Parallel Code Generation in MathModelica / An Object Oriented Component Based Simulation Environment

Peter Aronsson, Peter Fritzson
(petar,petfr)@ida.liu.se

Dept. of Computer and Information Science,
Linköping University, SE-581 83 Linköping, Sweden

Abstract. Modelica is an a-causal, equation based, object oriented modeling language for modeling and efficient simulation of large and complex multi domain systems. The Modelica language, with its strong software component model, makes it possible to use visual component programming, where large complex physical systems can be modeled and composed in a graphical way. One tool with support for both graphical modeling, textual programming and simulation is MathModelica.

To deal with growing complexity of modeled systems in the Modelica language, the need for parallelization becomes increasingly important in order to keep simulation time within reasonable limits.

The first step in Modelica compilation results in an Ordinary Differential Equation system or a Differential Algebraic Equation system, depending on the specific Modelica model. The Modelica compiler typically performs optimizations on this system of equations to reduce its size. The optimized code consists of simple arithmetic operations, assignments, and function calls.

This paper presents an automatic parallelization tool that builds a task graph from the optimized sequential code produced by a commercial Modelica compiler. Various scheduling algorithms have been implemented, as well as specific enhancements to cluster nodes for better computation/communication tradeoff. Finally, the tool generates simulation code, in a master-slave fashion, using MPI.

Keywords: Object Oriented Modeling, Visual Programming, Components, Scheduling, Clustering, Modelica, Simulation, Large Complex System

1 Introduction

Modelica is an a-causal, object-oriented, equation based modeling language for modeling and simulation of large and complex multi-domain systems [15, 8] consisting of components from several application domains. Modelica was designed by an international team of researchers, whose joint effort has resulted in a general language for design of models of physical multi-domain systems. Modelica has influences from a number of earlier object oriented modeling languages, for instance Dymola [7] and ObjectMath [9].

The four most important features of Modelica are:

– Modelica is based on equations instead of assignment statements. This permits a-causal modeling that gives better reuse of classes since equations do not specify a certain data flow direction. Thus a Modelica class can adapt to more than one data flow context.
– The possibility of having model components of physical objects from several different domains such as e.g. electrical, mechanical, thermodynamic, hydraulic, biological and control applications can be described in Modelica.
– Modelica is an object-oriented language with a general class concept that unifies classes, generics (known as templates in C++) and general subtyping into a single language construct. This facilitates reuse of components and evolution of models.
– The strong software component model in Modelica has constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex physical systems, and to some extent for software systems.

The class, also called model, is the building block in Modelica. Classes are instantiated as components inside other classes, to make it possible to build a hierarchical model of a physical entity. For instance, an electrical DC motor can be composed of a resistor, an inductance and a component transforming electricity into rotational energy, see Figure 1. The model starts with two import statements, making it possible to use short names for models defined in the **Modelica.Electrical.Analog.Basic** package (the first import statement) and using the short name for the **StepVoltage** model defined in the Modelica package **Modelica.Electrical.Analog.Sources** (the second import statement). The import statements are followed by several component instantiations, where modification of components is used. The modification of components is a powerful language construct that further increases the possibility of reuse. The next part of the model definition is the `equation` section, see Figure 1. It can consist of arbitrary equations, involving the declared variables of a model. It can also contain `connect` statements, which are later translated into equations that couple variables in different components together.
One tool for developing models, for simulation and for documentation is MathModelica [14]. It integrates Modelica with the mathematical engineering tool *Mathematica* and the diagram editor *Visio* to allow the user to work with models both in a powerful computer algebra system and by using component based modeling in a drag and drop/connect fashion in a graphical environment, see Figure 2. In MathModelica a model can also be entered in a Mathematica notebook document as ordinary text. For example, the `dcmotor` model can be simulated by:

```
Simulate[dcmotor,{t,0,50}];
```

The MathModelica environment then compiles the Modelica model into C, which is then linked with a solver into an executable file. The result from running the simulation consist of a number of variables changing over time, i.e. they are functions of time. In MathModelica these variables are directly available and can be, for instance, plotted, see Figure 3.
When a model described in Modelica is to be simulated the involved models, types and classes are first fed to a compiler. The Modelica compiler flattens the

```
model dcmotor
  import Modelica.Electrical.Analog.Basic.*;
  import Modelica.Electrical.Analog.Sources.StepVoltage;
  Resistor R1(R=10);
  Inductor L(L=0.01);
  EMF emf;
  Ground G;
  StepVoltage src;
equation
  connect(src.p,R1.p);
  connect(R1.n,L.p);
  connect(L.n,emf.p);
  connect(emf.n,G.p);
  connect(G.p,src.n);
end dcmotor;
```

Fig. 1. A simple model of a DCmotor described in the Modelica modeling language.

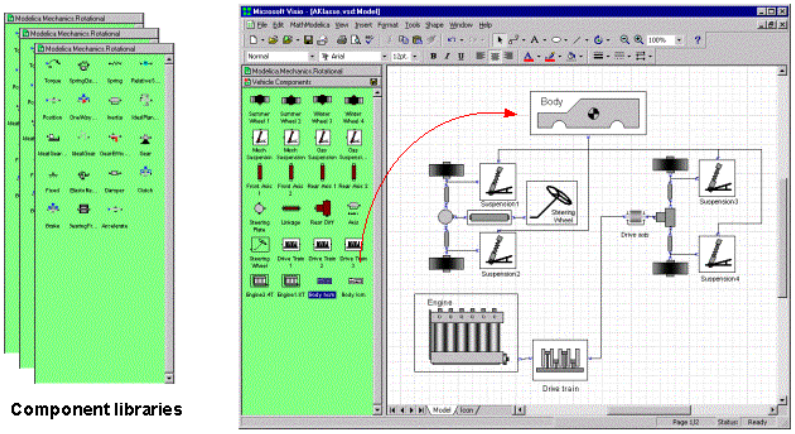

Fig. 2. Visual component based modeling in MathModelica.

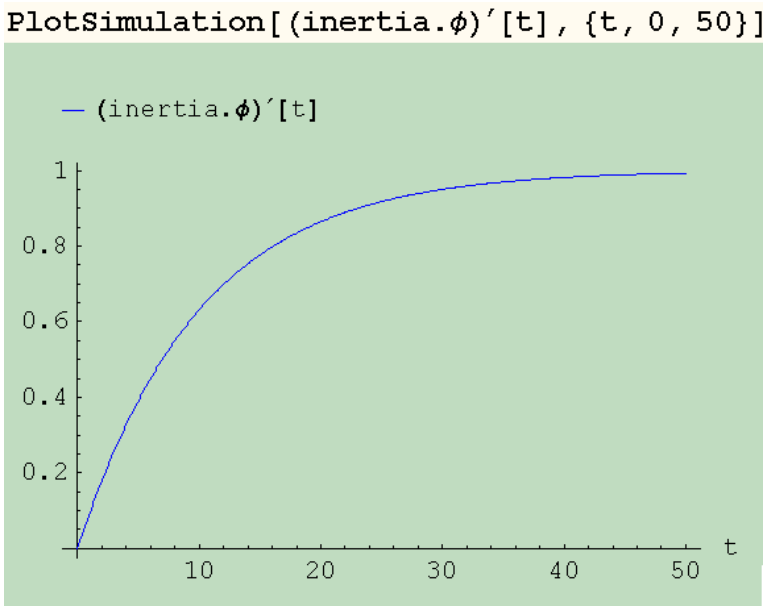

Fig. 3. A Notebook document containing a plot command.

object oriented structure of a model into a system of differential algebraic equations (DAE) or a system of ordinary differential equations (ODE), which during simulation is solved using a standard DAE or ODE solver. This code is often very time consuming to execute, and there is a great need for parallel execution, especially for demanding applications like hardware-in-the-loop simulation.

The flat set of equations produced by a Modelica compiler is typically sparse, and there is a large opportunity for optimization. A simulation tool with support for the Modelica language would typically perform optimizations on the equation set to reduce the number of equations. One such tool is Dymola [6], another is MathModelica [14].

The problem presented in this paper is to parallelize the calculation of the states (the state variables and their derivatives) in each time step of the solver. The code for this calculation consists of assignments of numerical expressions, e.g. addition or multiplication operations, to different variables. But it can also contain function calls, for instance to solve an equation system or to calculate sin of a value, which are computationally more heavy tasks. The MathModelica simulation tool produces this kind of code. Hence we can use MathModelica as a front end for our automatic parallelization tool. The architecture is depicted in Figure 4, showing the parallelizing tool and its surrounding tools.

To parallelize the simulation we first build a task graph, $G = (V, E)$ where each task $v \in V$ corresponds to a simple binary operation, or a function call. A data dependency is present between two tasks v_1, v_2 iff v_2 uses the result from v_1. This is represented in the task graph by the edge $e = (v_1, v_2)$. Each task

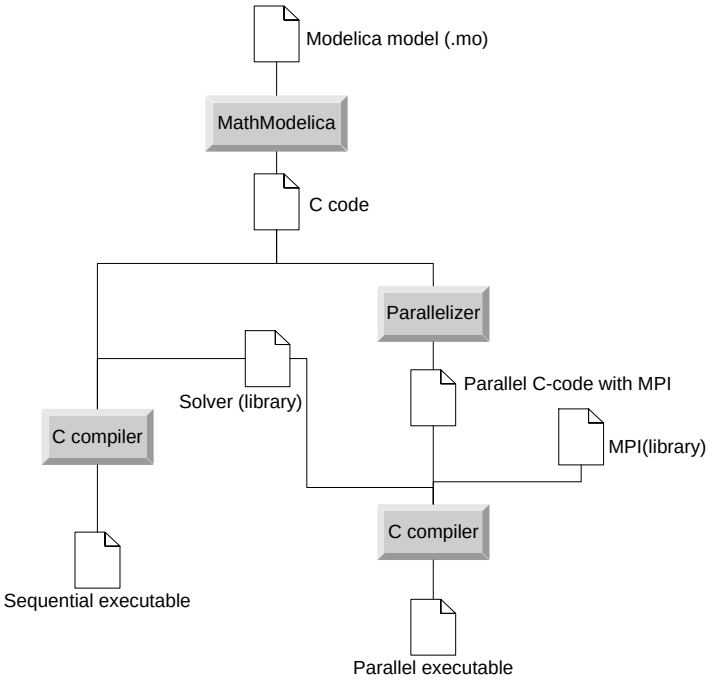

Fig. 4. The architecture of a Modelica simulation environment

is assigned an execution cost which corresponds to a normalized execution time of the task, and each edge is assigned a communication cost corresponding to a normalized communication time between the tasks if they execute on different processors. The goal is to minimize the execution time of the parallel program. This often means that the communication between processors must be kept low, since interprocessor communication is very expensive. When two tasks execute on the same processor, the communication cost between them is reduced to zero.

Scheduling and partitioning of such task graphs described above has been studied thoroughly in the past three decades. There exists a plethora of different scheduling and partitioning algorithms in the literature for different kinds of task graphs, considering different aspects of the scheduling problem. The general problem of scheduling task graphs for a multi-processor system is proven to be NP complete [16].

The rest of the paper is organized as follows: Section 2 gives a short summary of related work. Section 3 presents our contribution of parallelizing simulation code. In section 4 we give some results of our contribution, followed by a discussion and future work in section 5.

2 Related Work on Multiprocessor Scheduling

A large number of scheduling and partitioning algorithms have been presented in the literature. Some of them use a *list scheduling* technique and heuristics [1, 4, 5, 10, 12, 13], some have been designed specifically for simulation code [20]. A list scheduler keeps a list of tasks that are ready to be scheduled, i.e. all its predecessors have already been scheduled. In each step it selects one of the tasks in the list, by some heuristic, and assigns it to a suitable processor, and updates the list.

Another technique is called *critical path scheduling* [17]. The critical path of a task graph (DAG) is the path having the largest sum of communication and execution cost. The algorithm calculates the critical path, extracts it from the task graph and assign it to a processor. After this operation, a new critical path is found in the remaining task graph, which is then scheduled to the next processor, and so on. One property of critical path scheduling algorithms is that the number of available processors is assumed to be unbounded, because of the nature of the algorithm.

An orthogonal feature in scheduling algorithms is *task duplication* [11, 17, 21]. Task duplication scheduling algorithms rely on task duplication as a mean of reducing communication cost. However, the decision if a task should be duplicated or not introduces additional complexity to the algorithm, pushing the complexity up in the range $O(n^3)$ to $O(n^4)$ for task graphs with n nodes.

3 Scheduling of Simulation Code

Simulation code generated from Modelica mostly consist of a large number of assignments of expressions with arithmetic operations to variables. Some of the variables are needed by the DAE solver to calculate the next state, hence they must be sent to the processor running the solver. Other variables are merely temporary variables whose value can be discarded after the final use.

The simulation code is parsed, and a fine grained task graph is built. This graph, which has the properties of a DAG, can be very large. A typical application (a thermo-fluid model of a pipe, discretisized to 100 pieces), with an integration time of around 10 milliseconds, has a task graph with 30000 nodes. The size of each node can also vary a lot. For instance, when the simulation code originates from a DAE, an equation system has to be solved in each iteration if it can not be solved statically at compile time. This equation system can be linear or non-linear. In the linear case, any standard equation solver could be used, even parallel solvers. In the non-linear case, fixed point iteration is used. In both cases, the solving of the equation system is represented as a single node in the task graph. Such a node can have a large execution time in comparison to other nodes (like an addition or a multiplication of two scalar floating point values).

The task graph generated from the simulation code is not suitable for scheduling to multiprocessors, using standard scheduling algorithms found in literature. There are several reasons for this, the major reason is that the task graph is too fine grained and contains too many dependencies for getting good results on standard scheduling algorithms. Many scheduling algorithms are designed for coarse grained tasks. The granularity of a task graph is the relation between the communication cost between tasks and the execution cost of tasks. The large amount

of dependencies present in the task graphs also makes it necessary to allow task duplication in the scheduling algorithm. There are several scheduling algorithms that can handle fine grained tasks as well as coarse grained tasks. One such category of algorithms is non-linear clustering algorithms [18, 19]. A cluster is a set of nodes collected together to be executed on the same processor. Therefore all edges between two nodes that belong to the same cluster has a communication cost of zero. The non-linear clustering algorithms consider putting siblings[1] into the same cluster to reduce communication cost. But these algorithms does not allow task duplication. Therefore they are not producing well on this kind of simulation code.

A second problem with the task graphs generated is that in order to keep the task graph small, the implementation does not allow a task to contain several operations. For instance, a task can not contain both a multiplication and a function call. The simulation code can also contain Modelica `when` statements, which can be seen as a `if` statement without `else` branch. These need to be considered as one task, since if the condition of the `when` statement is true, all statements included in the `when` clause should be executed. An alternative would be to replicate the guard for each statement in the `when` clause. This is however not implemented yet, since usually the when statements are small in size and the need of splitting them up is low.

To solve the problems above, a second task graph is built, with references into the original task graph. The implementation of the second task graph makes it possible to cluster tasks into larger ones, thus increasing the granularity of the task graph. The first task graph is kept, since it is needed later for generating code. The two task graphs are illustrated in Figure 5.

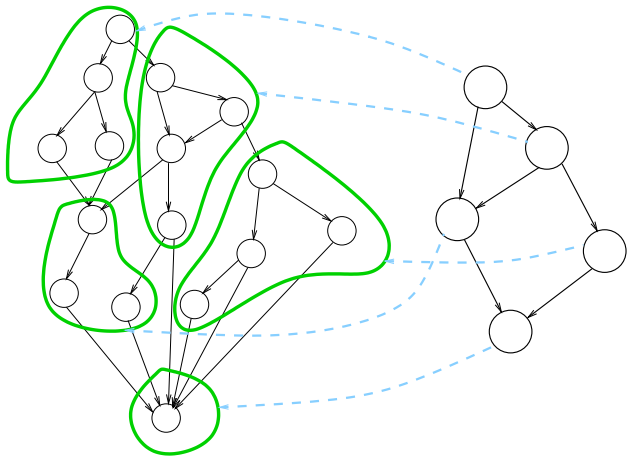

Fig. 5. The two task graphs built from the simulation code.

[1] A sibling *s*, to a task *n* is defined as a node where *n* and *s* has a common predecessor.

In this framework we have investigated several scheduling algorithms. Our early approaches found that the a task duplication algorithm called TDS [3] did not produce well. The main reason for this was the task granularity. The TDS algorithm can produce the optimal schedule if the task graph has certain properties, however fine grained task graphs as produced by our tool do not possess these properties.

We have also partially implemented a non-linear clustering algorithm called DSC [18], but measurements from this were not satisfying either. The clusters produced by the algorithm were too small, giving a parallel execution time much larger than the sequential execution time.

The combination of large fine grained task graphs and many dependencies makes it hard to find a low complexity scheduling algorithm that produce well. However, an approach that actually did produce speedup in some cases is a method we call *full task duplication*. The idea behind full task duplication is to prevent communication in a maximum degree, and instead of communicating values duplicate the tasks that produces the values. Figure 6 illustrates how the full task duplication algorithm works. For each node without successors, the complete tree of all its predecessors are collected into a cluster. Since all predecessors are collected no communication is necessary. The rationale of this approach is that, given a large fine grained task graph with many dependencies, it is cheapest to not communicate at all, but instead duplicate. The method also works better if the height of the task graph is low in relation to the number of the nodes, since the size of each cluster is dependent of the height of the task graph and the number of successors of each node.

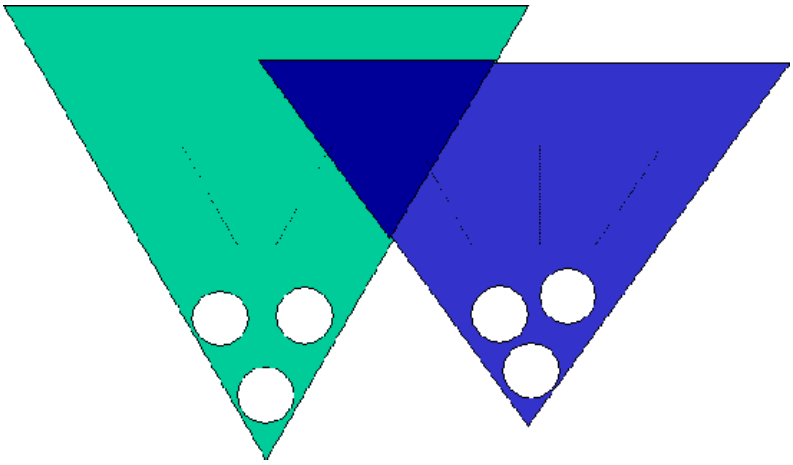

Fig. 6. By duplicating all predecessors, each cluster forms a tree

When all nodes without successors have been collected into clusters, a reduction phase is started to reduce the number of clusters until the number of clusters

matches the number of available processors. First, clusters are merged together as long as they do not exceed the size of the largest cluster. In this way, the clusters are load balanced since each cluster will be limited by the maximum cluster size. If the number of clusters is still larger than the number of available processors, the algorithm selects two clusters with the highest number of common nodes and merge them together. This is then repeated until the number of clusters matches the number of available processors.

4 Results

The early attempts of implementing standard scheduling algorithms did not produce speedup at all. Therefore, the full task duplication method was invented. The results we present here are theoretical results achieved by running the algorithm and measuring the parallel time of the program by calculating the size of the largest cluster. In the future, when we have fully implemented code generation with MPI calls, we will run the simulation code on different parallel architectures like Linux clusters and SMP (Shared Memory Processors) machines.

Figure 7 gives some theoretical results for a couple of different models. The *Pressurewave* examples are a thermo-fluid application where hot air is flowing through a pipe. The pipe is discretisized into 20 and 40 elements in the two examples. The *Robot* example is a multi-body robot with three mechanical joints with electrical motors attached to each joint. As shown in Figure 7, the results are better for the discretisized models than for the robot model. For the robot model, new scheduling methods are surely needed to get a speedup at all. The full task duplication algorithm could not produce better speedup than 1.27 for two processors.

The speedup figures are calculated as $speedup = P_{seq}/(P_{par} + CommCost)$ where P_{seq} is the sum of the execution cost of all nodes and P_{par} is the maximum cluster size. $CommCost$ is for simplicity reasons assumed to be zero, even if it is a large cost. Since these figures are only measurements on the scheduled task graph and not real timing measurements from a running application, we can make this simplification. The focus here is on explaining the complexity and special needs of scheduling simulation code from code generated from Modelica models.

Speedup	n=2	n=4	n=6	n=8
Pressurewave20	1.54	2.62	3.60	4.45
Pressurewave40	1.52	2.82	3.21	5.13
Robot	1.27	-	-	-

Fig. 7. Some theoretical speedup results for a few examples.

5 Discussion and Future Work

Simulation code generated from equation based simulation languages is highly optimized and very irregular code. Hence, it is not trivial to parallelize. The scheduling algorithms found in literature are not suited for fine grained task graphs of the magnitude produced by our tool. Therefore, new clustering techniques with task duplication are needed.
Due to the large task graphs, caused by the large simulation code files, the clustering algorithm must be of low complexity. The large number of dependencies in the task graphs also requires us to use task duplication to further decrease the parallel time of a task graph. Therefore, future work includes finding new scheduling algorithms that are suited for large fine grained task graphs with many dependencies and that use task duplication, and still have a low complexity.
The full task duplication algorithm can be further improved by cutting trees of at certain points and instead introduce communications. This will be further investigated in the near future to see if it is a fruitful approach to further reduce the parallel time of a simulation program.
Future work also includes parallelization of code using inline solvers and mixed mode integration [2]. This means that the system can be partitioned into parts, each with its own inline solvers, which reduces the dependencies between these parts. This will hopefully reveal more parallelism in the task graph, which will improve our results.

6 Acknowledgments

This work started with support from the Modelica Tools project in Nutek, Komplexa Tekniska System and continues in the EC/IST project RealSim.

References

1. A. Radulescu, A. J.C. van Gemund. FLB:Fast Load Balancing for Distributed-Memory Machines. Technical report, Faculty of Information Technology and Systems, Delft University of Technology, 1999.
2. A. Schiela, H. Olsson. Mixed-mode Integration for Real-time Simulation. In P. Fritzson, editor, *Proceedings of Modelica Workshop 2000*, pages 69–75.
3. S. Darbha, D. P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, vol. 9(no. 1):87–94, January 1998.
4. C. Hanen, A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. Technical report, Laboratoire Informatique Theorique Et Programmation, Institut Blaise Pascal, Universite P.et M. Curie, 1999.
5. C.Y. Lee, J.J. Hwang, Y.C. Chow, F.D Anger. Multiprocessor Scheduling with Interprocessor Communication Delays. *Operations Research Letters*, vol.7(no. 3), 1988.
6. *Dymola, http://www.dynasim.se*.
7. H. Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

8. P. Fritzson, V. Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *Proceedings of the 12th European conference on Object-Oriented Programming*, 1998.
9. P. Fritzson, L. Viklund, J. Herber, and D. Fritzson. High-level mathematical modeling and programming. *IEEE Software*, vol. 12(no. 4):77–87, July 1995.
10. G. Sih and E. Lee. Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4(no. 2), 1993.
11. G.L. Park, B. Shirazi, J. Marquis. DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems. In *Proceedings of Parallel Processing Symposium, 1997*.
12. J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *Journal on Computing*, vol. 18(vol. 2), 1989.
13. M. Y. Wu, D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, vol. 1(no. 3), 1990.
14. *MathModelica, http://www.mathcore.se*.
15. *The Modelica Language, http://www.modelica.org*.
16. R.L. Graham, L.E. Lawler, J.K. Lenstra and A.H. Kan. Optimization an Approximatioin in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, pages 287–326, 1979.
17. S. Darbha, D. P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *Transactions on Parallel and Distributed Systems*, vol. 9(no. 1), 1998.
18. T. Yang, A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *Transactions on Parallel and Distributed Systems*, vol. 5(no. 9), 1994.
19. V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.
20. B. E. Wells. A Hard Real-Time Static Task Allocation Methodology for Highly-Constrained Message-Passing Environments. *The International Journal of Computers and Their Applications*, II(3), December 1995.
21. Y-K. Kwok, I. Ahmad. Exploiting Duplication to Minimize the Execution Times of Parallel Programs on Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, vol. 9(no. 9), 1998.