Generation of Message-Passing Fortran 90 Code

Deliverable D8.2.2/1

The PREPARE Consortium

Release: 1.4
Date: October 4, 1994
Status: release
Confidentiality: public
Reference: PELAB-10-GeneratingF90

Contents

1 Introduction

This document describes the implementation of a Fortran 90 code generator[1] for the ObjectMath environment. Since no Fortran 90 compiler is available for our target system, we use Adaptor [1] to convert the generated code to Fortran 77. Because of limitations in Adaptor, we try to minimize the number of Fortran 90 features used. For applications which do not use intrinsic vector operations, the generated code will conform to the Fortran 77 standard, except for limitations on identifier and line lengths.

The application we use throughout this report models the dynamic behavior of the rigid bodies in a rolling bearing, by using ordinary differential equations (ODEs). The simulation is done by applying a existing, well known ODE-solver to the user-supplied differential equations.

Our efforts in making the application parallel concentrate on the user-supplied functions and parameters.

1.1 Organization of the Document

In the next section we will present an overview of the code generator. Section 3 describes the structure and format of the three different interfaces to the code generator. The internals of the code generator is discussed in sections 4, 5 and 6. Section 7 shows some statistics of code generated from a larger application. The final section presents implementation details.

2 ObjectMath Overview

The interaction of different tools in the ObjectMath environment is shown in Figure 1. The interactive environment is used for implementing the mathematical model in ObjectMath. Also, it is used to simplify, rewrite, and compute parts of the model by invoking Mathematica. When the model is complete, we extract the equations and write them to a text file. The code generator is invoked with the equation file as input. The output is a file containing a Fortran 90 subroutine. The subroutine computes the right-hand sides of the equations. This file is compiled and linked together with the ODE-solver and message passing routines. The program we get is capable of reading an input file containing start values and parameters and of simulating the implemented model according to these values. The program outputs the values of the state variables to a text file for each time step. Tools developed in PREPARE Task 8.8 can be used to present these as curves and diagrams.

Currently, the interaction between the interactive environment and the code generator is carried out by exporting and importing text files. In the future, the tools will be more integrated, so that the code generator can be invoked directly from the interactive environment.

[1] We have chosen Fortran 90 with message passing as a preliminary target language until HPF becomes available.

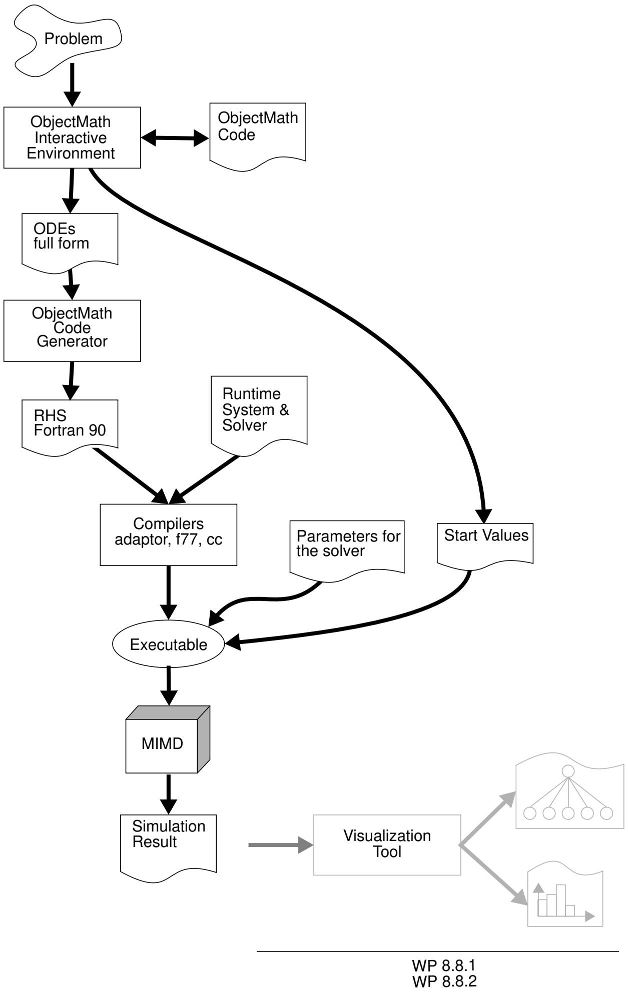

Figure 1: Structure of the ObjectMath environment.

3 Interfaces

The code generator is concerned with three interfaces: input from the ObjectMath interactive environment, output to a Fortran compiler, and the interface between the generated code and the existing solver. Each of them is described in the following subsections.

3.1 Format of Input

The input to the code generator consists of a textual representation of a Mathematica full form expression.[2] This expression is extracted automatically from the ObjectMath system. The variables in the expression are annotated by the types needed in the generated code [4, section 8.1.1], [6]. The grammar of an input expression is shown in Table 1. Since the equation

[2] Prefix notation with comma separated arguments inside brackets.

Normal form (without type annotations):

```
{ { x'[t] = y[t],
    y'[t] = -x[t] },
  { t, tstart, tend } }
```

Full form (input to the code generator):

```
List[
  List[
    Equal[Derivative[1][om$Type[x, om$Real]][om$Type[t, om$Real]],
          om$Type[y, om$Real]],
    Equal[Derivative[1][y][t], Minus[x[t]]]
  ],
  List[t, om$Type[tstart, om$Real], om$Type[tend, om$Real]]
]
```

Figure 2: This is an example consisting of a system of two differential equations which have the sine and cosine functions as solutions for $x[t]$ and $y[t]$.

part consists of first order differential equations, the right-hand side is always a derivative. For the semantics of the equations, see [13]. An example of a very simple input expression is shown in Figure 2.

⟨input expression⟩	⟶	`List[List[`⟨equations⟩`],List[`⟨x⟩ ⟨x_start⟩ ⟨x_end⟩`]]`
⟨equations⟩	⟶	⟨equation⟩ {`,` ⟨equation⟩}
⟨equation⟩	⟶	`Equal[Derivative[1][`⟨lhs variable⟩`][`⟨x⟩`],`⟨expr⟩`]`

The variables used in the input expression must be type annotated. `x` is the independent variable.

Table 1: Grammar of an input expression

In the future, it will be possible to parameterize the expressions so that the number of equations is reduced. E.g. if we want to model several similar rigid bodies, we model one body and instantiate the equations as many times as the number of bodies.

At this moment, we do not consider data parallelism. This is due to the characteristics of our application. Most of the arrays used in the application are of size 1×3 or 3×3, since we are dealing with physical three dimensional objects. These arrays are too small to benefit from data parallelism.

```
subroutine om_assign_worker (om_WorkerID, om_yin, om_yout)
        implicit none
        integer om_WorkerID, om_i
        real*8 om_yin(2), om_yout(2)
        real*8 y
        real*8 x
        real*8 om_T1

        go to (1, 2) om_WorkerID
        stop

 1      continue
        y = om_yin(2)
        x = y
        om_yout(1) = x
        go to 99
 2      continue
        x = om_yin(1)
        om_T1 = -x
        y = om_T1
        om_yout(2) = y
        go to 99
 99       continue
      end
```

Figure 3: Fortran 90 code generated from the equations in figure 2.

3.2 Format of Output

Presently, we generate a Fortran 90 function which can compute the expressions of the right-hand sides. From now on we call this function RHS. In Figure 3 we have generated the RHS from the simple input expression of Figure 2. An integer parameter is used for deciding which of the derivatives to compute. The decision where the separate expressions are computed is postponed to runtime to make load balancing possible. This makes it impractical to generate special code for packing and sending messages. It is better to go one step further and implement slightly more general communication routines in the runtime system and invoke these from RHS as needed.

There are unnecessary assignments in the generated code but we rely on the Fortran compiler to remove these by means of optimizations based on data flow analysis.

In addition to RHS we need some functions to set the starting point for the simulation. To make it possible for the programmer to use the variable names from the ObjectMath-model, these functions must also be generated by the code generator. Furthermore, since it is essential that the start values for the simulation can be changed without recompilation of the application, we generate a function which reads a value from a text file and assigns it to the right variable.

3.3 The Existing Solver

There are several ODE-solvers available. We have used a solver named LSODA from the ODE-solver package ODEPACK [11]. This solver is written in Fortran 77 by Alan C. Hindmarsh and Linda R. Petzold [12]. It is one of the solvers which implements BDF (backward differentiation formulas) methods, which are usually used to solve stiff ODEs [10].

Before calling the solver, the user have to supply several parameters, start conditions, and provide the solver with a function which computes the derivative of the current state, $\dot{\boldsymbol{y}}(t) = \boldsymbol{f}(\boldsymbol{y}(t), t)$. This function is the target of the parallelization efforts of the code generator. There is also a possibility for the user to provide the solver with an extra function that computes the Jacobian, instead of having the solver doing it internally (which is usually very expensive.) If the user can provide this function the computation time might be reduced drastically. The impact of this will be investigated in the future as a part of PREPARE Task 8.3.

4 Common Subexpression Elimination

There are four main reasons why we choose to do common subexpression elimination (CSE):

1. The textual size of the expression. There are problems of generating code for large expressions. In Fortran 90, both the line length and the number of continuation lines are limited.

2. The number of subexpressions in one large expression. We have experienced difficulties when compiling large expressions. Occasionally, the internal data structures inside the target language compiler have been too small to handle the large number of subexpressions we get.

3. Side-effect free functions. In Fortran 90, the common mathematical functions are defined as intrinsics which can be exploited by the compiler. Also, if we consider generating e.g. C++, the C++ compiler has no knowledge about common side-effect free functions.

4. User-defined functions. In the future, it will be possible to annotate functions with some information about side effects. This can, in addition to ordinary data flow analysis, be valuable for the ObjectMath compiler.

The CSE is done by traversing the expression bottom-up and generating a new assignments for each subexpression. Each assignment is saved in a hash table to make it faster to find common subexpressions. There is a more detailed description of this algorithm in [4, section 8.2.6].

5 Shape Conversion

The ObjectMath language, allows differently sized arrays as arguments to one single array operation. Fortran 90 is more restrictive in this respect, in that only scalar variables are promoted to arrays. To make the different environments similar, suitable code is generated to overcome the differences [4, section 8.3.3]. New temporaries are constructed where the converted arrays are stored. Operations can then be performed on equally shaped arrays.

6 Parallelization

When generating parallel code we must be fully aware of the structure of the application, how to partition it, the parts which need to communicate with each other, and also the target architecture.

6.1 Target Architecture

Our initial target architecture is a MIMD machine. We use computers from Parsytec including GC/el, PowerXplorer as well as their new GC/PowerPlus. The GC/el consists of a mesh of 16 T805 transputers. Each node contains one T805 transputer and 4 MB of memory. The compilers for this machine are made by ACE. The PowerXplorer also consists of 16 nodes in a mesh. Each node in this machine contains one PowerPC 601 which do the computation, and one transputer which is responsible for all communication. The two processors share 32 MB of memory. The GC/PowerPlus is going to have 64 nodes where each node has two PowerPC 601 and four T805 transputers (+ 32 MB of memory). Both the PowerXplorer and the GC/PowerPlus are currently delivered with compilers from Motorola.

6.2 Structure of the Application

We see in example 1, section 3.1, that the RHS function can be parallelized by making a task of each derivative computation and computing these in parallel. This can easily be done since the tasks in each time step are embarrassingly parallel. By using this method we get a repetitive fork/join communication graph (Figure 4) where each repetition is one call to RHS. The RHS is typically invoked several thousands of times in one simulation.

If we then look at the size of each derivative, there can be large differences leading to an unbalanced workload in the computation. As is typical in rigid bodies simulations, half of the derivatives are just trivial variables. (e.g. equation (2) in Figure 5).

Since the communication cost in a MIMD architecture is usually high compared with the processing speed, it is essential to maintain a large granularity size in order to utilize the processing power in each node. We have implemented some simple heuristics to increase the granularity size:

- All the trivial expressions, i.e. consisting of only one symbol, are packed together.
- When the expressions are too large, no splitting of the expressions is done.

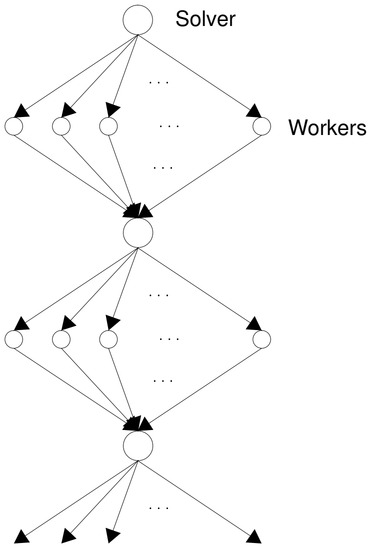

Figure 4: The structure of the communication in a parallelized RHS. The workers are each assigned one task which computes one derivative.

$$\dot{v} = \begin{cases} \texttt{(Bounce Expression)} & \text{If bounce} \\ -g & \text{Otherwise} \end{cases} \tag{1}$$

$$\dot{x} = v \tag{2}$$

Figure 5: Small example consisting of pseudo equations for a bouncing ball in one dimension. The `(Bounce Expression)` models the forces acting on the ball in the time interval when the ball is in contact with the ground. It can be arbitrary complex.

The scheduler will try to level the load but if there is one task which by itself loads one node of the machine more heavily then all the others together, a load imbalance will remain in the execution.

There will be further work on extracting subexpressions shared between several tasks as well as task splitting.

6.3 Scheduling

In most cases a real problem contains more differential equations than the target machine has nodes. Therefore we have a scheduler that is responsible for clustering tasks into *nr_worker* number of clusters, where *nr_worker* is the number of workers that are available on the machine. One method of doing this is to calculate the *weight* of each task and distribute them as evenly as possible. The *weight* is here a prediction of the estimated execution time. As the scheduler has the predicted execution time of each task and all tasks are currently independent of each other, it can use the very simple *largest-processing-time (LPT)* scheduling algorithm [2] to construct an efficient schedule. If t_{LPT} denotes the time for a LPT schedule, $t_{\rm opt}$ the optimal time and m the number of workers available, then

$$\frac{t_{LPT}}{t_{\rm opt}} \leq \frac{4}{3} - \frac{1}{3m} \qquad (3)$$

Thus, the schedule will never be worse than $\frac{4}{3}$ of the optimal schedule.

As we can see from equation (1) there may be conditions in the derivative expression. These may be impossible to predict which makes it difficult to estimate the execution time. During simulation, these conditions can cause the load on different nodes to vary in time and temporarily reduce the performance. We can avoid this by adapting the schedule to the varying load.

Inside one simulation there are repetitive calls to RHS. We make the assumption that the elapsed time for every RHS calculation is a decent prediction for the execution time required by the following RHS calculation. This may not always hold, but it is a good guess considering the number of repetitive evaluations that is done in this type of application. By using this measurement, which is based on the real load, we make new schedules at regular intervals. We could reschedule on the basis of measured imbalance of the workload, but because of the imbalance that can remain after a reschedule, this can cause frequent scheduling. Instead, we reschedule after a set number of calls, whether we need it or not. Since we use the LPT scheduling algorithm, this is a fairly inexpensive solution (amounting to about 10% of the total elapsed time, for the examples investigated.)

The LPT algorithm does not take the communication time into account. This is a problem since the number of calls (and therefore the number of communicated messages) is relatively large compared to the computation time. More effort should be spent in finding a scheduling algorithm which better utilizes the machine and tries to keep the number of workers down, since the communication cost is high.

6.4 Size of Messages

As the application, and also the number of ODEs grows, the size of the messages sent between the solver and all the workers becomes larger. This must be handled to make the application scalable. Currently, every variable that might be used is passed to the workers i.e. all variables in the state vector. This scheme is used because of the dynamic scheduling strategy. Since the work is redistributed semi-dynamically by the scheduler, we also have to compose the messages semi-dynamically at runtime. This composition of smaller messages instead of sending the whole state will be implemented in the future.

7 Generating Code for an Application

Looking at the simple example in section 3.1 and 3.2, we see that the resulting code is not very efficient; In each branch of the case-statement[3] we make a trivial assignment. I.e. the

[3] In Fortran 77 CASE does not exists. Instead we use computed GOTO

RHS function just moves data around. Unless we packed the expressions into one task, as we actually do, we would have computed the assignments in parallel on two processors, adding high communication overhead to the simulation time! We will see better behavior of the code generator by looking at a more reasonably sized problem.

The chosen application is a simple two-dimensional model of a cylindrical rolling bearing, earlier used in [3]. It was used to simulate the dynamic behavior of the outer ring and one rolling element in a bearing. We now extend it by adding rolling elements until we get ten of them. The application will still be small compared with a realistic model, but by doing this we approach the reality and at the same time we introduce more parallelism.

From the interactive ObjectMath environment, this model expands to 11859 lines of type annotated Mathematica full form code. From this, the code generator produces 10913 lines of Fortran 90 code. The CSE-elimination extracts 4642 common subexpressions and 4709 lines are variable declarations. The Fortran 90 statements are readable but not understandable since there is such a large amount of subexpressions extracted from their context. The performance of this model is discussed in [5].

If we instead generate serial Fortran 90 code, i.e. allowing the CSE-eliminator to use the entire right-hand side and not only equation by equation, we get 4301 lines of Fortran 90 code (1840 common subexpressions). Where does this reduction comes from? Different equations have apparently several large expressions in common. These can not be shared when the equations form separate tasks—hence the larger number of subexpressions. In order to reduce this number, we somehow have to extract large common subexpressions and compute them in parallel. This should be done prior to the computation of the right-hand sides as previously mentioned in section 6.2.

8 Implementation details

The code generator is implemented in C. We use Ast [8] for the internal representation, Rex [7] for the scanner, and Puma [9] to traverse and rewrite the internal structures.

The implementation consists of the following modules. (The number of lines for each module is shown in table 2):

Code Generator
Generates the Fortran 90 code.

OIR/Index
Modules where various routines to make, remove and rewrite internal structures are implemented.

Parser
A simple parse is needed to parse the Mathematica full form.

Verifier/Transformer
Is the parsed tree consistent? Various checks are done. We also make a few minor transformations concerning arrays here.

Parallelizer
It is here the parallelism in the code is analyzed.

Common Subexpression Eliminator (CSE)
Eliminates common subexpressions, as described in 4.

Type Deriv
The CSE needs the types of subexpressions for its generated variables.

Shape Converter
Arrays need to be reshaped to behave the same way as in ObjectMath.

Unparser
Outputs the Fortran 90 code.

Abbrev
Fortran 90 does not allow variable names of more than 31 characters. Longer names must be abbreviated.

Error Handling

Runtime System
Routines which invoke and support the computation.

Solver
We use LSODA from ODEPACK.

Parameter Reader
The solver needs some parameters which are read from a file at runtime. We also need start values for all variables.

RHS
The function which the solver need for computing the derivatives is a prewritten function which uses the message package and invokes the generated Fortran 90 code on each processor.

Message Package
A purpose built message package.

Timing Package
We need time measurement for the scheduling.

Scheduler
Implements the LPT scheduling at runtime.

Module	Lines of code
Code Generator	
OIR	251 Lines of Ast, 26 Lines of C
Index	702 Lines of C
Parser	164 Lines of Rex, 90 Lines of Puma, 686 Lines of C
Verifier/Transformer	261 Lines of Puma
Parallelizer	459 Lines of Puma
CSE	263 Lines of Puma, 253 Lines of C
Type Deriv	509 Lines of Puma
Shape Converter	129 Lines of Puma
Unparser	1043 Lines of Puma, 295 Lines of C
Abbrev	81 Lines of C
Error Handling	136 Lines of C
$\sum$	7797 Lines of code
Runtime System	
Solver	7584 Lines of prewritten Fortran 77.
Parameter Reader	168 Lines of Yacc-spec, 36 Lines of Flex-spec
Startup	543 Lines of C
RHS &	
Message Package &	
Scheduler	387 Lines of C
Time Package	119 Lines of C
$\sum$	1253 Lines of code

Table 2: The amount of code written for each module.

References

[1] Thomas Brandes. ADAPTOR: Language reference manual version 2.0. Report Adaptor 3, German National Research Center for Computer Science, March 1994.

[2] Edward G. Coffman, Jr and Peter J. Denning. *Operating System Theory*. Prentice Hall, 1973.

[3] The PREPARE Consortium. Dependency analysis and scheduling for ordinary differential equations. Ref. PELAB-2-Scheduling, rel. 1.1+, February 1994.

[4] The PREPARE Consortium. Design of the ObjectMath compiler and code generator. Ref. PELAB-3-Design, rel. 1.8, October 1994.

[5] The PREPARE Consortium. Performance measurements and evaluation of simulation of a simple roller bearing model. Ref. PELAB-9-SmallBearingPerformance, rel. 1.3, October 1994.

[6] The PREPARE Consortium. Static types for ObjectMath^c. Ref. PELAB-5-Types, rel. 1.5, August 1994.

[7] Josef Grosch. Rex — a scanner generator. Compiler generation report no. 5, Gesellschaft fuer Mathematik und Datenverarbeitung mbH, Forschungsstelle an der Universität Karlsruhe, Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe 1, Germany, December 1987.

[8] Josef Grosch. Ast — a generator for abstract syntax trees. Compiler generation report no. 15, Gesellschaft fuer Mathematik und Datenverarbeitung mbH, Forschungsstelle an der Universität Karlsruhe, Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe 1, Germany, September 1991.

[9] Josef Grosch. Puma — a generator for the transformation of attributed trees. Compiler generation report no. 26, Gesellschaft fuer Mathematik und Datenverarbeitung mbH, Forschungsstelle an der Universität Karlsruhe, Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe 1, Germany, November 1991.

[10] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer-Verlag, 1991.

[11] Alan C. Hindmarsh. ODEPACK, A systematized collection of ODE solvers. *IMACS Transactions on Scientific Computing*, 1:55–64, 1983.

[12] Linda Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM J. Sci. Stat. Comput.*, 4(1):136–148, March 1983.

[13] Stephen Wolfram. *Mathematica — A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, 1988.