A Task Merging Technique for Parallelization of Modelica Models

Peter Aronsson, Peter Fritzson
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden

Abstract

This paper presents improvements on techniques of merging tasks in task graphs generated in the ModPar automatic parallelization module of the OpenModelica compiler. Automatic parallelization is performed on Modelica models by building data dependency graphs called task graphs from the model equations. To handle large task graphs with fine granularity, i.e. low ratio of execution and communication cost, the tasks are merged. This is done by using a graph rewrite system(GRS), which is a set of graph transformation rules applied on the task graph. In this paper we have solved the confluence problem of the task merging system by giving priorities to the merge rules. A GRS is confluent if the application order of the graph transformations does not matter, i.e. the same result is gained regardless of application order.

We also present a Modelica model suited for automatic parallelization and show results on this using the ModPar module in the OpenModelica compiler.

1 Introduction

Parallel computers have been used in simulations for a long time. In fact, many of the large simulation applications are driving the parallel computer industry, like modeling and simulation of atomic explosions, or modeling and simulation for weather forecasting. These models are typically hand written for dedicated parallel computers. Modeling of such systems requires both knowledge of the modeling domain and knowledge in parallel programming. Thus, such models are mostly used by experts and the models tend to be used for a long period of time, since it is to expensive to change them.

In this paper we instead present techniques that enable a fully automatic approach to parallel simulation. We have developed an automatic parallelization tool for Modelica that can translate a Modelica model into a platform independent parallel simulation program. By having a fully automated process of producing the parallel simulation code, parallel simulation is opened up to a new set of users, with little or no knowledge of parallel programming or even parallel computers.

Our parallelization tool is plugged into the OpenModelica compiler developed at the Programming Environments Laboratory (PELAB) at Linköping University. Figure 1 presents an overview of the components of the OpenModelica compiler and the parallelization tool which is called ModPar. The OpenModelica compiler reads Modelica models and produces a set of variables, equations, algorithms, blocks, etc. This is fed into the ModPar module which performs a set of optimizations on the equations. First, simple algebraic equations on the form `a=b` are removed, which can substantially reduce the number of equations and variables of the system.

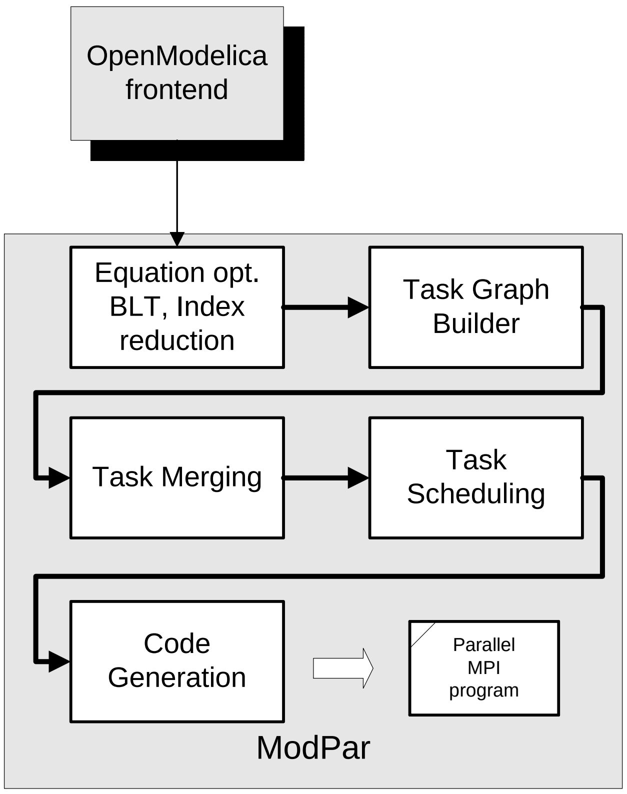

Figure 1. The ModPar Architecture.

The next optimization performed on the equations is the equation sorting. Equations are sorted in a Block Lower Triangular(BLT) form, resulting in a set of equation blocks, where each block consists of one or more equations that need to be solved simultaneously.

In conjunction with sorting the equations, index reduction using dummy derivatives is applied[6]. Index reduction is used on high index systems of equations, where some equations need to be differentiated in order to solve the system. The *index* of a system corresponds to how many times some equations needs to be differentiated before the set of equations can be transformed into an ODE (also called the underlying ODE).

A task graph is built, based on the sorted BLT form. A task graph is a Directed Acyclic Graph (DAG), with costs associated with edges and nodes. It is described by the tuple $G = (V, E, c, \tau)$ where

- V is a set of vertices (nodes), i.e. tasks in the task graph. A task is generated for each sub expression in the model equations. For instance, an addition between two scalar values `(a+b)` or a function call `(sin(x))` constitutes a task. In this paper tasks and nodes are used with the same meaning.
- E is a set of edges, which imposes a precedence constraint on the tasks. An edge $e = (v_1, v_2)$ indicates that node v_1 must be executed before v_2 and send data (resulting from the execution of v_1) to v_2.
- $c(e)$ gives the communication cost of sending the data along an edge $e \in E$.
- $\tau(v)$ gives the execution cost for each node $v \in V$.

The immediate predecessors (or parents) of a node **n** are all nodes having an edge leading to the node **n**. They are denoted by **pred(n)**. The immediate successors (or children) of a node **n** are all nodes having an edge leading to it from node **n**. They are denoted by **succ(n).** Similarly the predecessors of a node **n** is the transitive closure of **pred(n)**, i.e. the set of all tasks having a path leading to the node **n.** Analogously, the successors of a node **n** are all the tasks having a path leading to them from the node n. These sets are denoted $\text{pred}^m(n)$ and $\text{succ}^m(n)$ respectively.

Blocks containing more than one equation need to be solved before the task graph can be built. Such a block can either be a linear system of equations or a non-linear system of equations. For certain blocks the solution cannot be found at compile time and thus a numerical solver is integrated in the task graph itself. For example, the solution of a linear system of equations can be done in parallel, making the corresponding task possible to execute on more than one processor. Such tasks are referred to as *malleable tasks.*

The next step in the ModPar tool is to perform task merging and task clustering. Task clustering performs a mapping of tasks to virtual processors by forming clusters of tasks. This means that tasks that belong to the same cluster have a communication cost of zero, while tasks between clusters still have their original communication cost. Task merging differs from task clustering in the sense that tasks of the task graph are collapsed into a single node that represents the complete computational work of the included tasks. The data packets sent to and from the merged task are also combined. The goal of a task-merging algorithm is to increase the granularity, i.e., the relation between communication and execution cost of the task graph. This paper presents improvements on a task-merging algorithm based on earlier work in [1].

The result from the Task Merging algorithm is a new task graph with a smaller number of tasks (with larger execution costs). This is fed into the task-scheduling algorithm that maps the task graph onto a fixed number of processors. Each task in the task graph is assigned a processor(s) and starting time(s).

The final stage in the ModPar module is code generation. The ModPar outputs simulation code with MPI (Message Passing Interface) calls[7] to send and receive code between processors. Processor zero runs the numerical solver. In each integration step, work is distributed to other slave processors, which then calculate parts of the equations and send the result back to processor zero. Model parameters are only read once from file and distributed to all processors at the start of the simulation.

The rest of the paper is organized as follows. Section 2 introduces the method of merging tasks using a graph rewrite system formalism. Section 3 presents a Modelica application example suitable for parallelization, followed by results in section four. Section 5 presents the conclusions of the work and section six shows how the work relates to other contributions.

2 Task Merging using Graph Rewrite Systems

In previous work we have proposed a task-merging algorithm based on a graph rewrite system (GRS). A

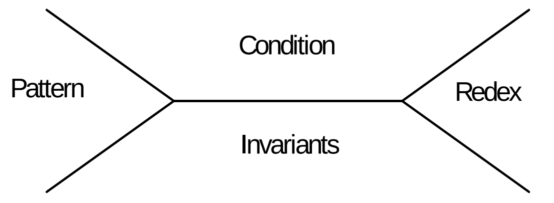

Figure 2. The X notation for GRS.

GRS is a set of graph transformation rules with a pattern, a condition, and a resulting sub-graph (*called redex*). We use a graphical notation (called the X-notation) depicted in Figure 2.

A GRS applies the transformation rules on the graph until there are no more matching patterns found in the graph. When this happens the GRS *terminates*. The termination of a GRS is an important property both theoretically and in practice. If it is not terminating, the GRS must be interrupted somehow in a practical implementation.

Our task merging rewrite rules are based on the condition that the *top level* of a task should not increase. The top level of a task is defined as the longest path from the task to a task without any ingoing edges, accumulating execution cost and communication cost along the path. The communication costs are described using two parameters, the bandwidth B and the latency, L. The communication cost of sending n bytes becomes $\boldsymbol{n / B + L}$. The transformation rules, first presented in [2] are given below.

1. The first and simplest rewrite rule is given in Figure 3. It merges a parent task that has only one child with the child. This can always be performed, i.e., without any condition, since such transformation will not reduce the level of parallelism in the task graph.

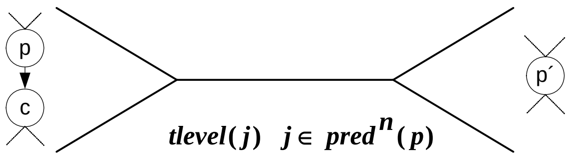

Figure 3. Merging of single children rule, called singlechildmerge.

2. The second rule handles *join nodes*, i.e., a task that has several incoming messages from a set of parent tasks, see Figure 4. The condition for this rule to apply is that the top level of task c does not increase when the transformation is performed. However, it is also necessary to make sure that *other successors* of the parents of the join node (p_{ij}) are not increasing their top levels. The rule therefore divides the parents into two disjoint sets, one that has successors fulfilling the condition and one that has successors increasing their top level by the merge and therefore not fulfilling the condition. The parents not fulfilling the condition are therefore not merged into the join task, c.

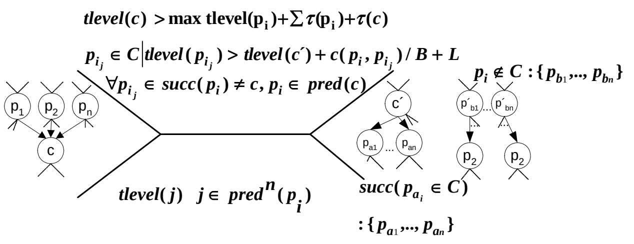
$tlevel(c) > \max tlevel(p_i) + \sum \tau(p_i) + \tau(c)$
$p_{i_j} \in C \mid tlevel(p_{i_j}) > tlevel(c') + c(p_i, p_{i_j}) / B + L$
$\forall p_{i_j} \in succ(p_i) \neq c, p_i \in pred(c)$
p_1
p_2
p_n
c
$tlevel(j) \quad j \in pred^{n}(p_i)$
c´
p_{a1}
p_{an}
$succ(p_{a_i} \in C)$
$: \{p_{a1}, .., p_{an}\}$
$p_i \notin C : \{p_{b1}, .., p_{bn}\}$
p'_{b1}
p'_{bn}
p_2
p_2

Figure 4. Rule of Merging of all parents to a task, called mergeallparents.

3. The third and final rewrite rule deals with *split nodes.* A split node is a node with several successors, or children. The transformation will replicate the split task and merge it with each individual successor task, c_i. However, the successor tasks can also have *other predecessors* for which the top level cannot be allowed to increase. Therefore, analogously as for the join task rewrite rule we also divide the successor tasks into two disjoint sets. The successor tasks that have other predecessors not increasing the top level are put in the set C. Thus, predecessors belonging to C are replicated and merged with the task c, while predecessors not belonging to C are left as they are.

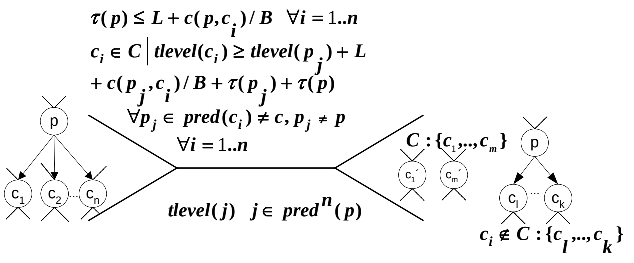
$\tau(p) \leq L + c(p, c_i) / B \quad \forall i = 1..n$
$c_i \in C \mid tlevel(c_i) \geq tlevel(p_j) + L$
$+ c(p_j, c_i) / B + \tau(p_j) + \tau(p)$
$\forall p_j \in pred(c_i) \neq c, p_j \neq p$
$\forall i = 1..n$
p
c_1
c_2
c_n
$tlevel(j) \quad j \in pred^{n}(p)$
$C : \{c_1, .., c_m\}$
c_1'
c_m'
p
c_l
c_k
$c_i \notin C : \{c_l, .., c_k\}$

Figure 5. Replicating a parent and merging into each child task, called replicateparentmerge.

An unanswered question so far has been if the GRS is *confluent* or not. A confluent GRS gives the same resulting graph independently of the order of the applied rules. In earlier work, we investigated empirically whether the GRS was confluent, but now we have found a counter example that the rewrite rules are not confluent as they appear above. There are several alternatives to try to remedy this fact:

1. One could limit the order of matching of the patterns on the task graph. An idea of this is for instance to traverse the graph once in a top down fashion to prevent the confluence problem to occur. It is however not clear if this would work or not, without a more thorough investigation.
2. Another alternative is to instead use the simpler rewrite rules first presented in [2]. This

approach might be taken for specific types of graphs, e.g. trees or forests, but in the general case, this is not sufficient. The simple rules did not succeed so well in reducing fine grained tasks graphs as produced by the task graph builder in ModPar.

3. A third, and the best practical alternative, is to give priorities to the rewrite rules. This means that a rewrite rule with a higher priority is always applied before other rules with lower priority. This will effectively prevent the GRS from being non-confluent, since only applications of transformations in priority order is allowed.

The priority order solution to the confluence problem was chosen in ModPar. The chosen priority is:

1. singlechildmerge
2. replicateparentmerge
3. mergeallparents

This means that the *singlechildmerge* rule has the highest priority and is always applied first. This rule is also the cheapest to apply since it does not have any condition, only a sub-graph pattern. Therefore, it makes sense to apply it with highest priority.

The second highest priority has the *replicateparentmerge* rule, thus giving the *mergeallparents* rule the lowest priority. The order between the last two rules is chosen so that rules limiting the amount of parallelism of the task graph are given lower priority. Since *mergeallparents* merges independent tasks (the successor of the parent), it reduces the amount of parallelism, which *replicateparentmerge* does not. Therefore, this order is chosen.

3 Application example

Lets consider a simple application example that can easily be scaled up using the *array of components* feature in Modelica. It uses the Modelica standard library and the one-dimensional `Rotational` package to create a flexible shaft. The shaft element is implemented as:

```
model ShaftElement "Element of a flexible
                    one dimensional shaft"
import Modelica.Mechanics.Rotational.*;[1]
   extends Interfaces.TwoFlanges;
   Inertia load;
   SpringDamper spring(c=500,d=5);
equation
   connect(load.flange_b,
            spring.flange_a);
   connect(load.flange_a,flange_a);
   connect(spring.flange_b,flange_b);
end ShaftElement;
```

[1] Unqualified imports are not recommended to use. They are used here for space considerations.

The `ShaftElement` model describes a one-dimensional shaft element with a spring and a damper. By instantiating this component as an array and connecting each array component to the next, we get a simple model of a flexible shaft.

```
model FlexibleShaft "model of a flexible
shaft"
import Modelica.Mechanics.Rotational.*;
  extends Interfaces.TwoFlanges;
  parameter Integer n(min=1) = 20 "number
of shaft elements";
  ShaftElement shaft[n];
equation
  for i in 2:n loop
    connect(shaft[I-1].flange_b,
            shaft[i].flange_a);
  end for;
  connect(shaft[1].flange_a, flange_a);
  connect(shaft[n].flange_b, flange_b);
end FlexibleShaft;
```

Finally, we create a test model to test our shaft.

```
model ShaftTest
  FlexibleShaft shaft(n=20);
  Modelica.Mechanics.Rotational.Torque
src;
  Modelica.Blocks.Sources.Step c;
equation
  connect(shaft.flange_a, src.flange_b);
  connect(c.outPort, src.inPort);
end ShaftTest;
```

The structural parameter **n** controls the number of element pieces of the model, i.e., the number of discretization points of the model. It is therefore directly proportional to the number of variables and equations of the model. Due to its simplicity and structure, it is suitable for parallelization.

4 Results

The confluence problem is successfully solved in this paper by introducing priorities on the task merging rules. These priorities makes the task merging GRS confluent, according to measurements made on a large set of random task graphs from the Standard Task Graph Set (STG)[10], as well as task graphs generated from the ModPar module.

The application example in section 3 can substantially be reduced in size but still reveal sufficient parallelism. When running the task-merging algorithm on the task graph produced from the example, it results in a set of independent tasks, which can then be allocated to a set of processors in a simple load balancing manner, i.e., evenly distributing them among the processors. Thus, for this example, no scheduling is even required. This reduction is possible since the graph rewrite rules allow replication of tasks, such that dependencies between tasks of the task graph are completely removed.

Table 1 shows the increase of granularity[2] when applying the task merging for another Modelica example from the Thermofluid package. With realistic figures on bandwidth (B) and Latency (L), we see a substantial increase of granularity.

Model	*Granularity before merge*	*Granularity after merge*
PressureWave (B=1, L=100)	0.000990	0.106
PressureWave (B=1, L=1000)	0.0000990	0.0562

Table 1. **Granularity before and after Task Merging.**

The status of the parallelization tool is that we can generate C code with MPI calls for execution of parallel machines, such as the Linux cluster *monolith* at NSC (Swedish National Supercomputer Center). We have successfully executed smaller examples on this cluster computer but without obtaining any speedups. The application example in Section 3 can only be translated in reasonable time with about 9000 equations (using 1000 discretization points), which is a bit too small for obtaining sufficient speedups. In order to handle larger system of equations, the equation optimization and other parts of the compiler must be implemented in a more efficient way. In addition, the amount of work per state variable in the Flexible Shaft example is not so large, so in order to get better speedups, other applications must be considered.

5 Conclusions

We have proposed improvements on earlier work of merging tasks in a task graph using a graph rewrite system formalism. Earlier improvements made the task merging GRS non-confluent, thus giving different results depending of order of application. We proposed several alternative solutions to make the GRS confluent and have chosen and implemented the best-suited solution for our application area, parallelization of simulation code from Modelica models.

The task merging technique is implemented in the ModPar module, a part of the OpenModelica compiler. It successfully reduces the number of tasks of task graphs built from Modelica simulation code to a suitable degree such that existing scheduling algorithms can succeed in producing parallel programs that give sufficient speedup.

6 Related Work

There is much work on scheduling of task graphs for multi-processors, like the DSC[11] algorithm, the TDS[4] algorithm or the Internalization algorithm[9], all working on unlimited number of processors, so called clustering algorithms. They all treat each task in the task graph as a non-preemptive atomic task, and do not consider merging of tasks. Therefore, they do not work well on very fine-grained task graphs.

There are other attempts to merge tasks, like the grain-packing algorithm[5]. The difference between this approach and ours is that our approach is iterative by nature and allows task replication.

Related work on parallelization of simulation code includes distributed simulation where the numerical solver is split into several parts, each handling a subset of the equations. The interaction between the subsystems is then delayed in time such that the subsystems becomes independent of each other in each time step. This division of the model equations into subsystems is implemented using a transmission line component in the system, giving the technique the name Transmission Line Modeling (TLM)[3].

Other related work on parallel simulation includes using parallel solvers, where the numerical solvers themselves are parallelized, like for instance Runge Kutta based solvers[8].

Acknowledgements

This work has been supported by the Swedish Foundation for Strategic Research (SSF), in the VISIMOD project and in the ECSEL graduate school, and by Vinnova in the GridModelica project.

References

[1] P. Aronsson, P. Fritzson, Automatic Parallelization in OpenModelica, Proceedings of 5th EUROSIM Congress on Modeling and Simulation, Paris, France, 6-10 Sep 2004. ISBN 3-901608-28-1

[2] P. Aronsson, P. Fritzson, Task Merging and Replication using Graph Rewriting, Tenth International Workshop on Compilers for Parallel Computers, Amsterdam, the Netherlands, Jan 8-10, 2003

[2] The relation between communication and execution cost of the task graph.

[3] Casella F. Maffezzoni C., The Transmission Line Modeling Method, EEE/OUP Series on Electromagnetic Wave Theory, 1995

[4] 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.

[5] B. Kruatrachue. Static Task Scheduling and Grain Packing in Parallel Processor Systems. PhD thesis, Dept. of Electrical and Computer Engineering, Oregon State University, 1987.

[6] S.E. Mattsson, G. Söderlind, Index reduction in differential-algebraic equations using dummy derivative, Scientific Computing Vol. 14 , Issue 3 1993

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[8] T. Rauber, G. Runger, Iterated Runge-Kutta Methods on Distributed Memory Multiprocessors. In Proceedings of First Aizu International Symposium on Parallel and Distributed Processing, pages 12-19. 1995.

[9] V. Sarkar. Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press, Cambridge, MA, 1989.

[10] Standard Task Graph Set (STG), http://www.kasahara.elec.waseda.ac.jp/schedule/, accessed 2004-12-02.

[11] 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.