High Level Mathematical Modeling and Parallel/GRID Computing with Modelica

Peter Fritzson
PELAB – Programming Environment Laboratory
Department of Computer and Information Science, PELAB
Linköping University, SE 581-32 Linköping, Sweden
{petfr}@ida.liu.se

Outline

• Complex Systems and High Performance Simulations
• Introduction to Modelica
• Overview of OpenModelica Environment
• Automatic fine-grained parallelization
• Explicit model-based parallelization
• Explicit parallel programming
Examples of Complex Systems

• Robotics
• Automotive
• Aircraft
• Living organisms
• Power plants
• Heavy Vehicles

Objectives

• Convenient high level mathematical modeling and simulation of complex systems
• High performance computation using parallel computers
Towards High-Level Parallel Modeling and Simulation

- Simulations are time-consuming
- Moore’s “Law”: (since 1965)
  - #devices per chip area doubles every 18 months
  - CPU clock rate also doubled every 18 months – until 2003, then: heat and power issues, limited ILP, ...
  \[ \rightarrow \text{superscalar technology has reached its limits, only (thread-level) parallelism can increase throughput substantially} \]
- The consequence:
  - Chip multiprocessors (+ clusters)
  - Multi-core, PIM, ... (for general-purpose computing)
- Need parallel programming/modeling
  - Automatic parallelization
  - Explicit parallel modeling and parallel programming

Three Approaches to Parallel Computation in Mathematical Modeling with Modelica

- Automatic fine-grained parallelization of mathematical models (ModPar)
- Coarse-Grained Explicit Parallelization Using Components (GRIDMOdelica)
- Explicit Parallel Programming Constructs in Modelica (NestStepModelica)
Background

Modelica – the Next Generation Modeling Language

Stored Scientific and Engineering Knowledge

Model knowledge is stored in books and human minds which computers cannot access

“The change of motion is proportional to the motive force impressed“
– Newton
The Form – Equations

- Equations were used in the third millennium B.C.
- Equality sign was introduced by Robert Recorde in 1557

\[ 14 \times \frac{2}{3} - 15 \times \frac{9}{8} = \frac{3}{4} - \frac{1}{6} \]

Newton still wrote text (Principia, vol. 1, 1686)

“The change of motion is proportional to the motive force impressed”

CSSL (1967) introduced a special form of “equation”:

- variable = expression
- \( v = \text{INTEG}(F)/m \)

Programming languages usually do not allow equations!

Modelica – The Next Generation Modeling Language

Declarative language

Equations and mathematical functions allow acausal modeling, high level specification, increased correctness

Multi-domain modeling

Combine electrical, mechanical, thermodynamic, hydraulic, biological, control, event, real-time, etc...

Everything is a class

Strongly typed object-oriented language with a general class concept, Java & MATLAB-like syntax

Visual component programming

Hierarchical system architecture capabilities

Efficient, non-proprietary

Efficiency comparable to C; advanced equation compilation, e.g. 300 000 equations, ~150 000 lines on standard PC
Modelica Language Properties

- **Declarative** and **Object-Oriented**
- **Equation-based**: continuous and discrete equations
- **Parallel** process modeling of real-time applications, according to synchronous data flow principle
- **Functions** with algorithms without global side-effects (but local data updates allowed)
- **Type system** inspired by Abadi/Cardelli
- **Everything is a class** – Real, Integer, models, functions, packages, parameterized classes....

Object Oriented Mathematical Modeling with Modelica

- The static *declarative structure* of a mathematical model is emphasized
- OO is primarily used as a *structuring concept*
- OO is *not* viewed as dynamic object creation and sending messages
- *Dynamic model* properties are expressed in a *declarative way* through equations.
- Acausal classes supports *better reuse of modeling and design knowledge* than traditional classes
Brief Modelica History

• First Modelica design group meeting in fall 1996
  • International group of people with expert knowledge in both language design and physical modeling
  • Industry and academia

• Modelica Versions
  • 1.0 released September 1997
  • 2.0 released March 2002
  • Latest version, 2.2 released March 2005

• Modelica Association established 2000
  • Open, non-profit organization

Modelica Conferences

• The 1st International Modelica conference October, 2000

• The 2nd International Modelica conference March 18-19, 2002

• The 3rd International Modelica conference November 5-6, 2003 in Linköping, Sweden

• The 4th International Modelica conference March 6-7, 2005 in Hamburg, Germany

• The 5th International Modelica conference September 4-5, 2006 in Vienna, Austria
Modelica Class Libraries - for Reuse
Visual View

![Modelica Class Libraries Diagram](image)

Modelica Model Example – Industry Robot

![Modelica Model Example Diagram](image)

Courtesy of ABB Corp. Research and of Martin Otter, DLR
Modelica Model Example
GTX Gas Turbine Power Cutoff Mechanism

Courtesy of Siemens Industrial Turbomachinery AB

Graphical Modeling /Visual Programming View
Multi-Domain (Electro-Mechanical) Modelica DCMotor Model

A DC motor can be thought of as an electrical circuit which also contains an electromechanical component.

```model DCMotor
Resistor R(R=100);
Inductor L(L=100);
VsourcenDC DC(f=10);
Ground G;
ElectroMechanicalElement EM(k=10,J=10, b=2);
Inertia load;
equation
  connect(DC.p,R.n);
  connect(R.p,L.n);
  connect(L.p, EM.n);
  connect(EM.p, DC.n);
  connect(DC.n,G.p);
  connect(EM.flange,load.flange);
end DCMotor
```

A DC motor can be thought of as an electrical circuit which also contains an electromechanical component.

Corresponding DCMotor Model Equations

The following equations are automatically derived from the Modelica model:

\[
\begin{align*}
0 &= DC.p.i + R.n.i \\
DC.p.v &= R.n.v \\
0 &= R.p.i + L.n.i \\
R.p.v &= L.n.v \\
0 &= L.p.i + EM.n.i \\
L.p.v &= EM.n.v \\
0 &= EM.p.i + DC.n.i \\
EM.p.v &= DC.n.v \\
0 &= DC.n.i + G.p.i \\
DC.n.v &= G.p.v
\end{align*}
\]

(load component not included)

Automatic transformation to ODE or DAE for simulation:

\[
\frac{dx}{dt} = f(x, u, t) \\
g\left(\frac{dx}{dt}, x, u, t\right) = 0
\]
The Translation Process

Modelica Model

Modelica Graphical Editor

Modelica Textual Editor

Modelica Model

Modelica Source code

Translator

Flat model

Analyzer

Sorted equations

Optimizer

Sorted equations

Code generator

Optimized sorted equations

C Compiler

C Code

Executable

Simulation

Modelica Acausal Modeling

- What is *acausal* modeling/design?
- Why does it increase *reuse*?

The acausality makes Modelica library classes *more reusable* than traditional classes containing assignment statements where the input-output causality is fixed.

- Example: a resistor equation:
  \[ R \cdot i = v; \]
  
  can be used in three ways:
  
  \[ i := v/R; \]
  \[ v := R \cdot i; \]
  \[ R := v/i; \]
Visual Model Design Using Connector Classes, Components and Connections

Connector Pin
Voltage v;
flow Current i;
End Pin;

Keyword flow indicates that currents of connected pins sums to zero.

A connect statement in Modelica
connect(Pin1,Pin2)
corresponds to

Pin1.v = Pin2.v
Pin1.i + Pin2.i = 0

Connection between Pin1 and Pin2

Common Component Structure as SuperClass

model TwoPin
"Superclass of elements with two electrical pins"
Pin p,n;
Voltage v;
Current i;
equation
v = p.v - n.v;
0 = p.i + n.i;
i = p.i;
end TwoPin;

Electrical Components Reuse TwoPin SuperClass

```model Resistor "Ideal electrical resistor"
extends TwoPin;
  parameter Real R "Resistance";
equation
  R*i = u
end Resistor;
```

```model Inductor "Ideal electrical inductor"
extends TwoPin;
  parameter Real L "Inductance";
equation
  L*der(i) = u
end Inductor;
```

Discrete-time vs. Continuous-time

- Continuous-time variables: change at any point in time
- Discrete-time variables only change at certain points in time
Modelica Discrete and Hybrid Properties

- Discrete event semantics based on *conditional equations* (if conditions ..., *when* conditions ..., etc.)
- Discrete event model follows *synchronous data flow* principle – events take no time
- Efficient *hybrid* modeling and simulation possible
- Can handle most *discrete formalisms*: FSA, DEV$S$, Petri Nets, State Charts, Queuing models, ...

Recent Book, 2004

Peter Fritzson
*Principles of Object Oriented Modeling and Simulation with Modelica 2.1*

Wiley-IEEE Press

940 pages

Book web page: [www.mathcore.com/drmodelica](http://www.mathcore.com/drmodelica)
The goals of the OpenModelica project is to:

- Create a complete Modelica modeling, compilation and simulation environment.
- Provide free software distributed in binary and source code form.
- Provide a modeling and simulation environment for research, teaching, and industrial purposes.
- Reference implementation of Modelica in Modelica

http://www.ida.liu.se/projects/OpenModelica
OpenModelica End-Users vs. Developers

- OpenModelica End-Users
  - People who use OpenModelica for modeling and simulation

- OpenModelica Developers
  - People who develop/contribute to parts in the OpenModelica environment including the OpenModelica compiler

OpenModelica Environment Architecture

- Interactive session handler
- Modelica Compiler
- Modelica Debugger
- Execution
- DrModelica OMNoteBook Model Editor
- Emacs Editor/Browser
- Graphical Model Editor/Browser
- Textual Model Editor
- Eclipse Plugin Editor/Browser
OpenModelica Client-Server Architecture
Callable via CORBA API

Interactive Session Handler – on dcmotor Example
(Session handler called OMShell – OpenModelica Shell)

```plaintext
>> simulate(dcmotor, startTime=0.0, stopTime=10.0)
>> plot([{load.w, load.phi}])
```

```plaintext
model dcmotor
  Modelica.Electrical.Analog.Basic.Resistor r1(R=10);
  Modelica.Electrical.Analog.Basic.Inductor i1;
  equation
    connect(v.p, r1.p);
    connect(v.n, g.p);
    connect(r1.n, i1.p);
    connect(i1.n, emf1.p);
    connect(emf1.n, g.p);
    connect(emf1.flange_b, load.flange_a);
end dcmotor;
```
## Event Handling by OpenModelica – BouncingBall

```model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;

  equation
    impact ≡ h ≤ 0.0;
    der(v) ≡ if flying then -g else 0;
    der(h) ≡ v;
    when {h ≤ 0.0 and v ≤ 0.0, impact} then
      v_new ≡ if edge(impact) then -e*pre(v) else 0;
      flying ≡ v_new > 0;
    reinit(v, v_new);
end when;
end BouncingBall;
```

## OpenModelica Eclipse Plugin in Action – Browsing and Building OpenModelica Compiler

- Browsing of Modelica/MetaModelica packages, classes, functions
- Automatic building of executables
- Separate compilation
- Syntax highlighting
- Code completion
- Code query support for developers
- Automatic indentation
Graphic Modelica Model Editor
(From MathCore; runs on Windows, Linux)

- Runs together with OpenModelica
- Free for university usage

OpenModelica OMNotebook Electronic Notebook with DrModelica

- Primarily for teaching
- Interactive electronic book
- Platform independent
- OMNotebook Does not need Mathematica
Interactive Contents in DrModelica Contains Examples and Exercises from Modelica Book

Cells with both Text and Graphics

First Basic Class

1. HelloWorld
   The program contains a declaration of a class called HelloWorld with two fields and one variable. The variable is a number, which is initialized in the constructor. The variable is used to simulate a state in a simulation. The method is called when the simulation starts, and it assigns the initial value to the variable.

   ```modelica
   class HelloWorld
       attribute integer intVal = 10; // Initial value for intVal
   end HelloWorld
   ``

2. Simulation of HelloWorld
   The method simulate is called after the constructor is called, and it simulates the state change.

   ```modelica
   function simulate(): Real
       // Simulate the state change
       return intVal += 1; // Increase intVal by 1
   end simulate;
   ```

Exercise 1

Using Algorithm Sections

Write a function `is_even`, which checks if a number is even by comparing it to 2 and returning true if it is.

```modelica
function is_even(n: Integer): Boolean
    return n % 2 == 0;
end is_even;
```
OpenModelica MetaModelica Compiler

- Supports extended subset of Modelica
- Used for development of OpenModelica compiler in Modelica
- Some MetaModelica Language properties:
  - Modelica syntax and base semantics
  - Pattern matching (named/positional)
  - Local equations (local within expression)
  - Recursive tree data structures
  - Lists and tuples
  - Garbage collection of heap-allocated data
  - Arrays (with local update as in standard Modelica)
  - Polymorphic functions
  - Function parameters to functions
  - Simple builtin exception (failure) handling mechanism

Parallelism in Modelica
Integrating Parallelism and Mathematical Models
Three Approaches

• **Automatic Parallelization of Mathematical Models (ModPar)**
  - Parallelism over the method.
  - Parallelism over time.
  - Parallelism over the model equation system
    - ... with fine-grained task scheduling
    - [Peter Aronsson’s PhD thesis June 14, 2006]

• **Coarse-Grained Explicit Parallelization Using Components**
  - Programmer structures the application into computational components using strongly-typed communication interfaces.
  - Co-Simulation, Transmission-Line Modeling (TLM)
  - Our GridModelica project

• **Explicit Parallel Programming**
  - Providing general, easy-to-use explicit parallel programming constructs within the algorithmic part of the modeling language.
  - NestStepModelica

**Automatic Parallelization of Mathematical Models (ModPar) (with Peter Aronsson)**

• Parallelism over the method.
• Parallelism over time.
• Parallelism over the model equation system
  - ... with fine-grained task scheduling
  - [Peter Aronsson’s PhD thesis June 14, 2006]
Modelica Simulations

- Simulation = solution of (hybrid) DAEs from models
  \[ g(\dot{X}, X, Y, t) = 0 \]
  \[ h(X, Y, t) = 0 \]
- In each step of numerical solver:
  - Calculate \( X \) in \( g \) (and \( Y \) in \( h \))
- Parallelization approach: perform the calculation of \( \dot{X} \) in parallel
  - Called parallelization over the system.

Automatic Generation of Parallel Code from Modelica Equation-Based Models

Clustered Task Graph

Thermofluid Pipe Application

Speedup

\[ 1 \quad 2 \quad 4 \quad 8 \quad 16 \quad \text{# Proc} \]
Multiprocessor Scheduling Problem

- **Given**
  - Task Graph \( G = (V,E, \tau, c) \)
  - A fixed number of processors \( P_1, \ldots, P_N \)
- **Find for each task**
  - A *processor assignment* (or several) and a *starting time* such that
  - Overall execution time is minimized

- **Problem**: Known scheduling algorithms perform bad on very fine grained task graphs.
- **Solution**: Increase granularity by merging tasks

Clustering v.s. Merging

Clustered Task Graph

Merged Task Graph
Fine-Grained Automatic Parallelization

Summary

• A task merging algorithm using graph rewrite system has been proposed and used
  • Improved patterns to increase applicability
• Can easily be integrated in existing scheduling tools.
• Successfully merges tasks considering
  • Bandwidth & Latency
  • Task duplication
  • Merging criterion: decrease Parallel Time, by decreasing tlevel (PT)
• Tested on examples from simulation code
• Speedup (on current examples) e.g. up to 4.5

GridModelica – Coarse-Grained Component-Level Parallel Simulation (with Kaj Nyström)

Very large system of equations with computational models from several domains:
• Mechanical domain
• Electrical domain
• Chemical domain
Outline of a Partitioned Model

SubSystem 1
Solver: Dassl
Stepsize: 0.1

SubSystem 2
Solver: Lsode2
Stepsize: 0.01

SubSystem 3
Solver: Euler
Stepsize: 0.001

SubSystem 4
Solver: LAPACK
Stepsize: 1.0

Transmission Line Modeling

• Connections in Modelica are ideal and information exchange takes no time
  • In the real world, this is not the case.

• The Idea: Let us use this time delay to decouple a model into smaller pieces

• Instead of one large equation system, we get two (or more). The solution to these systems is dependent only of the previous solutions to neighbouring systems.

• Result: Many smaller systems which can be solved independently
Transmission Line Modeling (2)

\[ c_1(t) = v_1(t) - T_{\text{lim}} + Z_f i_2(t - T_{\text{lim}}) \]
\[ c_2(t) = v_1(t) - T_{\text{lim}} + Z_f i_1(t - T_{\text{lim}}) \]
\[ P_1(t) = Z_f i_1(t) + c_1(t) \]
\[ P_2(t) = Z_f i_2(t) + c_2(t) \]

c_1, c_2 are the TLM-parameters
Tlim is the information propagation time
Zf is the implicit impedance

Advantages

- Partitioning at model level using drag and drop
- No knowledge in parallelization techniques needed
- Reduces model stiffness
- Domain independent (the Modelica way!)
- Separate solvers and settings possible
- ... and of course, better performance
Small Example Application

NestStepModelica – Explicit Parallel Programming in Modelica (w. Christoph Kessler and Mattias Eriksson)

- Introduce simple explicit parallel programming model in Modelica algorithmic code
- BSP – Bulk Synchronous Parallel Programming (master-slave)
- NestStep: BSP + Nested parallelism
- NestStepModelica: Modelica + NestStep runtime
**BSP – Bulk Synchronous Parallelism**

**BSP Model** (Bulk-Synchronous Parallelism) [Valiant 1990]

- **BSP computer:**
  - Abstract MIMD parallel computer \((p, L, g)\)
  - Group of \(p\) processors / threads (SPMD)
  - Message passing
  - Barrier synchronisation, overhead: \(L\)
  - Communication data rate: \(g\)

- **BSP program:**
  - Sequence of supersteps
  - \(t(\text{prog}) = \sum t(\text{step})\)

- **Superstep:**
  - Max. Computation time per processor: \(w\)
  - Max. Communication volume per proc.: \(h\)
  - \(t(\text{step}) = w + h g + L\)

**NestStep**

- **BSP**
- + shared variables / arrays
- + nested parallelism

- Set of extensions to existing languages
  - NestStep-Java [K.'99]
  - NestStep-C [K.'00, K.'04]
  - NestStep-Modelica
NestStep-Modelica: step statement

Shared Variables
- Integer x (mem = "shared");
  
  step();
  ...
  ... = ... x ...
  ...
  x = ...
  endstep();

Invariants:
- Superstep synchronicity:
  - All processors of an (active) group work on the same superstep at the same time.
- Superstep memory consistency:
  - At entry and exit from a superstep, all copies of a shared variable have the same value on all processors of the group.
  - Within a superstep, only its local value is visible.

Implementation Issues
- Spanning tree over all processors of the group
- 2 Phases:
  - "combine" – Communication upwards
  - "commit" – Communication downwards
- Barrier synchronisation included ;-)
Example: Parallel Prefix Sums

\[
\text{prefix}_i = \sum_{j=1}^{i} a_j \\
\text{for } i = 1, \ldots, N
\]

Example: \( a = \{ 1, 3, 1, 4 \} \)
\( \rightarrow \) prefix = \( \{ 0, 1, 4, 5 \} \), sum = 9

// Sequential in linear time:
sum := 0;
for \( i \) in 1:N loop
  prefix[\( i \)] := sum;
  sum := sum + a[\( i \)];
end for;

Parallel Prefix Sums in NestStepModelica (1)

function parPrefix "Compute prefix sums in parallel"
input Integer[: ] arr1 ( mem="shared", distr="block" );
output Integer[size(arr,1)] arr ( mem="shared", distr="block" ) := arr1;
protected
parameter Integer p = nProcessors();
Integer Ndp = N div p; // Assume p divides N for simplicity
Integer[Ndp] prefix; // Local prefix array
Integer myPreSum; // Local prefix offset for this processor
Integer sum ( mem="shared" ); // Shared, consistent at superstep boundary
Integer i, j;
algorithm
  ... // the parallel code comes here, see next slide
end parPrefix;
Parallel Prefix Sums in NestStepModelica (2)

```
algorithm
j := 1;    // In Modelica, arrays start at >=1
step();    // Start of BSP superstep
for i in arr loop // Iterate over local elements
    prefix[i] := sum;
    sum := sum + arr[i];
    j := j + 1;
    end for;
endstepReduce (result=sum, op=Operators.plus,
               prefixVar=myPreSum);
endstep(); // End of BSP superstep
j := 1;
step();
for i in arr loop // Put prefix sum results into arr
    arr[i] := prefix[i] + myPreSum;
    j := j + 1;
    end for;
endstep();
end parPrefix;
```

Speedup for Parallel prefix sums

- Parallel prefix 300*10^9 elements
- Speedup vs. Number of processors
- Xeon cluster Monolith, NSC Linköping
Nested BSP-Parallelism

- Parallelism-creating constructs can be nested:
  - statically: e.g. nested parallel loops
  - dynamically: e.g. parallel divide-and-conquer computations

- generates massive parallelism

- In NestStep: split the group (= fork a parallel process)
  
  ```
  neststep( nsubgroups = 2, 
             mysubgroup = ... );
  
  ....
  if (thisgroup().gid==0)  foo();
  else  bar();
  
  endneststep();
  ```

Summary NestStepModelica

- NestStep Run-time
  - Shared-Memory Programming on Message-Passing Systems
  - Alternative to OpenMP and UPC
  - Simple, deterministic memory consistency and synchronization model
  - Structured parallelism
  - Run-time system on top of MPI on Linux clusters [Soh06]:
    - Scalable implementation,
      up to 30x faster than OpenMP on Cluster-DSM with 32 processors

- NestStepModelica
  - Expose explicit parallelism at the language level
  - Encapsulated in the imperative parts of Modelica code
  - Front-end (NestStep-Modelica → C + NestStep RTS) as Modelica-Metamodel extension, under development
  - Plans for further targets:
    - CC-NUMA
    - Chip multiprocessors, e.g. IBM CELL
Conclusions

- Finegrained automatic parallelization of models
  - Now operational and give speedups. Could give more for larger applications and if solver sould not be a single-processor bottleneck
- Coarse-grained manual component-oriented model parallelization on the GRID (GRIDModelica)
  - Preliminary prototype started running.
- Explicit parallel programming in Modelica algorithmic code (NestStepModelica)
  - Run-time system gives speedups compared to OpenMP. Integration with OpenModelica compiler under way.

Contact

www.ida.liu.se/projects/OpenModelica
  Download OpenModelica and drModelica

www.mathcore.com/drmodelica
  Book web page, Download book chapter, DrModelica

www.modelica.org
  Modelica Association

peter.fritzson@ida.liu.se.com
OpenModelica@ida.liu.se