Generating Parallel Code in the ObjectMath Programming Environment

Deliverable D8.3.3/1

The PREPARE Consortium

Release: 1.6+
Date: April 13, 1996
Status: release
Confidentiality: public
Reference: PELAB-12-Fortran

Contents

Preface

This report describes the efforts made to generate parallel code within the ObjectMath programming environment. It constitutes Deliverable 8.3.3/1 from the partner PELAB in the PREPARE project.

We abandoned the goal of generating High Performance Fortran since this language, like all data parallel languages, is not a suitable intermediate language for our purposes. Instead, this work focuses on how to extract parallelism from ObjectMath equations and how to implement this parallelism in a message-passing runtime system.

The report has been written for dissemination in a general context. Therefore, the first chapters in this report describe rather basic ideas and terminology of parallel computing, ordinary differential equations, task graphs, etc.

Additionally, design documents of ObjectMath is enclosed. These are detailed specifications and documentations of design decisions made during the development of ObjectMath.

Enclosed:

1. The ObjectMath Internal Representation
2. The ObjectMath^c Language
3. Static Types for ObjectMath^c
4. Design of the ObjectMath Compiler and Code Generator

Chapter 1

Introduction

Parallel processing has become a widely used approach for increasing the throughput of modern computers. Development in this area has been stimulated by applications that need immense computing power, such as weather prediction, chemical modelling, and various mechanical simulation problems. Many of these problems are solved by dedicated application programs, which are hard to understand and even harder to port between different parallel architectures without major restructuring or loss in efficiency. Application problems should therefore be expressed at the highest possible level of abstraction to reduce the dependence between instances of machines and programs.

1.1 Overview

This report is organised as follows. After the introductory chapter which you are currently reading, there is a chapter about such widely differing subjects as parallel computers, ordinary differential equations, programming environments, and mechanical construction elements (rolling bearings). This provides the technical background and lies the groundwork for the rest of the report. Basic definitions to facilitate further discussions are presented in chapter 3. The next four chapters, numbered 4, 5, 6, and 7 comprise the core of this report. The first, chapter 4, expounds on the properties of a parallelising compiler and the general approach in this work. The next three chapters present the three approaches we take to find a solution to our parallelisation problem. Finally, everything is summarised in chapter 9.

Chapter 2

Background

This chapter provides a technical background in areas on which this work is based. It is not an exhaustive presentation but merely an overview to place ourselves on common ground.

In the first section, we take a look at the state of the art in parallel computing and give an answer to two questions:

- What is the current trend in computer architectures?
- Which programming models are used?

In the next section, we are given a short introduction to the topic of ordinary differential equations. We see how a numerical solver operates for solving such equations. Also, we touch upon domain specific potential parallelism that may exist.

Third, we present ObjectMath, the environment and framework in which context all of this research has been performed.

Next, there is an example application: A rolling bearing simulation application. We use this as the main test example throughout this report.

Last, we present efforts made by others in the fields of scheduling, mapping and clustering task graphs. Some of the present algorithms in this section are used in the following chapters.

2.1 Parallel Computing

Broadly speaking, a *parallel computer* is a set of processors which are able to work cooperatively to solve a problem. This definition is broad enough to include embedded transputer systems, clusters of workstation, shared memory machines, and supercomputers. When we talk about *parallel computing* we mean computational intensive applications which execute on a parallel computer.

Scientific applications, such as weather prediction, fluid dynamics, simulations of chemical reactions, etc. , still motivates the development of computers at the high end. Lately, large databases, video-on-demand, and virtual reality are applications that show an incresing demand for computational power. In contrast to the scientific applications, their main task is not to solve physical, mathematical, or chemical problems but to locate, transfer, translate and store large amount of data.

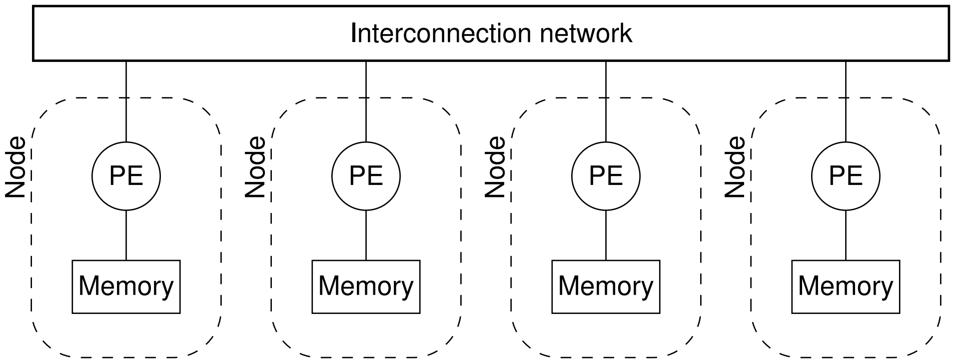


Figure 2.1: Distributed memory architecture

2.1.1 Computer Architectures

In the history of parallel computers many different architectures have been introduced. The current trend when talking about highly parallel[1] computers can be divided into three categories: distributed memory architectures, shared memory architectures, and others.

Distributed memory architectures comprise a set von Neumann computers. They communicate with each other via an *interconnection network* (figure 2.1). This can be anything from workstations interconnected with a local area network (LAN) to dedicated computer nodes, tightly coupled in a efficient topology by specially designed high speed communication links. Each computer executes its own program and communicates with other computers by explicitly sending and receiving messages over the network.

Shared memory architecture is our second category of parallel computer achitectures. Computers based on this achitecture are relatively inexpensive to build and usually have very low communication latency. Also, the shared memory programming model simplifies program development. This has made the shared memory architecture very popular.

A shared memory computer consists of a set of (often identical) processors sharing access

[1] We do not go as far as massively parallel.

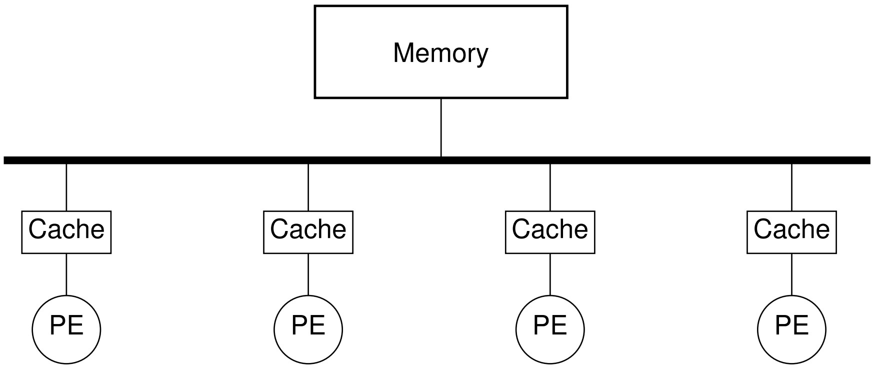


Figure 2.2: Shared memory architecture

to a common memory typically via a bus or a hierachical network (figure 2.2). If we look at current technology, memories do not show the same rate of yearly increase in speed as processors do. Therefore, this architecture suffers from scalability problems. The repeatedly used method to overcome this, is to associate a local *cache* to each processor. Access to data in the local cache becomes much faster than the common shared memory and hence locality will be more important than ever.

As a result, the difference between distributed memory architectures and shared memory architectures is just a question of the degree of locality.

In the last of our categories, the one labeled "others", we find vector computers, SIMD (Singel Instruction, Multiple Data) computers, and COMA (Cache Only Memory Access) computers among others. Indeed, each of these architectures is interesting in its own way. For instance, a SIMD computer may show vey good performance on problems with a high degree of regularity, e.g. image processing. Still, these architecture are either very expensive to build or highly specialised for a certain purpose. They will find it hard to survive on the market since the price/performance quotient becomes more and more important.

2.1.2 Programming Models

As programmers, we use a programming model to describe the parallelism of an architecture. The von Neumann's computer model with its basic instructions (read memory, write memory, and arithmetic operations) is at a low level. This is too low level from a today's programmers point of view. Instead, we will here identify three common models suitable to the earlier presented architectures. These are: *Message Passing*, *Data Parallelism*, and *Shared Memory*.

Messsage Passing

This is probably the most used parallel programming model.

- It gives the user control over each step of execution and communication,
- it is easy to keep message passing programs deterministic, and
- it is easy to map this model down onto distributed as well as shared memory architectures.

In this model, one or more tasks are created on each computer which then communicate with each other by sending messages. The message passing model is often implemented as a library of functions which can be invoked from a parallel program written in a common high level laguage. Example: *Parallel Virtual Machine (PVM)* and *Message Passing Interface (MPI)*.

Data Parallelism

Data parallelism presents another view on parallelism. It exploit parallelism in applications which contains operations that are applied on multiple data elements. The granularity of such program is generally small and the locality can be awesome to handle. To achieve adequate performance, the programmer often have to specify exactly how the data should be distributed. However, applications which contains many data parallel operations, fits well into this model.

Data parallelism is mostly implemented as language constructs. A data-parallel compiler is needed to translate the data-parallel constructs to appropriate primitives on the architectures. Typical examples of this model is *High Performance Fortran (HPF)* and *C** on Connection Machine.

Shared Memory

This is the third and last parallel programming model we touch upon. This model is easily implemented on shared memory architectures because of the many similarities between the model and the architecture. However, distributed memory architectures cannot be used as easily for this model since the communication latency usually becomes too great for achieving good performance.

An advantage of the shared memory model from the programmer's point of view is that the notation of data "ownership" is lacking, and hence there is no need to explicitly specify the communication of data. Different tasks can access data asynchronously. To control the access, semaphores, monitors and similar primitives are used. Not needing to specify data ownership can simplify program development. However, a disadvantage is that understanding and managing locality becomes more difficult. Also, writing deterministic programs can become more difficult.

2.2 Ordinary Differential Equations

2.2.1 Initial Value Problems

A large group of (mathematical) problems can be expressed as initial value problems (IVP) for a system of ordinary differential equations (ODEs) [3]. These problems can be solved numerically by applying a high quality, general, pre-written ODE-solver to the equation system and computing the desired solutions. Most existing solvers are inherently sequential due to the methods used. In the past, attempts have been made to restructure and adapt algorithms to extract some parallelism [5, 32].

The computed solution of an initial value problem consists of a large number of calculated approximations where each approximation depends on the previous one. To make the final result exact enough, the accuracy in every calculation must be kept at a high level, forcing the solver to take very small steps and approach the solution at a very slow pace. This makes such a computation very time-demanding. One method to shorten the execution time is to utilize parallelism.

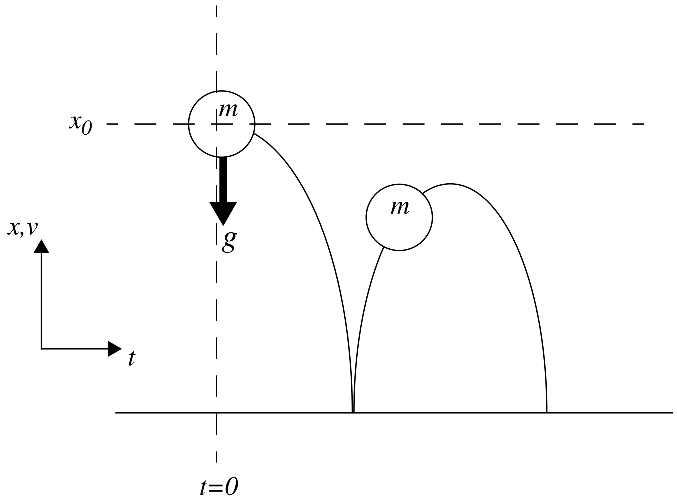
2.2.2 Numerical Solvers

The solution of a numerical ordinary differential equation needs the following:

- a general, carefully implemented algorithm (a solver) which approximates the solution to a given system of ODEs,

Example: A ball bounces on a surface. The law of gravity and the forces on the ball when it is in contact with the surface etc. are modelled with ODEs. The state is known when the time is zero and we calculate the position of the ball at different time points.

$$
\begin{array}{lll}
x' = v & & x(0) = x_0 \\
v' = \begin{cases} -g & \text{if } x \geq 0 \\ -g - \frac{k_{spring} \cdot x + k_{damp} \cdot v}{m} & \text{if } x < 0 \end{cases} & & v(0) = v_0 \\
x(t) = ? & &
\end{array}
$$



- an initial state,
- some controlling parameters, and
- solution points where solutions are desired (figure 2.3).

The initial state (often trivial to calculate) and control parameters are given once, at the beginning of the computation. The desired solution can be either a single point or a whole range of points where solutions are wanted. The system of ODEs is a function $\dot{\boldsymbol{y}}(t) = \boldsymbol{f}(\boldsymbol{y}(t), t)$ which calculates the first-order derivatives $\dot{\boldsymbol{y}}(t)$ for each state variable from a given state $\boldsymbol{y}(\mathrm{t})$. Thus the equation system consists of only first-order ODEs. The variable t is called the *free* variable and often represents the time in a simulation. The function, $\boldsymbol{f}$ should be side-effect free to allow as much parallelism as possible to be extracted. From now on we call this function RHS[2].

Scheme to Find a Solution

From a known point, $\boldsymbol{y}_n$, on the solution curve, the solver makes an approximation $\boldsymbol{y}_{n+1}$ to $\boldsymbol{y}(t_{n+1})$. This approximation is an extrapolation of either previously calculated points

[2] Right-Hand-Side

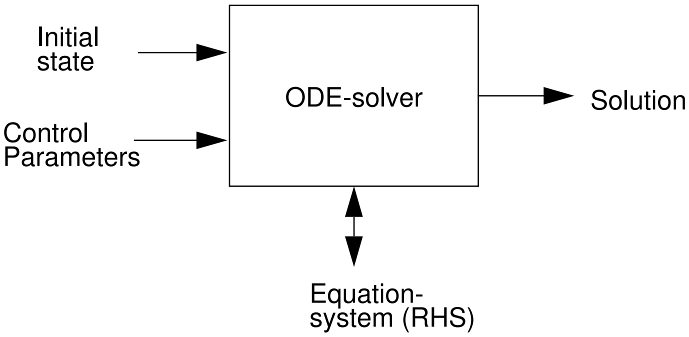


Figure 2.3: Structure of a ODE-solver for an initial value problem.

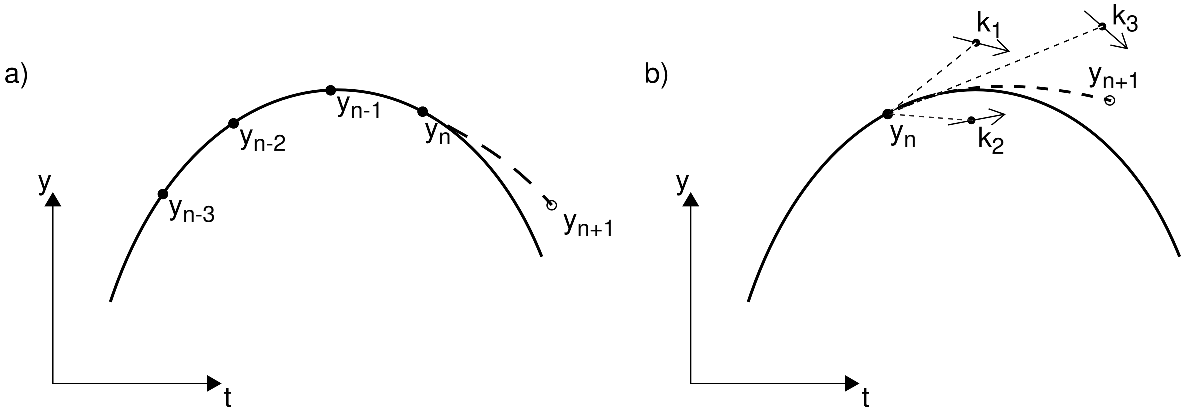


Figure 2.4: A step in solving an initial value problem. The left part is a multistep method and the right part is a single-step method

(multistep methods, figure 2.4a) or intermediate extrapolations (singlestep or Runge-Kutta methods, figure 2.4b) [32].

Each time an extrapolation is calculated, the function RHS is invoked. Since the solver usually needs to take a large number of steps to reach the desired solution, the communication between the solver and RHS is intense.

If the method used by the ODE-solver is implicit, the extrapolation point is dependent on itself and calculated by an iterative method. In that case it can be necessary to calculate the Jacobian matrix, $J = \frac{\partial f(y,t)}{\partial y}$. We can, of course, use the RHS-function to compute an approximation, but it is usually possible to provide the solver with an extra function dedicated to computing the Jacobian.

Exsisting Solvers

There are several ordinary differential equation solvers available. In this work, we have used a solver named LSODA from the ODE-solver package ODEPACK [14]. This solver is written in Fortran77 by Alan C. Hindmarsh and Linda R. Petzold [27]. It is one of the solvers which implements backward differentiation formulas (BDF), which are usually used to solve stiff ordinary differential equations [12].

2.2.3 Potential Parallelism

There are three ways to exploit parallelism when computing an initial value problem:

PAR–1: *Parallelism across the method.*

PAR–2: *Parallelism across time.*

PAR–3: *Parallelism across the system (or problem).*

The PAR–1 and PAR–2 techniques can not be exploited without redesigning the solver while the PAR–3 focus on the right-hand sideleaving the solver as it is.

2.3 ObjectMath

An alternative to the time-consuming explicitly parallel programming would bbe to specify the application problem in a notation at a high-level abstraction and let the specification be coverted automaically into an efficient program that performs the needed computation. This idea is the basis for the ObjectMath environment.

The ObjectMath programming environment is centred around the ObjectMath language which is a hybrid language, combining object-oriented constructs with computer algebra. This combination makes it a suitable language for representing and implementing complex mathematical models. Formulae and equations can be written in a notation that closely resembles conventional mathematics, while the use of object-oriented modelling makes it possible to structure the model in a natural way.

The well known computer algebra system named *Mathematica* has been chosen as a basis for ObjectMath. One advantage of this choice is that users who are familiar with the widespread Mathematica system can learn ObjectMath easily. The ObjectMath language consists of an object oriented layer on top of Mathematica, just like C++is compared to C. However, the ObjectMath language emphasises structured mathematical modelling more than operations on state, in contrast to object-oriented programming languages such as C++. The ObjectMath language has been, and still is, evolving based on feedback from users.

The ObjectMath programming environment is designed to be easy to use for application engineers, e.g. in mechanical analysis, who are not computer scientists. It is interactive and includes a graphical browser for viewing and editing inheritance hierarchies (figure 2.5). Other parts of the environment support routines for generation of numerical code and visualisation. Viklund et al. [34] gives a more thorough overview of the ObjectMath programming environment and its implementation.

2.3.1 Programming Environment

The ObjectMath programming environment consists of

- Compiler

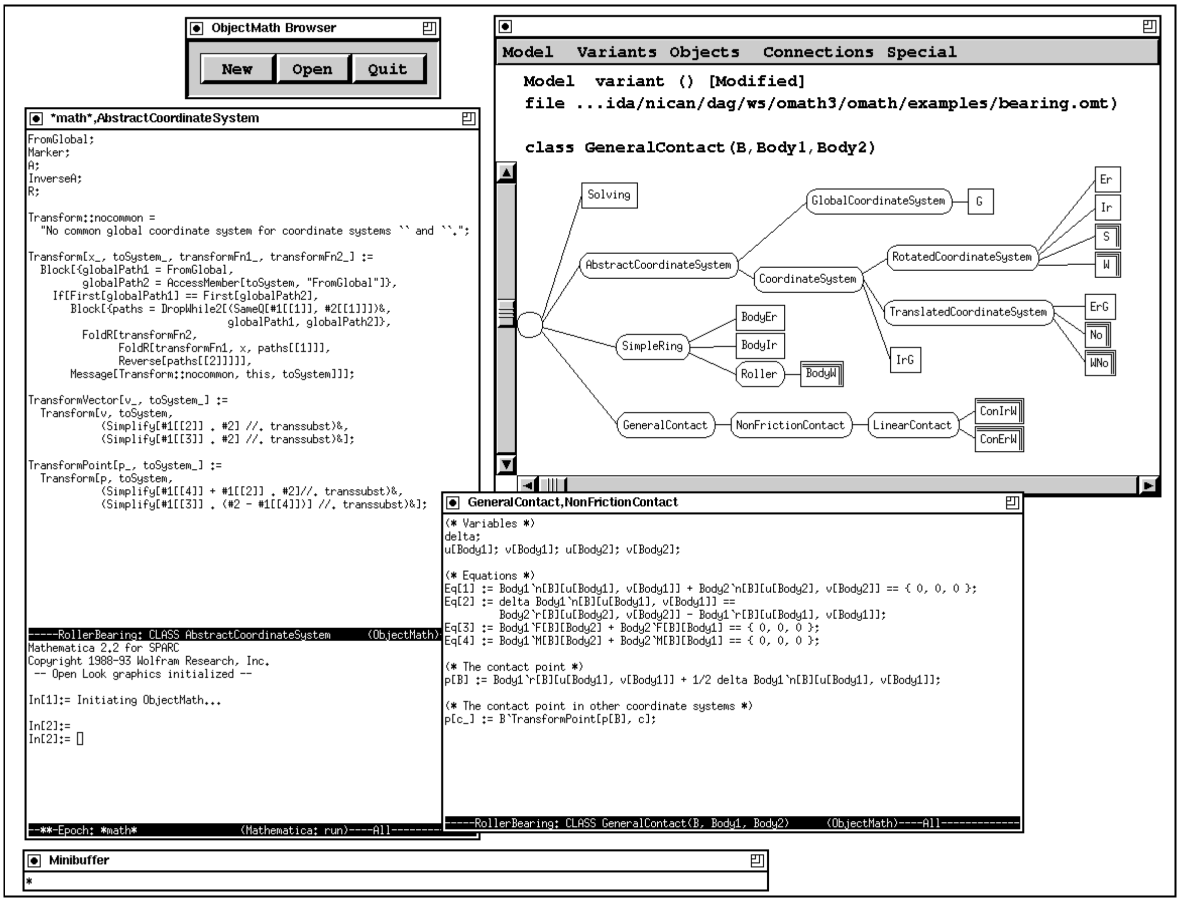


Figure 2.5: Screendump of the ObjectMath enviroment

- Structure Editor
- Text Editor
- Computer Algebra System

We the topic of the compiler for now, since this is explained in section 2.3.2.

The structure editor which has a graphical user interface (GUI) is used to view and edit the inheritance hierarchy of an ObjectMath program. It is implemented in C++ using the general GUI package InterViews.

GNU Emacs is the text editor of choise that the programmer use for writing code and equations that is contaioned whitin ObjectMath objects. Emacs is enhanced with an ObjectMath-mode which enables the exchange of information with the structure editor and Mathematica. Moreover, Emacs is also used as a front end to Mathematica.

As already mentioned, Mathematica is the computer algebra system in ObjectMath. It was chosen over other similar systems because it had better support for graphics and, at the time, had a better integrated environment than the possible alternatives.

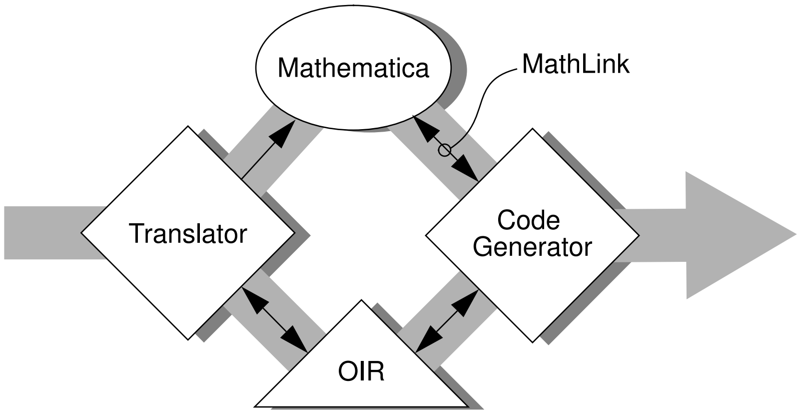


Figure 2.6: The ObjectMath compiler.

The user of ObjectMath has the ability to make advanced transformations of models expressed in ObjectMath both interactively and at compile-time. Therefore, the problem structure which preferrably was explicitly stated when a n application problem is specified, may become disguised and even lost to some degree during the process. Still, we do not want to limit the user's nor Mathematica's possibilities to simplify and transform the model. On the contrary, the more alternatives that are presented to the user, the better chances there are, to reach a good result.

Nevertheless, this complicates the analysis inside the ObjectMath compiler. The structure of the problem specification is, to some degree, kept throughout the automatic transformations inside the ObjectMath environment. However, this may have been disguised beyond recognition. It may be a difficult task to reveal the underlying structure of the problem. If this structure could be detected, it is a golden opportunity of finding a promising ordering of generated code and how to proceed efficiently with the calculation and thereby have the execution time reduced radically. When generating code for parallel computers this is the most important problem to focus on.

2.3.2 Software Architecture

The core of ObjectMath is the *compiler*. It consists of two main parts: The *translator* which translate ObjectMath programs into Mathematica, and the *code generator* which generates imperative code (Fortran 90or C++). After the translatoion phase have completed, the generated Mathematica code is transferred into a Mathematica process. This enables the programmer to interactively evaluate functions, apply translations and rewriting rules to expressions, etc. During the more autonomous code generation phase, the Mathematica process is still active. It is ready to step in, supplying the code generator with the power of a general computer algebra system whenever needed (figure 2.6).

Although the code generator is not invoked immediately after the translator has completed as in an ordinary compiler, both parts share a common address space. This enables the code generator to directly access ObjectMath's internal representation (OIR) where the type

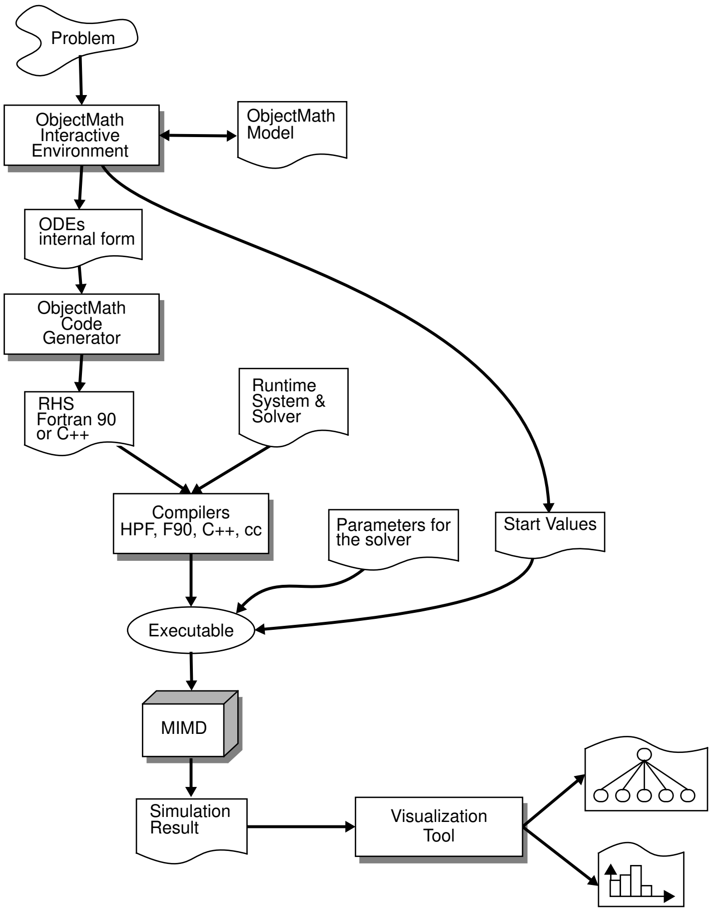


Figure 2.7: The overall structure of the ObjectMath environment.

information is stored. Algebraic expressions can be evaluated symbolically in Mathematica before code is generated from them. This can easily be handled, as the code generator communicates with Mathematica via the MathLink [36] protocol.

Figure 2.7 shows the overall data flow structure in the ObjectMath system. First, an application problem is described as an object oriented mathematical model. This model can be inspected and transformed in the interactive environment. Later, the generated code is is combined with library routines, compiled and run on a parallel computer. Finally, there is some support for visualisation of numerical results.

The Code Generator in More Detail

ObjectMath's code generator is invoked directly from Mathematica. It can generate code from either

1. a complete ordinary differential equation system, or

2. a single function

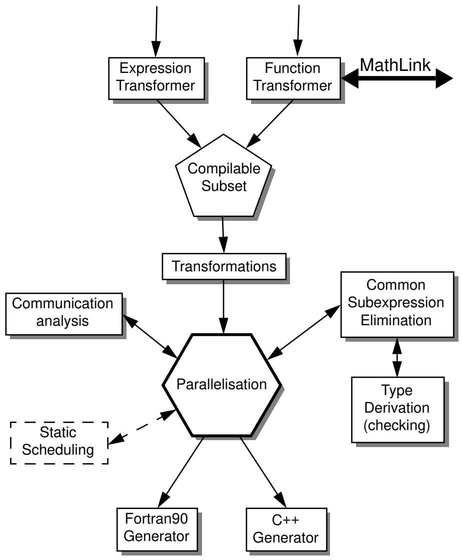


Figure 2.8: ObjectMath code generator in more detail. Modules in dashed lines is not fully implemented.

(see figure 2.8). When generating code from a function, the programmer can choose whether the body of the function should be symbolically evaluated inside Mathematica prior to code generation or not. After the input is guaranteed to consist of only compilable constructs, some preliminary transformations are applied. If the following parallelisation engine detects an ordinary differential equation system, the common subexpression eliminator with attached type derivator is invoked. Also, a data dependence graph is built that can be use in more exact and detailed analysis.

2.4 Rolling Bearings

As an example application in this report we are using simulation of a simplified two dimensional rolling bearing. The model originates from SKF Engineering & Research Centre

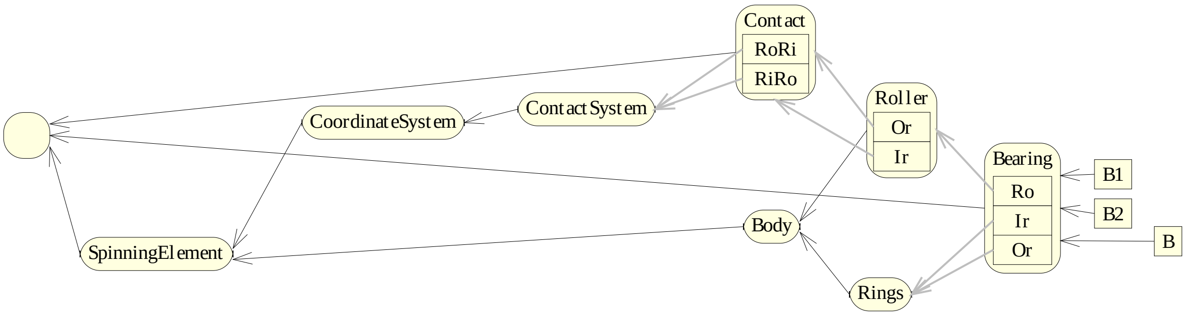


Figure 2.9: ObjectMath class hierarchy.

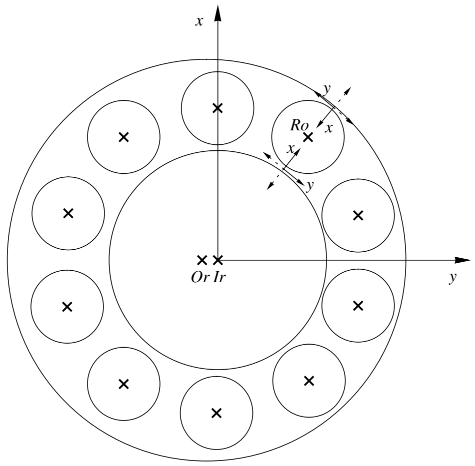


Figure 2.10: Geometry of the roller bearing.

B. V. [8] and has been specified and implemented using the ObjectMath environment (figure 2.9). The rolling bearing consists of ten rolling elements between an outer and an inner ring (figure 2.10). Each roller can be in contact with zero, one, or two rings depending on external forces and current state. Any interaction between rollers is ignored. The inner ring is fixed and the outer ring rotates at a constant speed and is affected by external load forces. There are no other restrictions of movement of the rollers except for contact forces and gravity.

The dynamic behaviour of this model has been expressed in equations which have been transformed, using Mathematica [35] in the ObjectMath environment, into first order ordinary differential equations. A form that the numerical ordinary differential equation solver accepts. During this transformation, several variables have been substituted and, thus, disappeared. Also, several standard mathematical transformations facilitating elimination of variables and other simplifications have been applied automatically. This substitution process results in a rather large expansion of the size of the final expressions (as compared to the original forms).

The chosen state variables in the transformed model represent normalised positions (1) and velocities (1) of each and every roller (10) in all degrees of freedom (3) plus the outer ring's position (1) and velocity (1) in all its degrees of freedom (2). ($(1+1)\times 10\times 3+(1+1)\times 2 = 64$ variables). Half of these are trivial to calculate (The derivative of the position is equal to the velocity, $\dot{x} = v$.) and therefore ignored in the following description.

Given a state (= set of state variables), the derivatives of the state variables are effectively calculated as follows (figure 2.11):

Calculate the contact forces. (Step 1 in figure 2.11.) This is the most time consuming part of the calculation. In a real application it would comprise strenuous integrations but in this example, the contact calculation is drastically simplified. Still, the structure of the calculation and its dependencies remains.

Apply each contact force to the bodies it affects. (Step 2) This is merely a redistribution of data. It introduce a difficult sharing of calculations in the problem.

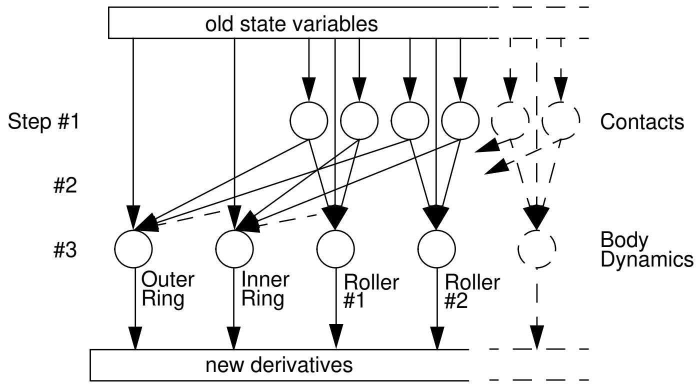


Figure 2.11: Structure of the calculation of the derivatives in the 2D bearing example.

Calculate velocities and accelerations of the rigid bodies. (Step 3) Relatively inexpensive tasks but still worthwhile to parallelise.

If we take a closer look at the first step, the calulation of contact forces, there are many entities that can be shared during the computation. Take for instance the interaction between two rigid bodies. The forces that the bodies exert on each other are of equal size but of opposite direction (Newton's third law). Therefore, only one of the forces needs to be calculated. Its negated value can then be used in the further calculations associated with the other body. Furthermore, there are many small expressions, such as $x^2, \sqrt{r}$, which are reused in many places. A computation which does not utilise these distinctive dependecies special to the application domain, can not be expected to be efficient.

This structure of the computation is, to some degree, kept throughout the automatic transformations inside the ObjectMath environment but it may have been disguised beyond recognition. One of the goals in this work is to find out whether it possible to automatically reveal the problem structure by using clustering techniques. If this structure could be detected, it is a golden opportunity of finding a promising[3] partitioning of the calculation and have the execution time reduced radically. For instance, take the interaction between two rigid bodies. The forces that the bodies exert on each other are of equal size but of opposite direction (Newton's third law) and need, therefore, be calculated only once and then used in the further calculations associated with the bodies.

2.5 Parallelism in Task Graphs

A large number of papers have been published in this (e.g. [1, 22, 29, 28, 4, 25, 26, 38, 17, 19, 18, 2]). In this section, we review a few of them and present their features and characteristics.

[3] a partition that can be of great use in later clustering and scheduling algorithms.

2.5.1 Scheduling

List Scheduling [1] is a well-known basic scheduling algorithm that quickly delivers a good schedule. The list scheduling algorithm works as follows:

1. All tasks are first given a unique priority. It is very common to use the length of the path to the exit node (the level) but other properties can also be used. If there are more than one task with a given priority, ties are broken randomly.

2. The tasks are then scheduled in priority order. This procedure consists of selecting the task with the highest priority and assigning it to the processor which offers the earliest finishing time for the task. This is repeated until all tasks have been scheduled.

The upper limit on the makespan is

$$\frac{t_{\text{sched}}}{t_{\text{optimal}}} = 2 - \frac{1}{m}$$

where m is the number of processors.

Kruatrachue and Lewis[21] introduce the *Duplication Scheduling Algorithm* which seems to be a promising extension of list scheduling. It is basically the same as list scheduling with the exception that when a step in the list scheduling algorithm introduce a communication, the *Duplication Scheduling Algorithm* can duplicate already scheduled tasks in order to minimise the makespan. Afterwards, an extra pass iterates through the schedule and removes redundant tasks. Also, their work focuses on fine grained task graphs.

2.5.2 Clustering

Internalization is an approach proposed by Sarkar [28]. This is a pass in a compile-time scheduling algorithm that finds a partition that minimises the critical path length of the task system, i.e. minimise the completion time of the task system on an unbounded number of processors.

The algorithm first places each task in the task graph in a separate cluster and sorts the edges in decreasing order. Then, it examines the edges in this sorted order and merge pairs of clusters as long as this does not cause an increase in the *critical path*. It terminates when no remaining merger could possibly reduce the critical path length. When computing the critical path length, all tasks in one cluster are forced to execute sequentially since they will all be assigned to the same processor.

The internalisation algorithm has an $O(|V|^2 \times (|V| + |E|))$ execution time which becomes $O(|V|^4)$ in the worst case. Sarkar shows that the algorithm finds the optimal partition for a restricted class of communication graphs, namely series-parallel graphs. Furthermore, he has shown that the internalisation algorithm has a worst-case performance bound of 2, relative to the optimal critical path.

After the internalisation pass, the tasks are actually assigned to the processors. Since, in our approach, task assignment has been delayed until runtime, we do not consider that problem here. Instead, the scheduling problem is reviewed in more detail in section 7.2.

The *Dominant Sequence Clustering* algorithm by Gerasoulis and Yang[10, 11, 37, 38] is also an algorithm that starts out with one task per cluster and tries to minimise the *critical path* by merging pairs of clusters. However, this algorithm's first goal is to reduce the length of the dominating sequence (= critical path)in every step. In addition, the time complexity is low. By sorting the nodes and using incremental updates the complexity is as low as $O((|V| + |E|)\,log|V|)$. The authors have also proved that the *Dominant Sequence Clustering* algorithm achieves an optimal clustering for a small group of coarse grained graphs.

If we look closer into the dominamt sequence clustering algorithm, there is an interesting phase which investigate the parents, p, of the currently clustered tasks, t. If t is the only child of p, p is moved to the cluster which contain t. This may be a profitable heuristic when the granularity of the task graph is large. However, in very fine grained task graphs the gain is negligible.

There are several other efforts made in the pursue of a solution to our problem. For instance, Hou et al. use Genetic Algorithms[15, 39] to solve the multiprocessor scheduling problem. Others try to reach a near optimal solution by gradually improving the solution by applying several different heuristics after each others [23, 30]

2.5.3 Mapping

By studying the more general problem of partitioning a graph without taking the temporal aspect into account (i.e., the *mapping* problem), many well-known algorithms have been proposed: Min cut-max flow [31, 23], spectral bisection [13, 7], Lagrange relaxation [33], graph parsing [24], simulated annealing. How well these methods fits in our topical problem is still to be investigated.

Chapter 3

Definitions

To keep our discussion precise, this chapter presents some basic definitions. The intention is neither to prove any theorems nor to show rigorous time complexity calculations. Instead, we need to establish some terms and notations since definitions found in papers may be in contradiction with each other.

In the first section, we define expressions represented as *data dependency graphs*. How these can be transformed into *task graphs* is presented in the second section. The next section is designated to define a parallel computer that can evaluate the transformed expression. It also reveal some of its features and limitations. Finally, *schedules* are defined in the last section.

The reader is expected to have basic knowledge in graph theory.

3.1 Expressions

An expression is represented as a directed acyclic graph (DAG). From here on we call this a *data dependency graph*. (example in figure 3.1)

Definition 3.2 *A* **data dependency graph** *is a tuple* $D = \langle V, E \rangle$ *where*

$$\begin{aligned} V &= \{v \in V \mid v \text{ denotes an atomic expression}\}\text{and} \\ E &= \{\langle v_1, v_2 \rangle \in E \mid \langle v_1, v_2 \rangle \text{ denotes a data dependency where } v_1 \text{ depends on } v_2\}. \end{aligned}$$

An *atomic expression* can be as small as a single addition or a large, complex, user defined function. From our point of view, the distinctive characteristics is that its computation is indivisible. Moreover, $v \in V$ contains no side-effects. This simplifies further work considerably. When using numerical solvers, this restriction regarding the right-hand side expression is often taken for granted.

3.3 Tasks and Task Graphs

Definition 3.4 *A* **Task** *is a sequential program with its own local memory.*

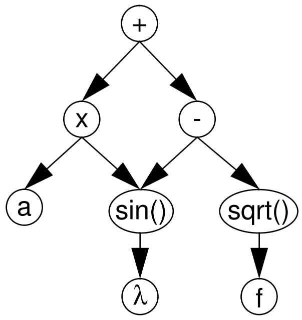


Figure 3.1: Example of a data dependency graph.

In effect, a task is a virtual von Neumann machine. The only way it can communicate with its environment (which may consist of other tasks) is through a number of well defined *ports*. Each task is referentially transparent, i.e., a task can be computed more than once and on different processors and still give the same result back.

If we interconnect a set of tasks by associating communication links to task ports, we get a *task graph*.

Definition 3.5 *A* **task graph** *is a tuple* $G = \langle T, C \rangle$ *where*

$$\begin{aligned} T &= \{t \in T \mid t \text{ denotes a task}\} \textit{and} \\ C &= \{\langle t_1, t_2 \rangle \in C \mid t_1, t_2 \in T \ \cap \ \text{data is sent from } t_1 \text{ to } t_2\}. \end{aligned}$$

When arranging tasks into a task graph, we can either take the temporal aspect into consideration or not. If we do, we get a directed *precedence graph* where the arcs define in which order the tasks are computed. If not, we get a undirected *connectivity graph* where the edges only indicate that communication takes place but not when.

The precedence graph forms a lattice, i.e.

$$\begin{aligned} \exists T_{\text{start}}, \forall T \in G &\mid T_{\text{start}} \prec T \\ \exists T_{\text{end}}, \forall T \in G &\mid T \prec T_{\text{end}} \end{aligned}$$

If this is not true, necessary pseudo tasks are added.

Scheduling algorithms operate on precedence graphs while *mapping algorithms* focus on the assignment of connectivity graphs to processors. We can transform the dependency graph to a precedence graph by merely reversing the arcs and removing tasks that obviously are empty (example in figure 3.2). Furthermore, if we replace the arcs with undirected edges, we get the connectivity graph.

In this report, we sometimes use the term *task graph* instead of the more precise terms *precedence graph* or *connectivity graph* when the context makes the meaning unambiguous.

3.6 Additional Definitions

We take definition of strongly connected components on directed graphs from...

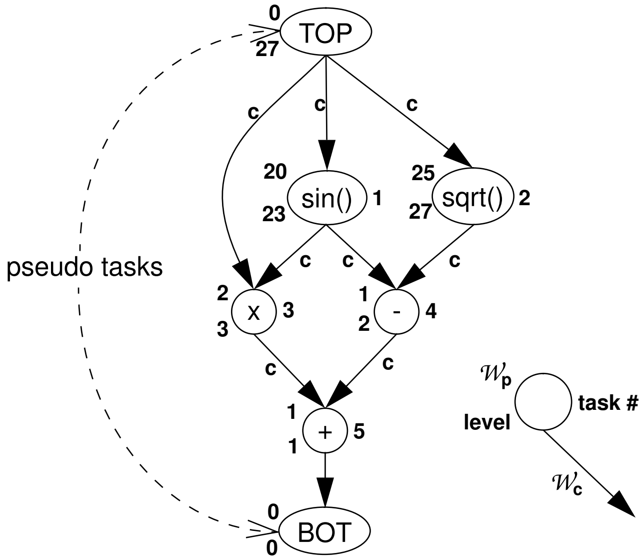


Figure 3.2: A precedence graph, obtained by transforming the data dependency graph in figure 3.1.

Definition 3.7 *A* **strongly connected component (SCC)** *is a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex in the set.*

The arcs that go from a vertex in one SCC to a vertex in another SCC are called *cross-component arcs* and the graph where the vertices consist of the SCCs and the only arcs are cross-component arcs is called the *reduced dependence graph*.

We define cost functions on *task graphs*:

Definition 3.8 *Let* $\mathcal{W}_p(t)$ *represent the* computation cost *of* $t \in T$ *measured in elapsed time.*

Definition 3.9 *Let* $\mathcal{W}_c(c)$ *represent the* communication cost *of* $c \in C$ *measured in elapsed time.*

The cost of a task which consists of only a single operation can be estimated automatically. The Cost for larger, perhaps external, tasks have to be supplied by the user.

We use the following definition for the *granularity* [10] of an arbitrary task graph:

Definition 3.10 *The* **granularity** *of* G *is defined as*

$$g(G) = \min_{t \in T} \{\mathcal{W}_p(t) / \max_{u \in T} \mathcal{W}_c(\langle t, u \rangle)\}$$

Also, we need a name for the longest path in a graph.

Definition 3.11 *A longest path from source to sink is a* **critical path** *of a directed acyclic graph.*

3.12 Parallel Computers

Definition 3.13 *A* **parallel computer** *is a set of interconnected general von Neumann processors.*

Generally, we always have to make considerably simplifications to the communication model to make scheduling and mapping problems tractable. We assume that,

- adequate communication bandwidth is provided,
- all processors are identical, and
- all the processors are interconnected in a clique[1], either by a shared memory or communication links.

We do *not* consider

- routing,
- variations in latencies between different processors, or
- network contention.

3.14 Schedules

Definition 3.15 *A* **schedule** *is a description of the work to be done by each processor at every moment in the scheduled time period.*

We use a Gannt chart to represent a schedule. Each processor has a time axis and executing tasks are marked as intervals on the axes. $\emptyset$ is used to represent idle time (see example in figure 3.3).

Definition 3.16 **Makespan** *is the time of completion of the last task in the schedule.*

[1] every processor is connected to all other processors

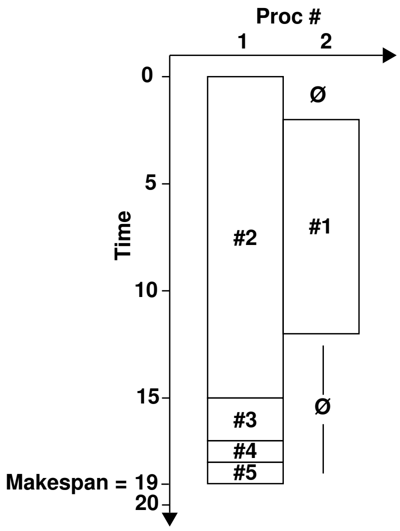


Figure 3.3: Schedule of the task graph in figure 3.2.

Chapter 4

Towards a Parallelising Compiler

"Parallel programming is and will always be performance programming."
Marc Snir, IBM

The main goal of this work is to automatically parallelise applications in which the main computation is to solve a system of ordinary differential equations. By restricting the set of problems that we focus our parallelising efforts upon, domain specific information may assist the compiler to produce a more efficient program. For instance, the communication patterns for an algorithm used to solve a certain set of problems can often guide the analysis in the compiler to produce a result better tailored to execute on a parallel computer.

Moreover, we use ObjectMath as the programming environment. Not only does it support the programmer during the design and implementation phases, but also does the attached computer algebra system facilitate general symbolic expression transformations and partial evaluation at compile time.

4.1 Application Domains

In the field of scientific computing, the applied solving algorithms can be used to classify the problems. For instance, we may observe similarities between different partial differential equation (PDE) solvers and thus may treat PDEs as one class of problems. Another class may be the set of ordinary differential equations (ODEs) and so on, and so forth. Of course, we can elaborate the classification scheme much more, but for our purpose, i.e. to parallelise the solving process, this simple classification is enough.

Henceforth, an *application domain* refers to a class in this classification, e.g. the class of ODEs, while an *application* means a specific problem, e.g. a rolling bearing simulation.

4.2 ODE-Solvers Revisited

Programs that solve ordinary differential equation problems usually consist of

1. a solver and

2. a function (henceforth called RHS) which calculate the derivative of the current state.

The solver is general to the application domain, whereas the RHS is specific to the application.

The possibilities to parallelise the solver itself are omitted here since this is a very difficult to do without weakening the mathematical properties (tolerance, stability, etc.). This is left to the mathematical community. Instead, we focus our efforts on the equation system. By making thorough analysis of the dependencies in the ordinary differential equation system, we may be able to detect parallelism, rearrange and partition the equations accordingly, and thereby reduce the time needed to compute the solutions.

To achieve good performance, we must make as much analysis as possible at compile time, reducing the execution time of the resulting program to a minimum. The more that is known about the problem and algorithms, the more efficient comilation is possible. Therefore, we do not hesitate if we can make use of domain specific information.

4.3 Computer Architecture

We assumed that we have a parallel computer that adhere to the main stream of computers outlined in section 2.1 and that it has an operating system in which it is possible to express task parallelism.

4.4 Coarse or Fine Grained Tasks?

Which size in terms of computational work should the tasks have in a parallel computation?

If we, on the one hand, choose to keep the tasks large, the communication becomes fairly inexpensive, relatively speaking, but the final solution will probably not scale very well due to lack of tasks to keep additional processors busy. If we, on the other hand, focus our parallelising efforts at a fine grained level, the time spent in communication will become large compared to the time spent in computation and we end up with far too much overhead time.

The answer to this question depends, among other things, directly on the given hardware architecture. The level of granularity in the solution should be influenced by the actual cost of communication. If we use a architecture with low latency and high bandwidth, e.g. a shared memory architecture, the size of the tasks may be smaller than if we use a distributed memory architecture with message passing[1]. Therefore, we must have means to specify and/or control the existing parallelism in our problems to fully utilise any given architecture. One extreme is to let the user do everything manually, either by using built-in support for parallelism in the programming language or even by only using external message passing libraries. The other extreme is to hide everything from the user and do all parallelisation automatically. The main approach taken in this work is to make the parallelisation process as automatic as possible. in order to speed up the programming process and be able to handle increasingly complex applications.

[1] We assume here that message passing on distributed memory machine have a higher latency and lower bandwidth compared to a consistent cache protocol in a shared memory machine.

Let us take a look at a few of the parameters that influence the grain size of our computation:

Communication latency is a predetermined parameter that sets a lower limit on the amount of work each task contain. It is useless to communicate with a task which reside on another processor if it is faster invoke the task directly on the same processor. Of course, this implies that tasks may be duplicated.

Communication bandwidth is another predetermined parameter. This sets an upper limit on communicated data. Since we are limiting our efforts to the domain of ordinary differential equations, the data that is communicated is of rather small size. Therefore, we are not taking this parameter into account.

Possibility to duplicate tasks. We can achieve a better result if it is possible to duplicate tasks. For instance, if we have n tasks allocated on separate processors where n is fairly large and all these tasks get some input data from a tiny task, T, it is probably faster if T is duplicated n times and distributed among the processors than if T was evaluated once and the result was communicated to every cluster.

```
  T             T T T
 /|\      ->    | | |
1 2 3 ... n     1 2 3 ... n
```

4.5 Our Efforts

In the search for parallelism, efforts have been made at three different levels:

Equation System Level The equations are gathered in strongly connected components which are solved independently.

Equation Level Each equation forms a separate task. These tasks can be distributed and computed in parallel.

Clustered Task Level We view each subexpression as a separate task and use clustering and scheduling efforts to distribute the work.

The following three chapters (5, 6, and 7) present these levels, one by one.

Sometimes, the dependencies in the calculations can be very intricate or the model presented to the parallelising compiler can be written in an imperative way. In such cases, it is very difficult to automatically parallelise the computation. To facilitate analysis, the programmer can explicitly instruct the parallelising compiler where the control of execution is to be passed from one processor to another. In chapter 8 we look upon an explicit interface to the ObjectMath compiler.

Chapter 5

Equation System Level

In this chapter, we study potential parallelism in ordinary differential equations at the equation system level of granularity. We investigate possibilities to identify tightly coupled equation systems within the problem. If such systems are found, we try to separate them and solve them independently of each other.

5.1 Analysis

We create an *equation dependency graph* by using the equations in the ordinary differential equation system as vertices. Arcs represents dependencies between equations:

Definition 5.2 *Given two first order ordinary differential equations,*

$$\dot{y}_1 = f_1(V_1, t) \tag{5.1}$$
$$\dot{y}_2 = f_2(V_2, t) \tag{5.2}$$

where V_1 and V_2 are minimal sets of state variables. Equation 5.1 is dependent on 5.2 iff $y_2 \in V_1$.

From this graph we extract its strongly connected components. (definition 3.7). The obtained reduced graph is henceforth called the *equation system dependency graph* and is used when we analyse the examples in section 5.4.

Transforming the equation system dependency graph into a task graph, we get an *equation system task graph.* Each vertex in this graph is a task that can compute the right-hand side of the ordinary differential equation system in the corresponding vertex in the former graph. Arcs in this graph represent, as usual in task graphs, data transfers between the tasks.

We attach a solver to each task in the equation system task graph (figure 5.1). The ODE-solver can proceed to compute a solution to the equation system as soon as enough data on the input arcs is present. The order in which the separate systems of are solved is only partially determined by the dependencies. The final ordering of can be made either at compile-time or at run-time depending on characteristics of the model and on wanted properties of the generated program.

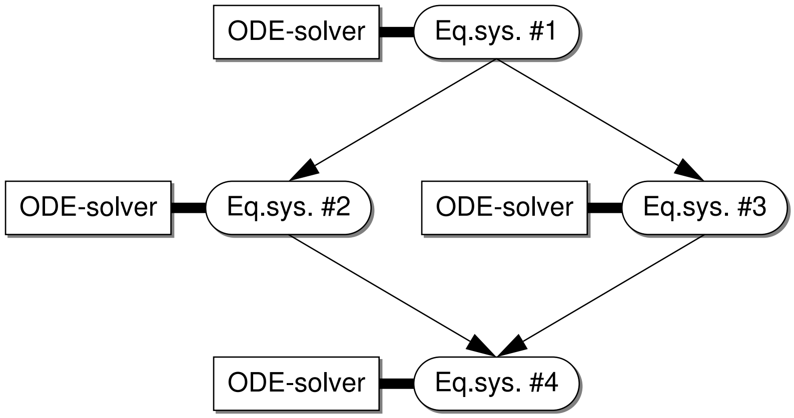


Figure 5.1: Network of equation systems. Each of them with an attached ODE-solver.

Furthermore, splitting up the equation system will have effects on the numerical solver. A reduced size of the equation system will reduce both the sizes in many of its internal structures and the computational work performed in each step. Thereby, its execution time can be shortened.

5.3 Scheduler

Given the equation dependence graph, the scheduler finds an order in which the computation can be done. This is a rather straight forward task.

If the intermediate results between the solution of two systems of ODEs become very large or is not needed by later analysis, we have the opportunity of exploiting pipeline parallelism to accomplish parallel computation with a smaller granularity. In this case we maintain a buffer where we keep the generated, intermediate results which still have not been used by all succeeding ODE-solvers. The synchronisation is done at a lower level[1] and we can also limit the size of the buffer in order to save memory space if the amount of data grows.

5.4 Examples

5.4.1 Servo

As a first attempt to apply the equation system level strategy, we regard a very simple control theory example. It consist of four coupled servos (see figure 5.2). $G_{1,2,3,4}(s)$ are the Laplace transforms of the motors' transition functions. $G_4(s)$ represents a correct servo-motor whereas the three $G_{1,2,3}(s)$ are servo-motors which malfunctions.

$$G_1(s) = G_2(s) = G_3(s) = \frac{1}{s(s+1)} \tag{5.3}$$

[1] The level can actually be arbitrary small. It is based on the resolution of the free variable and varies according to the step sizes in the both solvers.

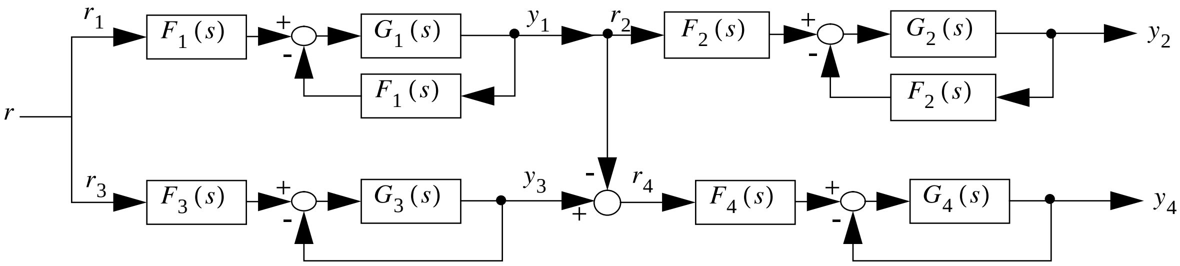


Figure 5.2: Four motor servo example

$$G_4(s) = \frac{1}{s(s+0.5)} \tag{5.4}$$

The regulators $F_1(s), \ldots, F_4(s)$ are similarly defined as

$$F_1(s) = F_2(s) = 29.7\frac{s+1}{s+9} \tag{5.5}$$

$$F_3(s) = F_4(s) = 29.7\left(1 - \frac{8s+28.7}{s^2+9s+29.7}\right) \tag{5.6}$$

The formulae for the regulators and motors are in accordance with reality to a certain degree, but the way the parts are connected together is in this example entirely fictitious. However, this is a good example for showing parallelism at equation system level.

After we have made transformations and subexpression substitutions in ObjectMath, we can study the structure of the equation system dependency graph (visualised in figure 5.3). A full listing of the ordinary differential equations can be found in appendix B). We see that there are eight rather small (1-3 equations) SCCs. Also, we can see their mutual dependences, their direction and which SCCs that are involved. From this, it is easy to determine which variables that represent cross-component arcs and must be kept as intermediate results.

[Code examples]

Fritzson & Andersson [9] presents a slightly different partitioning of this problem and a simulation of the solving process on an iPCS/2.

5.4.2 Rolling Bearing

In this example, we use the bearing simulation model described in section 2.4. From a mathematical point of view, this is a much more complex model compared to the servo example.

As in the previous example, the model is transformed in ObjectMath and the first order ordinary differential equations are extracted. Figure 5.4 shows the equation system dependency graph for one roller. Here we see only two strongly connected components. The small one contains only one equation, the equation that defines the rotation angle, T_1. All the other equations are gathered into one large SCC. Since the roller is perfectly round, T_1 is of

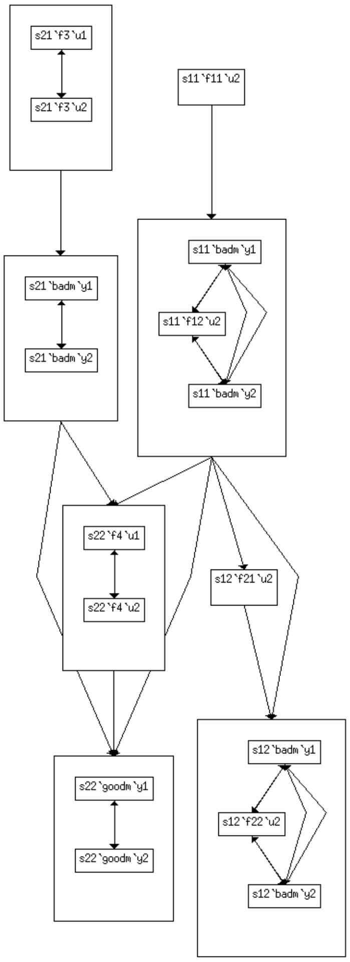


Figure 5.3: The dependencies of the ODEs in the servo example.

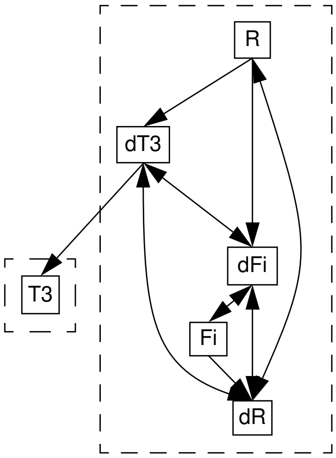


Figure 5.4: The dependencies of the ODEs in the bearing example.

no interest to us, and thus the equation can be discarded. As we add the outer and inner rings and the other rollers, the only SCCs that can be created in this equation dependence graph is the equations for the rotation angle of each roller, $T_{1,2,\ldots,10}$. The rest of the equations are agglomerated into one single SCC, a kernel of equations which is impossible to partition at this level.

5.5 Discussion

Is the dependence analysis we have done useful in our parallelising process? If we assume that it is possible to gain speed by partitioning the ODE-system, then the dependence analysis is always useful. Thus the question we really should ask is, *are there real problems that can be partitioned?*

We identified eight SCCs in the servo example. A modest level of parallelism. In the bearing example, there was only one SCC interesting to us. Also worth noting is that this model originates in a real simulation problem unlike the servo example. To use this analysis as the only guide for parallelisation seems to be futile since a real problem does not yield much parallelism at this level.

Though the dependence analysis appears to be of little use, the analysis can be useful in collaboration with other analyses. Suppose we use an implicit solver and need to calculate the Jacobian matrix. Elements in the matrix which are always zero and not necessary to calculate can easily be deduced from the dependence graph, at compile time. This gives us the possibility to generate a function for computing the Jacobian that is specialised for the current problem.

Another situation where this equation level dependence analysis can be useful appears when we split up the solution temporally. The dependences can be dependent on the free variable (here: the time) and by looking at each part separated from the others, several dependences may disappear. We can select a suitable partitioning by detecting non-continuous functions in the generated right-hand side function. Recall the bouncing-ball example in section 2.2.1: When the ball is not in contact with the surface, the term contributing with the force from the table (second term in the second row of equation 2.2.1) can be ignored.

However, splitting and merging equation systems at runtime affects the solving process. Since we have taken on a black-box view of the solver, we end our analysis of this approach here.

Chapter 6

Equation Level

In this chapter, we view the ordinary differential equations as tasks and try to utilise the parallelism by distributing the computation of these tasks dynamically at runtime.

6.1 Code Generation

The analysis at compile time is quite rudimentary in this version of the code generator. The user is responsible that the equation system has the following form (Mathematica notation):

```
{  { Derivative[1][y[1]][t] == ...
     Derivative[1][y[2]][t] == ...
     ...
     Derivative[1][y[n]][t] == ... }
   { t, tstart, tend } }
```

That is, the user has to make all necessary transformations to reach a state where only first order ODEs in normal form remains. From this state, it is rather straight forward to generate code. See figure 6.1 for a corresponding Fortran 90 code template. There is one function for each ODE, except for the most trivial equations, such as $\dot{y}_i = y_j$. These are gathered into a single function which is not participating in the latter scheduling phase. To be able to dynamically choose which derivative that is currently going to be computed, calls to all generated functions are collected in a select statement in a separate, all-embracing function.

Before any code is generated, common subexpressions are eliminated. There are two reasons why this operation is carried out:

1. *Elimination of recurrent invocations of pure, side effect free functions.*

   The ObjectMath compiler has greater knowledge about user defined functions than the subsequent commodity compiler has. Also, C++ compilers can not optimise calls to standard mathematical functions, such as `sin()` and `sqrt()`. In Fortran-compilers, these functions exist as intrinsics and can be optimised accordingly.

2. *Size reduction of the right-hand side expression.*

The size of the right-hand side expression is often well over the allowed 39 continuation lines in Fortran 90. The only possibility to make a Fortran compiler to accept such large expressions, is to split them into smaller subexpressions. A similar thing occur in C++ compilers. Even if there are no limits on the source code formatting, the compiler's internal symbol table for temporary variables tends to overflow when expressions becomes this large.

```
subroutine rhs(eqno, y, ydot)
  integer              eqno
  real(double), in  :: y(n)
  real(double), out :: ydot
  select case (eqno)
  case (1)
    ydot = rhs_1(y)
  case (2)
    ydot = rhs_2(y)
  ...
  case (n)
    ydot = rhs_n(y)
  end select
end rhs

real(double) function rhs_1(y)
  real(double), in  :: y(n)
  ...
  rhs_1 = ...
end rhs_1

real(double) function rhs_2(y)
  ...
```

Figure 6.1: Code fragment of Fortran 90 code generated from ObjectMath

6.2 Runtime System

The only currently existing runtime system for executing code generated from ObjectMath on a parallel computer in ObjectMath is domain specific. It is designed for solving ordinary differential equation systems and works as follows.

First, the numerical ordinary differential equation solver is assigned to a "supervisor" node. The solver is started on this node and remains there throughout the whole computation. Any internal states and data inside the solver are ignored.

Second, computation servers, "workers", that are compliant to orders from the supervisor, are started on all remaining processors. They follow a simple schema:

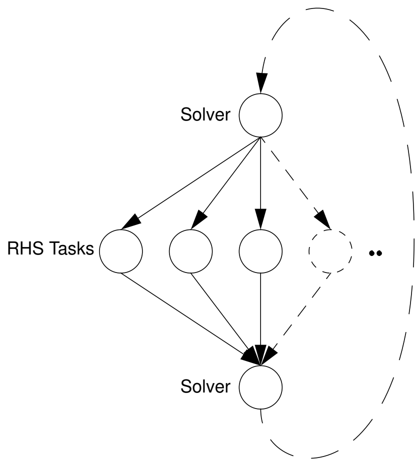


Figure 6.2: A task graph of an ordinary differential equation solution step in the current version of ObjectMath. All right-hand side tasks are independent of each other.

```
init
FOREVER {
  get orders
  IF (finished) THEN exit
  FOREACH request IN orders
    start timer
    do request
    stop timer
  END
  compose reply
  reply
END
```

The supervisor takes on the rôle as the client and send computation request to the workers. A scheduler in the supervisor is responsible for balancing the load of the workers equally (see figure 6.3).

6.2.1 Task Distribution

The distribution of tasks can be done in two different ways; Either we do it as dynamically as possible as in the *Supervisor–Worker model*, or we try harder to minimise the amount of data that is sent around as in the *Semi-Dynamic Task Assignment model*.

Supervisor–Worker model

This is an execution model that is rather easy to implement. It has the most dynamic scheme where a supervisor process hands out one task at a time until all tasks in the task graph are executed. A *worker* executes its assigned task, hands back the result to the supervisor and

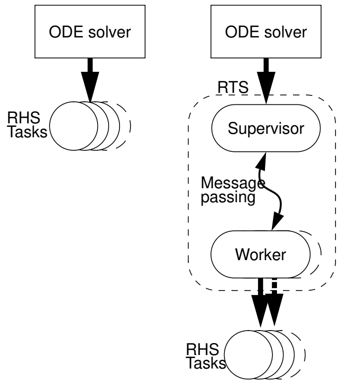


Figure 6.3: The serial version and the parallel runtime system of ordinary differential equation slution in ObjectMath. Neither the solver nor the right-hand side tasks are influenced by the parallel runtime system.

continues by listen for further orders. In this model, difficulties in data placement can be suppressed by keeping all data on the supervisor by default and copy the needed data to and from the workers at the same time the tasks are handed out and the result is returned. The communication latency will require rather large tasks since the time lag from that a worker has finished executing one task until it receives a new task is not easy to hide.

Also, we might end up sending the same data items to and from one processor if two related tasks are executed by the same worker. This last inefficiency may be possible to partially circumvent by additional runtime analysis made by the supervisor.

Semi-Dynamic Task Assignment Model

This model is domain specific since it utilises the computational behaviour of the problem, In this model, the task assignment and data paths are determined by the scheduler beforehand. When the schedule has been determined, all tasks are distributed to their assigned workers. After all tasks have been executed and the results have been collected, the task assignment might be altered. I.e., the assignment is static throughout one simulation step but may change in between. The scheduling algorithm induces increased execution time overhead but we can limit this by taking domain knowledge into account.

From the laws of physics of the simulated bodies, we assume that there is locality in the values of the state variables and, thus, in the execution cost of the tasks. In other words, things will remain the same to some extent. On these grounds, we can decrease the frequency of the dynamic rescheduling and make several simulation steps using the same task assignment and without great loss in efficiency.

6.2.2 Load Balancing

To accomplish load balancing, the result packets are augmented with the measured execution time of each task (figure 6.4). The scheduler uses this information as the estimated cost to

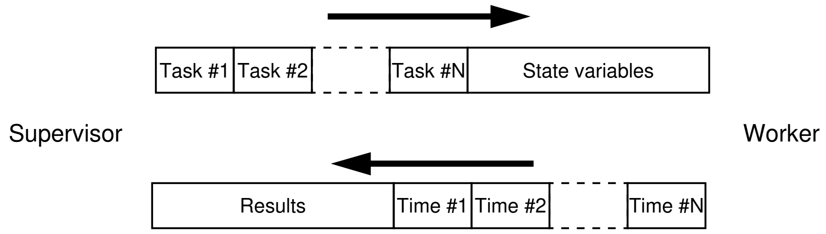


Figure 6.4: Contents of the packets sent between the supervisor and the workers.

execute the same task during the next step. If the scheduler causes too much overhead in execution time by rescheduling every step, the scheduler may only sum up the measured times and calculate an average execution time for the last N invocations. It will then reschedule every N:th call to the right-hand side. The value of N is currently set by the user but could be controlled by the scheduler.

We use Coffman's [6] *largest-processing-time* (LPT) scheduling algorithm for this scheduling problem, His algorithm simply schedules the tasks in size order:

> The largest task among the tasks that have not been scheduled yet is assigned to the worker that has least work to do. This is repeated until all tasks are assigned to a processor.

The upper limit on the execution time of this straightforward algorithm is

$$\frac{t_{\text{sched}}}{t_{\text{optimal}}} = \frac{4}{3} - \frac{1}{3m}$$

where m is the number of workers.

6.3 Bearing Example

As mentioned earlier, out of the 64 state variables in the model, half of them are trivial to calculate. They are gathered in one single task. The remaining 32 state variables, three for each roller plus two for the outer ring, involve calculations of contact forces. The main problem here is that the outer ring may be in contact with each and every roller. Therefore, the outer ring's computation has to incorporate the calculation of possible contacts with every roller. Thus, the size of the expressions related to the outer ring tend to be N times the roller's expression size (N is the number of rollers). This will result in a rather unbalanced schedule when the number of processors is increased. This was one of the problems that initiated the task clustering approach (chapter 7).

6.3.1 Measurements

The generated code have been executed on both a Parsytec GC/PP-128 (appendix A) and a Sun SPARC Center 2000 (figure 6.5). Maximum performance on the Parsytec GC/PP-128 (330 invocation of the right-hand side per second) is reached when using four processor.

The performance drops as we add more processors. This behaviour is caused by the high communication cost on this machine. The Sun SPARC Center 2000 is a shared memory architecture and has, relatively speaking, a much lower communication cost. Its optimum performance for this example lies above the available 20 processors.

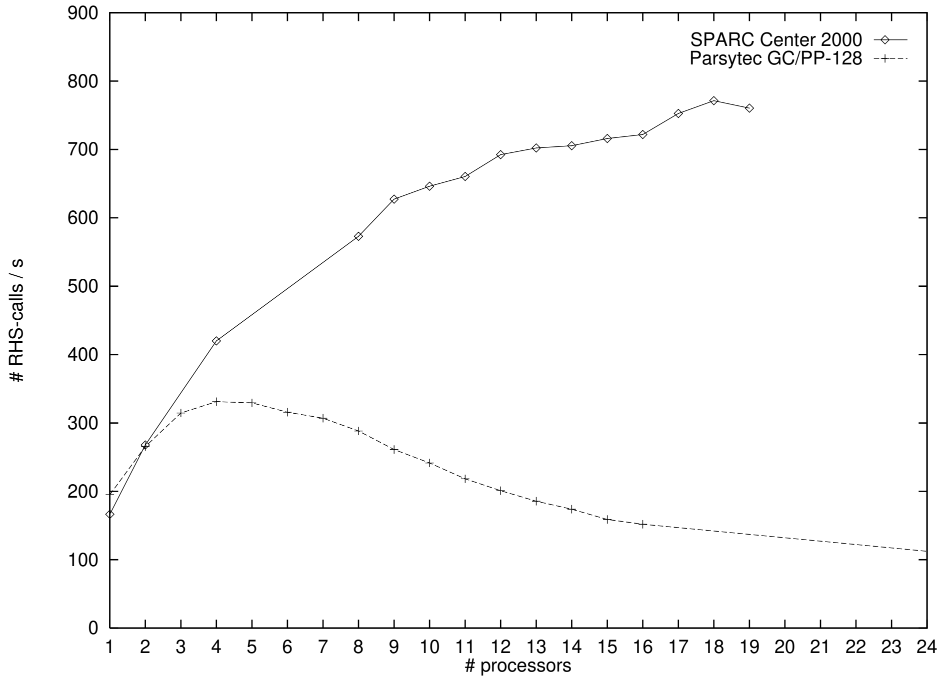


Figure 6.5: Speedup curves for the 2D bearing example computed on the Sun SPARC Center 2000 and the PARSYTEC GC/PP-128.

Chapter 7

Clustered Task Level

In this chapter we focus on fine grained task graphs. We view the atomic operations in the specified problem model as tasks and use task graph algorithms both at compile time and at run time to attain an efficient computation of the problem.

The work is divided into two parts:

1. Clustering the graph into a coarse grained graph which can be
2. efficiently scheduled onto a parallel computer.

This division enables us not only to reduce the intertask communication but also to adapt dynamic characteristics of the problem by, in the later stage, using different scheduling algorithms. The clustering is made once at compile time and thus not as time critical as the following scheduling stage. The scheduling stage takes place repeatedly at runtime to encompass the dynamic characteristics of the problem (figure 7.1). Both stages strive for the best possible solution to the problem, i.e. the shortest schedule and the best utilisation of the computer.

We name the first part the *Clustering stage* and the second part the *Scheduling stage*. In the following two sections we will take a closer look on these two separate stages.

7.1 Clustering Stage

If the *granularity* of the task graph is very small, communication and scheduling is far too expensive compared to the computational work of a single task. We must, therefore, gather tasks in the task graph into *reasonably* large units of computation, *clusters*, before target code is generated. But how do we gather tasks into clusters and what is a *reasonable* size of a cluster?

The later scheduling stage does not enforce any tight constraints on the result of the clustering algorithm. If the clustering algorithm performs badly, the computational performance after the scheduling is likely to be bad also. Since the two stages (clustering and scheduling) are separated and we want to minimise the makespan of the final schedule, i.e., the output of the scheduling stage, it is difficult to express a good cost function which can be used in the clustering algorithm.

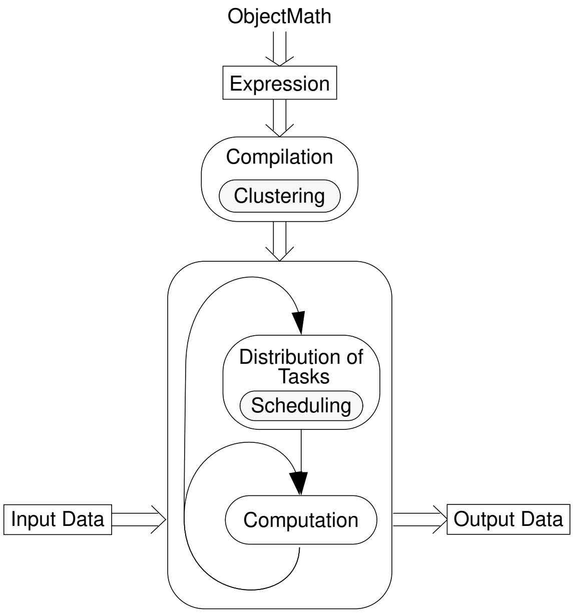


Figure 7.1: Where the clustering and scheduling takes place in our approach.

7.1.1 Implementation

An implementation of clustering algorithms is carried out in an experimental version of ObjectMath. (We do not want to incorporate anything in the regular ObjectMath environment until it has been well tested.) As in the previous approaches, the model is parsed and internal structures are built. A data dependency graph is built on top of the existing internal form which is immediately transformed into a task graph. We have now enough internal structures for implementing the clustering algorithms.

Current efforts are focused on the *List Scheduling* and *Dominant Sequence Clustering* algorithms (outlined in section 2.5). Since the runtime system is not completed yet, we evaluate the efficiency of the generated clustering using the calculated total *makespan*. That is, we do not actually "run" a clustered task graph but instead make an estimation of its execution time assuming a computer with an infinite number of nodes.

7.2 Scheduling Stage

If we have acquired a good clustering what is then the point of scheduling? Is it not better to make a greater effort at compile time and then remove the expensive runtime scheduling stage? Nevertheless, due to various dynamic characteristics such as varying cost of computations ($\mathcal{W}_p$), dynamic load balancing, varying computer sizes, parameterised models, etc. this is not possible. Therefore the scheduling stage is retained.

This section presents the needed dynamic characteristics of a runtime scheduler and what algorithms can be used. But first, we enlarge upon the dynamic characteristics of the scheduling problem.

7.2.1 Characteristics

As mentioned, the following dynamic characteristics of the problem inhibit us from making a final schedule at compile time:

Varying cost of computations. Expressions may contain several conditional expressions with predicates that cannot be determined at compile time. If a task contains such an expression, the cost (= execution time) is impossible to estimate beforehand.

Load balancing. As a result of difficulties in cost estimation there are also problems regarding load balancing. By measuring the execution time and redistributing the tasks at run time, we can achieve a better load balance than if we perform this at compile time.

Varying computer sizes. We need to execute the simulation program on parallel computers with different numbers of processors without recompilation.

Parameterised models. The used application may be parameterised. That is, at compile time, the size of the task graph will be unknown. Instead it will consist of N copies of a known task graph, where N is determined at runtime.

7.2.2 A Runtime Scheduler

From the dynamic characteristics of the problem we can conclude that an efficient runtime scheduler is necessary to minimise the total execution time. However, this introduces more problems:

Execution time. A runtime scheduler must induce minimal overhead on the execution time. We have to limit the time complexity of the chosen scheduling algorithm to keep the overhead as low as possible.

Message passing. So far we have not touched upon the underlying communication software. Since the runtime scheduler may dynamically reschedule a task to another processor, it must also handle marshalling[1] and transferring of data to and from the tasks.

Furthermore, to save communication bandwidth, various optimisations, such as gathering data from several tasks into one network packet may be applied to decrease the communication time.

Clearly, if the clustered tasks are dynamically relocatable without any limitations, the message passing problem will call for a substantial amount of extra execution time just figuring out which data items should be sent where.

The runtime system proposed here is domain specific since it utilises the computational behaviour of the problem. The task assignment and data paths are determined by the scheduler before any task is executed. After all tasks have been executed, the task assignment might be altered. That is, the assignment is static throughout one simulation step but may change

[1] packing and unpacking of structured data

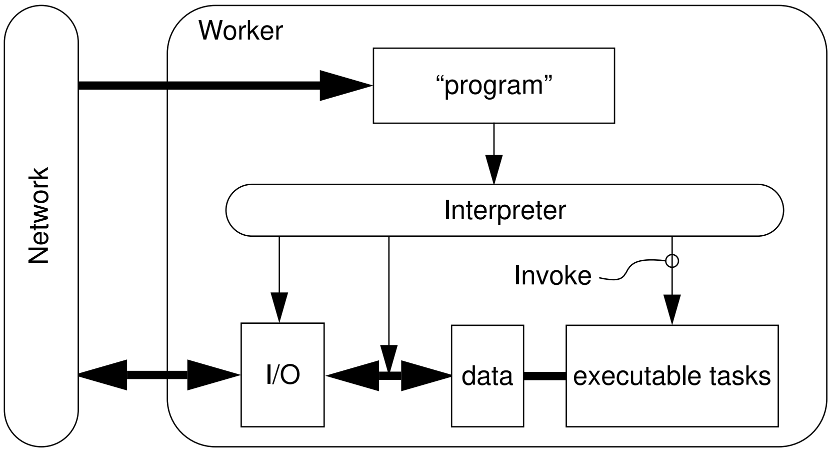


Figure 7.2: Structure of a more general runtime system

in between. The scheduling algorithm and the message passing problem induces increased execution time overhead but we can limit this by taking domain knowledge into account (Cf. section 6.2.1).

7.2.3 Implementation

A new runtime system is currently being implemented. It will assimilate the information from the preceding clustering stage and manage to distribute its given task in a transparent and efficient manner. The load balance is retained using a scheduler as in the current ObjectMath environment. The kernel of the runtime system is built as a tiny and very simple interpreter. It understands a small set of specialised instructions such as receive and send data, pack and unpack variables, measure the execution time, etc. Thus, the scheduling activities can be concentrated to one processor which will minimise the amount of scheduling information stored on every node/processor (see figure 7.2 and 7.3). This will eliminate the inconsistency problems that arise when dealing with distributed algorithms.

Of course, this approach does not preclude distributed scheduling and load balancing algorithms. We can still implement such algorithms if, e.g., we otherwise would encounter difficulties in scalability. Furthermore, the "interpreter" is easy to expand with more instructions in the future. For instance, we may want to incorporate support for interrupts and remote handlers.

Currently, the *List Scheduling* algorithm (described in section 2.5.1) is used to schedule the tasks.

Although implementing an interpreter as the kernel of a runtime system is a general technique and is easy to extend, there are drawbacks. The interpreter top loop will itself induce a certain amount of overhead on the execution time. By limiting the size of the instruction set and not implementing variables, memory management and other time consuming features, the overhead can be kept small.

The following thing can be hidden from the user:

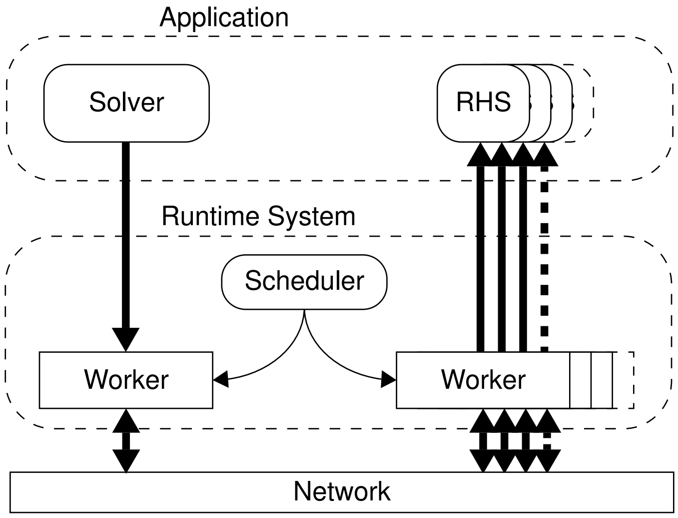


Figure 7.3: An example of how an ordinary differential equation application interfaces to our runtime system

- Data placement
- Scheduling algorithms
- Communication

Data Placement

To efficiently make data available at the right moment to the right task, the runtime system must be able to dynamically relocate data as well as tasks. Therefore, for scheduling purposes, we need to globally identify a variable and its location. A general approach is to implement some sort of limited shared memory concept. To make this efficient, it must be supported by the code generator.

Scheduling Algorithms

Communication

The amount of information that is sent by the runtime system tends to be small. Just a few variables for each task. To minimise network latency, data items that are sent at the same point in time, having the same sender and receiver, are packed together and presented as one data item to the network. The (un)packing can either be done by the runtime system or by the supplied message passing library depending on which is the most efficient. Another detail regarding the communication topic is whether messages should be tagged or not. Further investigation needs to be done to evaluate suitable alternatives.

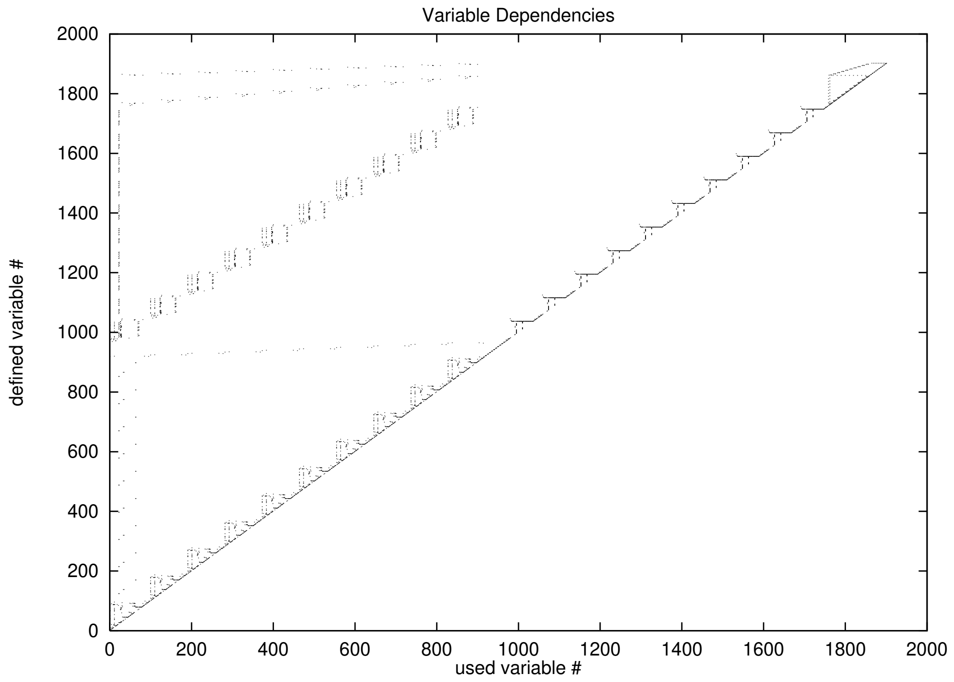


Figure 7.4: Data dependencies of every subexpression in the 2D bearing example.

7.3 Example: 2D Bearing Simulation

We apply the clustering effort on the 2D bearing model outlined in section 2.4. Since we have not completed the implementation of the new runtime system, there is no results from execution of the clustered and scheduled program. Nevertheless, we evaluate our clustering efforts by using the makespan estimation in the clustering stage,

7.3.1 Subexpression Dependencies

Let us have a look on the dependency graph of all subexpressions. Since the number of nodes in the graph exceed 2000, the dependency graph is not included in this report. Instead, the adjacency matrix is shown in figure 7.4.

First, we note that the matrix is sparse. There are in fact, rather few dependencies present. Second, one can see that there still is a structure in the calculation by just looking at the matrix. Third, the dependency graph is very shallow and wide. The longest path in the graph contains 14 nodes while the total number of nodes (= subexpression) is 2020.

Additionally, one can detect that there are temporary variables which are used in several expressions (appears as dots on the same horizontal line within the matrix). These consist mostly of tiny expressions such as calculating the square of a state variable or the like. To achieve an efficient parallelisation, such small expressions should be duplicated whenever its result is needed.

The lower section of the matrix ($V \leq 918$) consist of variables associated to the computation of each rollers radial acceleration. In a small slice ($918 < V \leq 968$), their angular acceleration is calculated. Each roller's spin is calculated using dependencies in the upper area ($968 < V \leq 1758$). And the last part of the dependency matrix ($1758 < V \leq 1902$) is when the outer ring's accelerations are computed.

7.3.2 Execution Cost

As an estimation of execution and communication costs ($\mathcal{W}_p$ and $\mathcal{W}_c$) used in the clustering algorithms, we have made measurements on a PARSYTEC GC/PP-128 (Appendix A). The communication latency in this machine is in the range of 5000–6000 times it take to make a single floating point addition. If we use these figures in the scheduling algorithms, we would end up with one single task after the clustering stage.

We intentional overlook the fact that each made communication will consist of several dependencies. It is difficult to take this into consideration in the algorithms but we may simulate its influence by an appropriate, general reduction of the communication cost.

7.3.3 Algorithms

Starting with the *List Scheduling* algorithm, we have used the level as priority and scheduled highest level first using estimated time (HLFET in [1]). In figure 7.5, we have invoked the scheduler with different estimation of the communication cost to investigate its influence on the makespan. At $\mathcal{W}_c \approx 3800$ the makespan make make a large drop. This indicates that larger clusters are formed by agglomerating smaller ones, eliminating some levels of communication. Still, the makespan of the serial schedule is as low as 8071 and is reached only when $\mathcal{W}_c < 1200$ units. When $\mathcal{W}_c$ pass 5600, the communication is too expensive compared to the computation in the tasks and the schedule collapse into a single serial task. Also, we see a similar drop in the number of used schedules (virtual processors) at $\mathcal{W}_c \approx 3800$. From 3800 to 5600 the scheduler uses only 16–30 clusters.

We see the same behaviour when we use the *Dominant Sequence Clustering* algorithm. The drop of the makespan is even more emphasised and the numbers of clusters falls to a constant value of 23 (figure 7.6). Also, this algorithm has a generally lower makespan as well as lower maximum number of clusters than the previous one. Even so, the makespan climbs up to values around three times the makespan of the serial schedule.

7.3.4 Conclusion

Optimising algorithm, as these algorithms are, should detect the optimum (here: serial schedule) earlier. One reason is the common subexpressions. Although these algorithms find shared expressions and introduce communication whenever it is necessary to keep the makespan short, trivial subexpression should not be shared at all. They should be duplicated. In these cases, the communication is far too expensive compared to the computation. Frequent sharing of such small tasks has a negative influence on the makespan. To solve this, we have either to introduce a new phase, before the clustering stage, where small tasks are duplicated on beforehand, or to use a clustering algorithm which can duplicate tasks when needed.

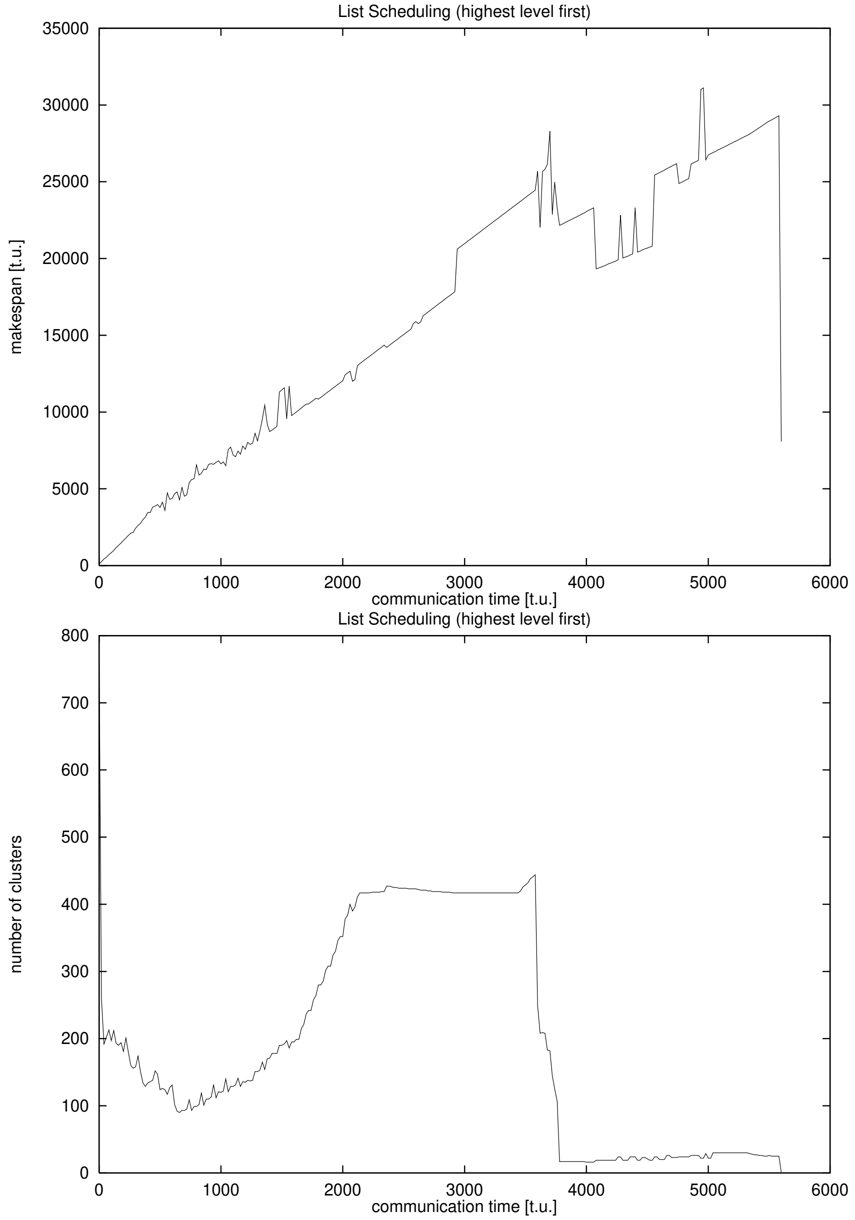


Figure 7.5: *makespan* and numbers of clusters in the *List Scheduling* algorithm using various communication latencies in the 2D bearing example. t.u. (= time unit) is the approximate time it takes to perform one floating point addition.

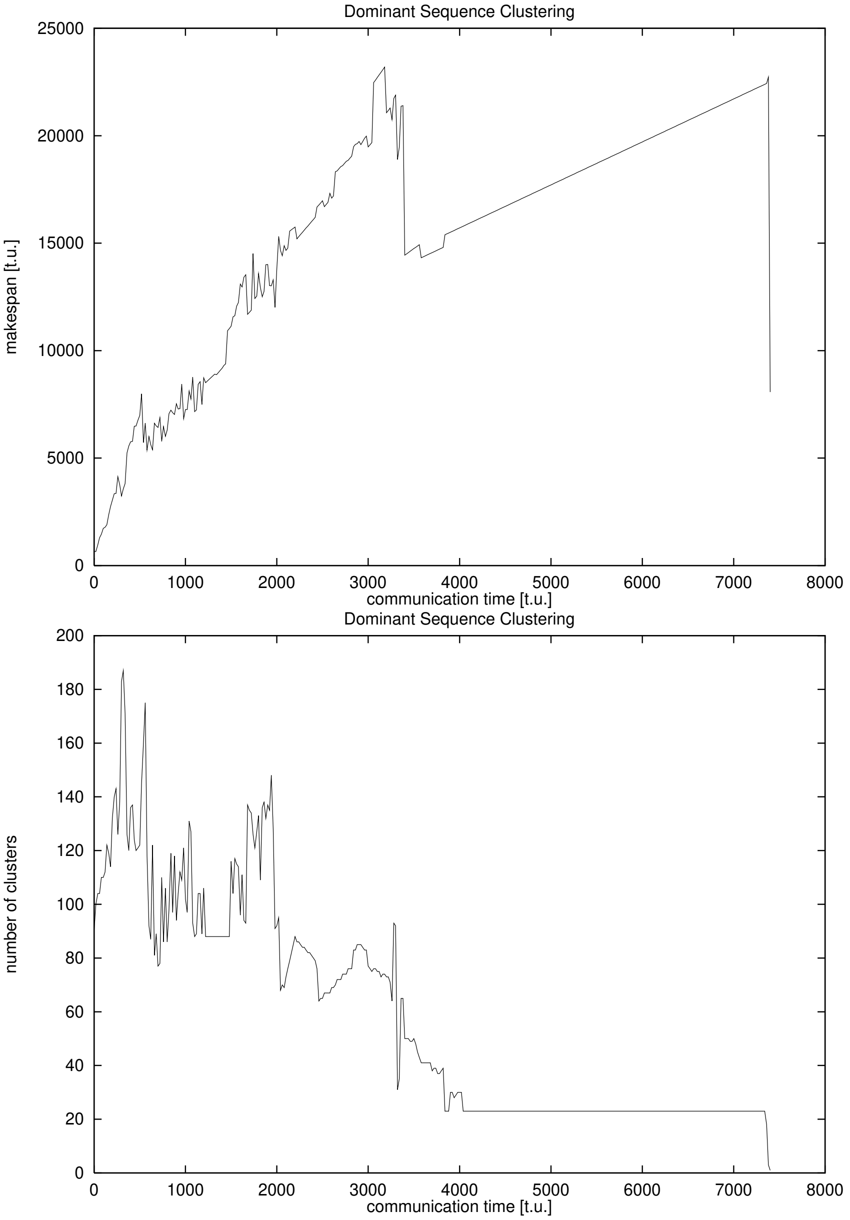


Figure 7.6: *makespan* and numbers of clusters in the *Dominant Sequence Clustering* algorithm using various communication latencies in the 2D bearing example. t.u. (= time unit) is the approximate time it takes to perform one floating point addition.

Another reason of the poor result is the fine granularity of the task graph. Neither *List Scheduling* nor *Dominant Sequence Clustering* performs well on fine grained tasks. We have to investigate and analyse what actually happens when parameters such as granularity reach unexpected (small) values. Are there any well defined limits of these parameters? We have to widen our view and investigate and evaluate other algorithms in our search for an adequate result.

Chapter 8

Explicit Parallelism

In contrast to the automatic or semi-automatic approaches presented in the previous three chapters, the parallelism here is explicitly induced by the user.

There are situations where the user explicitly wants to instruct the runtime system where and when the computation is to be performed in parallel. If the user expresses the problem using imperative constructs in ObjectMath, the compiler has no means to parallelise the problem. Since the approaches described in the previous chapters deal only with *pure* expressions, the user must encapsulate used imperative constructs in side-effect free, pure functions. In this chapter we leave the functional approach and instead describe an explicit interface to the runtime system which enables the user to gain more control of how the parallel computation should proceed.

In this chapter, code examples are written in Fortran 90 and High Performance Fortran.

8.1 Language Extensions

To explicitly express parallelism, ObjectMath's language is extended with a ForAll-construct:

`ForAll[`*expr*`, `*iterator(s)*`]`

The expression, *expr*, must be an assignment and *iterator(s)* is a single iterator or a list of iterators. Example:

```
ForAll[ y[[i]] := rhs[x[[i]], t], {i, 1, 20}]
```

Here, 20 instances of the `rhs` function are invoked in parallel, each with its own element of the vector `x`. The return values is assigned to the vector `y`.

To simplify the code generation, there must be no dependencies among the different instances of the right-hand side of the assignment.

The `ForAll` does not affect the data distribution used in computations on a parallel computer. Currently, this decision is left to the runtime system. In the ODE case, the data is preferably kept centralised, while it may be distributed in other cases.

8.2 Compiling ForAlls

There are three alternatives when generating code for this construct:

1. serial Fortran 90 code for one processor,
2. parallel Fortran 90 code which interface to a message passing runtime system, and
3. data parallel HPF code.

8.2.1 Serial Fortran 90 code

In this case, code is generated by iterating over the iterator variables in a number of for-loops assigning temporary variables and, afterwards, assigning the real variables in a new set of for-loops. Generated code fragment for the previous example:

```
do om_i=1,20
  om_T1(om_i) = foo(x(om_i), t)
end do
do om_i=1,20
  y(om_i) = om_T1(om_i)
end do
```

8.2.2 Parallel Fortran 90 code

In this case, a runtime system is supplied that takes care of the message passing and also schedules the tasks (here: the instances of the `ForAll` loop. The following code is generated from the Mathematica `ForAll` expression above. Types for communicated data:

```
common /<model>/ om_P1_id
integer om_P1_id

type om_P1_in
  sequence
  integer(om_int) p1
  real(om_real) p2
  real(om_real) p3
end type om_P1_in

type om_P1_out
  sequence
  real(om_real) p1
end type om_P1_out
```

Extra variables:

```
external om_parallel

type(om_P1_in) om_T1(n)
type(om_P1_out) om_T2(n)
```

The `ForAll` loop:

```
do i=1,20
  om_T1%p1 = i
  om_T1%p2 = x(i)
  om_T1%p3 = t
end do
call om_parallel(om_P1_id, 20, om_T2, om_T1)
do i=1,20
  y = om_T2%p1
end do
```

And finally, the expression:

```
subroutine om_P1(om_res, om_arg)
  use om_<model>
  implicit none
  type(om_P1_in) om_arg
  type(om_P1_out) om_res

  !   Unpack args
  i = om_arg%p1
  x(i) = om_arg%p2
  t = om_arg%p3

  ! Evaluate the body
  y(i) = rhs(x(i), t)

  !   Pack result
  om_res%p1 = y

end subroutine om_P1
```

The communicated variables are unpacked to the same variables in the receiver as they were packed from in the sender. That is, the environment for the expression will be the same irrespective of the used processor. Additionally, there must be a subroutine that initialise the runtime system with information about message sizes and available functions.

In the example, the expression that was evaluated in parallel was a single function. The scheme could easily be expanded to include arbitrary expressions by encapsulating these in an automatically generated function.

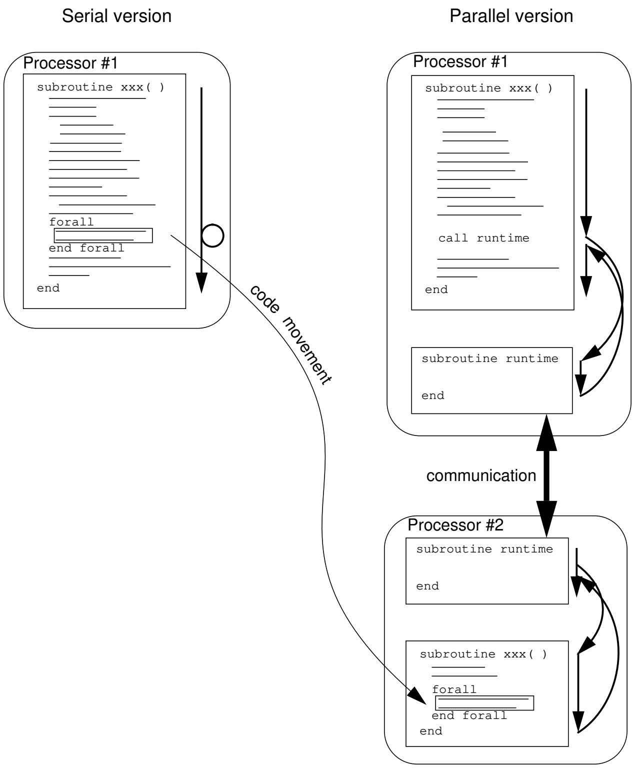


Figure 8.1: Example of how generated parallel code is restructured compared to the serial version

8.2.3 High Performance Fortran (HPF)

In High Performance Fortran the runtime system is already supplied and there is data parallel constructions in the language [16]. Nevertheless, there are several constraints in HPF which make these difficult to use when generating code from ObjectMath. An example:

> All procedures that are invoked inside an HPF `ForAll` statement have to be *pure*. Inside such a procedure, global variables must not be changed (since this violates the *pureness* of the procedure). In the ObjectMath compiler, variables that are not local to a function but may be local inside an object, are transformed into global variables. Therefore, a model that is modular in the object oriented sense may not be possible to parallelise.

Moreover, several HPF compilers today only implements a subset of HPF and/or do not utilise the parallelism that exists in the sources (e.g., the `!HPF$ independent` directive is often ignored).

There are two possibilities to generate HPF code:

1. use the `ForAll` statement, or
2. use the `!HPF$ independent` directive.

In the first alternative the code will look like this:

```
forall (om_i=1:20)
  y(om_i) = rhs(y(om_i), a(om_i), a)
end forall
```

The function `rhs` (and all functions and subroutines that `rhs` invokes) must be declared pure.[1] [20]

In the other alternative, the generated code is identical to serial code except for the independent declaration:

```
! HPF$ independent
  do om_i=1,20
    om_T1(om_i) = rhs(x(om_i), t)
  end do
  do om_i=1,20
    y(om_i) = om_T1(om_i)
  end do
```

This version eases the constraints on the body of the loop. To conform to High Performance Fortran, the independent directive must be true; different instances of the loop must not interfere.

[1] A pure subroutine is not permitted at the top level in a ForAll construct. However, a pure function may invoke a pure subroutine.

Chapter 9

Summary

This work has focused on automatic parallelisation of ordinary differential equation applications.

An ordinary differential equation system consisting of 64 equations was used as a test example. The equations originate in a simplified 2D rolling bearing simulation. They have been transformed, using a computer algebra system, to a form which allowed the application of a numerical ordinary differential equation solver.

The potential parallelism in the ordinary differential equation system was investigated on three different levels:

- Equation system level (chapter 5),
- Equation level (chapter 6), and
- Clustered Task level (chapter 7).

On the equation system level the degree of parallelism is rather low. Our efforts at this level seems futile, at least when we consider the bearing test application. Moving to the equation level, the possible parallelism increase slightly. Still, dependencies in the given problem often prohibit parallelism and reduce the achieved performance. Finally, at clustered task level, the parallelism is radically increased. However, the size and connectivity of the obtained dependency graph makes graph partitioning and scheduling intractable.

The clustering algorithms *List Scheduling* and *Dominant Sequence Clustering* was evaluated. Both algorithms showed poor performance on the entangled, fine grained task graph which was extracted from the bearing example. Using *List Scheduling* as the scheduling algorithm in the runtime system turned out to be a good choice.

Furthermore, a new runtime system is currently being implemented. To make this as general as possible, it is built on a kernel containing a tiny interpreter.

To handle code written in an imperative way, an explicit, parallel language construct, ForAll, was introduced. Also, this is useful when the user want to have more control over the distribution of the parallel computation.

9.1 Future Work

First, a better algorithm for the clustering stage has to be implemented. As indicated in section 2.5, there are several other algorithms and methods to evaluate. In particular, we must consider task duplication possibilities. Unless an algorithm is found that can produce adequate clustering of the transformed task graph, the approach of this work has to be altered.

Second, the new runtime system has to be completed. The reason is not only to be able to better evaluate the result from the clustering stage, but also to be able to define the actual cost of the clustered task graph. We can not define the cost if we do not know exactly how the task graph can be computed. The runtime system can be used in other application domains as well.

Other runtime systems should be investigated. Lately, several parallel languages and programming models has emerged, many of them with a built-in runtime system. What functionality do they offer? Perhaps there are similarities we can take advantage of.

In the explicit interface, updates to global data is handled in a very primitive way or not at all. Further analysis of data dependencies would facilitate a more automatic and efficient distribution of data.

Moreover, the only data items that are communicated in the parallel Fortran 90 code generated from the an explicit `ForAll` are the arguments and results to the expression, i.e. variables that occur inside the `ForAll` expression. A more user-friendly approach would be to analyse the dependencies between the functions and possibly generate code that maintains consistency between global variables in the different processors used.

Looking further into the future, other application domains should be supported. For instance, partial differential equation problems is a large application domain which often allow a high degree of parallelism.

Appendix A

Parsytec GC/PP-128

The Parsytec GC/PP-128 is a parallel computer with distributed memory. The computer consists of 64 nodes interconnected in a mesh topology. Each node consists of two 80 MHz PowerPC 601 processors and, for communication, four T805 transputers. On each node there is 32 Mbyte of memory.

Computation Cost

These figures are only very rough approximations based on measurements. We only regard floating point operations since this comprise the greater part of an ordinary differential equation application. The figures are normalised to the time it takes to perform a single floating point addition.

operation	*timeunits*
assign	0
$+, -, \times, >, \geq, <, \leq$	1
$\div$	4
$\sqrt{x}$	11
x^y	25
$\sin, \arcsin, \sinh, \cos, \arccos, \cosh, \exp, \log$	15
$\tan, \arctan, \tanh, \cot, \coth, \sec, \csc$	30

Communication Cost

latency	141 μs
bandwidth	3.2 Mbyte/s

Appendix B

Servo Example

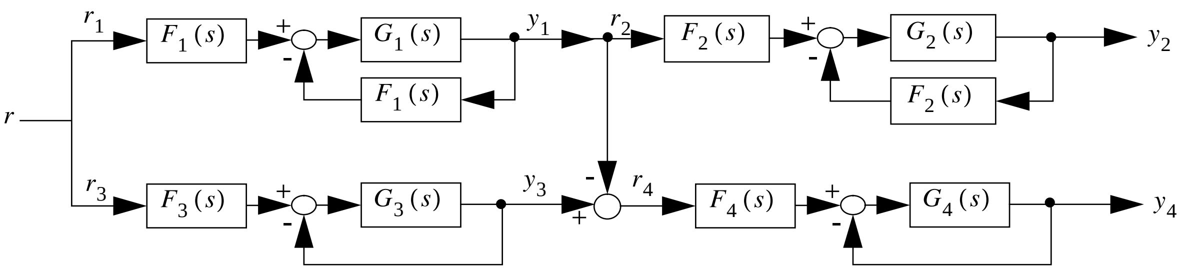


Figure B.1: Four motor servo example

Motors

There are two different motors. G_4 is the original motor for which all regulators were designed. G_1, G_2, and G_3 are motors which are malfunctioning. Their response time is lower.

$$G_1(s) = G_2(s) = G_3(s) = \frac{1}{s(s+1)} \tag{B.1}$$

corresponds to

$$\dot{x_a} = u \tag{B.2}$$
$$\dot{x_b} = u - x_b \tag{B.3}$$
$$y = x_a - x_b \tag{B.4}$$

and

$$G_4(s) = \frac{1}{s(s+0.5)} \tag{B.5}$$

corresponds to

$$\dot{x_a} = 2u \tag{B.6}$$
$$\dot{x_b} = 2u - 0.5x_b \tag{B.7}$$
$$y = x_a - x_b \tag{B.8}$$

Regulators

$$F_1(s) = F_2(s) = 29.7\frac{s+1}{s+9} \tag{B.9}$$

corresponds to

$$\begin{aligned} \dot{x_a} &= 29.7u && \text{(B.10)} \\ \dot{x_b} &= 8u - 9x_b && \text{(B.11)} \\ y &= x_a - x_b && \text{(B.12)} \end{aligned}$$

and

$$F_3(s) = F_4(s) = 29.7\frac{s^2+s+1}{s^2+9s+29.7} \tag{B.13}$$

corresponds to

$$\begin{aligned} \dot{x_a} &= x_a && \text{(B.14)} \\ \dot{x_b} &= x - 9x_b - 29.7x_a && \text{(B.15)} \\ u &= 29.7(x - 8u_a - 28.7u_b) && \text{(B.16)} \end{aligned}$$

Bibliography

[1] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[2] P. Bouvry, J. Chassin de Kergommeaux, and D. Trystram. Efficient solutions for mapping parallel programs. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *EURO-PAR'95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 379–390. Springer-Verlag, August 1995.

[3] Francois E. Cellier. *Continuous System Modeling*. Springer-Verlag, 1991.

[4] Vipin Chaudhary and J. K. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Transaction on Parallel and Distributed Systems*, 4(3):328–346, March 1993.

[5] Yi-Ling F. Chiang, Ji-Suing Ma, Kuo-Lin Hu, and Chia-Yo Chang. Parallel multischeme computation. *Journal of Scientific Computing*, 3(3):289–306, 1988.

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

[7] R. Van Driessche and D. Roose. An improved spectral bisection algorithm and its application to dynamic load balancing. *Parallel Computing*, 21:29–48, 1995.

[8] Dag Fritzson and Peter Fritzson. Equational modeling of machine elements — applied to rolling bearings. Technical Report LiTH-IDA-R-91-05, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, March 1991.

[9] Peter Fritzson and Niclas Andersson. Generating parallel code from equations in the ObjectMath programming environment. In Jens Volkert, editor, *Parallel Computation*, volume 734 of *Lecture Notes in Computer Science*, pages 217–232. Springer-Verlag, 1993.

[10] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *IEEE Transaction on Parallel and Distributed Systems*, 16:276–291, 1992.

[11] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transaction on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

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

[13] B. Hendrickson and R. Leland. An improved spectral load balancing method. In *6th SIAM Conf. on Parallel Processing for Scientific Computing*, pages 953–961, Philadelphia, 1993. SIAM.

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

[15] Edwin S. H. Hou, Nirwin Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transaction on Parallel and Distributed Systems*, 5(2):113–120, February 1994.

[16] High Performance Fortran Forum (HPFF). High performance fortran language specification, May 1993.

[17] M. Ashraf Iqbal and Shahid H. Bokhari. Efficient algorithms for a class of partitioning problems. *IEEE Transaction on Parallel and Distributed Systems*, 6(2):170–185, February 1995.

[18] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. Technical Report 95-037, University of Minnesota, Department of Computer Science, Minneapolis, June 1995.

[19] George Karypis and Vipin Kumar. Parallel multilevel graph partitioning. Technical Report 95-036, University of Minnesota, Department of Computer Science, Minneapolis, June 1995.

[20] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* Scientific and Engineering Computation Series. The MIT Press, 1994.

[21] Boontee Kruatrachue and Ted Lewis. Grain size determination for parallel processing. *IEEE Software*, pages 23–32, January 1988.

[22] Joseph Y-T Leung, Oliver Vornberger, and James D. Witthoff. On some variants of the bandwidth minimization problem. *SIAM Journal on Computing*, 13(3):650–667, August 1984.

[23] Virginia Mary Lo. Heuristic algorithms for task assignment in distributed system. *IEEE Transaction on Computers*, 37(11):1384–1397, November 1988.

[24] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32(9):1073–1078, September 1989.

[25] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning. Probably a TechReport, August 1994.

[26] Chao-Wei Ou, Sanjay Ranka, and Geoffrey Fox. Fast and parallel mapping algorithms for irregular problems. Probably a TechReport, December 1994.

[27] 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.

[28] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Reasarch Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, Massachusetts, 1989.

[29] Chien-Chung Shen and When-Hsiang Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transaction on Computers*, c-34(3):197–203, March 1985.

[30] Gilbert C. Sih and Edward A. Lee. Declustering: A new multiprocessor scheduling technique. *IEEE Transaction on Parallel and Distributed Systems*, 4(6):625–637, June 1993.

[31] Harold S. Stone. Multiprocessor scheduling with aid of network flow algorithms. *IEEE Transaction on Software Engineering*, SE-3(1):85–93, January 1977.

[32] P. J. van der Houwen and B. P. Sommeijer. Iterated Runge-Kutta methods on parallel computers. *SIAM J. Sci. Stat. Comput.*, 12(5):1000–1028, September 1991.

[33] Peter Värbrand. *Generalized Assignment Type Problems: Models and Solution Procedures.* Ph.D. dissertation, Linköping University, September 1988.

[34] Lars Viklund, Johan Herber, and Peter Fritzson. The implementation of ObjectMath — a high-level programming environment for scientific computing. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction — 4th International Conference, CC '92*, volume 641 of *Lecture Notes in Computer Science*, pages 312–318. Springer-Verlag, 1992.

[35] Stephen Wolfram. *Mathematica — A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, second edition, 1991.

[36] Wolfram Research, Inc, P.O. Box 6059, Champaign, IL, 61826-6059, USA. *MathLink Reference Guide*, 1993. Version 2.2.

[37] Tao Yang. *Scheduling and Code Generation for Parallel Architectures.* Ph.D. dissertation, Rutgers, The State University of New Jersey, May 1993.

[38] Tao Yang and Apostolos Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transaction on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[39] Michalewicz Zbigniew. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer Verlag, 1994.