Software ENgineering

SEN

Parallel, Distributed-Memory Implementation of a Sparse-Grid Method for Time-Dependent Advection-Diffusion Problems

C.T.H. Everaars, F. Arbab, B. Koren

REPORT SEN-R0306 AUGUST 31, 2003

ISSN 1386-369X

Parallel, Distributed-Memory Implementation of a Sparse-Grid Method for Time-Dependent Advection-Diffusion Problems

ABSTRACT
A workable approach for modernizing existing software into parallel/distributed applications is through coarse-grain restructuring. If, for instance, entire subroutines of legacy code can be plugged into a new structure, the investment required for the re-discovery of the details of what they do can be spared. The resulting renovated software can then take advantage of the improved performance offered by modern parallel/distributed computing environments, without rethinking or rewriting the bulk of their existing code. Our approach is simple and is in fact a cut-and-paste method. First, we try to identify and isolate components in the legacy source code (the cut). Second, we glue them together by writing coordinator modules (glue modules) with the help of a coordination language (the paste). We have used Manifold as the glue language. Manifold is a general purpose coordination language developed at CWI (Centrum voor Wiskunde en Informatica) in the Netherlands and is specially designed to express cooperation protocols among components in component based systems. Our point of departure is an existing sequential C code for computational fluid dynamics (CFD). This C source code deals with a time-dependent advection-diffusion problem solved with a sparse-grid technique. Applying our cut-and-paste method to this program results in a generally applicable coordinator module that can restructure our sequential programs into a parallel application (i.e. it can run on a shared memory machine) as well as a distributed application (i.e. it can run on a cluster of workstations). We also give some performance results.

2000 Mathematics Subject Classification: D3.3, D.1.3, D.3.2, F.1.2, I.1.3
1998 ACM Computing Classification System: 68N15, 68Q10, 65N50, 65N55, 65N99, 76M25, 76N15
Keywords and Phrases: Parallel and distributed computing, coordination languages, software renovation, software re-usability, protocol library, multi-grid methods, sparse-grid methods, computational fluid dynamics, advection-diffusion problems, sparse-grids combination techniques.

Parallel, Distributed-Memory Implementation of a Sparse-Grid Method for Time-Dependent Advection-Diffusion Problems

C.T.H. Everaars, F. Arbab and B. Koren
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
email: Kees.Everaars@cwi.nl, Farhad.Arbab@cwi.nl and Barry.Koren@cwi.nl

August 18, 2003

ABSTRACT
A workable approach for modernizing existing software into parallel/distributed applications is through coarse-grain restructuring. If, for instance, entire subroutines of legacy code can be plugged into a new structure, the investment required for the re-discovery of the details of what they do can be spared. The resulting renovated software can then take advantage of the improved performance offered by modern parallel/distributed computing environments, without rethinking or rewriting the bulk of their existing code. Our approach is simple and is in fact a cut-and-paste method. First, we try to identify and isolate components in the legacy source code (the cut). Second, we glue them together by writing coordinator modules (glue modules) with the help of a coordination language (the paste). We have used Manifold as the glue language. Manifold is a general purpose coordination language developed at CWI (Centrum voor Wiskunde en Informatica) in the Netherlands and is specially designed to express cooperation protocols among components in component based systems.

Our point of departure is an existing sequential C code for computational fluid dynamics (CFD). This C source code deals with a time-dependent advection-diffusion problem solved with a sparse-grid technique.

Applying our cut-and-paste method to this program results in a generally applicable coordinator module that can restructure our sequential programs into a parallel application (i.e. it can run on a shared memory machine) as well as a distributed application (i.e. it can run on a cluster of workstations). We also give some performance results.

2000 ACM Computing Classification: D3.3, D.1.3, D.3.2, F.1.2, I.1.3.
2000 Mathematics Subject Classification: 68N15, 68Q10, 65N50, 65N55, 65N99, 76M25, 76N15.

Keywords and Phrases: Parallel and distributed computing, coordination languages, software renovation, software re-usability, protocol library, multi-grid methods, sparse-grid methods, computational fluid dynamics, advection-diffusion problems, sparse-grids combination techniques.

Note: Work carried out under the projects Coordination Languages (SEN3) and Computing and Control

(MAS2). Funding for this project was provided by the National Computing Facilities Foundation (NCF), under project number NRG 2001.06.

Contents

1 Introduction

A key area in software modernization is renovating aging software systems to take advantage of today's parallel and distributed computing environments. Interestingly, not all "aging software" consists of the dusty decks of the so-called legacy systems inherited from the programming projects of the previous decades. A good deal of such software is still being produced today in on-going programming projects that, for one reason or another, prefer to use a tried and true language like ANSI C with which they have gained some expertise, rather than to struggle their way through uncharted territories of parallel and distributed programming tools and languages such as PVM, PARMACS, MPI, or even High-Performance C. A good deal of both categories of such software can benefit from a restructuring that allows them to take advantage of the increased throughput offered by the modern parallel or distributed computing platforms.

A workable approach for modernizing such existing software into parallel/distributed applications is through coarse-grain restructuring. If, for instance, entire subroutines of legacy code can be plugged into a new structure, the investment required for the re-discovery of the details of what they do can be spared. The resulting renovated software can then take advantage of the improved performance offered by modern parallel/distributed computing environments, without rethinking or rewriting the bulk of their existing code. Our approach is simple and is in fact a cut-and-paste method. First, we try to identify and isolate components in the legacy source code (the cut). Second, we glue them together by writing coordinator modules (glue modules) in a coordination language (the paste). We have used Manifold as the glue language. Manifold is a general purpose coordination language especially designed to express cooperation protocols among components in component based systems.

Our point of departure is an existing sequential C code for computational fluid dynamics (CFD). This C source code deals with a time-dependent advection-diffusion problem solved with a sparse-grid technique and was developed at CWI by a group of researchers in the department of Modeling Analysis and Simulation, within the framework of the NWO-funded project "Sparse Grid Methods for Transport Problems".

The developers of the program found their algorithms to be effective (good convergence rates) but inefficient (long computing times). As a remedy, they looked for methods to restructure their code to run on multi-processor machines and/or to distribute their computation over clusters of workstations.

Applying our cut-and-paste method to the program results in *one* generally applicable coordinator module that can restructure the sequential program into a parallel application (which can run on a shared memory machine) as well as a distributed application (which can run on a cluster of workstations).

The rest of this paper is organized as follows. In §2 we describe the sparse-grid method for a time-dependent advection-diffusion model problem. This section can be read independently from the rest and can be skimmed (or skipped) without problems. In §3 we give a brief introduction to the **MANIFOLD** language. In §4 we present the simplified pseudo-program as distilled from the original ANSI C program, explore its structure and try to identify and isolate its software components. This leads us to a new concurrent scheme for the simplified pseudo-program. In §5, we describe the paste phase in the software renovation process and present our generic gluing modules written in the **MANIFOLD** coordination language. The actual restructuring of the original sequential program can be found in §6. In §7 we compare the performance results before and after the restructuring. Finally, the conclusion of the paper is in §8.

2 Introduction to the Problem

2.1 Sparse-Grid Techniques

Systems of partial differential equations of advection-diffusion-type play a prominent role in, e.g., the mathematical modeling of pollution of the atmospheric air, surface water and ground water. The nature of these equations and the necessity of modeling transport over long time spans, require very efficient algorithms. In the past, much research has been done on developing efficient solvers, tailored integrators for stiff systems of ordinary differential equations and other time stepping techniques. This has already led to significant progress. However, for realistic modeling, computer capacity (computing time and memory) is still a severe limiting factor. This limitation is felt particularly in the area of global air and water pollution modeling with its huge numbers of grid points at each of which many calculations must be carried out. To greatly reduce the number of grid points, of course without loss of accuracy and in combination with an appropriate, efficient time-stepping process, a so-called sparse-grid technique may be applied. A derivation of the complexity and accuracy estimates of sparse-grid methods is given in [1].

2.2 Sparse-Grid Combination Techniques

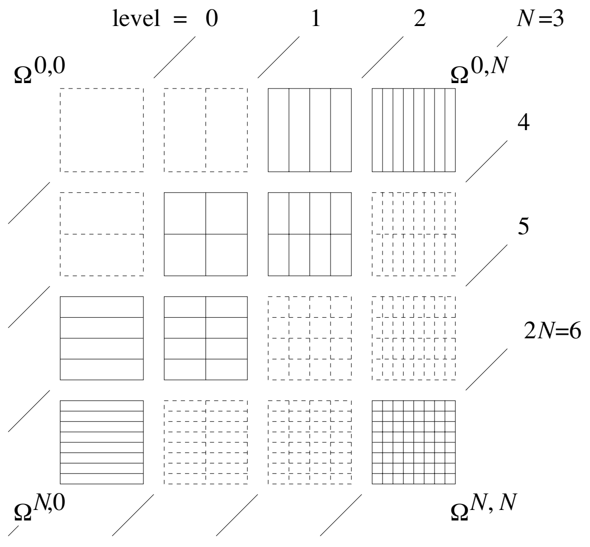
In [2], Griebel, Schneider and Zenger show that the sparse-grid complexity and representation error can also be achieved by the so-called sparse-grid combination technique. This technique combines solutions on conventional grids of different mesh widths in different directions into a representation on the conventional, full grid. The coefficients of the combination are chosen such that there is a canceling in leading-order error terms. The combination technique is attractive because, asymptotically, it can yield a smaller spatial error for a given complexity than a single-grid approach can [3, 1]. Consider a problem of spatial dimension d that is solved on a single grid with spatial discretization of order p, i.e., on a single grid with mesh width h the spatial error is $O(h^p)$. The single-grid problem has a complexity $O(h^{-d})$. With the combination technique, a spatial error of order $O(h^p(\log h)^{d-1})$ can be obtained with a complexity $O(h^{-1}(\log h)^{d-1})$, i.e., an asymptotically first-order complexity with only a slightly larger error than for the single-grid approach. An advantage of the combination technique relative to the sparse-grid technique, as introduced in [1], is that the former involves a straightforward discretization and solution of the partial differential equations on conventional grids while the latter requires discretization through a set of hierarchical basis functions, leading to a linear algebra problem with nearly full matrix. Convergence proofs of the combination technique for elliptic problems are given in [3, 4], error analyses can be found in [5]. In [6], a pointwise error analysis is given for the representation error that is inherent in the combination technique.

In the combination technique, problems are solved on conventional grids. These problems are all independent of each other. Therefore, the combination technique is inherently parallelizable. For a successful parallelization, see [7]. Examples of other publications on distributed computing for sparse-grid-type methods are [8, 9, 10, 11, 12, 13, 14, 15, 16].

2.3 The Current Sparse-Grid Combination Technique

In sparse-grid combination techniques, several solutions on different grids are combined to obtain a solution which has the accuracy corresponding to a much finer grid. The current two-dimensional combination technique is based on a family of grids as shown in Figure 1. Grids in the family of grids are denoted by $\Omega^{l,m}$ where superscripts label the level of refinement relative to the *root grid* $\Omega^{0,0}$. The mesh-widths in

x- and y-direction of $\Omega^{l,m}$ are $h_x = 2^{-l}H$ and $h_y = 2^{-m}H$, where H is the mesh width of the uniform root grid $\Omega^{0,0}$. We denote the mesh width of the *finest grid* $\Omega^{N,N}$ by h. The mesh sizes h_x and h_y are dependent on the particular grid $\Omega^{l,m}$ in the family of grids, h is not. In the time-dependent combination



Notation	Description
$\Omega^{N,N}$	finest grid of mesh width $h = 2^{-N} H$
$\Omega^{l,m}$	semi-coarsened grid, of mesh widths $h_x = 2^{-l} H$ and $h_y = 2^{-m} H$
u	continuous, exact solution
$R^{l,m}$	restriction operator that maps onto $\Omega^{l,m}$
$P^{N,N}$	prolongation operator that maps onto $\Omega^{N,N}$
$U^{l,m}$	semi-discrete approximate solution on $\Omega^{l,m}$

Figure 1: Grid of grids.

technique the initial profile $u(x, y, 0)$ is restricted, by injection, to the grids $\Omega^{N,0}$, $\Omega^{N-1,1}$, $\cdots$, $\Omega^{0,N}$ and to $\Omega^{N-1,0}$, $\Omega^{N-2,1}$, $\cdots$, $\Omega^{0,N-1}$. Then, independent of each other, these rather coarse representations are all integrated in time. Next, at a chosen point in time, the coarse approximations are prolongated with q-th order interpolation onto the finest grid $\Omega^{N,N}$, where the integrated solutions are combined to obtain a more accurate solution. The notation is summarized in Figure 1.

Starting from the exact solution u, the combination technique as introduced in [2] constructs a grid function $\widehat{u}^{N,N}$ on the finest grid $\Omega^{N,N}$ in the following manner:

$$\widehat{u}^{N,N} \equiv \sum_{l+m=N} P^{N,N} R^{l,m} u \; - \sum_{l+m=N-1} P^{N,N} R^{l,m} u. \tag{1}$$

The corresponding so-called *representation error* $r^{N,N}$ is

$$r^{N,N} \equiv \widehat{u}^{N,N} - R^{N,N} u. \tag{2}$$

Analogously, assuming exact time integration and semi-discrete solutions $U^{l,m}$ resulting from a spatial discretization, the combination technique constructs an approximate solution $\widehat{U}^{N,N}$ on the finest grid $\Omega^{N,N}$ from the coarse-grid approximate solutions according to

$$\widehat{U}^{N,N} = \sum_{l+m=N} P^{N,N} U^{l,m} - \sum_{l+m=N-1} P^{N,N} U^{l,m}. \tag{3}$$

Let $d^{l,m}$ denote the discretization error on grid $\Omega^{l,m}$, i.e.,

$$d^{l,m} \equiv U^{l,m} - R^{l,m} u. \tag{4}$$

Then the total error $e^{N,N} = \widehat{U}^{N,N} - R^{N,N}u$ in $\widehat{U}^{N,N}$ is written as

$$e^{N,N} = r^{N,N} + \widehat{d}^{N,N}, \tag{5}$$

where the *combined discretization* error $\widehat{d}^{N,N} = \widehat{U}^{N,N} - \widehat{u}^{N,N}$ is given by

$$\widehat{d}^{N,N} = \sum_{l+m=N} P^{N,N} d^{l,m} - \sum_{l+m=N-1} P^{N,N} d^{l,m}. \tag{6}$$

In [6] the representation error $r^{N,N}$ is analyzed. In [17], a detailed derivation is given of the combined discretization error $\widehat{d}^{N,N}$ for pure advection problems, and in [18] for advection-diffusion problems.

2.4 The Current Time Integration Method

Here, time integration is done implicitly, with the Rosenbrock solver ROS3. This scheme is a variation to the ROS2 scheme presented in [19] and belongs to a family of schemes discussed on p. 233 of [20]. The ROS3-scheme is third-order accurate and A-stable. Because our spatially discrete problems are stiff due to the diffusion term, A-stability is a desirable property. The ROS3-scheme is factorized such that the third-order accuracy is preserved. In the current work we use directional factorization, separating the horizontal and vertical coupling. This leads to attractive savings in the required computational work since it reduces the two-dimensional linear algebra to one-dimensional linear algebra. The step size of the ROS3-scheme is controlled such that the solution error is fixed at some *tolerance* during the time integration. For further details about the scheme we refer to [18] and also to [21].

2.5 The Model Problem

We consider the constant-coefficient advection-diffusion equation

$$u_t + u_x - \varepsilon\,(u_{xx} + u_{yy}) = 0 \tag{7a}$$

on the spatial domain $[-1,1] \times [-1,1]$ and take $u(x,y,0) = 0$ as initial solution. As boundary conditions we impose

$$u(-1,y,t) = \begin{cases} 0, & y < 0 \\ \frac{1}{2}, & y = 0 \\ 1, & y > 0 \end{cases}, \quad u_y(x,\pm 1,t) = 0, \quad u(1,y,t) = 0. \tag{7b}$$

For $\varepsilon = 10^{-2}$ the solution at $t = 1$ is shown in Figure 2. It possesses a horizontal and a vertical grid-aligned solution layer. The thickness of both layers is proportional to $\sqrt{\varepsilon}$ as $\varepsilon \to 0$.

3 The Manifold Coordination Language

3.1 An Overview

In this section, we give a brief overview of **MANIFOLD**[1]. **MANIFOLD** is used to develop concurrent software, regardless of whether it runs on a parallel or a distributed platforms. **MANIFOLD** is not a

[1]For more information, refer to our html pages located at http://www.cwi.nl/projects/manifold/manifold.html.

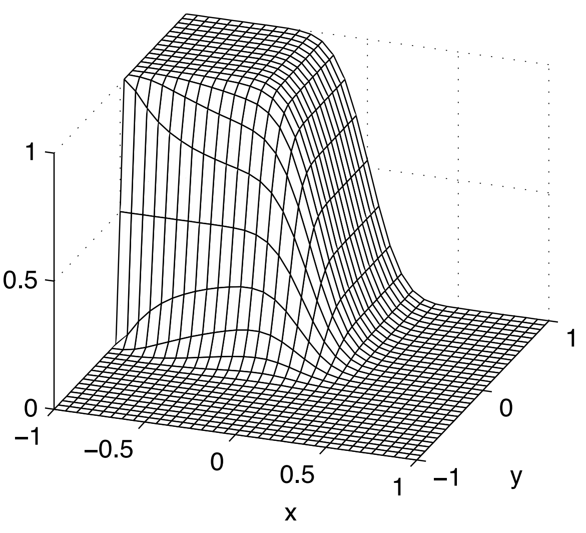


Figure 2: Solution of model problem 1 at $t = 1$ for $\epsilon = 0.01$.

parallel programming language; it is a *coordination language* as opposed to a *computation language* [22]. **MANIFOLD** is a *complete* language (as opposed to a language extension, like Linda [23]) for programming the cooperation protocols of concurrent systems. These protocols describe the routing of the information between various processes that comprise a concurrent application, and the dynamic changes that take place in such routing network in reaction to events.

MANIFOLD is based on the IWIM (*Idealized Worker Idealized Manager*) model of communication [24]. The basic concepts in the IWIM model (and thus also in **MANIFOLD**) are *processes*, *events*, *ports*, and *channels* (in **MANIFOLD** called streams). In IWIM, a process can be regarded as a *worker* process or a *manager* (or coordinator) process. An application is built as a (dynamic) hierarchy of worker and manager processes. Lowest in the hierarchy are pure worker processes that do not do any coordinating activities. Highest in the hierarchy are pure coordinators. A process between the lowest and highest level may consider itself a worker doing a task for a manager higher in the hierarchy, or a manager coordinating processes lower in the hierarchy.

Programming in **MANIFOLD** is a game of dynamically creating process instances and (re)connecting the ports of some processes via streams (asynchronous channels), in reaction to observed event occurrences. Its style reflects the way one programmer might discuss his interprocess communication application with another programmer on a telephone (let process a connect process b with process c so that c can get its input; when process b receives event e, broadcast by process c, react to that by doing this and that; etc.). As in this telephone call, processes in **MANIFOLD** (in this case b and c) do not explicitly send or receive messages to or from other processes. Processes in **MANIFOLD** are treated as black-boxes that can only read or write through the openings (called ports) in their own bounding walls. It is the responsibility of a worker process to perform a (computational) task. A worker process is not responsible for the communication that is necessary for it to obtain the proper input it requires to perform its task (it simply reads this information from its own input port), nor is it responsible for the communication that is necessary to deliver the results it produces to their proper recipients (it simply writes this information to its own output port). In general, *no process in IWIM is responsible for its own communication with other processes.* It is always the responsibility of a third party—a coordinator process or *manager*—to arrange for and to coordinate the necessary communications among a set of worker processes. This third party sets up the communication channel between the output port of one process and the input port of another process, so that data can flow through it. This setting up of the communication links from the *outside* (exogenous coordination) is very typical in **MANIFOLD** and has several advantages. One important advantage is that it results in a clear separation of the modules responsible for computation (the workers) from the modules responsible for coordination (the managers). This strengthens the modularity and enhances the re-usability of both types of modules (see [25, 24, 26]).

A **MANIFOLD** application consists of a (potentially very large) number of processes that run as threads bundled up (automatically or under user control) in one or more operating-system-level processes (called task instances in **MANIFOLD**). The different task instances in a **MANIFOLD** application can run on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming languages. Some of them (the so-called non-compliant atomic processes) may not know anything about **MANIFOLD**, nor the fact that they are cooperating with other processes through **MANIFOLD** in a concurrent application.

The **MANIFOLD** system consists of a compiler called **MC**, a runtime system library, a number of utility programs, libraries of built-in and predefined processes [27], a link file generator called **MLINK** and a runtime configurator called **CONFIG**. **MLINK** uses the object files produced by the (**MANIFOLD** and other language) compilers to produce link files needed to compose the application-executable files for each required platform. At the runtime of an application, **CONFIG** determines the actual host(s) where the processes that are created in the **MANIFOLD** application will run.

The system has been ported to several different platforms (e.g., IBM RS60000 AIX, IBM SP1/2, Solaris, Linux, Cray, and SGI). The system was developed with emphasis on portability and support for heterogeneity of the execution environment. It can be ported with little or no effort to any platform that supports a thread facility functionally equivalent to a small subset of the Posix threads, plus an inter-process communication facility roughly equivalent to a small subset of PVM [28].

The **MANIFOLD** system automatically takes care of the data conversion necessary for communication in a heterogeneous environment. These conversions are only done when the receiving process really attempts to use the data. When data are simply to be passed on to another process on another machine, conversion is not necessary and does not take place.

The **MANIFOLD** terminology is given in the next section. It is beyond the scope of this paper to present all the details of the syntax and semantics of the **MANIFOLD** language. For that we refer to [27].

3.2 The Terminology

Coordination processes are always written in the **MANIFOLD** language and are called manifolds. A manifold definition is a process type, a template from which we can make process instances (i.e., manifold processes). It consists of a *header* and a *body*.

The header of a manifold begins with keyword `manifold`, followed by its name, the number and types of its parameters, and the names of its input and output ports that are used for information exchange with other process instances.

The body of a manifold definition can be written in the **MANIFOLD** language (in which case the body is a *block*), or as an ordinary C function. A manifold whose body is a block is called a *regular* manifold. When its body is written as a C function it is called an *atomic* manifold. Atomic manifolds interface with the **MANIFOLD** world through a special ANSI C interface library.

The inner logic of a block is always expressed in terms of an *event driven finite state machine.* In this machine, a finite number of states are defined, each defining a sequence of actions. An event in **MANIFOLD** is considered as an atomic message, broadcast by a manifold process in its environment or internally posted within the manifold process itself. Occurrences of broadcast events can be picked up by **MANIFOLD** processes in this environment (i.e., by all running processes in the same **MANIFOLD** application) in which case they are stored in their private event memories. Based on the events found in its event memory and some other conditions (see below) the finite state machine of the manifold process jumps from one state to another and performs its associated actions. Because, initially, there is a high priority event, named `begin`, available in the event memory of a process instance, the first state visited in a state machine (thus in a block) is the so-called `begin` state, and its actions are performed. This event driven jumping from one state to another goes on and on until the state machine arrives in some termination state.

Syntactic Structure of a Block

Syntactically a block consists of an optional *local declaration part* followed by a finite number of *states* (at least the `begin` state should be present). It is easy to recognize a block because it is always placed between a pair of curly braces (the symbols { and }).

In the local declaration part of a block, we can declare events (we want to use in that block), we can create (and activate) process instances from manifolds by using the syntactic construct *process* `process_name` *is* `manifold_name`, and we can use declarative statements like `save`, `hold`, `ignore`, and `priority` (we explain their meaning later as we need them) and more.

Each state has a *state label* and *state body* separated by a colon. The label of a state defines the *necessary* condition under which a transition to that state is possible. It is an expression that can match observed event occurrences in the event memory of the manifold instance. The simplest state label is the name of an event. If we have an event with name `x` in the event memory of the executing manifold and we have a state with the state label `x`, then the necessary condition for a state transition to the `x` state (i.e., the state with the state label `x`) is fulfilled. Whether or not the state transition really takes place depends also on two other conditions as we will see below. The body of a state can be:

a) a block

b) a control structure

c) a pipeline

Transition to a state whose body is a block causes the running process instance to enter the block and make a transition to its `begin` state. Because we are already in a block, this action results in a nested block. Blocks can be nested arbitrarily deep. Note that with the possibility of nested blocks we can set up our state machine in a modular way as a set of substate machines (as we will do), each with its own scope rules (indicated by its braces of the blocks). Because of the possibility to use nested blocks, states to which we jump are not necessarily always found in the current block. State transition to states in ancestor blocks of the current block are also allowed. All state transition rules in **MANIFOLD** are well defined and are intuitive as we will see later when we explain the source code.

All familiar control constructs like conditional "if" constructs, loop constructs, and the "this after that" operator ";" are available in the **MANIFOLD** language. All these constructs are constructed out of the standard block and event handling mechanisms of the **MANIFOLD** language. Syntactically a control construct is equivalent to a block.

A pipeline is syntactically one of the following four constructs

1) an expression
2) a primitive action
3) a connection specification of processes
4) a number of pipelines separates by commas and enclosed in a pair of parentheses forming a so called *group*.

An expression is a sequence of actions that, optionally, yields a single value. The value of an expression, if any, is a process, a port of a process, a manifold, an event, or a manner call (see below).

The primitive actions are the basic operations in **MANIFOLD**. The most important (primitive) actions we can perform in pipelines are (1) creating and activating process instances, (2) broadcasting events (with the action `raise`) or putting them in a process' own event memory (with the action `post`), (3) connecting processes to each other by setting up streams among their ports (by the action denoted by the arrow `->`) (see below).

In its most simple form a connection specification of processes looks like `a -> b`, where `a` and `b` are process names. With this notation we denote that the (the output port) of process `a` is connected to the (input port of) process `b` by the primitive action denoted by the arrow, so that data produced by process `a` can flow to the consumer process `b`.

Manners

A manner is a parameterized subprogram that optionally can return a value. Just like a manifold, a manner also consists of a header and a body. As for the subprograms in other languages, the header of a manner essentially defines, after the keyword `manner`, its name and the types and the number of its parameters. In the same way as with a manifold body, a manner body can also be written as a block or as an ordinary C function. A manner whose body is a block is called a *regular* manner. When its body is written as a C function, it is called an *atomic* manner. The atomic manners interface with the **MANIFOLD** world through the same ANSI C interface library used by atomic manifolds. Upon transition to a state whose body is a manner, a running manifold process creates a new invocation of the manner and enters the block that constitutes the body of the manner. When this body is a block its sub-state machine takes over the control. When this body is a C function this function executes and returns.

Semantics of a Pipeline

A pipeline is a construct that defines:

- a set of manner calls to be executed upon a transition to the state that contains it;
- a set of primitive actions to be performed in that state;
- a set of processes as the ones whose events can pre-empt the pipeline (pre-emptive sources, see below); and

- the termination condition for the pipeline (see below).

Transition to a state whose body is a pipeline causes the running process instance to construct the pipeline. The construction of a pipeline means that all actions specified therein are performed to their completion. When a particular primitive action is considered as being done (complete), then this is clearly given in its description in the reference manual [27]. A pipeline is *constructed* upon transition to its corresponding state. The construction of a pipeline is considered to be an atomic activity (i.e., it cannot be interrupted or interleaved). Because all values used in a pipeline must be evaluated before it can be constructed, all manner calls (with or without a return value) in a pipeline are performed during its construction.

A pipeline (and the state that contains it) becomes *pre-emptable* once it is fully constructed. This means that occurrences of events can cause a transition out of the current state (that contains the pipeline), into another state. When this happens, the current state is said to have been *pre-empted* (i.e., we are kicked out of it). An event occurrence can cause a transition out of the current state only when the following three conditions are satisfied.

- The pipeline must be pre-emptable (thus it must be fully constructed).
- The event occurrence must match with one of the labels of other states.
- The events must come from a source that belongs to the so-called set of the pre-emptive sources of the pipeline. The pre-emptive sources of the pipeline are the processes used in some way in the pipeline (e.g., processes mentioned in its expressions, or processes used as parameters in its primitive actions).

Termination of a pipeline (see below) is a special case of pre-emption and thus can happen only after it becomes pre-emptable.

Units can flow through the stream connections made in a pipeline only after it is constructed. Specifically, all streams defined in a pipeline must be created and/or connected before any units can flow through any one of the new connections made in the pipeline.

By definition, a pipeline terminates (1) when all streams it constructs or connects are broken (on at least, one end; see [27] for the details about when a stream connection breaks) and (2) all processes mentioned as parameters of the primitive action named `terminated`, have been terminated.

Pre-emptability and termination of a state whose body is a pipeline is determined by the pre-emptability and termination of its pipeline. Thus saying "the pipeline of state `x` is pre-empted" is the same as saying "state `x` is pre-empted". The same goes for termination.

Pre-emption of a pipeline pre-empts all of its streams (i.e., the streams are dismantled in some way depending on their types).

Sequential Composition of State Bodies

In **MANIFOLD** we can use the semicolon to separate different state bodies to construct a single composite state body. We do this frequently in our source code. The syntax of such a state is

"`state_label: body-1; body-2; ...  etc.`"

and its semantics is "switch to this state (and dismantle the previous one) when the conditions for transition are satisfied and execute the state bodies one after the other". However, this sequential composition is done in a special way. Once the first body is complete the *whole* state becomes pre-emptable. Thus, when the conditions for transition to another state are satisfied, we do not perform the second state body, but instead jump to that other state.

4 The Cut

In this section we explore the structure of the ANSI C program. The program consists of a data definition section, a main program and some 33 subroutines with a total length of some 3500 lines. Instead of its full source code, we give only the relevant part of the C code for the sparse-grid method, viz., a schematized version of the main program, and the subroutine `subsolve`. With this small part of the C code we can explain the essential implementation aspects of the sparse-grid method, as well as its actual restructuring into a concurrent application.

```
 1 /* SeqSourceCode.c */
 2
 3 int root, level;
 4 double le_tol;
 5
 6 /* Declaration of the huge global data structure (arrays) */
 7
 8 /**************************************************/
 9 int main (int argc, char *argv[])
10 {
11   int i, j, lm, l;
12
13   /* Root level (i.e. refinement level of coarsest grid) */
14   root = atoi(argv[1]);
15   /* Additional refinement above the root level */
16   level = atoi(argv[2]);
17   /* The tolerance of the integrator. */
18   le_tol = atof(argv[3]);
19
20   /* Initialization data structure and some initial computations */
21
22   /* The heavy computational work  */
23   for (lm = level - 1; lm <= level; lm++) { /* loop over the grid level */
24     for (l = 0; l <= lm; l++) { /* loop over the grids belonging to a certain grid level */
25       subsolve(l, lm-l);
26     }
27   }
28
29   /* Prolongation work */
30   ...
31 }
32
33 /**************************************************/
34 void subsolve (int l, int m)
35 {
36   /* Heavy computational work on grid (l, m) */
37   ...
38
39   /* The results are stored in the global data structure */
40   ...
41 }
```

On lines 3-6, some global variables are declared followed on line 6 with the actual global data structure that contains the grid data. On lines 13-18 the global variables declared on lines 3-6 are set with values read from the command line at the time the program is executed. After that, the program continues with initializing the data structure and with some initial computation (line 20).

After this a nested iteration starts (the nested for-loop on lines 22-27) in which the subroutine `subsolve` is called. In this nested loop the grids on the main diagonal $\Omega^{N,0}$, $\Omega^{N-1,1}$, $\cdots$, $\Omega^{0,N}$ and the grids on the subdiagonal $\Omega^{N-1,0}$, $\Omega^{N-2,1}$, $\cdots$, $\Omega^{0,N-1}$ of Figure 1 are all visited and on each of these grids (a grid is specified by two integers; see the two integer arguments of `subsolve` on line 25) the subroutine `subsolve` is performed. `subsolve` is a very computing intensive routine. In this routine a linear system of equations ($Ax = b$) is solved for every time step. Moreover, this A matrix must be built up in the program which takes a long time. Also the adaptive time step in the time integrator (the Rosenbrock solver) is something that must be computed again and again.

After the nested loop, the coarse approximations on the grids laying on the above mentioned main diagonal and subdiagonal are known and are prolongated on line 29 onto the finest grid $\Omega^{N,N}$ (i.e. the lower right corner of Figure 1) to obtain a more accurate solution for it. With this the program comes to an end.

Because it is our aim to restructure this ANSI C program into a concurrent (parallel or distributed) structure we have special interest in the subroutines which possess concurrent properties. In general, we can say that every grid subroutine with the property that it reads and writes data only from and to its own grid, can be restructured to run concurrently. In our program it turns out that `subsolve` has this property and because it is also very computing-intensive, it is a good candidate to run concurrently on all the grids to be visited in the nested for-loop.

A simple way to restructure our sequential ANSI C program into a concurrent one is to introduce a workers-pool (containing a number of workers) when we arrive at the heavy computations that can be done concurrently. Each worker in the workers-pool performs the same operation `subsolve` on a different data segment of the global data structure independently of the others. In a program built according to this scheme, none of the computational processes actually runs concurrently until it reaches a concurrent region. Then the multiple workers (i.e., the parallel or distributed threads) in the workers-pool begin, and the program runs concurrently. When the program exits the concurrent region, only one single computational process continues (now we run sequentially) in which the prolongation work is performed.

In the next section we explain how we have organized the restructuring of the sequential program into a parallel/distributed program by a master/worker protocol and discuss its implementation in **Manifold**.

5 The Paste

As explained in §4, the crux of our restructuring is to allow the computations done in `subsolve` on every single grid visited in the nested loop, to be carried out in a separate process. These processes can then run concurrently in **Manifold** as separate threads executed by different processors on a multi-processor hardware (e.g., a multi-processor SGI machine), or in different tasks on a distributed platform (e.g., a network of workstations), or a combination of the two.

We have organized the restructuring according to a master/worker protocol in which the master performs all the computations in the sequential source code except the work embodied in `subsolve`, which is done by the workers. In **Manifold**, we can easily realize this master/worker protocol in a generic way, where the master and the worker are parameters of the protocol. In this protocol we describe only how instances of master and worker process definitions should communicate with each other. For the protocol, it is irrelevant to know what kind of computation is performed in the master or the worker. What is indeed important for the protocol is that the input/output and the event behavior of the master and the worker comply with the protocol. E.g., the master should write the data needed by the worker to its own output port and the worker, connected by a third party (a manager) to this port, should read this information from its own input port. Furthermore, according to this protocol, the coordinator can create a worker only when the master raises an event to request for its creation.

We give an informal description of this Master/Worker protocol in §5.1 followed by its actual implementation in §5.2. In §5.3 we give a stepwise description of the behavior interface of the master and worker.

5.1 The Glue

The master/worker protocol we use can be described as follows. In a coordinator process we create and activate a master process that performs all computations in the main C program of the sequential version, except the computation to be carried out by `subsolve`.

Each time the master arrives at the point where it has to do the `subsolve` work, it delegates this work to a worker in a workers-pool. The master makes its wish known to the coordinator by raising an event (`create_pool`)[2]. The coordinator reacts to this event by jumping to a state where it waits for requests coming from the master to create a worker for the workers-pool. Each time the master needs another worker for the workers-pool it raises an event (`create_worker`) to signal the coordinator to create one. Because the master wants to use the worker, it needs to know its identity. The coordinator makes this identity available to the master by sending its reference via a stream. The master waiting for its workers, receives a worker reference, activates it and takes care that the worker receives all necessary information so that it can do its job. The master writes this information on its output port which is connected by the coordinator to the input port of the worker, so that the latter can read it from this port. In this way, a pool of workers, created by the coordinator, is set to work by the master, each worker performing a relaxation computation. Before the master can continue its work, it must wait until all the workers are done with their relaxations and are ready to die, which they signal by raising an event (`dead_worker`). The master does not want to count those events by itself, but delegates the organization of this rendezvous (i.e., a synchronization point) by raising an event (`rendezvous`) to signal the coordinator to make the proper arrangements. In the meantime, the master takes a nap and waits for the event (`a_rendezvous`) raised by the coordinator (which is now responsible for counting the `dead_worker` events) to acknowledge the successful rendezvous. After this rendezvous, the master reads from its input port the computational results of the workers. This is made possible by the coordinator which has set up a stream between the output port of each worker and the input port of the master. Hereafter, the master proceeds with prolongation work and is done.

5.2 The Implementation of the Gluing Modules

The **Manifold** source code of our master/worker protocol is given below.

```
// protocolMW.m

#include "MBL.h"

#include "rdid.h"

#include "protocolMW.h"

#define IDLE terminated(void)

/***********************************************************************/
manner Create_Worker_Pool(process master <input, dataport | output, error>,
                          manifold Worker(event) )
{
  save *.
  ignore death.

  auto process now is variable(0).
  auto process t is variable(0).

  event death_worker.

  priority create_worker > rendezvous.

  begin: (MES("begin"), preemptall, IDLE).

  create_worker: {
    hold Worker.

```

[2]We give the names of the events as used in the **Manifold** source code (see §5.2) in parentheses.

```
30     process worker is Worker(death_worker).
31
32     stream KK worker -> master.dataport.
33
34     begin: now = now + 1;
35            (MES("create_worker: begin"),
36             &worker -> master -> worker -> master.dataport, IDLE).
37   }.
38
39   rendezvous: {
40     begin: (preemptall, IDLE).
41
42     death_worker: t = t + 1;
43                     if (t < now) then (
44                       post(begin)
45                     ) else (
46                       post(end)
47                     ).
48   }.
49
50   end: (MES("rendezvous acknowledged"), raise(a_rendezvous)).
51 }
52
53 /**********************************************************************/
54 export manner ProtocolMW(process master <input, dataport | output, error>,
55                          manifold Worker(event) )
56 {
57   save *.
58
59   begin: terminated(master).
60
61   create_pool: Create_Worker_Pool(master, Worker); post(begin).
62
63   finished: halt.
64 }
```

First we make some remarks about the syntax of the code in §5.2.1 and in §5.2.2 we describe how it runs.

5.2.1 Syntax of the Gluing Code

Looking syntactically at the source code and using the MANIFOLD terminology explained in §3.2 we observe the following:

- The source code describes two regular manners named respectively `Create_Worker_Pool` (lines 11-51) and `ProtocolMW` (lines 53-64). There are no manifolds defined in this source file.
- In the header of `Create_Worker_Pool` (lines 12-13) we see that it has two parameters. The first parameter is a process named `master` which has besides the normal standard ports (`input`, `output`, `error`) also a port named `dataport`. The second parameter is a manifold named `Master` (note the capital) and has an event parameter.

  In the header of `ProtocolMW` (lines 54-55) we also see these two parameters.
- The body of `Create_Worker_Pool` (lines 14-51) consists of a local declaration part (lines 15-23) and four states (lines 25-50) namely the `begin` state (line 25), the `create_worker` state (lines 27-37), the `rendezvous` state (lines 39-48) and the `end` state (line 50).

  In the local declaration part, we see the two declarative statements `save` and `ignore` on lines 15 and 16, and the creation of process instances `now` and `t` (lines 18-19) using the syntactic construct *process* `process_name` *is* `manifold_name`. Further, we see an event declaration (line 21) and an event priority declaration (line 23).

  The body of the `begin` state is a pipeline.

  The body of the `create_worker` state is a block that has its own local declaration part with three declarations on lines 28, 30 and 32. The block has only a `begin` state (lines 34-36), whose body consists of two pipelines to be constructed sequentially (see the semicolon on line 34). The first pipeline is after the colon on line 34 and the second one is on lines 35-36.

The body of the `rendezvous` state is also a block but it does not have a local declaration part. This block has two states, namely the `begin` state (line 40) and the `death_worker` state (lines 42-47). The body of this `begin` state is a pipeline. The body of the `death_worker` state consists of a pipeline (specified after the colon on line 42), followed by an "if" construct (lines 43-47).

The body of the `end` state consists of a pipeline.

The body of `ProtocolMW` (lines 56-66) consists of a local declaration part (lines 57-59) (with two declarations) and has three states (lines 61-65), namely the `begin` state (line 61), the `create_pool` state (line 63), and the `finished` state (line 65). The bodies of all these states are pipelines.

Note that in the `create_pool` state (line 63) the `Create_Worker_Pool` manner is called.

- The text on line 1, starting with `//` and denoting the name of the MANIFOLD source file, is a comment and is ignored by the MANIFOLD compiler.

- In the MANIFOLD language we can group all commonly used definitions of manifolds, manners, events, etc., inside a so called header file and simply include this file, in the same syntax as that of the ANSI C pre-processor, in any program that needs to use those definitions.

  On line 3 we include the file `MBL.h` which contains the definitions of MANIFOLD built-in library.

  On line 5 we include the file `rdid.h` which contains the definitions of some manners we use to print messages from independent running processes on the computer screen in an ordered way.

  On line 7 we include the header file `ProtocolMW.h` (see its contents below) which contains the definitions of our protocol manner `ProtocolMW` (lines 3-4) and some global events (line 5) we use in the file `ProtocolMW.m`.

```
1 // protocolMW.h
2
3 manner ProtocolMW(process master <input, dataport | output, error>, manifold Worker(event) ) elsewhere.
4
5 extern event create_pool, create_worker, rendezvous, a_rendezvous, finished.
```

- Line 9 defines a pre-processor macro in the same syntax as that of the ANSI C pre-processor.

- The keyword `export` in front of the manner `ProtocolMW` (line 54) states that this manner can be used in other source files which import this MANIFOLD definition.

5.2.2 The Gluing Code at Work

In this section we explain the internal working of the state machines of the master/worker protocol. From the informal description of the master/worker protocol in §5.1 we already known that the master and the worker communicate via the events defined in the header file `protocolMW.h`. Because the master and worker manifolds are implemented as C functions and the fact that the events defined in the header file are meaningful only in the MANIFOLD world, we must make them available in the C world as well. In the task instance of the master we do this using a routine defined in the MANIFOLD application programmers interface library. For the worker we do it via its parameter list, because a worker must be able to run as a remote worker. We make the events available under the same names as used in the header file.

To clarify the way they co-operate with each other following the master/worker protocol `protocolMW`, in this section we provide cross-references to the source code lines of the master and the worker within parentheses.

We first discuss the manner `ProtocolMW` (lines 54-64) followed by the manner `Create_Worker_Pool` (lines 12-51) which is used by the former.

The actual manifold (named `Main`) that does the restructuring of the sequential source code invokes (as we see in §6.2) the `ProtocolMW` manner in its `begin` state. As a result, we enter the block of this manner (lines 56-64). Upon entering a block, the statements in its local declaration part are performed. In this case the only statement in this part is the `save` which states that we can switch only to states in this block (i.e., the `begin`, `create_pool` or `finished` states respectively on lines 59, 61 and 63). Other possible event occurrences are saved and can be handled (if necessary) outside this block.

After performing the local declaration part of the entered block the MANIFOLD run-time system automatically posts an occurrence of the predefined high-priority event `begin` in the event memory of the caller (as we will see this is `main` in §6.2) which causes a transition to the `begin` state. There must always be a `begin` state (i.e., a state with a single `begin` as its label) in every block. This insures that upon entering a block, at least this one state can be visited (i.e., the actions in its state body are performed), regardless of any other event occurrences that may or may not be present in the event memory.

In the `begin` state (line 61) we wait until the already active process instance `master` (received as parameter on line 54) terminates. Because we have mentioned `master` (as an argument of the `terminate` primitive) in the state body, we also make this state sensitive to events that are raised by `master`. Because `master` does not terminate, the net result of the action in the `begin` state is that we wait there until there is an event occurrence for which we have a matching event label. Because `master`, which is a process wrapper around the C code (excluding the `subsolve` work), arrives after some sequential computational work (initialization) at the point where it has to do the work embodied in `subsolve`, it raises an event named `create_pool` to signal that it needs a workers-pool to delegate that work to (master: 4(a)). This event pre-empts the `begin` state and causes a transition to the `create_pool` state (line 61). In this state the manner `Create_Worker_Pool` (lines 11-51) is called with the process instance `master`, and the manifold `Worker` (which the protocol manner `ProtocolMW` itself has received as a parameter on lines 54-55) as its actual parameters.

The manner `Create_Worker_Pool` conducts the workers in the pool and takes care that they can do their computations properly. When the workers in the pool are done, they die and the manner returns. Afterwards (denoted by the semicolon on line 61) we post the `begin` event so that we jump again to the `begin` state (line 61) where we wait for events. Another event will arrive soon because the `master` raises the event `finished` (master: 5) to denote that it does not need workers anymore. This causes a jump to the `finished` state (line 63), where the primitive action `halt` effectively returns the flow of control from the manner to its caller. The master is still running and is also done after performing the final prolongation computations.

The manner `Create_Worker_Pool` (lines 11-51) called on line 61 works as follows. Upon entering its block, first the statements in its local declaration part are performed (lines 15-23).

Line 15 is a declarative statement which states that we can switch only to states specified in this block (lines 14-51).

Line 16 is another declarative statement which states that `death` events can be removed from the event memory of the executing manifold instance, upon departure from the block (at line 51).

On lines 18-19 we create and activate two process instances, respectively named `now` and `t`, of the predefined manifold `variable` (defined in the MANIFOLD built-in library), and initialize them with 0. We use these variables respectively for counting the number of created instances of the `Worker` manifold (we count them on line 34 with `now` which is a mnemonic for Number Of Workers) and for counting the number of dead workers (by counting their `death_worker` events on line 42). Note that, **MANIFOLD** obviously knows only processes; there are no data structures in **MANIFOLD**, not even the simplest kind, a variable.

On line 21, a local event named `death_worker` is declared.

Because it can happen that both events `create_worker` and `rendezvous` are available in the event memory of the executing manifold instance that calls this manner, we state with the `priority` declarative statement that jumping to the `create_worker` state has a higher priority than jumping to the `rendezvous` state.

The first state we visit in this manner is the `begin` state (line 25). There, we do the following: we print the message `"begin"` on the screen to indicate that we are in this state; we state by the primitive action `preemptall` that all events for which we have a handling state label can pre-empt the `begin` state; and we wait (due to the word IDLE) for the termination of the special pre-defined process `void`. In the **MANIFOLD** language we express this by `terminated(void)` as can be seen from the meaning (line 9) of the IDLE macro (line 25). Because the special process `void` never terminates, this effectively causes a hang in the `begin` state until it detects an event in the event memory of the process instance where this manner is invoked and for which it has a state label. An event will come soon, because `master` is expected to raise the event `create_worker` every time it wants another worker in the workers-pool (master: 3(b)). This event pre-empts the `begin` state and causes a state transition to the `create_worker` state.

In the `create_worker` state (lines 27-37) a number of workers are set to work in a workers-pool. The body of this state is a block. In its local declaration, we use the `hold` statement on line 28 so that we can handle events coming from `Worker` instances outside the scope in which those instances are known (we intend to count their `death_worker` events in the `rendezvous` state on line 42); otherwise, the instances of `Worker` are known only in the block in which they are defined (lines 27-37). On line 30, we create a process named `worker` and pass it to the local event `death_worker` declared on line 21.

The `death_worker` event is an event the worker must raise to inform the manner `Create_Worker_Pool`, that it finished its job and is going to die (worker: 4).

The declarative statement on line 32 states that all stream connections between the output port of `worker` and the input port of the `master` (this input port is named `dataport`) must be of type `KK` (i.e., Keep-Keep). When streams of this type are used in a state they are not dismantled (i.e., disconnected from their sources and sinks) once the state is pre-empted. Normally, streams are `BK` (i.e., Break-Keep) streams which means that the stream is disconnected from its producer automatically, as soon as it is disconnected from its consumer, but disconnection from its producer does not disconnect the stream from its consumer.

In the `begin` state of the state `create_worker`, the stream configuration on line 36 is constructed and we wait for events (due to the word IDLE) from the `master` (`create_worker` and `rendezvous` are possible events). In the stream configuration we see that the process identification of `worker` (denoted by `&worker`) is sent through a stream (the first $\rightarrow$ on line 36) to the already active `master`. The `master` receives this reference to `worker` and sends all the information `worker` requests through a stream (the second $\rightarrow$ on line 36) to `worker`. The `worker` process promptly reads the information it receives from `master` (worker: 1), does its job (worker: 2), and sends its computed results (worker: 3) through a stream (the third $\rightarrow$ on line 36) to the `dataport` port of `master` (denoted by `master.dataport`). The `master` process reads this and stores the results in the global master space (master: 3(f)). Due to the word `IDLE` (line 36) we stay in the state on line 34 until `master` again raises a `create_worker` event. This event pre-empts this `begin` state (line 34) which dismantles the streams in this state and causes a transition to the `create_worker` state where the whole sequence starts again. Dismantling of the streams means, in this case, that all the streams on line 36 are broken at their sources (because they have the default type BK) with the exception of the stream for which the worker is the source; this stream is KK (see line 32) and must stay intact because when the worker is a remote worker this stream is used to transport its computed results to the master. This is how all workers are created and set to work in the pool.

The next event to be handled is the `rendezvous` event. This event is raised by `master` (master: 3(g)) after it reads the computed results of the remote workers (master: 3(f)) and causes a transition to the `rendezvous` state which has two (sub)states: the `begin` state (line 40) and the `death_worker` state (line 42). In its `begin` state, we wait for the `death_worker` events. Each time a `death_worker` is detected, it is counted (line 42). As long as we have fewer `death_worker` events than the number of created workers (i.e., the value of `now` on line 34) we post the `begin` event (line 44) which causes a transition back to the `begin` state (line 40) where we wait for other `death_worker` events. Otherwise, we post `end` (line 46) which causes a state switch to the `end` state (line 50). In this state we print a message on the screen, raise the event `a_rendezvous`, and the `Create_Worker_Pool` manner returns.

Note that the coordination scheme in `ProtocolMW` can also handle a more demanding master. Just imagine that we a have `master` that instead of raising `finished` wants to introduce another workers-pool to delegate some work to. It could easily raise the event `create_pool` to denote that, in which case we jump again to the `create_pool` state and another pool is created. In [8] we have exploited this facility in a slightly different master/worker protocol.

5.3 Behavior of Master and Worker

The behavior of the master is given below. The line numbers between parentheses in this section refer to the MANIFOLD source code `protocolMW.m` in §5.2.2. Moreover, we refer to the process instance that invokes the `protocolMW` manner, as *the coordinator* (i.e., the instance of the manifold `Main` (line 13) in §6.2).

1. Make the extern events defined in the header file `protocolMW.h` available to the master so that it can communicate with `protocolMW` in `protocolMW.m`.
2. Perform some initialization work (optional).
3. Perform some work concurrently by creating a pool of workers and charge each with a computational job. Do this as follows:
   (a) Request a coordinator process to create an empty pool of workers by raising the `create_pool` event (line 63).
   (b) Request this coordinator process to create a worker in this pool by raising the event `create_worker` (line 27).
   (c) Read a unit containing the process reference (identification) of a created worker from your own input port and activate it. (This unit, `&worker`, is sent through the first stream (`->`) on line 36 in `protocolMW.m` to the master).
   (d) Write the information, which the worker needs to do its job, on your own output port.
   (e) Repeat steps a, b, c and d for each worker as needed. (In this way a pool of workers is created and set to work.)
   (f) Collect the computational results from the workers (read those results from your own input port)

(g) Raise the event `rendezvous` to request the coordinator to organize a rendezvous (line 39).

(h) Wait for the event `a_rendezvous` raised by the coordinator to acknowledge a successful rendezvous (line 50).

4. Repeat step 3 as many times as needed and raise at the end of this repetition the event `finished` (line 63) to inform the coordinator process that the master does not need workers anymore.

5. Perform some final sequential computations (optional).

The behavior of the worker is described below. Here, the `death_worker` event is introduced via the first argument of the worker.

1. Read the information you need to do your job, from your own input port.

2. Do the computational job.

3. Write the computed results to your own output (master: 3(f)).

4. Raise the event `death_worker` to signal to the coordinator that you are done and are going to die (line 42).

6 The Restructuring

6.1 The Actual Master and Worker Manifolds

From the description of the master/worker protocol it is clear that we have to adapt the `main` routine and the `subsolve` routine of the sequential source code (line 9 and line 34 in §4) in such a way that they comply with the behavior of respectively the master and the worker as described in §5.3. In fact, this is all we have to do. All the other subroutines (31 in total) in the sequential source code are left untouched. The adapted routines can run as atomic processes, i.e., as separate threads (light-weight processes [29]) within operating-system level processes. The master and worker manifolds (written as atomic processes) are in fact C wrappers around the original C subroutines of the sequential version. In the source code below we give the new versions of `main` and `subsolve`.

```
 1 /* ResSourceCode.c */
 2
 3 #include "adid.h"
 4 #include "AP_interface.h"
 5 #include "ResSourceCode.h"
 6
 7 int root, level;
 8 double le_tol;
 9
10 /* Declaration of the huge global data structure (arrays) */
11
12 AP_Event create_pool, create_worker, rendezvous, a_rendezvous, finished;
13
14 /*******************************************************************/
15 void global_init_stuff(void)
16 {
17   int err;
18
19   create_pool = AP_AllocateEvent();                          P(create_pool)
20   err = AP_InitHeaderEvent(create_pool, "create_pool");      I(err)
21
22   create_worker = AP_AllocateEvent();                        P(create_worker)
23   err = AP_InitHeaderEvent(create_worker, "create_worker");  I(err)
24
25   rendezvous = AP_AllocateEvent();                           P(rendezvous)
26   err = AP_InitHeaderEvent(rendezvous, "rendezvous");        I(err)
27
28   a_rendezvous = AP_AllocateEvent();                         P(a_rendezvous)
29   err = AP_InitHeaderEvent(a_rendezvous, "a_rendezvous");    I(err)
30
```

```
 31   finished = AP_AllocateEvent();                                P(finished)
 32   err = AP_InitHeaderEvent(finished, "finished");               I(err)
 33 }
 34
 35 /**************************************************************************/
 36 void Master (AP_Port port)
 37 {
 38   int i, j, lm, l, err, ar[3];
 39   AP_Unit u, ua[2];
 40   AP_Tuple tup;
 41   char buf[200];
 42
 43   AP_Process p = AP_AllocateProcess();
 44   AP_Event r = AP_AllocateEvent();
 45   AP_EventPatternSet eps = AP_AllocateEventPatternSet();
 46   AP_Process q = AP_AllocateProcess();
 47
 48   int dataport = AP_LocalPortId("dataport");
 49
 50   char mesg[300];
 51
 52    /* Step 1 */                                                    M("Welcome")
 53   global_init_stuff();
 54
 55   err = AP_PortGetUnit(port, &u, NULL);                         I(err) P(u)
 56
 57   err = AP_FetchTuple (u, &tup);                                I(err)
 58
 59   err = AP_DeallocateUnit(u);                                   I(err)
 60
 61   /* Root level (i.e. refinement level of coarsest grid) */
 62   err = AP_FetchString(tup.vec[1], buf, sizeof(buf) - 1);
 63   root = atoi(buf);
 64
 65   /* Additional refinement above the root level) */
 66   err = AP_FetchString(tup.vec[2], buf, sizeof(buf) - 1);
 67   level = atoi(buf);
 68
 69   /* The tolerance of the integrator */
 70   err = AP_FetchString(tup.vec[3], buf, sizeof(buf) - 1);
 71   le_tol = atof(buf);
 72
 73   /* Step 2 */
 74   /* Initialization data structure and some initial computation */
 75   ...
 76
 77   /* Step 3(a) */
 78   /* Serve out the subsolve routines */
 79   err = AP_Raise(create_pool);                                  I(err)
 80   for (lm = level; lm >= level - 1; lm--) {
 81     for (l = lm; l >= 0; l--) { // The loop is now reversed
 82       /* Step 3(b) */
 83       err = AP_Raise(create_worker);                            I(err)
 84
 85       /* Step 3(c) */
 86       err = AP_PortRemoveUnit(AP_INPUT, &u, NULL);              I(err) P(u)
 87
 88       err = AP_DerefProcess(p, u, NULL, NULL);                  I(err)
 89       err = AP_Activate(p);                                     I(err)
 90
 91       /* Step 3(d) */
 92       ar[0] = l;
 93       ar[1] = lm - l;
 94       ar[2] = root;
 95
 96       ua[0] = AP_FrameIntegerArray((int *) ar, 3);
 97
 98       ua[1] = AP_FrameDouble(le_tol);
 99
100       tup.size = 2;
101       tup.vec = ua;
102
103       // Frame this tuple
104       u = AP_FrameTuple(&tup);
105
106       err = AP_PortPlaceUnit(AP_OUTPUT, u, NULL);               I(err)
107
108       err = AP_DeallocateUnit(u);                               I(err)
109
110     }
111   }
112
113   /* Step 3(e) is the nested loop on lines 79-80 */
114
115   /* Step 3(f) */
116   // Collect the computational results from the workers
117   for (lm = level - 1; lm <= level; lm++) {
118     for (l = 0; l <= lm; l++) {
119       err = AP_PortRemoveUnit(dataport, &u, NULL);
120       /* Stored the unit in the global data structure */
121       u2gds(u);
122       err = AP_DeallocateUnit(u);                              I(err)
123     }
124   }
125
126   /* Step 3(g) */
127   /* Request the coordinator to start its rendezvous */
128   err = AP_Raise(rendezvous);                  I(err)
129
130   /* Step 3(h) */
131   /* Make a mask over the EM to pick out the "a_rv" event
132      and wait for the acknowledge from a Create_Worker_Pool manner
133      to signal a succesfull rendezvous */
134   err = AP_EventPatternSetInsert(eps, a_rendezvous, NULL);      I(err)
135   err = AP_DeleteWaitEvent(eps, r, p);                          I(err)
136
137   err = AP_DeallocateProcess(p);                                I(err)
138   err = AP_DeallocateProcess(q);                                I(err)
139   err = AP_DeallocateEvent(r);                                  I(err)
140   err = AP_DeallocateEventPatternSet(eps);                      I(err)
141
142   /* Step 4 */
143   err = AP_Raise(finished);
144
145   /* Step 5 */
146   /* Prolongation work */
147   ...                                                           M("Bye")
148 }
149
150 /**********************************************************************/
151 void Worker(AP_Event e)
152 {
153   int root;
154   double le_tol;
155
156   AP_Unit u;
157   int ar[3], l, m, err;
158   AP_Tuple tupIn;
159
160   /* Step 1 */                                                  M("Welcome")
161   /* Receive l, m, root, and le_tol */
162   err = AP_PortRemoveUnit(AP_INPUT, &u, NULL);                  I(err) P(u)
163   err = AP_FetchTuple(u, &tupIn);
164   err = AP_FetchIntegerArray(tupIn.vec[0], ar, 3);              I(err)
165   l = ar[0]; m = ar[1]; root = ar[2];
166   err = AP_FetchDouble(tupIn.vec[1], &le_tol);
167   err = AP_DeallocateUnit(u);                                   I(err)
168
169   /* Step 2 */
170   u = subsolve(l, m, root, le_tol);
171
172   /* Step 3 */
173   err = AP_PortPlaceUnit(AP_OUTPUT, u, NULL);                   I(err)
174   err = AP_DeallocateUnit(u);                                   I(err)
175
176   /* Step 4 */
177   err = AP_Raise(e);                                            I(err) M("Bye")
178 }
179
180 /**********************************************************************/
181 AP_Unit subsolve (int l, int m, int root, double le_tol)
182 {
183   AP_Unit u;
184
185   /* Heavy computational work on grid (l, m) */
186   ...
187
188   /* Results are stored in unit "u" which is returned */
189   ...
190   return u;
191 }
```

The master manifold named `Master` (lines 35-148) is a rewriting of the routine `main` of the sequential version. We have changed the name of `main` into `Master` because the concurrent version of our sequential program has its own `main`. `Master` follows exactly the steps 1-5 of the behavior of the master (see §5.3).

The worker manifold named `Worker` (lines 150-178) is a wrapper around a new version of the routine `subsolve` (lines 180-191). It follows exactly the steps 1-4 of the behavior of the worker (see §5.3).

We mention the different steps of the behavior of the master and the worker in §5.3 as comment in the source code of the master and worker to facilitate its comprehension of it. Therefore, we do not give a long description of this source code but only some remarks and a short general description of how events are raised (broadcast), how they are read from the event memory, and how units (data) are read and written from and to ports.

- Atomic manifolds interface with the MANIFOLD world through a special ANSI C interface library. The routines of this interface are recognized by the `AP_` prefix. In this interface we find the routines for raising events and reading and writing data units from and to ports.
- We have written a separate routine named `global_init_stuff` (lines 14-33) which is called on line 53 in the master in order to make available the external events defined in the header file `protocolMW.h` (step 1 in the master).
- In the `main` routine of the sequential version we read some command line arguments from `argv` (lines 13-18 of `SeqSourceCode.c.v` in §4). `Master`, which will run as a separate process, now reads these arguments during runtime from its port parameter (line 36). It turns out that this parameter port is the `argv` of the `main` in the MANIFOLD world (line 15 in `mainprog.m` below). On line 55 we read a unit `u` from this port. This unit has a special form. It is a so-called tuple (a C struct). This tuple contains the values of `root`, `level`, and `le_tol`. On lines 57-71 we see how these variables are assigned to the different tuple fields.
- Events in atomic manifolds (i.e., manifolds written in a conventional programming language) are raised with the C routine `AP_Raise` (e.g., 127). To handle events, we first must insert them in a so-called *event pattern set* (with `AP_EventPatternSetInsert`; e.g., line 134). After this, we can search, with `AP_DeleteWaitEvent` (e.g., 135) in a blocking way through the event memory of the calling process instance to find out if one of the events contained in the event pattern set is available there. When an event is found it is deleted.
- Reading and writing units from and to ports are done, respectively, with `AP_PortRemoveUnit` (e.g., line 86) and `AP_PortPlaceUnit` (e.g., line 106). Other routines used in `ResSourceCode.c` reveal their purposes by their names or are not discussed because they are not critical for the understanding of the main activities in the source code. For the details we refer to [27].
- In the ANSI C source code, we use a number of macros for error checking (see e.g., the macro `I` on line 20) or for printing to the screen in an ordered fashion (e.g., the macro `P` on line 19). Their definitions are given in the header file `adid.h` which is included in `ResSourceCode.c` on line 3.
- The file with the prototypes of the master and the worker is below.

  ```
  // ResSourceCode.h

  #include "AP_interface.h"

  extern void Master(AP_Port p);

  extern void Worker(AP_Event e);
  ```

  Because this file is included both in the file (line 5) `ResSourceCode.c` as well as in the manifold source file where we instantiate the master and the worker (line 3 in the file `mainprog.m` below), we are sure that a mismatch between their formal and actual parameters results in syntax errors issued by the C compiler.

6.2 The Concurrent Version

Using the manifold `ProtocolMW` together with the two actual master and worker parameters just described, we can construct the following small MANIFOLD program which finally changes our original sequential application into a concurrent version.

```
 1 // mainprog.m
 2
 3 //pragma include "ResSourceCode.h"
 4
 5 #include "protocolMW.h"
 6
 7 manifold Worker(event) atomic.
 8
 9 manifold Master(port in p) port in input. port in dataport. port out output. port out error.
10   atomic {internal. event create_pool, create_worker, rendezvous, a_rendezvous, finished}.
11
12 /***************************************************/
13 manifold Main(process argv)
14 {
15   begin: ProtocolMW(Master(argv), Worker).
16 }
```

We briefly explain this source code.

On line 3 we include the header file `ResSourceCode.h` to insure that a parameter mismatch will result in syntax errors.

On line 5 we include the header file `protocolMW.h` which contains the definitions of our protocol manner `protocolMW` and the external global events (see §5.2.1). Note that we also use this header file in `ProtocolMW.m` (see §5.2) to insure that a compilation error will be generated by the MANIFOLD compiler if there is a mismatch between the definition of `ProtocolMW` and the way it is called (on line 15).

Line 7 defines the worker manifold named `Worker`, which takes an event argument, and states (through the keyword `atomic`) that it is not implemented in the MANIFOLD language, but in another programming language such as ANSI C, C^{++}, or Fortran. The keyword `internal` states that the function that constitutes the body of this manifold is to run as a thread within an operating system level process. Because we do not explicitly specify the ports of the manifold, it has the default set (i.e., the `input`, `output` and `error` ports).

The same holds for the master manifold `Master` (lines 9-10), except that it has an input port argument (`port in`) (through which the command line arguments are read into the application) and its ports are explicitly specified (the default set plus an additional input port named `dataport`). Because the external global events defined in `protocolMW.h` are to be exchanged between the master and the rest of the MANIFOLD application, we also specify those events between brackets on line 10.

Lines 12-16 define the manifold named `Main`, which has only one state: the `begin` state. In this state, the `ProtocolMW` manner is called with the master and the worker manifold as actual arguments (line 15). Because `ProtocolMw` expects as its first argument a process type argument, the manifold `Master` is automatically converted to an active process which reads from its argument `argv` the command line arguments. `argv` itself is received as an argument of the `main` manifold (line 13) which is automatically instantiated and activated by the MANIFOLD runtime system.

After this, the instance of `Main`, the instance of the master `Master`, and all the necessary instances of the worker `Worker`, run concurrently.

6.3 Running the Concurrent Version

The source files that contain the MANIFOLD program (i.e., `mainprog.m` and `protocol.m`) must be compiled with the MANIFOLD compiler, named MC. This compiler generates from each MANIFOLD source code a C source file which is subsequently compiled by a normal C compiler to an object file. These object files are linked with the object files obtained from the ANSI C files of the master and worker (i.e., `ResSourceCode.c`), the object files of the original sequential source code excluding the `main` and `subsolve` routines, and with some other C source files necessary to provide the inter-task information (these latter files are generated by the MANIFOLD linker named MLINK). In order to facilitate this whole procedure, the linker in the **MANIFOLD** system generates a *makefile*, which is meant to be used as a black-box by recursive make commands in programmer-defined makefiles that finally create the executable files suitable for the appropriate platforms.

Process instances in a **MANIFOLD** application always run as separate threads (light-weight processes [29]) within an operating-system level process. This latter heavy-weight process is called a task instance in **MANIFOLD**. Process instances are bundled in task instances either automatically or under user control. When all process instances of a **MANIFOLD** application run as threads in the same task instance, the application executes in parallel (i.e., not distributed). We can, however, also bundle the process instances in such a way that each worker is housed in a separate task instance. This mapping of process instances in task instances, which can be fully specified by the user, is considered to be a separate stage in the application construction and is described in a file which is input for the **MANIFOLD** linker **MLINK**. In the example below, we arrange it such that each worker is housed in a separate task instance (line numbers have been added).

```
 1 # mainprog.mlink
 2
 3 {task *
 4   {perpetual}
 5   {load 1}
 6   {weight Master 1}
 7   {weight Worker 1}
 8 }
 9 {task mainprog
10   {include mainprog.o}
11   {include protocolMW.o}
12 }
```

In this file, we specify that a task instance is considered to be "full" when its load exceeds 1 (line 5) and that the weight of an instance of `Worker` or `Master` is also 1 (lines 6-7). The net result of this is that each task instance will house only one `Worker` or `Master` instance and thus instances of `Worker` or `Master` end up in different instances of the task named `mainprog` (line 9).

After this task composition stage the final stage in application construction can start: this is the runtime configuration stage. In that stage we define the mapping of tasks to hosts. This mapping too, is described in a file and is the input for the **MANIFOLD** runtime configurator named **CONFIG**. Suppose we need in our application in addition to the master, five workers; then we expect, with the above input file for **MLINK**, that at most (as we will see) six task instances come into existence during the run. Each of these task instances houses either a master or a worker instance. However, it can happen that a worker is already done before another worker is introduced by the master. In that case, the task instance (which is a heavy weight process) that has housed the freshly expired worker does not have a load of 1 anymore and is in principle ready to welcome a new worker. However, the standard behavior of a **MANIFOLD** task instance is that it dies when there are no thread processes running in it. To inhibit this task instance termination behavior, and to keep an empty task (i.e., task with load zero) alive for new workers, we use the keyword `perpetual` in the input file for the **MANIFOLD** linker **MLINK** (line 4). With this task termination behavior it can happen that we need less than six machines to run an application with five workers, which is more efficient. Therefore, when we start up the first task instance on the machine we are sitting behind (this so-called "start-up" machine is in our case `bumpa.sen.cwi.nl`), we have to organize five other machines for the possible other five task instances that are forked during the run. In the file below these additional machines are specified.

```
{host host1 diplice.sen.cwi.nl}
{host host2 alboka.sen.cwi.nl}
{host host3 altfluit.sen.cwi.nl}
{host host4 arghul.sen.cwi.nl}
{host host5 basfluit.sen.cwi.nl}
{locus mainprog  $host1 $host2 $host3 $host4 $host5}
```

Here, we define five variables `host1`, `host2`, up to `host5`, which we set to, respectively, `diplice.sen.cwi.nl`, `alboka.sen.cwi.nl` up to `basfluit.sen.cwi.nl`. These are the names of computers located at different places and connected via a network. The last line in the file states that the instances of the task named `mainprog` can be started on any of these machines.

Running the restructured program, using the task composition stage and run-time configuration described above, the application executes in a *distributed* fashion and produces the following chronological output.

```
bumpa.sen.cwi.nl 262146 140 1048087412 175834 mainprog Master(port in) ResSourceCode.c 136 -> Welcome
basfluit.sen.cwi.nl 1572865 79 1048087412 275851 mainprog Worker(event) ResSourceCode.c 351 -> Welcome
basfluit.sen.cwi.nl 1572865 79 1048087412 366117 mainprog Worker(event) ResSourceCode.c 370 -> Bye
arghul.sen.cwi.nl 1310721 79 1048087412 385644 mainprog Worker(event) ResSourceCode.c 351 -> Welcome
basfluit.sen.cwi.nl 1572865 90 1048087412 414473 mainprog Worker(event) ResSourceCode.c 351 -> Welcome
arghul.sen.cwi.nl 1310721 79 1048087412 483301 mainprog Worker(event) ResSourceCode.c 370 -> Bye
basfluit.sen.cwi.nl 1572865 90 1048087412 511798 mainprog Worker(event) ResSourceCode.c 370 -> Bye
altfluit.sen.cwi.nl 1048577 79 1048087412 520315 mainprog Worker(event) ResSourceCode.c 351 -> Welcome
arghul.sen.cwi.nl 1310721 90 1048087412 552362 mainprog Worker(event) ResSourceCode.c 351 -> Welcome
altfluit.sen.cwi.nl 1048577 79 1048087412 600215 mainprog Worker(event) ResSourceCode.c 370 -> Bye
bumpa.sen.cwi.nl 262146 140 1048087412 637649 mainprog Master(port in) ResSourceCode.c 337 -> Bye
arghul.sen.cwi.nl 1310721 90 1048087412 639482 mainprog Worker(event) ResSourceCode.c 370 -> Bye
```

We show only the "Welcome" and "Bye" messages from the master and its workers during the run (see lines 52, 147, 160, and 177 in §5.2.2). We don't show the computational results of this distributed run. These are written to a file and are exactly the same as in the sequential version.

Each of these output lines has the following structure. It starts with a long label followed by a `->` before the actual message. The label shows, respectively, the machine on which the task instance runs, the identification of the task instance, the identification of the process instance, a time stamp that is expressed as two numbers (these numbers are the seconds and microseconds past since midnight (0 hour), January 1, 1970) the name of the task, the name of the manifold, the name of the MANIFOLD source file and the line number where the message is produced. With such a label in front of an actual message, we always know *who* is printing, *what*, *where* and *when*.

When we look at the above output we see that not all the machines specified in the input file for the configurator are used. This is due to the *perpetual* termination behavior of a task instance and the fact that workers die before new ones are introduced in the workers-pool

Because the number of task instances that are forked, varies during the run and each task instance runs on a separate machine, the number of machines varies in exactly the same way as the number of task instances does. From the output, given above, we can make a graph that shows the number of machines needed during the dynamic expansion and shrinking of our application in its short run (about half a second). It is shown in Figure 3.

To execute our application in *parallel* such that all workers are in the same task instance, we simply change the load on line 5 of `mainprog.mlink` to 5, and do the linking phase again.

Note that the different mappings in the task composition stage and the run-time configuration stage do not affect the semantics of the MANIFOLD source code.

7 Performance Results

We have carried out a number of experiments. Every run of our sequential or restructured application needs a number of parameters. These are (see lines 13-18 in §4), the refinement level of the coarsest grid (we have used 2), the additional refinement level (we have used 0 through 15), the tolerance in the integrator (we have used 1.0e-3 and 1.0e-4). Hereafter we will called runs with these tolerances respectively the "1.0e-3 run" and the "1.0e-4 run".

The relationship between the additional refinement level l and the number of workers w is $w = 2l + 1$ (i.e., the workers on the grids on the main diagonal and subdiagonal to the left in Figure 1). Thus the total number of workers plus master is $2l + 2$. As argued, this latter number is an upper bound for the number of machines used during a distributed run.

We have run and compared the performance results of the sequential and the concurrent versions of our application on a cluster of 32 single processor workstations. Such a cluster is big enough to run the

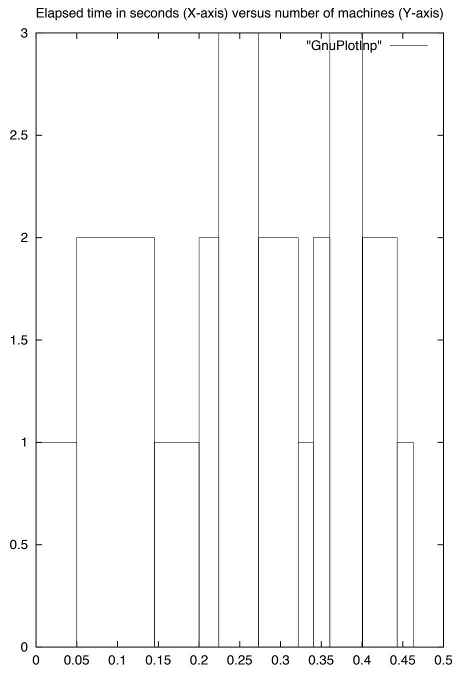


Figure 3: The ebb & flow during the example run of our restructured application.

application with $l = 15$. Unfortunately, in our institute no homogeneous cluster of workstations of that size is available. All the machines in our cluster have an AMD Athlon Processor and a cache size of 256Kb. However 24 machines have a clock cycle of 1200Hz, 5 machines have a clock cycle of 1400Hz, and 3 machines have a clock cycle of 1466Hz. Although these machines have different CPU speeds, their speeds are of the same order of magnitude.

The workstations in the cluster are connected to each other by a switched Ethernet (100 Mbps).

The experiments were done at night. However, even then, we do not have a guarantee that we are the only user. There are always unpredictable effects such as network traffic and file server delays, etc. Furthermore some users of the machines in the cluster, run their own job(s) at night, run screen savers or have runaway Netscape jobs. All this causes differences in performance on identical hardware. These unknown effects cannot be eliminated and are always reflected in our computational results. To even out such "random" perturbations, we ran the two versions of the application five times and collected their *elapsed* or *wall-clock* times (i.e., the actual time the application program runs as it would be measured by a user sitting at the terminal with a stopwatch). The timing measurements were obtained using the UNIX utility `/bin/time`. The results for the 1.0e-3 runs are given in Table 1 and those for 1.0e-4 are in Table 2. Also the average computing time is given. The speedup is given in Table 3 and the weighted average of numbers of machines used during a run is given in Table 4.
For our analysis of the results, we distinguish the following categories of overhead introduced by our restructuring:

- The overhead introduced by the unpredictable effects of working in a multi-user environment. These

	level	*result 1*	*result 2*	*result 3*	*result 4*	*result 5*	*average*
sequential	0	0.02	0.05	0.05	0.04	0.01	0.03
	1	0.04	0.04	0.05	0.03	0.03	0.04
	2	0.06	0.07	0.06	0.05	0.06	0.06
	3	0.11	0.13	0.10	0.11	0.10	0.11
	4	0.20	0.18	0.19	0.22	0.19	0.20
	5	0.42	0.40	0.41	0.38	0.40	0.40
	6	0.86	0.82	0.78	0.90	0.94	0.86
	7	1.87	1.86	1.71	1.71	2.37	1.90
	8	4.10	4.16	3.93	3.83	5.34	4.27
	9	9.16	11.69	9.34	9.21	11.98	10.28
	10	25.63	26.16	23.19	22.89	22.81	24.14
	11	59.91	61.03	57.74	57.68	53.19	57.91
	12	147.96	150.45	140.53	159.00	129.42	145.47
	13	354.49	356.18	334.64	332.50	310.66	337.69
	14	851.78	873.37	815.21	799.96	752.80	818.62
	15	2018.08	2035.48	1920.55	2341.51	1779.49	2019.02
distributed	0	10.44	10.48	10.61	4.70	10.11	9.27
	1	13.99	14.70	14.18	13.58	13.96	14.08
	2	13.33	12.91	13.07	13.21	12.95	13.09
	3	15.55	6.56	5.70	5.95	5.54	7.86
	4	11.12	11.39	11.57	11.66	11.52	11.45
	5	15.28	14.97	17.86	19.17	19.72	17.40
	6	20.62	20.85	32.95	31.12	28.99	26.91
	7	26.53	24.07	32.49	31.86	29.91	28.97
	8	29.90	29.73	29.94	29.81	30.92	30.06
	9	22.10	27.44	22.18	25.81	21.66	23.84
	10	20.49	19.62	18.98	19.92	30.07	21.82
	11	34.41	33.52	30.30	32.88	36.81	33.58
	12	52.05	50.30	49.19	50.47	51.92	50.79
	13	96.11	72.34	69.94	70.62	67.41	75.28
	14	172.31	121.79	104.28	112.98	109.63	124.20
	15	210.39	263.33	236.33	254.88	333.51	259.69

Table 1: The Cluster elapsed times (in seconds) for the 1.0e-3 runs.

effects are totally out of our control in a multi-user environment without dedicated machines.

- The overhead introduced by the concurrency itself (i.e., the overhead of making a sequential program run as a concurrent program).
- The overhead of the coordination layer (i.e., the actual implementation of the overhead of the concurrency).

From the tables we conclude the following:

Looking at Table 1 and Table 2 we see that the differences between the five results are not so big. Probably the effects of working in a multi-user environment are minimal in comparison with the other overhead.

We also see that for the performances with $l < 10$ there is no gain in time for the 1.0e-3 and the 1.0e-4 runs (i.e., the speedup is less than 1.0). Probably the useful computational work done by the workers is too little in comparison with the overhead of the concurrency plus the overhead of the coordination layer.

	level	*result 1*	*result 2*	*result 3*	*result 4*	*result 5*	*average*
	0	0.02	0.01	0.02	0.03	0.01	0.02
	1	0.03	0.05	0.07	0.05	0.04	0.05
	2	0.06	0.08	0.08	0.09	0.06	0.07
	3	0.16	0.13	0.15	0.15	0.15	0.15
	4	0.29	0.29	0.32	0.32	0.28	0.30
	5	0.62	0.69	0.73	0.68	0.69	0.68
	6	1.45	1.58	1.69	1.48	1.47	1.53
	7	3.32	3.87	3.83	3.31	3.33	3.53
sequential	8	7.71	8.53	8.64	7.60	7.70	8.04
	9	18.26	21.03	21.16	18.12	26.43	21.00
	10	46.60	54.07	52.73	45.81	58.99	51.64
	11	116.61	133.40	131.60	120.78	118.47	124.17
	12	289.61	320.81	322.39	283.57	289.45	301.17
	13	686.03	777.77	781.20	684.02	695.59	724.92
	14	1666.30	1882.28	1868.40	1654.06	1684.07	1751.02
	15	3905.65	4418.47	4410.31	3893.35	3962.64	4118.08
	0	10.57	10.48	3.12	10.53	3.69	7.68
	1	15.33	14.54	13.85	10.88	10.62	13.04
	2	12.94	12.92	13.01	12.99	13.08	12.99
	3	14.29	5.71	5.87	5.72	5.60	7.44
	4	11.95	11.66	12.46	11.67	12.43	12.03
	5	15.44	14.95	20.72	15.18	15.68	16.39
	6	20.42	20.34	24.21	20.62	19.77	21.07
	7	29.18	30.28	29.96	27.45	26.51	28.68
distributed	8	30.35	30.70	29.96	30.23	30.19	30.29
	9	26.01	23.38	26.99	28.75	26.05	26.24
	10	35.53	35.84	41.14	44.33	36.48	38.66
	11	46.96	46.09	47.68	45.54	45.23	46.30
	12	68.77	61.04	62.48	64.04	68.78	65.02
	13	180.84	99.49	103.60	94.52	167.93	129.28
	14	217.17	213.33	269.71	221.69	214.00	227.18
	15	526.54	433.24	560.62	440.87	634.47	519.15

Table 2: The Cluster elapsed times (in seconds) for the 1.0e-4 runs.

In Table 4 we see for the $l \geq 10$ runs a gain in time. For those levels the average speedup for the levels 10 through 15 ranges from 1.1 to 7.8 for the 1.0e-3 runs and from 1.3 to 7.9 for the 1.0e-4 runs. However, we also see in Table 4 that this time reduction is accomplished by a growing number of machines. Their averages range for the 1.0e-3 runs from 5.5 to 12.2 machines and for the 1.0e-4 runs from 5.7 to 13.3. Furthermore, we see that the average speedup in a run always lags behind the average number of machines.

Looking at Table 3 and Table 4 we notice that for the levels 12 and higher the speedup is about half of the weighted number of machines used. From this we conclude that for those levels the overhead of the concurrency plus the overhead of the coordination layer seems to be of the same order of magnitude as the actual useful computational work.

Note that the weighted number of machines used during a run can be very different from the actual number of machines used in the application. E.g., the “result 5” run in Table 2 for level $l = 15$ has a weighted average of 11 as shown in Table 3 but from Figure 4 we see that there are moments during the run when it used 32 machines.

	level	*result 1*	*result 2*	*result 3*	*result 4*	*result 5*	*average*
	0	1.9	1.9	1.9	1.8	1.9	1.9
	1	2.3	2.5	2.3	2.4	2.3	2.4
	2	2.8	3.0	2.9	2.7	2.8	2.8
	3	2.3	2.8	2.8	3.0	2.7	2.7
	4	3.3	3.3	2.7	2.9	2.5	2.9
	5	3.3	2.9	3.8	4.0	4.0	3.6
	6	3.2	3.1	3.4	3.6	3.4	3.3
	7	3.6	3.6	3.6	3.6	3.7	3.6
1.0e-3 run	8	3.7	3.9	3.6	3.7	3.4	3.7
	9	4.2	3.7	3.9	4.1	4.5	4.1
	10	5.8	5.7	5.5	5.6	4.8	5.5
	11	6.1	6.1	6.7	6.2	6.2	6.3
	12	7.5	7.7	7.8	7.6	7.6	7.6
	13	7.8	10.1	9.9	10.0	11.0	9.8
	14	8.7	11.9	13.0	12.2	12.9	11.7
	15	15.0	11.7	12.7	12.1	9.7	12.2
	0	1.9	1.9	1.7	1.9	1.9	1.9
	1	2.4	2.5	2.4	2.4	2.1	2.4
	2	2.8	2.9	2.7	2.7	2.8	2.8
	3	2.3	2.7	3.2	2.6	2.4	2.6
	4	3.6	2.7	3.5	2.4	2.5	2.9
	5	3.2	3.5	3.4	3.1	3.1	3.3
	6	3.3	3.6	3.6	3.6	3.5	3.5
	7	3.6	3.7	3.7	3.9	3.5	3.7
1.0e-4 run	8	3.6	3.8	4.0	3.9	4.0	3.9
	9	4.8	4.8	4.9	4.8	4.5	4.8
	10	6.0	6.1	5.8	5.0	5.7	5.7
	11	7.5	7.6	7.7	7.6	7.6	7.6
	12	9.3	10.5	10.5	9.9	9.2	9.9
	13	8.0	13.3	13.1	13.9	8.5	11.4
	14	13.5	13.8	11.2	13.3	13.6	13.1
	15	12.8	15.2	12.2	15.4	11.0	13.3

Table 3: The weighted average of machines per second for the 1.0e-3 and 1.0e-4 runs.

8 Conclusions

Our cut-and-paste restructuring essentially consists of picking out the computation subroutines in the original ANSI C code (the cut), and gluing them together with coordination modules written in **Manifold** (the paste). No rewriting of, or other changes to, these subroutines is necessary: within the new structure, they have the same input/output and calling sequence conventions as they had in the old structure. The **Manifold** glue modules, representing a master/worker protocol, are separately compiled programs that have no knowledge of the computation performed by the ANSI C modules – they simply encapsulate the protocol necessary to coordinate the cooperation of the computation modules running in a parallel/distributed computing environment.

It is remarkable that we can realize the master/worker protocol in such a generic way where the master and the worker manifolds themselves are parameters of the protocol. With the possibility of using different manifolds as actual values for the formal manifold parameters of another manifold, we can easily build meta-coordinators in **Manifold**.

The unique property of **Manifold** which enables such high degree of modularity is inherited from

	level	*result 1*	*result 2*	*result 3*	*result 4*	*result 5*	*average*
	0	0.0	0.0	0.0	0.0	0.0	0.0
	1	0.0	0.0	0.0	0.0	0.0	0.0
	2	0.0	0.0	0.0	0.0	0.0	0.0
	3	0.0	0.0	0.0	0.0	0.0	0.0
	4	0.0	0.0	0.0	0.0	0.0	0.0
	5	0.0	0.0	0.0	0.0	0.0	0.0
	6	0.0	0.0	0.0	0.0	0.0	0.0
	7	0.1	0.1	0.1	0.1	0.1	0.1
1.0e-3 run	8	0.1	0.1	0.1	0.1	0.2	0.1
	9	0.4	0.4	0.4	0.4	0.6	0.4
	10	1.3	1.3	1.2	1.1	0.8	1.1
	11	1.7	1.8	1.9	1.8	1.4	1.7
	12	2.8	3.0	2.9	3.2	2.5	2.9
	13	3.7	4.9	4.8	4.7	4.6	4.5
	14	4.9	7.2	7.8	7.1	6.9	6.6
	15	9.6	7.7	8.1	9.2	5.3	7.8
	0	0.0	0.0	0.0	0.0	0.0	0.0
	1	0.0	0.0	0.0	0.0	0.0	0.0
	2	0.0	0.0	0.0	0.0	0.0	0.0
	3	0.0	0.0	0.0	0.0	0.0	0.0
	4	0.0	0.0	0.0	0.0	0.0	0.0
	5	0.0	0.0	0.0	0.0	0.0	0.0
	6	0.1	0.1	0.1	0.1	0.1	0.1
	7	0.1	0.1	0.1	0.1	0.1	0.1
1.0e-4 run	8	0.3	0.3	0.3	0.3	0.3	0.3
	9	0.7	0.9	0.8	0.6	1.0	0.8
	10	1.3	1.5	1.3	1.0	1.6	1.3
	11	2.5	2.9	2.8	2.7	2.6	2.7
	12	4.2	5.3	5.2	4.4	4.2	4.6
	13	3.8	7.8	7.5	7.2	4.1	5.6
	14	7.7	8.8	6.9	7.5	7.9	7.7
	15	7.4	10.2	7.9	8.8	6.2	7.9

Table 4: Speedup of the 1.0e-3 and 1.0e-4 runs.

its underlying IWIM model in which the communication is set up from the *outside*. The core relevant concept in the IWIM model of communication is isolation of the computational responsibilities from communication and coordination concerns, into separate, pure computation modules and pure coordination modules. This is why the **MANIFOLD** modules in our example can coordinate the already existing computational ANSI C subroutines, without any change. The master and worker manifolds used in the concurrent version just call C functions which are in fact (wrappers around) the C functions of the sequential program.

It is interesting, as illustrated in our restructuring, that we are able to abstract away the details of the computation; that it is possible to focus on the invariant (hidden) properties of our programs, and that we can compile those invariant properties as coordination patterns in **MANIFOLD**. In fact, we compile structure. This coordination structure (compiled **MANIFOLD** coordinators) can transparently run the same computation modules on parallel shared-memory or distributed clusters of workstation platforms. The nice thing in this distillation process is that we end up with *one* tangible piece of code that represents the common coordination structure. Such glue modules (coordinators) can then be compiled separately and stored in what we may call a "protocol library", ready for reuse.

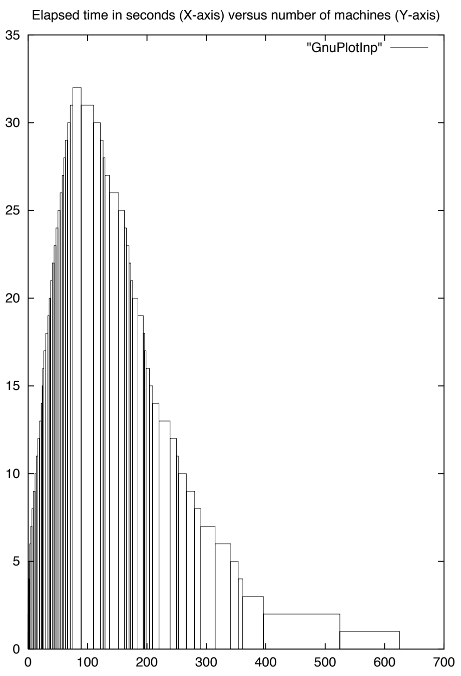


Figure 4: The ebb & flow during the "result 5" run of our restructured application for level 15.

References

[1] C. Zenger. Sparse grids. In W. Hackbusch, editor, *Notes on Numerical Fluid Mechanics*, volume 31, pages 241–251. Vieweg, Braunschweig, 1991.

[2] M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In R. Beauwens and P. de Groen, editors, *Iterative Methods in Linear Algebra*, pages 263–281. North-Holland, Amsterdam, 1992.

[3] H. Bungartz, M. Griebel, D. Röschke, and C. Zenger. Pointwise convergence of the combination technique for the laplace equation. *East-West J. Numer. Math.*, 2:21–45, 1994.

[4] C. Pflaum. Convergence of the combination technique for second-order elliptic differential equations. *SIAM J. Numer. Anal.*, 34:2431–2455, 1997.

[5] C. Pflaum and A. Zhou. Error analysis of the combination technique. *Numerische Mathematik*, 84:327–350, 1999.

[6] B. Lastdrager and B. Koren. Error analysis for function representation by the sparse-grid combination technique. Technical Report MAS-R9823, CWI, 1998. Available on-line http://db.cwi.nl/rapporten/index.php?jaar=1998&dept=13.

[7] M. Griebel. The combination technique for the sparse grid solution of pde's on multiprocessor machines. *Parallel Processing Letters*, 2:61–70, 1992.

[8] C.T.H. Everaars, F. Arbab, and B. Koren. Parallel, distributed-memory implementation of sparse-grid methods for three-dimensional fluid-flow computattions. Technical Report SEN–R0039, CWI, 2000. Available online http://db.cwi.nl/rapporten/index.php?jaar=2000&dept=15.

[9] C.T.H. Everaars, F. Arbab, and B. Koren. Using coordination to restructure sequential source code into a concurrent program. In *Proceedings of the International Conference on Software Maintenance, Florence*, pages 342–351, 2001.

[10] C.T.H. Everaars and B. Koren. Using coordination to parallelize sparse-grid methods for 3D CFD problems. *Parallel Computing*, 24(7):1081–1106, 1998. Special issue on coordination languages for parallel programming.

[11] C.T.H. Everaars, B. Koren, and F. Arbab. Dynamic process composition and communication patterns in irregularly structured applications. In Jose Rolim et al., editor, *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 1046–1054. Springer-Verlag, Berlin, 1999.

[12] C.T.H. Everaars, F. Arbab, and B. Koren. Dynamic process composition and communication patterns in irregularly structured applications. *Concurrency: Practice and Experience*, 12:157–174, 2000. Extended version.

[13] M. Griebel. Parallel multigrid methods on sparse grids. In W. Hackbusch and U. Trottenberg, editors, *Multigrid methods III*, volume 98 of *International Series of Numerical Mathematics*, pages 211–221. Birkhäuser, Basel, 1991.

[14] M. Griebel. A parallelizable and vectorizable multi-level algorithm on sparse grids. In W. Hackbusch, editor, *Parallel Algorithms for Partial Differential Equations*, volume 31 of *Notes on Numerical Fluid Mechanics*, pages 94–100. Vieweg, Braunschweig, 1991.

[15] B. Koren, P.W. Hemker, and C.T.H. Everaars. Multiple semi-coarsened multigrid for 3D CFD. In *Proceedings of the 13th AIAA Computational Fluid Dynamics Conference, Snowmass Village, CO, 1997*, pages 892–902, Reston, VA, 1997. American Institute of Aeronautics and Astronautics. AIAA-paper 97-2029.

[16] B. Koren, P.W. Hemker, and C.T.H. Everaars. Sparse-grid solution of the steady Euler equations of gas dynamics. In *Proceedings of Fourth European Computational Fluid Dynamics Conference*, volume 2 of *Computational Fluid Dynamics*, pages 252–257. Wiley, Chichester, 1998.

[17] B. Lastdrager, B. Koren, and J.G. Verwer. The sparse-grid combination technique applied to time-dependent advection problems. *Appl. Numer. Math.*, 38:377–401, 2001.

[18] B. Lastdrager, B. Koren, and J.G. Verwer. Solution of time-dependent advection-diffusion problems with the sparse-grid combination technique and a Rosenbrock solver. *Comput. Meth. Appl. Math.*, 1:86–98, 2001.

[19] J.G. Verwer, E.J. Spee, J.G. Blom, and W. Hundsdorfer. A second-order Rosenbrock method applied to photochemical dispersion problems. *SIAM J. Sci. Comput.*, 20:1456–1480, 1999.

[20] K. Dekker and J.G. Verwer. *Stability of Runge-Kutta Methods for Stiff Nonlinear Differential Equations.* Elsevier North-Holland, Amsterdam, 1984.

[21] D. Lanser, J.G. Blom, and J.G. Verwer. Time integration of the shallow water equations in spherical geometry. *J. Comput. Phys.*, 171:1–21, 2001.

[22] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communication of the ACM*, 35(2):97–107, 1992.

[23] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[24] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag, April 1996.

[25] F. Arbab. The influence of coordination on program structure. In *Proceedings of the* 30^{th} *Hawaii International Conference on System Sciences.* IEEE, 1997.

[26] F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaars. Reusable coordinator modules for massively concurrent applications. *Software: Practice and Experience*, 28(7):703–735, 1998. Extended version.

[27] F. Arbab. Manifold version 2: Language reference manual. Technical report, CWI, 1996. Available on-line http://www.cwi.nl/ftp/manifold/refman.ps.Z.

[28] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1994.

[29] B. Nicols, D. Buttlar, and J.P. Farrell. *Pthreads Programming.* O'Reilly & Associates, Inc., Cebastopol, CA, 1996.