Evaluation of Static Scheduling Algorithms Improved with Partial Strict Triggering of Program Graph Nodes

Milan OJSTERŠEK*, Viljem ŽUMER*, Ljubo PIPAN**, Aleksander KVAS*, Boris BENKO*
*University of Maribor, The Faculty of Electrical Engineering and Computer Science
Smetanova 17, 2000 Maribor, Slovenia
**University of Ljubljana, The Faculty of Electrical and Computer Engineering
Tržaška 25, 1000 Ljubljana, Slovenia
tel. (+386) 62 25 461, fax (+386) 62 212-013, email: ojstersek@uni-mb.si

Abstract
Almost all coarse grained program graph nodes don't need all of their input operands at the beginning of their execution. Thereafter they can be scheduled a bit earlier. This type of program graph nodes triggering is called partial strict triggering. The missing operands will be requested later during the execution. Coarse grained program graph nodes send their output operands to all successors, as soon as they produce them. Successors of coarse grained program graph nodes will be scheduled earlier too, because they will receive their input operands sooner. An evaluation of improved (with partial strict triggering of coarse grained program graph nodes) CPM, VL and DSH scheduling algorithms is presented in this paper.

1. Introduction
Optimal execution of parallel programs which are executed on a parallel computer system depends on partitioning programs into modules and scheduling those modules for the shortest possible execution time. In this paper, we present three efficient algorithms for scheduling modules to the processing units of parallel computer system. Algorithm for the partitioning programs into modules is discussed in [6]. We will make only a brief description of it in chapter 2.

We have improved three well known static scheduling algorithms (CPM, VL and DSH) with partial strict triggering of program graph nodes. We compare our improved algorithms with original CPM static scheduling algorithm.

This paper is organised as follows. First we give a brief description of our partitioning algorithm. In the Section 3 we present partial strict triggering of program graph nodes. Next a brief description of improved scheduling algorithms is given. In the section 5, we present the model of a macro dataflow computer which support partial strict triggering of program graph nodes. Performance evaluation of improved scheduling algorithms with execution of program graph on the macro dataflow computer simulator is done in the section 6. And finally Section 7 presents concluding remarks as well as directions for future research work.

2. Description of the Partitioning Algorithm
We have used similar grain size determination algorithm as Sarkar's "internalisation" algorithm [8], which clusters nodes together to minimise the schedule length on an unbounded number of processors. The algorithm initially places each task in a separate cluster and considers the arcs in descending order according to the amount of data transferred over each arc. Given arc $A_{ij}$ connecting nodes $N_i$ and $N_j$, the algorithm merges the clusters containing these nodes to
internalise' any communications between nodes in these respective clusters. This merging step is accepted if it does not increase an estimate of the parallel execution time of the program graph on an infinite number of processors, where nodes in the same cluster are constrained to be executed on the same processor. The partitioning process in 'internalisation' algorithm is stopped when cluster’s execution time is equal or greater than one percent of ideal parallel execution time. A cluster must also satisfy a convexity constraint, which ensures that a cluster can run to completion once all its inputs are available. Our opinion is that the size of clusters (execution time) is computer architecture dependent, so our partitioning algorithm stops further aggregation into bigger clusters when the size of a cluster is equal or greater than maximal granularity of cluster which depends on the organisation of the macro dataflow computer architecture.

3. Partial Strict Triggering of Program Graph Nodes

![Diagram of partial strict triggered node](image)

**Figure 1:** Example of partial strict triggered node

Now, let us consider a group of program nodes that are connected with the precedence relation. If the sum of communication delays for transferring operands between nodes is greater than the sum of their execution times, it is possible to achieve fastest execution time with joining them in a larger program grain. Of course, there are other techniques for grain size determination, but it is interesting that joining small grains into larger one enables the processing element to start with execution of a new grain without all input operands. An example is shown in Figure 1. We aggregate fine grain program graph nodes (1, 2, 3) into a coarse grained program graph node. The new node N can start with the execution on a free processing unit immediately after operand number 1 is available, although operand number 2 is not present. This operand must be available at least 10 time units after the beginning of the node N execution to avoid the node execution delay. The operand number 4 is sent to the successor nodes 20 time units before the end of the execution of node N, so the triggering of successors which use this operand is faster.

We have introduced two new attributes for each communication arc. \( C_s \) represents operand’s relative sending time. It defines the time from the moment when the operand is sent to the end of a node execution. \( C_r \) represents operand’s relative receiving time. It defines the time from beginning of a program graph node execution to the moment when this operand must be present to avoid delaying the node’s execution. Operands which have \( C_r \) equal zero are called *strict operands* and must be present before the execution of their program graph nodes. Operands which have \( C_r \) greater than zero are called *nonstrict operands* and must be present between time from beginning of a program graph node execution and \( C_r \).

The following notation is used throughout this paper:

- \( G \) : graph \( G(\mathcal{N}, \mathcal{E}) \),
- \( \mathcal{N}(G) \) : set of nodes in \( G \),
- \( \mathcal{E}(G) \) : set of arcs in \( G \),
- \( n \) : number of nodes in \( G \),
- \( n_i \) : \( i \)-th node in \( G \); \( i=1...n \),
- \( T(n_i) \) : execution time of node \( n_i \),
- \( \mathcal{N}_S(n_i) \) : set of successors of the node \( n_i \),
- \( n_{S_i} \) : number of successors of the node \( n_i \),
- \( C(n_i, n_j, n_{ii}, n_{oj}) \) : communication delay time between nodes \( n_i \) and \( n_j \); \( n_i \) is source and \( n_j \) is destination, \( n_{ii} \) is a number of particular input operand of node \( n_j \), \( n_{oj} \) is a number of particular output operand of node \( n_i \),
- \( C_s(n_i, n_j, n_{ii}, n_{oj}) \) : relative sending time of operand between nodes \( n_i \) and \( n_j \); \( n_i \) is source and \( n_j \) is...
destination, nii is a number of particular input operand of node ni, noj is a number of particular output operand of node ni.

\[ CR(n_i, n_j, n_{ii}, n_{noj}) : \text{relative receiving time of operand between nodes } n_i \text{ and } n_j \text{ is source and } n_j \text{ is destination, } n_{ii} \text{ is a number of particular input operand of node } n_j, \text{ noj is a number of particular output operand of node } n_i; \]

\[ O(G) : \text{set of output nodes of graph } G, \]

\[ CP(n_j) : \text{the length of exit path for node } n_j. \]

4. Description of Scheduling Algorithms with Partial Strict Triggering of Program Graph Nodes

CPM scheduling algorithm \([4,6,7]\) assigns the program graph nodes to the processing elements on the basis of the priority list scheduling method. A priority weight for each node is a length of the longest path from program graph node to the terminal node. The original CPM algorithm computes the priority weight (the length of exit path) of each node without using communication delay time for transferring operands between nodes. We have defined new priority weight which is computed by using execution time of a node and communication delay times, \( C_S, C_R \) of it’s input and output operands. The length of exit path for node \( n_i \) (\( CP(n_i) \)) is computed on the following way:

\[ \forall n_i \in O(G): CP(n_i) = T(n_i) \]

\[ \forall n_i \not\in O(G) \land \forall n_j \in Ns(n_i) \land \forall n_j \not\in CP(n_i): \]

\[ CP(n_j) = \max_{j} \{ CP(n_j) - C(n_i, n_j, n_{ii}, n_{noj}) + CR(n_i, n_j, n_{ii}, n_{noj}) - T(n_j) \} \]

(3.1)

In improved CPM algorithm we also assure that all nonstrict operands are present in time from beginning of a program graph node execution to the moment when these operands must be present to avoid delaying the node’s execution. If we can’t assure this condition for nonstrict operand, we assign its \( C_T \) to zero. Time complexity of improved CPM algorithm is \( O(n^2) \), where \( n \) is number of program graph nodes.

Vertically Layered (VL) scheduling algorithm \([2,3,6]\) is based on two philosophies:

i) assigning concurrently executable nodes to separate processing units and

ii) assigning nodes connected serially to the same processing element.

Main idea of this algorithm is the distribution of program graph nodes into vertical layers, where nodes constituting a single layer, can be allocated to a processing element. Actual allocation is done in two phases: the separation and optimisation phase. In separation phase critical path (CP) is identified and nodes on CP are assigned to one vertical layer. Rearranging is done in an iterative manner as follows: nodes on longest directed path emanating from an arc in a node that is already assigned, are assigned to the next available vertical layer.

The separation phase does not take into account communication delays among PEs. The optimisation phase rearranges nodes by considering inter-PE communication delays. For example, if two subsets of nodes are arranged in two distinct vertical layers, and there is the transitory relationship between them, there will be inter-PE communication costs associated with the execution of two vertical layers. In order to improve overall execution time, we can consider combining the subsets of nodes into a single vertical layer. This eliminates the communication time between two layers and the overall execution time is now the sum of execution times of subsets. However, if the new execution time results in a larger delay, two subsets are assigned to different PEs. We improved VL algorithm optimisation phase that rearranges nodes by considering communication delay time, \( C_S \) and \( C_R \). We also assure that all nonstrict operands are present in time from beginning of a program graph node execution to the moment when these operands must be present to avoid delaying the node’s execution. If we can’t assure this condition for nonstrict operand, we assign its \( C_T \) to zero. Time complexity of improved VL algorithm is \( O(n^4) \), where \( n \) is number of program graph nodes.
Duplication scheduling heuristic algorithm [1,5,6] maximises parallelism and minimises communication delays by insertion of ready program graph nodes into idle time slots of Gantt chart and by duplication of critical program graph nodes. We have focused our research efforts to three main extensions of the original heuristic:

- We have converted program graphs that have nonstrict operands into program graphs that have all strict operands. A new type of program graph is called a strictly triggered program graph. Each program graph node of original program graph is transformed into several nodes of strictly triggered program graph. We call them a group of strict nodes. All nodes of a group have to be executed on the same PE. PE pre-empts the execution of a group if in the execution phase a nonstrict operand is not present. While this operand is not present, PE executes other groups. When a nonstrict operand arrives PE continues with the execution of the pre-empted group.

- We proposed a new processing units allocation scheme, which greatly improves the efficiency of scheduling. An original DSH algorithm uses a length of longest path from the node to the exit node and number of immediate successors as a priority weight for processor allocation. We introduced predecessor selection scheme (PSS) and successor selection scheme (SSS). Both selection schemes are trying to decrease the amount of communication overhead between nodes by scheduling similar tasks on the same processing unit. The term similar is used to refer to tasks that are expected to communicate with each other, either directly or indirectly. PSS considers two tasks as similar, if one of the tasks is a direct or an indirect predecessor of the other one. SSS considers two tasks as similar if they have at least one successor task in common, i.e. they both send at least one message to the same destination.

- We have also extended DSH algorithm to take into account a finite number of communication channels between processing units. In this way, the accuracy of the results obtained from simulation of coarse grained program graph execution on the model of parallel computer has been improved.

Time complexity of improved DSH algorithm is $O(n^6)$, where $n$ is number of program graph nodes.

5. The Model of Macro Dataflow Computer

![Figure 2: The Model of a Macro Dataflow Computer](image)

We have introduced our model of a macro dataflow computer (figure 2) which supports partial strict triggering of coarse grained program graph nodes. Our model of a macro dataflow computer (MMDC) is a loosely coupled multiprocessor with processing elements (PE) and an interconnection network.

![Figure 3: A Structure of a Processing Element of the MMDC](image)
Every PE (figure 3) contains an input queue, a local multiprocessor memory, a memory processor, a scheduling processor, and a node execution processor. Program graphs are loaded on the model of a macro dataflow computer at the beginning of their execution. Inputs of nodes (tokens) are matched together in the memory processor. When all required inputs of a particular node are present, the memory processor forms the executable node (activity) and sends a node number to the scheduling processor. The scheduling processor selects the executable nodes on the basis of a Gantt chart which has been produced by the static scheduling algorithm. It sends activities to the node execution processor. The node execution processor executes activities. If a nonstrict operand of executed activity has not arrived in time when it is required, the node execution processor preempted execution of this activity and executes another activity. When the nonstrict operand of preempted activity arrives in the memory processor, the memory processor sends the address where the activity has been preempted into queue of preempted activities, which are in multiport memory. When the node execution processor finish with the execution of temporary executed activity, continues with the execution of the preempted activity. The node execution processor also sends results of computed activities to memory processor if the successor node is in the same PE or to other PEs.

In our simulation we assume that the model of a macro dataflow computer has the following features:

- All PEs have equal performances.
- A PE executes nodes and communicates with other units simultaneously.
- All PEs are connected with communication network, which has a finite number of communication channels.
- A memory space, needed for matching tokens in sets of tokens, forming activities, scheduling and executing activities of program graphs that are executed on the MMDC, must be less than the capacity of available memory space in the model of the MMDC.

6. Performance Evaluation of Improved Scheduling Algorithms

![Figure 4: Comparison of improved scheduling algorithms with original CPM algorithm. The total execution time of program graph produced with usage of: (1) original CPM scheduling algorithm is represented by Tcpmo(PG), (2) improved CPM scheduling algorithm is represented by Tcpmi(PG), (3) improved VL scheduling algorithm is represented by Tvl(PG), (4) improved DSH scheduling algorithm is represented by Tdsh(PG).](image)

We have evaluated improved scheduling algorithms with a simulation of a program graph execution on the MMDC. Program graph has been consisted from 200 nodes, 374 operands (306 strict operands and 68 nonstrict operands). It's level of granularity (average execution time of nodes/average communication delay time of operands) has been 0.092. We have used program graph and Gantt charts produced by original CPM algorithm and improved algorithms as simulation inputs. A comparison of ratios between total execution times of program graph with usage of original CPM algorithm and improved scheduling algorithms is depicted on the figure 4. As shown in figure 4 all improved scheduling algorithms outperform original CPM algorithm. Improvements have been from 54% to 213%. From the figure 4 we can see that the improved DSH algorithm has given the best improvement but it have the highest computational complexity (O(n^6)). VL algorithm has given better results than improved CPM algorithm, but it has higher computational complexity than CPM algorithm.

We have observed that improvements of our algorithms is higher if level of granularity is lower than one. If level of granularity is greater
than one then improvements are only few percents. In that case we use improved CPM algorithm, because it has the lowest computation complexity.

7. Conclusion

This paper presented performance evaluation of three different static scheduling algorithms improved with partial strict triggering of program graph nodes. The new model of a macro dataflow computer which supports partial strict triggering of program graph nodes is also presented. Simulation results of program graph execution on the MMDC with usage of Gantt charts produced by improved scheduling algorithms showed promising improvement over original CPM algorithm.

In our future research we will incorporate scheduling algorithms into our integrated programming environment for static partitioning and scheduling of time critical tasks which are executed on the macro dataflow realtime computer [9].

8. References


