Program Scheduling in Look-Ahead Reconfigurable Parallel Systems with Multiple Communication Resources

Eryk Laskowski
Institute of Computer Science
Polish Academy of Sciences
01-237 Warsaw, Ordona 21, Poland
laskowsk@ipipan.waw.pl

Abstract

The paper presents new graph structuring algorithms for look-ahead reconfigurable multi-processor systems. This architectural model is based on preparation of inter-processor link connections in advance in redundant communication resources (i.e. crossbar switches) in parallel with program execution, which enables elimination of connection reconfiguration time overheads. Application programs are partitioned into sections, which are executed using connections prepared in redundant communication resources. Parallel program structuring for execution in such systems incorporates task scheduling and graph partitioning problems. Presented algorithms apply two-phase approach, in which program task scheduling is solved by modified ETF heuristics and, in a second phase, a new iterative clustering heuristics is used for graph partitioning. The experimental results are presented, which compare performance of several graph partitioning heuristics for such environment.

1. Introduction

The paper is related to a new kind of parallel system model, based on dynamically reconfigurable connections between processors. This new approach, called *look-ahead dynamic inter-processor connection reconfiguration* [3, 5] is a multi-processor architectural model, which has been proposed to eliminate connection reconfiguration time overheads. It is based, at the hardware level, on multiple, redundant communication resources. These resources (multiple crossbar switches) are used for dynamic link connection setting in parallel with program execution and run-time look-ahead reconfiguration. It is combined with the new program execution control strategy. Application programs, designated for execution in such systems, are scheduled and partitioned into sections. During execution of some sections of a program, connections for the next program sections are prepared in spare crossbar switches. Preparing link connections in advance in parallel with program execution allows reduction (or total elimination) of connection reconfiguration time overheads, thus it can provide a time-transparent dynamic link connection reconfiguration.

Efficient execution of programs in the look-ahead reconfigurable system requires appropriate program structuring, which consists in task scheduling and program partitioning. The algorithm presented determines, at the compile time, the program schedule, partition into sections and the number of crossbar switches that provide time transparency of inter-processor connection reconfiguration. It is based on list scheduling and iterative task clustering heuristics. The article focuses on the partitioning phase, for which we present a new, refined section clustering heuristics. The new algorithm, intended for application in multi-crossbar systems, is controlled by communication resource utilization estimation functions, which allow to adapt its functioning to system and program parameters. The paper presents several new variants of such functions.

The paper consists of four parts. The first part describes the idea of look-ahead dynamic link reconfiguration. In the second part new parallel program execution paradigms and their impact on the scheduling methodology are presented. In the third part program graph scheduling and partitioning algorithms are discussed. The last part presents experimental results obtained with the use of proposed scheduling algorithms.

2. The look-ahead reconfigurable multi-processor systems

The look-ahead dynamic reconfigurable parallel system, investigated in the paper, contains multiple crossbars as redundant communication resources. It is a multiprocessor system with distributed memory and with point-to-point communication based on message passing (Fig. 1).

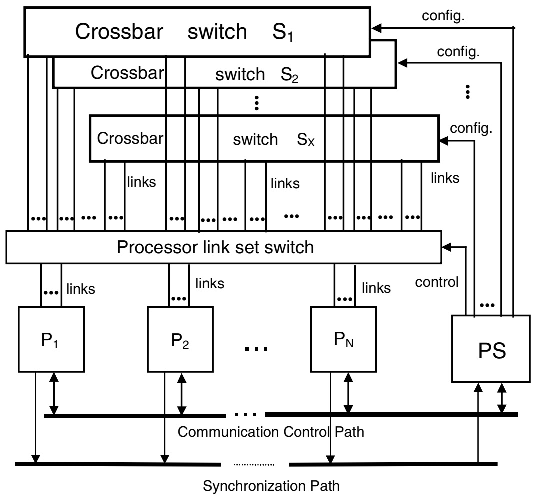


Fig. 1. Look-ahead reconfigurable system with multiple connection switches

Worker processors (P_1 – P_N) have sets of communication links (L_1 – L_K) connected to the crossbar switches S_1 … S_X. by the Processor Link Set Switch. This switch is controlled by the Control Processor (*PS*). The switches S_1 … S_X are interchangeably used as the active and configured communication resources. *PS* collects messages on the section execution states in worker processors (link use termination) sent via the Control Communication Path. The simplest implementation of such path is a bus but more efficient solution can assume direct links connecting worker processors with the *PS*. Depending on the availability of links in the switches S_1… S_X, *PS* prepares connections for execution of next program sections, in parallel with current execution of sections. Synchronization of states of all processors in clusters for next sections is performed using the hardware Synchronization Path [6]. When all connections for a section are ready and the synchronization has been reached, *PS* binds all links of processors, which will execute the section, with the look-ahead configured connections in a proper switch. Then, it enables execution of the section in involved worker processors.

Three program execution control strategies can be identified which differ in granularity of control:
1) synchronous, with inter-section connection switching controlled at the level of all worker processors in the system,
2) asynchronous processor-restrained, where inter-section connection switching is controlled at the level of dynamically defined worker processor clusters,
3) asynchronous link-restrained, with granularity of control at the level of single processor links.
In the synchronous strategy, processes in all processors in the system have to be synchronized when they reach the end of section points. In the asynchronous processor-restraint strategy, process states in selected subsets of processors are synchronized when they reach the end of use of all links in a section. In the asynchronous link-restraint strategy, the end of use of links in pairs of processors has to be synchronized. In this paper we assume the asynchronous processor-restrained strategy.

A parallel program is represented by a Directed Acyclic Graph (DAG), where nodes represent computation tasks and directed edges represent communication. The weight of node represents task execution time, the weight of edge is a communication cost. The graph is assumed to be static and deterministic. Program is executed according to the *macro-dataflow* [4] model.

3. Program structuring algorithms in the look-ahead configurable environment

In the look-ahead reconfigurable environment the schedule determines task execution order and program partitioning into sections. Schedule is defined as task-to-processor and communication-to-link assignment with specification of starting time of each task and each communication. Partition is defined as communication-to-resources assignment. Both schedule and partition have to preserve the precedence constraints coming from the program graph and from assumed execution model.

3.1. The program schedule representation

In the paper, a program with specified schedule is expressed in terms of the Assigned Program Graph (APG), see Fig 2a. APG assumes the synchronous communication model (CSP-like). Two kinds of nodes are used in an APG: the code nodes (which correspond to tasks in DAG, rectangles in Fig. 2a) and communication instruction nodes (circles in Fig. 2). Activation edges are shown as vertical lines in Fig. 2a, communication edges as horizontal lines (solid for inter-processor, dashed for intra-processor communications).

Asynchronous, non-blocking communications in a look-ahead reconfigurable environment are modeled in APG as activation paths on the sender processor. They are used for sending a message to the link subgraph and as activation paths on the receiver processor, which transmit a message from a link to the processor. The processor link works independently of the processor and others links, so it is modeled as a subgraph (marked as L_{i1} on Fig. 2a), parallel to the computation path.

The Communication Activation Graph (CAG) is a program graph partitioning representation. CAG contains information necessary for graph partitioning, thus it enables an easier partitioning analysis. This graph is composed of nodes, which correspond to external communication edges of the APG program graph, and of

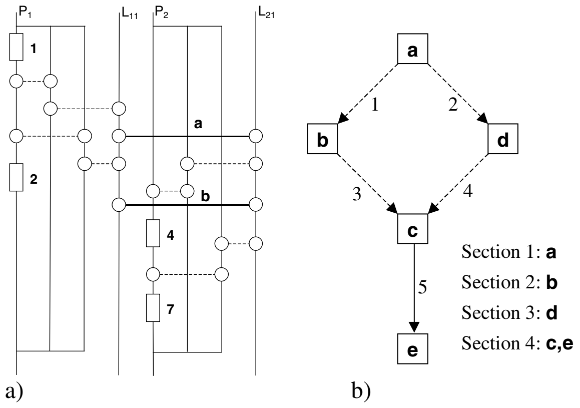


Fig. 2. a) Modeling of scheduled *macro-dataflow* graph by the APG. b) Communication Activation Graph partitioned into sections.

edges, which correspond to activation paths between communication edges of the APG. Exemplary partitioning is shown in Fig. 2b (edges, which are section boundaries are denoted by dashed lines).

Program sections are defined by identification of such subgraphs in the APG or in CAG that the validity conditions hold. The following validity conditions assure correct execution of many sections in parallel in the system with the look-ahead created connections:

a) Section subgraphs corresponding to program sections are mutually disjoint in respect to external communication edges. Each communication belongs to one and only one section.

b) The edges, which connect nodes contained in a section subgraph define a connected subgraph when considered as undirected.

c) All nodes on each path, which connects two nodes belonging to a section subgraph belong to the same section.

d) A correct partition shows stability of inter-processor link connections inside sections. Processor link connections inside section subgraphs do not change.

3.2 The new scheduling algorithm

Program structuring algorithms, presented in the paper, apply a two-phase approach to solve the problem of scheduling and graph partitioning in the look-ahead reconfigurable environment [2]. In the first phase, a list scheduling algorithm is applied to obtain a program schedule with a reduced number of communications and minimized program execution time. In the second phase, scheduled program graph is partitioned into sections for execution in the assumed environment.

The scheduling algorithm, used in the first phase of program structuring, is based on the ETF /Earliest Task First/ heuristics, proposed by Hwang et al. [1]. In our

```
Procedure Ready($n_i$, $P_i$)
Time := 0
For each $n_j \in$ Predecessors $n_i$
     $T_{Arrive}$ := finishing time of task $n_j$
     $P_j$ := processor which task $n_j$ is scheduled on
     If $P_j \neq P_i$ Then
          $T_{Arrive}$ := $T_{Arrive}$ + $c_{j,i}$ {cost of comm. from $T_j$ to $T_i$}
          If $P_i$ and $P_j$ are connected Then
               send := link of $P_j$ connected to $P_i$
               recv := link of $P_i$ connected to $P_j$
          Else
               send := last recently used link of $P_j$
               recv := last recently used link of $P_i$
               If time since last use of link $L_{j,send}$ or
                    link $L_{i,recv}$ in previous configuration $< c_R$
               Then
                    $T_{Arrive}$ := $T_{Arrive}$ + $c_R$
          EndIf
          Allocate communication $e_{j,i}$ on
                         links $L_{j,send}$ and $L_{i,recv}$
     EndIf
     If Time < $T_{Arrive}$ Then
               Time := $T_{Arrive}$
EndFor
Return Time
```

Fig. 3. The *Ready* procedure used in scheduling algorithm.

implementation of ETF a system with look-ahead dynamically created connections is assumed. We take into account links contention and the limited number of links in each processor.

Modification of ETF consists in new formula used for evaluation the earliest starting time (*Ready* procedure, see [1] for details). The flow chart of *Ready* procedure used in our modified ETF algorithm is given in Fig. 3. *Ready* (n_i, P_i) returns the time when the last message for task n_i will arrive at processor P_i. Additional time overhead (c_R in Fig. 3) represents the start delay of communication when network topology should be changed and there is no sufficiently long time gap after last communication to do reconfiguration in advance and without delaying program execution. These link reconfiguration time overheads are minimized by reduction of the number of link reconfigurations.

3.3. The partitioning algorithm

A second phase of the program structuring is the graph partitioning algorithm (Fig. 4). It finds program graph partitioning into sections and assigns a crossbar switch to each section. It also finds the minimal number of switches, which allow program execution without reconfiguration time overheads. The algorithm starts with an initial partition, in which each section is built of a single communication and all sections are assigned to the same crossbar switch. In each step, a vertex of CAG is selected and then the algorithm tries to include this vertex to a union of existing sections determined by edges of the current vertex. The heuristics tries to find such a union of sections, which doesn't break rules of graph partitioning. The union, which gives the shortest program execution time is selected.

IEEE
COMPUTER
SOCIETY

```
Begin
B := initial set of section, each section is composed of
       single communication and assigned to crossbar 1
curr_x := 1    {current number of switches used}
finished := false
While not finished
     Repeat until each vertex of CAG is visited and there
     is no execution time improvement during last β steps
       v := vertex of CAG which maximizes the selection
           function and which is not placed in tabu list
       S := set of sections that contain communications
                  of all predecessors of v
       M := Find_sections_for_clustering(v, S)
       If M ≠ ∅ Then
         B := B - M
         Include to B a new section built of v and com-
         munications that are contained in sections in M
       Else
         s := section that consists of communication v
         Assign crossbar switch (from 1..curr_x) to
                                                section s
         If there are time overheads Then
           curr_x := curr_x + 1
           Break Repeat
         EndIf
       EndIf
     EndRepeat
     finished := true
EndWhile
End
```

Fig. 4. The general scheme of the graph partitioning algorithm.

The program execution time is estimated by simulated execution of the partitioned graph in a modeled look-ahead reconfigurable system. An APG graph with a valid partition is extended by subgraphs, which model the look-ahead reconfiguration control. The functioning of the Communication Control Path, Synchronization Path and the Control Subsystem *PS* are modeled as subgraphs executed on virtual additional processors. Weights in the graph nodes correspond to latencies of respective control actions, such as crossbar switch reconfiguration, bus latency, and similar.

When section clustering doesn't give any execution time improvement, the section of the current vertex is left untouched and crossbar switch is assigned to it. The choice of the switch depends on program execution time. When the algorithm cannot find any crossbar switch for section that allows creation of connections with no reconfiguration time overhead, then current number of switches used (*curr_x* in Fig. 4) is increased by 1 and the algorithm is restarted.

The vertices can be visited many times. The algorithm stops when all vertices have been visited and there hasn't been any program execution time improvement in a number of steps. The heuristics manages a list of last visited vertices (*tabu list,* Fig. 4), which prevents the algorithm from frequent visiting small subset of all vertices.

A vertex selection heuristics is applied at each iteration step. It selects vertex, which maximizes the value of selection function $Z(v)$. The following APG and CAG graph parameters are taken into account during computation of value of selection function:

a) the *critical path* CP of APG,
b) the *delay* D of vertex of CAG,
c) value of *reconfiguration criticality* Q for the investigated vertex,
d) the dependency on links use between communications.

Critical path of APG is established in the graph partitioned into sections according to the best recent partition found.

The delay $D(v)$ of vertex v is defined as follows:

$$D(v) = I_v / (su_v - max(eu_{P(v)}))$$

where (as shown in Fig. 5):

$I_v = (s_v - max(e_{P(v)}))$ – the length of reconfiguration interval of vertex v

$P(v)$ – parents of vertex v,

e_v – finishing time of vertex v,

s_v – starting time of vertex v,

su, eu are starting and finishing times, respectively, in APG with reconfiguration time overheads neglected.

The choice of the vertex for visiting depends on the reconfiguration time overheads. These overheads are measured by applying the *critical point of reconfiguration* heuristics. For every communication v, the value of reconfiguration burden $C(v)$ is computed:

$$C(v) = I_v / t_R$$

where t_R is reconfiguration overhead.

The value Q of *reconfiguration criticality* function for vertex v is equal to sum of reconfiguration burden of all vertices whose reconfiguration interval is overlapping with reconfiguration interval of v (see Fig. 5):

$$B(t) = \sum C(v_i) : i = 1 \ldots n,\ s_i > t > max(e_{P(i)})$$
$$Q(v) = max(B(t) : s_v > t > max(e_{P(v)}))$$

The communication vertices of CAG are classified into three disjoint sets depending on their relationship in processor link use. The first set G_1 contains vertices which

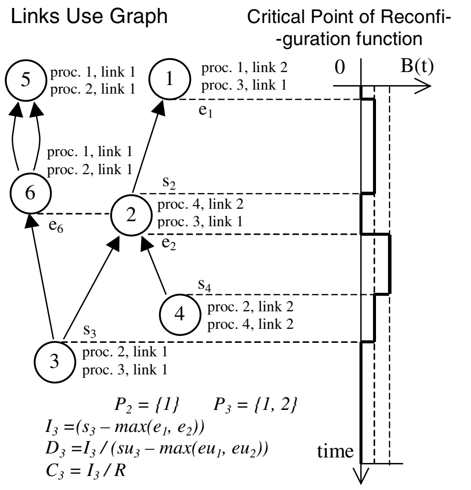


Fig. 5. Evaluation of the delay and critical point of reconfiguration based on the link use graph.

IEEE COMPUTER SOCIETY

use the same links as one of their parent vertices (two edges between vertices v_5, v_6; Fig. 5). The second set G_2 contains vertices, which cannot be clustered into single section with their parents because of conflicting link connection requirements. The third set G_3 contains vertices which do not belong to any of previous sets. When visiting a vertex, the set G_1 is first considered, because it is advised to include such communication vertices into the same section. The set G_3 is next considered. The set G_2 is considered as the last one.

By considering the described APG graph parameters and the link use dependencies between communication vertices, several vertex selection heuristics have been identified (iter0÷iter4). They select a vertex with the biggest value of selection function Z, which is computed as follows:

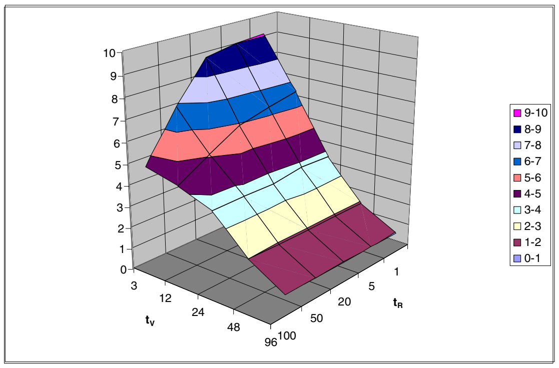
iter0 – Z = Q,
iter1 – Z = c_1 Q + c_2 D + c_3 CP /linear combination of CP, D, Q; c_1, c_2, c_3: arbitrary constants/,
iter2 – sort vertices according to value G, CP, D, Q, then assign first vertex the biggest Z value,
iter3 – Z = D,
iter4 – Z = CP.

4. Experimental results

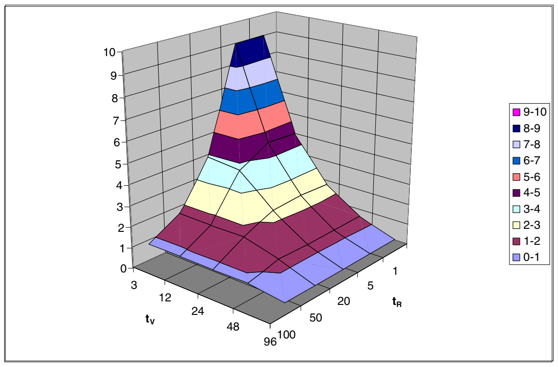
The experiments concerned with this paper evaluated the execution efficiency of programs for different program execution control strategies and system parameters and the influence of the number of redundant link connection switches on reduction of reconfiguration time overheads. During experiments, the performance of presented graph structuring algorithms and the usability of vertex selection heuristics was investigated.

The results were obtained for two families of exemplary program graphs: **test7** (randomly generated graphs, which model applications with irregular communication patterns), **test8** (graph with communication patterns of numeric applications), executed in the *look-ahead* and in *on-request* system, with parameters: number of processors 4, 8, 12, number of proc. links 2, 4, synchronization via bus or a hardware barrier, reconfiguration time of a single connection $\mathbf{t_R}$ and section activation time $\mathbf{t_V}$ in range 1-100.

The program execution speedup as a function of parameters of reconfiguration control ($\mathbf{t_R}$ and $\mathbf{t_V}$), on the example of *test7* and *test8* graphs is shown in Fig. 6, 7. For low values of control overhead $\mathbf{t_V}$ crossbar redundancy has big influence on total program execution time. Multiple crossbar switches used with the look-ahead control strongly reduce reconfiguration time overheads. When $\mathbf{t_V}$ is bigger, the *look-ahead* strategy suffer from section activation time overheads and reduction of execution time is not so substantial. The larger is the number of links in a processor, the look-ahead method is



a) – 6 crossbars



b) – 2 crossbars

Fig. 6. Speedup for *test8* program in the *look-ahead* environment (12 proc., 4 proc. links) for different values of system parameters t_R, t_V.

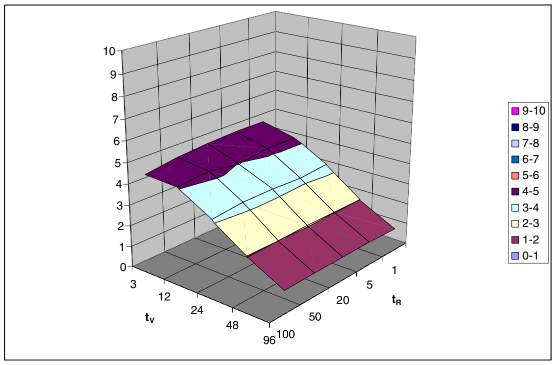


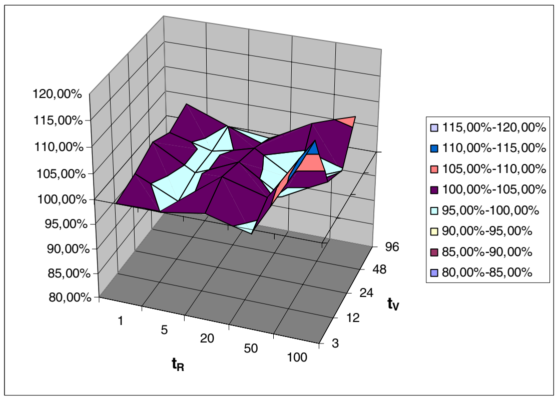
Fig. 7. Speedup for *test7* program in the *look-ahead* environment (12 proc., 4 proc. links, 6 crossbars) for different values of t_R, t_V.

successfully applicable for wider range of reconfiguration and activation time parameters than with the *on-request* reconfiguration.

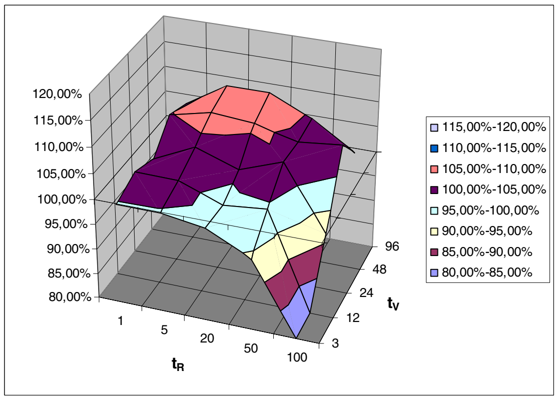
Fig. 8 shows the relative performance of structuring algorithm with different versions of vertex selection heuristics used. Our experiments show that the performance depends mainly on parameters of program graphs (their granularity) and parameters of reconfiguration control subsystem ($\mathbf{t_R}$ and $\mathbf{t_V}$). For programs with fine-grain parallelism and intensive reconfiguration during execution (i.e. all graphs from *test7* family in our experiments) there is only a small difference between iter0÷iter4 heuristics. The difference in total execution time of partitioned program was up to 6.5%. For such graphs reconfiguration control subsystem is overloaded by incoming reconfiguration requests and changing of the order of selection of vertices during program structuring could not improve the performance.

For programs with coarse-grain parallelism and small reconfiguration requirements (i.e. *test8*), the choice of vertex selection heuristics plays important role in optimal program execution. When reconfiguration efficiency of the system is low (Fig. 8a, $\mathbf{t_R}$ = 100) the best results are obtained by *iter1* and *iter2* heuristics, which combine several parameters during evaluation of the selection function. The worst results are obtained by heuristics that use only one parameter during evaluation of selection function (i.e. *iter4, iter0, iter3).* For system with low efficiency of section activation control ($\mathbf{t_V}$ = 96) the *iter4* heuristics gives the best results (Fig. 8b). The *iter3* heuristics gives in almost all cases the worst results.

The analysis of experimental results for wider selection of program graphs has shown that methods *iter0, 1, 2* behaves better than others. These heuristics use *reconfiguration criticality*, thus we could deduce that this APG graph parameter is the most important.



a) test8 – iter1/2 against iter0/3/4



b) test8 – iter4 against iter0/1/2/3

Fig. 8. Relative speedup obtained by graph structuring algorithms for different vertex selection heuristics (average – 100%) in system with 12 processors, 4 proc. links, 6 crossbars.

5. Conclusions

Several iterative graph structuring algorithms for the look-ahead reconfigurable multi-processor system have been presented in the paper. During experiments we have found the relationship between program time parameters, number of switches and the effective speedup achieved. Increasing the number of switches allows to reduce (or even completely eliminate) time overheads connected with connection creation. This effect is especially visible in systems with low reconfiguration efficiency or for fine-grain parallel programs with high reconfiguration demands. For a program for which inter-processor connection reconfiguration completely overlaps with program execution, the multi-processor system behaves as a fully connected processor structure. Experiments show that reduction of time overheads, introduced by section activation and termination control, is possible by application of hardware synchronization at the system architecture level. The future works will focus on partitioning algorithm improvements and further optimizations in section clustering and mapping of communication to resources.

References

[1] Jing-Jang Hwang, Yuan-Chien Chow, Frank D. Angers, Chung-Yee Lee; *Scheduling Precedence Graphs in Systems with Interprocessor Communication Times*, Siam J. Comput., Vol. 18, No. 2, pp. 244-257, April 1989.
[2] E. Laskowski *Fast Scheduling and Partitioning Algorithm in the Multi-Processor System with Redundant Communication Resources*, PPAM 2001
[3] E. Laskowski, M. Tudruj, *A Testbed for Parallel Program Execution with Dynamic Look-Ahead Inter-Processor Connections*, Proc. of the 3rd International Conference on Parallel Processing and Applied Mathematics PPAM '99, Sept. 1999, Kazimierz Dolny, pp. 427-436.
[4] El-Rewini H., Lewis T. G., Ali H. H. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall 1994
[5] M. Tudruj, *Look-Ahead Dynamic Reconfiguration of Link Connections in Multi-Processor Architectures*, Parallel Computing '95, Gent, Sept. 1995, pp. 539-546.