, $w_x$; (c) the maximum communication time incurred along any of task’s ingoing edges, $\max_i \{C_{i,x}\}$; and (d) the maximum communication time incurred along any of task’s outgoing edges, $\max_i \{C_{x,i}\}$. These parameters are combined into the task selection rule as follows. Primary criterion for task selection is based on the relative task mobility (defined by Eq. (5)). The task $n_x \in UT$, selected for scheduling in the $k$-th scheduling step, satisfies $rmt^{k-1}(n_x) \leq rtm^{k-1}(n_i)$ for any $n_i \in UT$. In this way, the scheduling preference is given to the heavy unscheduled tasks which lay on the critical or near-critical paths of the $STG(p,k-1)$, assuming that these tasks have more effect on the overall execution time than the tasks with larger relative mobility. In the case of tie, the secondary criterion is applied. It chooses the task for which the value $cw(n_x) = \max_i \{C_{i,x}\} + w_x + \max_i \{C_{x,i}\}$ is maximal among all tasks satisfying the primary criterion. This criterion simultaneously incorporates both the execution and communication aspects, giving the highest priority to the tasks which are heavy and/or intensively communicate with other tasks. If still a tie exists, it is broken randomly.

The relative task mobility is a dynamic (i.e., changing) parameter, and it has to be recalculated at each scheduling step for all unscheduled tasks. To do this, values $cp(STG(p,k-1))$ and $lp^{k-1}(n_x)$, have to be known. These values can be calculated according to Eqs. (1)–(4). Application of Eq. (1) and Eq. (2) leads to downward and upward topological traversing of the $STG(p,k-1)$, starting from node $s$ and node $q$, respectively.

4.2. $p$-edge selection rule

To perform $p$-edge selection, at the $k$-th scheduling step, first it is necessary to identify all $p$-edges in the $STG(p,k-1)$ that are valid with respect to the selected unscheduled task $n_x$. In other words, all nodes in the $STG(p,k-1)$ need to be marked with an appropriate type mark ($P$, $S$, or $I$) with respect to $n_x$. This can be done by marking initially all nodes as $I$, and then to perform upward and downward topological traversing of the $STG(p,k-1)$ starting from the $n_x$. All tasks visited during upward
traversing are marked as \( P \), and all tasks visited during downward traversing are marked as \( S \).

The primarily criterion for \( \rho \)-edge selection considers the length of the longest path which will pass through the \( n_x \) when \( n_x \) is placed into the \( \rho \)-edge, i.e., \( \text{lp}^k(n_x) \). The selected \( \rho \)-edge is the one for which the \( \text{lp}^k(n_x) \) is minimal. With this \( \rho \)-edge selection criterion we try to build a schedule in a way to maintain the partial schedule length as small as possible for as long as possible. The \( \text{lp}^k(n_x) \), which comes as a result of placing \( n_x \) into the \( \rho \)-edge \((n_i,n_j)\), can be determined without constructing and traversing the STG\((p,k)\), but using only the local information about the \( n_x \) and \((n_i,n_j)\) in the STG\((p,k-1)\). After \( n_x \) is placed into \((n_i,n_j)\), the following expressions will become valid:

\[
\text{Impred}^k(n_x) = \text{Impred}^{k-1}(n_x) \cup \{n_j\},
\]

\[
\text{ImSucc}^k(n_x) = \text{ImSucc}^{k-1}(n_x) \cup \{n_i\}.
\]

Placing \( n_x \) into \((n_i,n_j)\) does not affect the exit distances of the \( n_x \)'s successors, and the entry distances of the \( n_x \)'s predecessors. So, to calculate \( l^k(n_x) \) and \( e^k(n_x) \) we can use Eq. (1) and Eq. (2) in which the terms \( l^k(n_x) \) and \( e^k(n_x) \) are substituted with \( l^{k-1}(n_x) \) and \( e^{k-1}(n_x) \), respectively. By substituting \( l^k(n_x) \) and \( e^k(n_x) \) into Eq. (3), \( \text{lp}^k(n_x) \) is finally obtained.

The secondary criterion for \( \rho \)-edge selection examines the width of the \( \rho \)-edge (defined by Eq. (6)). Among the \( \rho \)-edges satisfying the primary criterion, the secondary criterion chooses the one with maximal width. By placing the task into the "widest" \( \rho \)-edge, this criterion tries to uniformly distribute the load within the processors, and also among the processors. If still a tie exists, it is broken randomly.

**Example.** All the graphs STG\((2,4)\), STG\((2,5)\), and STG, shown in Fig. 2(a), Fig. 3(b) and Fig. 1(c), respectively, are derived from the TG in Fig. 1(a) by using the TSF's selection policy. Let us consider more closely how the STG\((2,4)\) is transformed into the STG\((2,5)\). Among the unscheduled tasks in the STG\((2,4)\) tasks \( n_2 \) and \( n_4 \) have the same minimal relative mobility (see Table 1), and consequently both of them pass the primarily task selection criterion. The final selection is performed by the secondary task selection criterion which chooses the \( n_4 \), since \( \text{cw}(n_2) = 3 \) and \( \text{cw}(n_4) = 4 \). The \( \rho \)-edges in the STG\((2,4)\) which are valid with respect to the \( n_4 \) are \(((n_1,n_3),(n_5,q),(s,n_6),(n_6,n_7))\) (see Fig. 3(a)). For all these edges the algorithm calculates the width and the length of the longest path that will pass through the \( n_4 \) if \( n_4 \) is placed into that edge. The

<table>
<thead>
<tr>
<th>((n_i,n_j))</th>
<th>( \text{wd}^k((n_i,n_j)) )</th>
<th>( \text{lp}^k(n_4) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((n_3,n_2))</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>((n_3,n_4))</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>((s,n_4))</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>((n_6,n_4))</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>
calculated values are given in Table 2. The \((n_x,q)\) is selected because it yields the minimal \(I_p^k(n_x)\).

**The Complexity.** The time complexity of TSF is determined by the time complexity of the topological traversing of the partially scheduled task graph, which have to be performed at each step in order to calculate parameters required for task and \(p\)-edge selection. As the complexity of topological traversing the DAG is \(O(v + e)\), where \(v\) is the number of nodes, and \(e\) is the number of edges, and the number of steps is \(v\), TSF’s complexity is \(O(v(v + e))\).

4.3. **TSF for scheduling on an unbounded number of processors**

Although the TSF is conceived to be an algorithm for scheduling on a bounded number of processors, it can be used as a clustering algorithm for scheduling on an unbounded number of processors, also. In this case, the number of \(p\)-chains is initially set to be equal to the number of computation tasks. Thus, the algorithm, at each step, can make a choice either to cluster the selected unscheduled task \(n_x\) with the tasks already scheduled in a \(p\)-chain, or to insert \(n_x\) into an empty \(p\)-chain. If \(n_x\) is placed into an empty \(p\)-chain, the length of the longest path passing through \(n_x\) will not be changed, i.e., \(I_p^k(n_x) = I_p^{k-1}(n_x)\). On the other hand, if \(n_x\) is placed into a non-empty \(p\)-chain, \(I_p^k(n_x)\) could be reduced (since \(n_x\) and tasks in the selected \(p\)-chain are grouped into the same cluster, some communication edges are zeroed). When the \(I_p^k(n_x)\) is reduced, the partial schedule length could be reduced, also (see Eq. (4)). Since the TSF’s \(p\)-edge selection rule minimizes the \(I_p^k(n_x)\), the following property is satisfied at the \(k\)-th step, \(k = 1, \ldots, v\), of the TSF:

\[
\text{cp(STG}(v,k) \leq \text{cp(STG}(v,k-1)).
\]

This property (named by Gerasoulis et al. [3] as the monotonicity of decrease in parallel time) is recognized, by several researchers [8,13], as an important characteristic of a clustering algorithm, without backtracking capability, which has a goal to minimize the schedule length.

5. **Simulation results**

We have conducted comparison, via simulation, of tree different multiprocessor scheduling techniques: list scheduling, Sarkar’s two-step scheduling, and chaining. These scheduling techniques are based on different principles. Both the list scheduling and chaining are single-pass techniques, which schedule one task at a time. The list scheduling can be characterized as a low-complexity local scheduling heuristic, since it examines only the ready tasks (i.e., unscheduled tasks with all their predecessors scheduled) in order to decide which task is to be scheduled next. The chaining represents a global scheduling heuristic, since at each scheduling step it considers all unscheduled tasks. However, this global scheduling perspective is partially attained at the price of added computational complexity. The DLS and TSF are taken as the representative algorithms of the list scheduling and chaining approach. The time complexity of the DLS is \(O(pv^2)\), while the complexity of TSF is \(O(v(v + e))\), where \(p\) is the number of
processors, \( v \) is the number of tasks, and \( e \) is the number of edges. On the other hand, Sarkar's two-step approach represents a clustering-based scheduling. To implement such algorithm we use DSC for the first step, and the Modified List Scheduling (MLS) algorithm [8] for the second step. The complexity of the DSC is \( O((v + e)\log v) \), and the complexity of the MLS is \( O(pB(v + e)) \), where \( B \) is the number of clusters formed in the first step. In the sequel, this two-step scheduling algorithm will be denoted as DSC/MLS.

We have generated 320 random task graphs with different number of nodes and graph parallelism, by using the method outlined in [1]. The graph parallelism is defined as \( gp(G) = (1/(c'(G)))(\sum^{e}_{i=0} w_i) \), where \( c'(G) \) is the length of the graph \( G \) when the edge weights are not included. Each graph contains between \( 80 \) and \( 250 \) nodes, and the node and edge weights are uniformly distributed over \( [10–200] \). The graph parallelism of the generated task graphs is within the range \([2–16]\).

We have carried out a number of simulations by scheduling the generated random task graphs onto a simulated fully-connected multiprocessor architecture, and by using the scheduling algorithm, the number of processors, and the amount of inter-task communication as parameters. Each simulation is characterized by a 4-tuple \( \text{EXP} = < TG, A, p, \alpha > \), which means that algorithm \( A \in \{ \text{DLS, TSF, DSC/MLS} \} \) is used to schedule the task graph \( TG \), which edge weights are multiplied by \( \alpha \in R^+ \), onto \( p \) processors. We introduce the parameter \( \alpha \) in order to analyze the algorithms' performances under different communication/execution ratios. Our simulations have been conducted for three \( \alpha \) values: \( \alpha = 0.5 \) (characterizing computation intensive applications), \( \alpha = 1 \) (characterizing balanced communication and computation applications), and \( \alpha = 2 \) (characterizing communication intensive applications), and five \( p \) values, \( p \in \{4, 6, 8, 10, 12\} \). For each simulation we have recorded: the degree of parallelism \( dp(\text{EXP}) = (gp(TG))/p \), and the obtained schedule length \( (sl(\text{EXP})) \).

Fig. 4 and Fig. 5 show the average percentage improvement of TSF over DLS and DSC/MLS, respectively, as a function of the degree of parallelism, for the three \( \alpha \) values. As can be seen, TSF gives better scheduling performances over the other two scheduling algorithms. Fig. 4 shows that if the computation dominates the communica-

![Graph showing average percentage improvement of TSF over DLS](image_url)
tion ($\alpha = 0.5$), then the improvement of TSF over DLS is relatively small (2%–3%). As the communication increases, this improvement decreases, for the case of low degree of parallelism, while for the medium and high degree of parallelism the improvement increases. The peak percentage improvement of the TSF over DLS is about 4%, for $\alpha = 1$, and 6.5%, for $\alpha = 2$. These results show that the TSF’s ability to make “more informed” decisions than the DLS, becomes important when the communication dominates or is balanced with the computation, and when the graph parallelism is greater than the system parallelism. Otherwise, when either the communication or the degree of parallelism is low, the list scheduling mechanism, which strictly follows the topological ordering of the task selection and forces the load balancing, is sufficient to attain a good schedule.

Fig. 5 shows that TSF outperforms the DSC/MLS for all studied levels of communication and degree of parallelism. Although the percentage improvement of the TSF over the DSC/MLS, on average, decreases with the increasing of the communication, TSF is still significantly better than the DSC/MLS even for high communication ($\alpha = 2$). Obviously, the reduction of parallel time, obtained after the first step of DSC/MLS, is

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\nu$</th>
<th>$e$</th>
<th>$CPU_{TSF}$</th>
<th>$CPU_{DSC/MLS}$</th>
<th>$CPU_{DLS}$</th>
<th>TSF/(DSC/MLS)</th>
<th>TSF/DLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40</td>
<td>60</td>
<td>0.02 sec</td>
<td>0.01 sec</td>
<td>0.01 sec</td>
<td>-4.83%</td>
<td>9.7%</td>
</tr>
<tr>
<td>20</td>
<td>155</td>
<td>270</td>
<td>0.19 sec</td>
<td>0.09 sec</td>
<td>0.03 sec</td>
<td>-10.9%</td>
<td>1.1%</td>
</tr>
<tr>
<td>40</td>
<td>610</td>
<td>1140</td>
<td>3.74 sec</td>
<td>1.47 sec</td>
<td>0.31 sec</td>
<td>-12%</td>
<td>1%</td>
</tr>
<tr>
<td>60</td>
<td>1365</td>
<td>2610</td>
<td>22.5 sec</td>
<td>8.41 sec</td>
<td>1.30 sec</td>
<td>-8.5%</td>
<td>0.7%</td>
</tr>
<tr>
<td>80</td>
<td>2420</td>
<td>4680</td>
<td>79.2 sec</td>
<td>39.0 sec</td>
<td>4.87 sec</td>
<td>-5.7%</td>
<td>0.27%</td>
</tr>
<tr>
<td>100</td>
<td>3775</td>
<td>7350</td>
<td>234.3 sec</td>
<td>121.9 sec</td>
<td>12.24 sec</td>
<td>3.8%</td>
<td>0.17%</td>
</tr>
<tr>
<td>120</td>
<td>5430</td>
<td>10620</td>
<td>520.1 sec</td>
<td>337.4 sec</td>
<td>26.05 sec</td>
<td>3.7%</td>
<td>0.01%</td>
</tr>
<tr>
<td>Graph parallelism</td>
<td>$\alpha = 0.5$</td>
<td>$\alpha = 1$</td>
<td>$\alpha = 2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2–6</td>
<td>$-0.26%$</td>
<td>$0.23%$</td>
<td>$1.55%$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6–10</td>
<td>$0.84%$</td>
<td>$1.36%$</td>
<td>$1.94%$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9–12</td>
<td>$0.65%$</td>
<td>$1.43%$</td>
<td>$2.07%$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12–16</td>
<td>$0.86%$</td>
<td>$1.23%$</td>
<td>$1.76%$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>$0.50%$</td>
<td>$1.01%$</td>
<td>$1.80%$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

not sufficient to compensate the performance loss during the second step, caused by a coarse granularity of the formed clusters.

To supplement simulation results we have considered an important task graph in the area of numerical computation, the task graph of LU decomposition of band matrices based on Gaussian elimination with partial pivoting [7]. We have considered the task graphs generated for a class of the band matrices of order $n \times n$ with $n - 1$ diagonals. For this case, the number of tasks $v$ is about $3n^2/8$, and the number of edges $e$ is about $3n^2/4$. In our study we have assumed that the average communication/execution ratio is 1. The improvement ratio of the TSF over DLS and DSC/MLS, as well as the total CPU time spent in scheduling these graphs for $p = 4$ processors on Silicon Graphics Indy workstation are given in Table 3. Although the difference in the CPU time between the TSF and two competitive algorithms seems to be significant, time complexity of all three algorithms, with respect to $n$, is the same, i.e., $O(n^4)$, since for this type of the task graphs $O(v) = O(e) = O(n^2)$.

Finally, we have examined the performance of the TSF for scheduling on an unbounded number of processors. To do this, we have scheduled the 320 random graphs by using the TSF (for $p = v$ processors), and DSC algorithms. The average improvement ratio of the TSF over DLS, for different levels of graph parallelism and communication/execution ratios is given in Table 4. The improvement of the TSF over DSC increases with the increasing of the communication, but the performance difference between two algorithms is rather small (less than 2% in all analyzed cases). On the other hand, an advantage of the DSC over TSF is a smaller time complexity, i.e., $O((v + e)\log v)$ vs. $O(v(v + e))$. However, it should be pointed out that the TSF's selection policy is optimized to be used for scheduling on a bounded number of processors. The existence of the chaining heuristic that will be more efficient than the TSF for an unbounded number of processors remains an open question.

6. Conclusion

A schedule for $p$ processors is a partition of the task set of a given task graph into $p$ disjoint task execution sequences such that the execution time is minimized. We have presented a new compile-time single-pass scheduling approach, called chaining, that presents a framework for scheduling task graphs with communication delays onto fully-connected multiprocessor architecture containing an arbitrary number of processors. Chaining is based on 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 the chaining approach is that it does not impose any constraints about the ordering according to which tasks are selected for scheduling. This scheduling freedom allows us to obtain a number of variants on the algorithm by specifying different rules by which the task and insertion location within the current partial schedule are selected. One of these variants, the Task Selection First (TSF) algorithm, has been presented, and its performances have been compared, by simulation, with a list scheduling algorithm and an algorithm based on the Sarkar’s two-step scheduling approach. TSF is a greedy algorithm that, at each step, initially selects the “most critical” task, and then chooses the location for task placement so that the extension of the partial schedule length is minimized. Simulations have carried out by altering over the graph parallelism, the communication/execution ratio, and the number of processors. Analysis show that TSF is superior over the two-step scheduling algorithm. The performance advantage of TSF over the list scheduling algorithm becomes significant when the communication dominates the computation, and the graph parallelism is larger than the number of available processors. Finally, we do not suggest that the TSF is the best or the only one chaining heuristic that could yield good results. Other, perhaps better chaining heuristics, with more sophisticated selection policies could yield even better results. We now intensively work in this direction.

Acknowledgements

The authors gratefully acknowledge the constructive discussions they had with Z. Radenkovic. Special thanks are due to Dr. M. Stojcev, who read and criticized the early drafts of this paper. Finally, the authors would also like to thank the anonymous referees for their careful review of the paper.

References