Improving Efficiency of Program Graph Scheduling with Partial Strict Triggering of Program Graph Nodes

Milan Ojsteršek and Aleksander Kvas
FERI Maribor, Smetanova 17, Maribor, Slovenia
Phone: +386 2 220 7451, Fax: +386 2 251 1178
E-mail: ojstersek@uni-mb.si

Keywords: parallel computers, program graphs, scheduling

Received: June 15, 2003

An efficient scheduling of a parallel program onto the processors is critical for achieving a high performance from a parallel computer system. The scheduling problem is known to be NP-hard and heuristic algorithms have been proposed to obtain optimal and sub optimal solutions. The partitioning algorithm partitions an application into tasks with appropriate grain size and represents them in the form of a directed acyclic graph (DAG). The nodes of the resulting DAG are then scheduled onto the processors of a parallel computer system. We can see that 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 operand 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 CPM, VL and DSH scheduling algorithms is done in this paper. We have improved them with partial strict triggering of coarse grained program graph nodes.

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 a parallel computer system. An algorithm for the partitioning programs into modules is discussed in (Ojsteršek 1994). We will make only a brief description of it in chapter 2.

The general problem of multiprocessor scheduling can be stated as scheduling a set of partially ordered computational tasks onto a multiprocessor system so that a set of performance criteria will be optimised. The difficulty of the problem depends heavily on the topology of the program graph representing the precedence relations among the tasks, the topology of the multiprocessor system, the number of parallel processors, the uniformity of the node processing time and the performance criteria chosen. In general, the multiprocessor scheduling problem is computationally intractable even under simplified assumptions. Because of this computational complexity issue, many heuristic algorithms have been proposed to obtain optimal and suboptimal solutions to various scheduling problems (Palis et al. 1996, Gerasoulis & Yang 1992, Darbha & Agrawal 1998, Park & Chloe 2002).

The scheduling of programs onto parallel computer system can be achieved using three approaches: static, dynamic and hybrid. The distinction indicates the time at which the scheduling decisions are made. With *static* scheduling, information regarding the program graph representing the program must be estimated prior to execution. In static scheduling, each node of a program graph has a static assignment to a particular processor, and each time that task is submitted for execution, it is assigned to that processor. In *dynamic* scheduling, the parallel processor system must attempt to schedule tasks on the fly. Thus, the scheduling decisions are made while the program is running. The disadvantage of dynamic scheduling is the overhead incurred to determine the schedule while the program is running. *Hybrid* scheduling technique are a mix of static and dynamic methods, where some preprocessing is done statically to guide the dynamic scheduler and/or reduce the amount of undeterminism.

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 Section 3 we present partial strict triggering of program graph nodes. Next, a brief description of improved scheduling algorithms is given in Section 4. In Section 5 we present a model of a macro dataflow computer which supports partial strict triggering of program graph nodes. Performance evaluation of improved scheduling algorithms with execution of program graph on the macro dataflow computer sim-

ulator is done in Section 6. And, finally, Section 7 presents 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 (Sarkar 1989), 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 satisfies 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

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

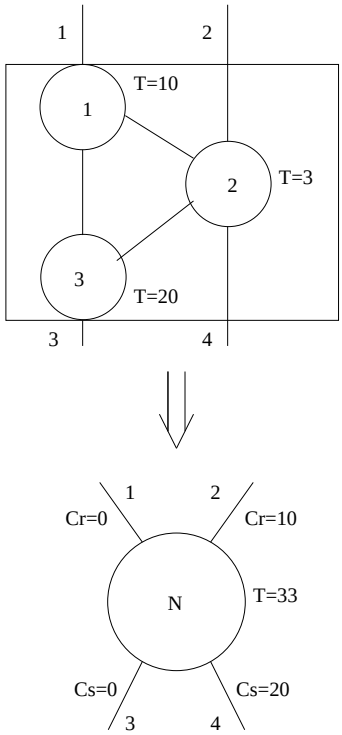

Figure 1: Example of partial strict triggered node

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 to 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 *non-strict 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 $G(N, E)$,
$N(G)$	: set of nodes in G,
$E(G)$	: set of arcs in G,
n	: number of nodes in in G,
n_i	: i-th node in $G; i = 1..n$,
$T(n_i)$	: execution time of node n_i,
$N_s(n_i)$	: set of successors of the node n_i,
ns_i	: number of successors of the node n_i,
$O(G)$	: set of output nodes of graph G,
$CP(n_j)$	: the length of exit path for node n_j,
$I(n_{ij})$	: set of input operands of n_j that originates from n_i,

$C(n_i, n_j, nii, noj)$: communication delay time 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 n_j, noj is a

number of particular output operand of node n_i,

$C_S(n_i, n_j, nii, noj)$: relative sending time of ope-rand 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 n_j, noj is a number of particular output operand of node n_i,

$C_R(n_i, n_j, nii, noj)$: relative receiving time of operand between nodes n_i and n_j, n_i is source and n_j is destination, nii is a number of parti- cular input operand of node n_j, noj is a number of particular output operand of node n_i.

4 Description of scheduling algorithms with partial strict triggering of program graph nodes

CPM scheduling algorithm (Kohler 1975, Ojsteršek 1994, Shirazi et al. 1990) 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 and C_R of its input and output operands. The length of exit path for node n_i ($CP(n_i)$) is computed in the following way:

$$\begin{aligned}&\forall n_i \in O(G) : CP(n_i) = T(n_i)\\ &\forall n_i \notin O(G) \wedge \forall n_j \in N_s(n_i) \wedge \forall n_j \; \exists \, CP(n_j) :\\ &CP(n_i) = MAX_{j=1,ns_i}(CP(n_j)\\ &\qquad +C(n_i, n_j, nii, noj) - C_R(n_i, n_j, nii, noj)\\ &\qquad -C_S(n_i, n_j, nii, noj) + T(n_i)) \qquad (4.1)\end{aligned}$$

In the improved CPM algorithm we also make sure 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_R to zero. Time complexity of original CPM algorithm is $O(n^2)$, where n is number of program graph nodes. Time complexity of improved CPM algorithm is also $O(n^2)$.

Vertically Layered (VL) scheduling algorithm (Hurson et al. 1990, Kvas et al. 1994, Ojsteršek 1994) is based on two philosophies:

1. assigning concurrently executable nodes to separate processing units and
2. 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 processing elements (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 times C_S and C_R. Original equations (Hurson et al. 1990) that are used to compare communication and execution time between nodes contain simple communication time delays. Now, if we use partial strict triggering, we can change the communication delays in the following way:

$$\begin{aligned}C_u(n_i, n_j) = MAX_{nii \, \in \, I(n_{ij})}(&C(n_i, n_j, nii, noj)\\ &-C_S(n_i, n_j, nii, noj) - C_R(n_i, n_j, nii, noj))\end{aligned} \qquad (4.2)$$

Then we use C_u in equations instead of C. 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_R to zero. Time complexity of original VL algorithm is $O(n^4)$, where n is number of program graph nodes. Time complexity of improved VL algorithm is also $O(n^4)$.

Duplication scheduling heuristic algorithm (Benko et al. 1995, Kruaratrachue & Lewis 1988, Ojsteršek 1994) maximises parallelism and mini-mises 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 original DSH algorithm is $O(n^4)$, where n is number of program graph nodes. Time complexity of improved DSH algorithm is $O(n^6)$.

5 The model of macro dataflow computer

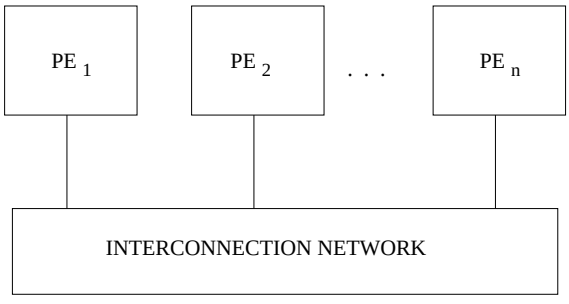

Figure 2: The Model of a Macro Dataflow Computer

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.

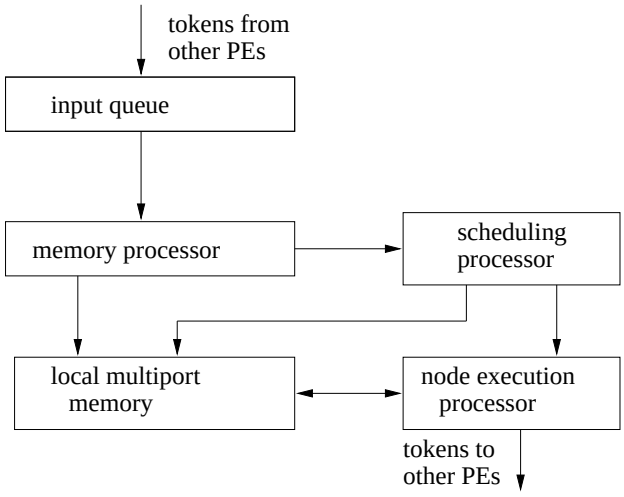

Figure 3: A Structure of a Processing Element of the MMDC

Every PE (Figure 3) contains an input queue, a local multiport 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 have not arrived in time when it is required, the node execution processor preempts 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 finishes with the execution of temporary executed activity, it then 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

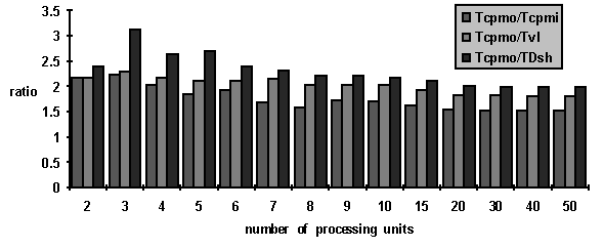

Figure 4: Comparison of improved scheduling algorithms with original CPM algorithm. The total execution time of program graph produced with usage of: (1) original CPM scheduling algorithm is represented by $T_{cpmo}(PG)$, (2) improved CPM scheduling algorithm is represented by $T_{cpmi}(PG)$, (3) improved VL scheduling algorithm is represented by $T_{vl}(PG)$, (4) improved DSH scheduling algorithm is represented by $T_{dsh}(PG)$.

We have evaluated improved scheduling algorithms with a simulation of a program graph execution on the MMDC. Program graph consisted of 200 nodes and 374 operands (306 strict operands and 68 nonstrict operands). Its 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 in Figure 4. As shown in Figure 4 all improved scheduling algorithms outperform original CPM algorithm. Improvements have been from 35% to 60%. From Figure 4 we can see that the improved DSH algorithm gave the best improvement but it have the highest computational complexity ($O(n^6)$). VL algorithm obtained better results than improved CPM algorithm, but it has higher computational complexity than the CPM algorithm.

We have observed that improvements of our algorithms are 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 previous work we build an integrated programming environment for static partitioning and scheduling of time critical tasks which are executed on the macro dataflow realtime computer (Ojsteršek & Žumer 1992). In our future research we will incorporate scheduling algorithms into the new version of integrated programming environment.

References

[1] Benko B., Ojsteršek M., Žumer V. (1995) Improvement of Duplication Scheduling Heuristic Algorithm with Nonstrict Triggering of Program Graph Nodes. *Proceedings of First Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, Aizu-Wakamatsu, Fukushima, Japan, IEEE Computer Society Press, p. 227-233.

[2] Darbha S. and Agrawal D. P. (1996) Optimal Scheduling Algorithm for Distributed-Memory Machines. *IEEE Trans. on Parallel and Distributed Systems*, January, vol. 9, No. 1, p. 87-95.

[3] Gerasoulis A. and Yang T. (1992) A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *J. Parallel and Distributed Computing*, vol. 16, p. 276-291.

[4] Hurson A. R., Lee B., Shirazi B., Wang M. (1990) A Program Allocation Scheme for Data Flow Computers. *Proc. of the 1990 Intern. Conf. on Parallel Processing*, University Park, Peen, Pennsylvania State Univ., vol. 1, p. 415-423.

[5] Kohler W. H. (1975) A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems. *IEEE Trans. on Computers*, December, p. 1235-1238.

[6] Kruaratrachue B., Lewis T. (1988) Grain Size Determination for Parallel Processing. *IEEE Software*, January, p. 23-33.

[7] Kvas A., Ojsteršek M., Žumer V. (1994) Eva-luation of Static Program Allocation Schemes for Macro Dataflow Computer. *Proceedings of the 20th EUROMICRO Conference*, Liverpool, England, IEEE Computer Society Press, p. 573-580.

[8] Ojsteršek M., V. Žumer V. (1992) Improving a Time Critical Task Execution Time Using an IPRESPS. Microprocessing and Microprogramming, Amsterdam, 34, 1-5, p. 197-200.

[9] Ojsteršek M. (1994) Partitioning and Scheduling Program Graphs onto Parallel Computer System. PhD thessis. University of Maribor, Faculty of Technical Sciences Maribor, June 1994 (in slovene).

[10] Palis M. A., Liou J. C., Wei D. S. L. (1996) Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Trans. on Parallel and Distributed Systems*, January, vol. 7, No. 1, p. 46-55.

[11] Park C.-I. and Choe T.-Y. (2002) An Optimal Scheduling Algorithm Based on Task Duplication. *IEEE Trans. on Computers*, April, vol. 51, No. 4, p. 444-448.

[12] Sarkar V. (1989) Partitioning and Scheduling Parallel Programs for Execution on MultiProcessors, MIT Press, 1989.

[13] Shirazi B., Wang M., Pathak G. (1990) Analysis and Evaluation of Heuristic Methods for Static Task Scheduling. *Journal of Parallel and Distributed Computing*, October, No. 10, p. 222-232.