Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Parallel, Distributed-Memory Implementation of Sparse-Grid Methods for Three-Dimensional Fluid-Flow Computations

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

Software Engineering (SEN)

SEN-R0039 December 2000

Report SEN-R0039
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Parallel, Distributed-Memory Implementation of Sparse-Grid Methods for Three-Dimensional Fluid-Flow Computations *

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

ABSTRACT
A workable approach for modernization of 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 two different pieces of existing sequential Fortran code from computational fluid dynamics (CFD). The first Fortran source code deals with a standard CFD problem solved with the so-called sparse-grid method and the other solves the same problem but now with the so-called semi-sparse-grid method. Applying our cut-and-paste method to these two programs results in *one* generally applicable coordinator module that can restructure both sequential programs into parallel applications (which now run on a shared memory machine) as well as distributed applications (which now run on a cluster of workstations). Before we give the actual restructuring of the two sequential programs, we first test the coordinator module with a simplified "toy" example. 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 reusability, protocol library, multi-grid methods, sparse-grid methods, computational fluid dynamics, three-dimensional flow problems.

*Partial funding for this project was provided by the National Computing Facilities Foundation (NCF), under project number NRG 98.04.

Note: Work carried out under the projects Coordination Languages (SEN3) and Industrial Processes (MAS2).

Contents

1 Introduction 3

2 The Problem and the Multigrid Method 4
2.1 Equations 4
2.1.1 Continuous Equations 4
2.1.2 Discretized Equations 5
2.2 Sparse-grid Methods 6
2.2.1 Standard Multigrid 6
2.2.2 Multiple Semi-coarsened Multigrid 8

3 The Manifold Coordination Language 9

4 The Cut 11

5 The Paste 14
5.1 The Glue 15
5.2 The Implementation 17
5.2.1 MANIFOLD Terminology 18
5.2.2 The Gluing Modules 21
5.2.3 How it Runs 23
5.2.4 The Behavior Interface of the Master and Worker 26

6 The Test 28
6.1 The Sequential Source Code of Toy Example 28
6.2 The Concurrent Version of the Toy Example 30
6.3 Running the Toy Example 37
6.4 The Performance 39

7 The Restructuring 40

8 The Performance Analysis 41
8.1 Ebb & Flow 41
8.2 Performance Results 43
8.2.1 Shared Memory Performance Results 44
8.2.2 Distributed Memory Performance Results 47

9 Conclusions 48

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 Fortran 77 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 Fortran. 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 modernization of 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 two different pieces of existing sequential Fortran code from computational fluid dynamics (CFD). These two pieces of code were developed at CWI by a group of researchers in the department of Numerical Mathematics, within the framework of the BRITE-EURAM Aeronautics R&D Programme of the European Union. Both implement a multi-grid solution algorithm for the Euler equations representing three-dimensional, steady, compressible flows. In the first piece of code, the problem is solved using the so-called sparse-grid method, and the other uses the so-called semi-sparse-grid method. The developers of these programs 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 these two programs results in *one* generally applicable coordinator module that can restructure both sequential programs into parallel applications (which run on a shared memory machine) as well as distributed applications (which now run on a cluster of workstations). We have reported earlier about the restructuring of these Fortran programs [1]. However, the coordinator modules developed there were only able to restructure the source code into a parallel application.

Clearly, the details of the computational algorithms used in the original program are too voluminous to reproduce here, and such computational detail is essentially irrelevant for our restructuring. Instead, we use a simplified pseudo-program here that has the same logical design and structure as the original program

The rest of this paper is organized as follows. In §2 we give the sparse-grid solution problem for the steady, 3-D Euler equations of gas dynamics and discuss different solving methods. 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. It is beyond the scope of this paper to present all the details of the syntax and semantics of the MANIFOLD language. In §4 we present the simplified pseudo-program

as distilled from the original Fortran 77 program, explore its structure and try to identify and isolate software components in it. 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. In §6 we test those generic gluing modules with a "toy" example that has the same structure as the original sequential Fortran code and we also give some performance results. The actual restructuring of the two original sequential programs can be found in §7. In §8 we compare the performance results before and after the restructuring. Finally, the conclusion of the paper is in §9.

2 The Problem and the Multigrid Method

One of the major challenges in science and technology is the fast numerical solution of partial differential equations. Important examples of such equations are those of fluid mechanics. When partial differential equations are solved numerically, they must be discretized, i.e., their solution, which is a set of functions defined over an area, is approximated by a set of – say – $\mathcal{O}(N^d)$ real numbers, where d is the space dimension of the problem ($d = 1, 2$ or 3). Thus, the original differential equations are transformed into a system of $\mathcal{O}(N^d)$ algebraic equations with the aforementioned $\mathcal{O}(N^d)$ real numbers as the unknowns. For $d = 3$ the size of the system can be very large. To solve these large systems, various techniques have been developed. Among these, the multigrid methods are optimal in the sense that the amount of computational work to solve the algebraic system is only linear with the number of unknowns. For all other known solution methods, the amount of work grows faster than linearly with the number of unknowns. For literature on multigrid techniques, see, e.g., [2, 3, 4], where [2] is recommended for an elementary introduction.

Novel multigrid techniques to speed up the solution of systems of discrete equations are the so-called sparse-grid techniques; see [5] and its references. Sparse-grid techniques are very attractive from the viewpoint of computational efficiency, particularly for 3-D problems. The gain in efficiency is achieved through a strong reduction of the number of grid points. Of course, this goes at the expense of numerical accuracy. Fortunately, the sparse-grid-of-grids approach has a better ratio of discrete accuracy over number of grid points [6] than a standard multigrid method (which in turn already has a much better performance in this sense than a single-grid method).

This section is organized as follows: First we discuss the sparse-grid solution method for the steady, 3-D Euler equations of gas dynamics [7, 8]. In §2.1, we introduce the discrete equations under consideration and in §2.2, we describe the concept of sparse-grid methods.

2.1 Equations

2.1.1 Continuous Equations

We consider the flow of a perfect, di-atomic gas (air, e.g.) in three dimensions (3-D). The unknown quantities that describe the gas flow are the gas velocity components in the three coordinate directions, u, v and w; the gas density ρ; and the gas pressure p. Neglecting friction forces, the gas flow is described by the steady, 3-D Euler equations

$$\frac{\partial f(q)}{\partial x} + \frac{\partial g(q)}{\partial y} + \frac{\partial h(q)}{\partial z} = 0, \tag{1a}$$

in which q is the so-called state vector

$$q = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{pmatrix}, \tag{1b}$$

with e the sum of internal and kinetic energy, satisfying the perfect-gas relation $e = \frac{1}{\gamma - 1}\frac{p}{\rho} + \frac{1}{2}\left(u^2 + v^2 + w^2\right)$, and in which $f(q)$, $g(q)$ and $h(q)$ are the so-called flux vectors

$$f(q) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ \rho u(e + \frac{p}{\rho}) \end{pmatrix}, \quad g(q) = \begin{pmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vw \\ \rho v(e + \frac{p}{\rho}) \end{pmatrix}, \quad h(q) = \begin{pmatrix} \rho w \\ \rho wu \\ \rho wv \\ \rho w^2 + p \\ \rho w(e + \frac{p}{\rho}) \end{pmatrix}. \tag{1c}$$

2.1.2 Discretized Equations

The above equations are too intricate to be integrated by pen and paper only. Fortunately, good tools are available to integrate the equations numerically. For this, usually, the equations are discretized in the integral form

$$\oint_{\partial\Omega^\star} \left(f(q)n_x + g(q)n_y + h(q)n_z\right) \mathrm{d}s = 0, \tag{2}$$

where $\partial\Omega^\star$ is the boundary of an arbitrary subdomain $\Omega^\star$ of the computational domain Ω, and where n_x, n_y and n_z are the x-, y- and z-components, respectively, of the outward unit normal on $\partial\Omega^\star$. The equations represent the laws of conservation of mass, momentum and energy, respectively.

We can divide the computational domain into a finite number of virtual cells (finite volumes) and then require that the integral form of equation (1a) is satisfied for each of these finite volumes. Denoting the finite volumes by $\Omega_{i,j,k}$, $i = 0, 1, \ldots, i_{\max}$, $j = 0, 1, \ldots, j_{\max}$, $k = 0, 1, \ldots, k_{\max}$, this leads to the following system of equations

$$\oint_{\partial\Omega_{i,j,k}} \left(f(q)n_x + g(q)n_y + h(q)n_z\right) \mathrm{d}s = 0, \quad \forall i, j, k. \tag{3}$$

So, per finite volume we have five numbers which represent the gas flow in that volume: the values of the three velocity components and the values of density and pressure. The five values are found by solving for each finite volume: the system of five equations (3).

As finite volumes, arbitrarily shaped hexahedra are considered, the structured subdivision being such that – if existent – $\Omega_{i\pm1,j,k}$, $\Omega_{i,j\pm1,k}$ and $\Omega_{i,j,k\pm1}$ are the neighboring volumes of $\Omega_{i,j,k}$. The type of finite-volume method applied is the cell-centered one. Following the so-called Godunov approach [9], along each cell face $\partial\Omega_{i,j,k}$, the flux vector is assumed to be constant and to be determined by a uniformly constant left and right state, q^l and q^r, only. Doing so, the flux evaluation is identical to the numerical solution of the 1-D Riemann problem for a non-isenthalpic perfect-gas flow. For this, we apply the 3-D extension of the 2-D P-variant [10] of Osher's approximate Riemann solver [11]. For the left and right cell-face states, we take the first-order accurate approximations

$$\begin{pmatrix} q^l_{i+\frac{1}{2},j,k} \\ q^r_{i+\frac{1}{2},j,k} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i+1,j,k} \end{pmatrix}, \quad \begin{pmatrix} q^l_{i,j+\frac{1}{2},k} \\ q^r_{i,j+\frac{1}{2},k} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i,j+1,k} \end{pmatrix}, \quad \begin{pmatrix} q^l_{i,j,k+\frac{1}{2}} \\ q^r_{i,j,k+\frac{1}{2}} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i,j,k+1} \end{pmatrix}. \tag{4}$$

At a later stage, these approximations can be replaced by higher-order accurate ones, in which case also limiters can be introduced.

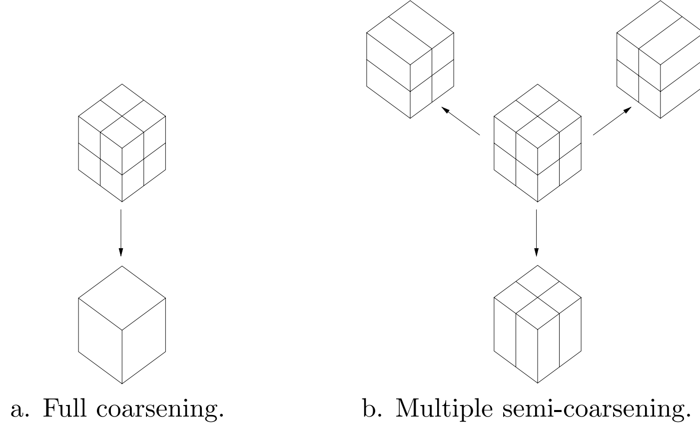
2.2 Sparse-grid Methods

In summary, by discretizing the flow problems, we create a set of – say – N^d finite volumes $\Omega_{i,j,k}$, N^d gas states $q_{i,j,k}$ and N^d nonlinear equations of the form (3). As mentioned in the introduction, N^d may be very large, particularly in 3-D ($d = 3$). All methods to solve such large systems of equations are iterative: a guessed initial solution is improved step-by-step during the solution process. As also mentioned in the introduction, most iterative methods have the drawback that the rate of convergence to the final numerical solution decreases with increasing N^d. The reason is that for larger systems of equations, not only the number of equations increases, but – mostly – also the effect of the separate iterations (solution corrections) decreases. Multigrid methods are capable of alleviating this problem; they can accelerate the iteration processes. How this is done can be briefly explained in the following way. Suppose we want to solve a 3-D flow problem on a grid with 128^3 finite volumes (i.e., in the present case, a system of 5×128^3 unknowns). To solve this system of equations, we invoke the help of a corresponding, twice-coarser grid with 64^3 finite volumes. Given the initial guess of the flow solution on the 128^3-grid, one can start the iteration by substituting this guess into (3). Then, in each finite volume one gets five defect values (one value for the mass defect, three values for the momentum defect and one for the energy defect). By solving the (eight times cheaper) coarse-grid flow problem, extended with righthand sides obtained by proper summation of local fine-grid defect values, one finds a correction to the fine-grid solution. Because it comes from the coarse grid, this correction cannot completely remove the fine-grid solution error, but it *can* remove the important low Fourier-frequency parts of the fine-grid solution error. The remaining, high Fourier-frequency parts can – in principle – be removed by an appropriate smoothing algorithm (the smoother). One may now argue that since the 64^3 problem is still large, the above two-grid algorithm is still expensive. This can be fixed by also considering the corresponding 32^3-problem and, if desired, also the corresponding 16^3-problem, etc. Doing so, one applies a multigrid algorithm. On each of the different coarser grids, one effectively reduces a different part of the spectrum of the fine-grid solution error.

The multigrid method outlined above is a standard multigrid method, i.e., in going from a fine grid to the next coarser grid, the number of cells is halved in each coordinate direction, which leads to the strong reduction in the number of grid points by a factor 8 per coarsening. A significant difficulty now with standard multigrid methods for 3-D problems, compared to 2-D problems, is that in 3-D, the requirements to be imposed on the smoother are much more severe. In 3-D, standard coarsening implies restriction from each set of $2 \times 2 \times 2$ cells to a single cell only. Because the set of eight cells can support more high-frequency errors than the two-dimensional 2×2–set, 3-D standard multigrid imposes stronger requirements on the smoother than 2-D standard multigrid. Standard multigrid may not perform satisfactorily for 3-D generalizations of 2-D problems, for which it does perform well. A fix to this might be found in deriving a more powerful smoother, keeping the other components of the multigrid method the same. A more natural remedy is not to apply standard, i.e., full coarsening, but to use multiple semi-coarsening instead. Figures 1a and 1b show standard coarsening and multiple semi-coarsening, respectively.

2.2.1 Standard Multigrid

In this section we first describe in more detail the standard 3-D multigrid algorithm. We use the 3-D generalization of the optimal 2-D multigrid approach that was originally described in [10].



a. Full coarsening. b. Multiple semi-coarsening.

Figure 1: Two types of 3D coarsenings.

As the smoothing technique for the first-order discretized Euler equations, we prefer to apply collective symmetric point Gauss-Seidel relaxation. *Point* refers to the property that during the update of the local state vector $q_{i,j,k}$, all other state vectors are kept fixed. *Collective* refers to the property that the update of $q_{i,j,k}$ is done for all of its five components simultaneously. Further, *symmetric* means that after a relaxation sweep (i.e., an update of all state vectors $q_{i,j,k}$) in one direction, a new sweep in the reverse direction is made. The four different symmetric relaxation sweeps that are possible on a regular 3-D grid, are performed alternatingly. At each volume visited during a relaxation sweep, the system of five nonlinear equations is approximately solved by (exact) Newton iteration. This relaxation method is simple and robust.

As the standard multigrid method we apply the nonlinear version (the so-called full approximation scheme [4], abbreviated as FAS), preceded by nested iteration (also called full multigrid [4], which is abbreviated as FMG). For this we construct a nested set of grids such that each finite volume on a coarse grid is the union of $2 \times 2 \times 2$ volumes on the next finer grid (full coarsening, figure 1a). Let $\Omega_0, \Omega_1, \ldots, \Omega_{\lambda_{\max}}$ be the sequence of such nested grids, with Ω_0 the coarsest and $\Omega_{\lambda_{\max}}$ the finest grid. Then, nested iteration is applied to obtain a good initial solution on $\Omega_{\lambda_{\max}}$, whereas nonlinear multigrid is applied to converge to the solution on the finest grid, $q_{\lambda_{\max}}$. The first iterand for the nonlinear multigrid cycling is the solution obtained by nested iteration. We proceed to discuss both stages in more detail.

The nested iteration starts with a user-defined initial estimate for q_0, the solution on the coarsest grid. To obtain an initial solution on a finer grid $\Omega_{\lambda+1}$, first the solution on the coarser grid Ω_λ is improved by a single nonlinear multigrid cycle. Hereafter, this solution is prolongated to the finer grid $\Omega_{\lambda+1}$. These steps are repeated until the highest level (finest grid) has been reached.

Let $N_\lambda(q_\lambda) = 0$ denote the nonlinear system of first-order discretized equations on Ω_λ, then a single nonlinear multigrid cycle is recurrently defined by the following steps:

1. Improve on Ω_λ the latest obtained solution q_λ by applying n_{pre} relaxation sweeps.

2. Compute on the next coarser grid $\Omega_{\lambda-1}$ the right-hand side $r_{\lambda-1} = N_{\lambda-1}(q_{\lambda-1}) - I_\lambda^{\lambda-1} N_\lambda(q_\lambda)$, where $I_\lambda^{\lambda-1}$ is a restriction operator for right-hand sides.

3. Approximate the solution of $N_{\lambda-1}(q_{\lambda-1}) = r_{\lambda-1}$ by applying n_{FAS} nonlinear multigrid cycles. Denote the approximation obtained as $\tilde{q}_{\lambda-1}$.

4. Correct the current solution by: $q_\lambda = q_\lambda + \tilde{I}^{\lambda}_{\lambda-1}(\tilde{q}_{\lambda-1} - q_{\lambda-1})$, where $\tilde{I}^{\lambda}_{\lambda-1}$ is a prolongation operator for solutions.

5. Improve again q_λ by applying n_{post} relaxations.

Steps (2),(3) and (4) form the coarse-grid correction (all three are skipped on the coarsest grid). The efficiency of a coarse-grid correction depends in general on the coarseness of the coarsest grid. The restriction operator $I^{\lambda-1}_{\lambda}$ and the prolongation operator $\tilde{I}^{\lambda}_{\lambda-1}$ are defined in [8].

2.2.2 Multiple Semi-coarsened Multigrid

In the case of the semi-coarsened multigrid method we also use FAS as the basic multigrid algorithm, and on each grid we apply collective symmetric point Gauss-Seidel relaxation as the smoothing technique. In the semi-coarsened multigrid method, however, we replace the sequentially ordered set of grids $\Omega_\lambda, \lambda = 0, \dots, \lambda_{\max}$, by a partially ordered set of grids $\Omega_{l,m,n}$, $l = 0, 1, \dots, l_{\max}$, $m = 0, 1, \dots, m_{\max}$, $n = 0, 1, \dots, n_{\max}$, with $\Omega_{0,0,0}$ the coarsest and $\Omega_{l_{\max},m_{\max},n_{\max}}$ the finest grid. Now, the level of grid $\Omega_{l,m,n}$ is defined as the sum $l+m+n$. The nesting and the semi-coarsening relation between these grids is described in [12, 13].

Also here, nested iteration is applied to obtain a good initial solution on the finest grid. We proceed to discuss the present nested iteration and nonlinear multigrid iteration in more detail. The nested iteration starts with a user-defined initial estimate on the coarsest grid, $\Omega_{0,0,0}$, i.e., at level 0 ($= 0 + 0 + 0$). The estimate is improved by relaxation. The approximate solution $q_{0,0,0}$ is prolongated (level-by-level) to all grids up to and including level 3 (i.e., to all grids $\Omega_{l,m,n}$ for which $l + m + n = 3$, with $l \le l_{\max}$, $m \le m_{\max}$ and $n \le n_{\max}$). The 3-D prolongation is according to formula (29) in [5] (see Appendix A in [8] for the implementation in the present 3-D Euler context). Next, the solution $q_{1,1,1}$ is improved by a single nonlinear multigrid cycle and prolongated to all grids up to and including level 6 (i.e., to all grids $\Omega_{l,m,n}$ for which $l+m+n = 6$, with $l \le l_{\max}$, $m \le m_{\max}$ and $n \le n_{\max}$). For simplicity, we assume that $l_{\max} = m_{\max} = n_{\max}$. Then, the above process can be repeated in a straightforward manner up to and including level $3l_{\max}$.

A single nonlinear multigrid cycle at level $l + m + n$ is recurrently defined by the following steps:

1. Improve the solutions at level $l + m + n$ by applying n_{pre} relaxation sweeps.

2. Compute on all grids at the next coarser level, $(l + m + n) - 1$ the same right-hand sides as in the standard multigrid method, but use another restriction operator, viz., the one described in Appendix B of [8]. (The restriction of defects is still natural, i.e., by summation over all sub-cells.)

3. Approximate the solutions at the coarser level $(l+m+n)-1$ by applying a single nonlinear multigrid cycle at level $(l + m + n) - 1$.

4. Correct the current solutions at level $l + m + n$ by one of two alternative correction prolongations. One prolongation can be seen as an extension to 3-D and to systems of equations, of the prolongation due to Naik and Van Rosendale [14]. (It uses prolongation weights that are proportional to the absolute values of the restricted defect components.) The other correction prolongation is the one proposed in [5]. (It is the correction-prolongation version of the solution prolongation described

in Appendix A of [8], using fixed prolongation weights.) In Appendix C of [8], both correction prolongations are described explicitly.

5. Improve the solutions at level $l+m+n$ by applying n_{post} relaxation sweeps.

When multiple semi-coarsening is applied to solve a system of equations defined on the single, finest grid $\Omega_{l_{\max},m_{\max},n_{\max}}$, and when all coarser grids $\Omega_{l,m,n}$, level$\equiv l+m+n < l_{\max}+m_{\max}+n_{\max}$ contribute to the solution process, we speak of *full-grid-of-grids* semi-coarsening. hexahedron, in figure full-grid-of-grids semi-coarsening is that many grid cells are needed in total. With N^3 the total number of cells on the finest grid $\Omega_{l_{\max},m_{\max},n_{\max}}$, in 3-D, asymptotically standard multigrid uses $\frac{9}{8}N^3$ grid cells versus $8N^3$ cells for the full-grid-of-grids approach. An efficiency improvement can be achieved by thinning out the grid-of-grids, i.e., by deleting fine grids. Then, if no finest grid is available any more, accurate approximations can be constructed by extrapolation [5, 15, 16]. Most ambitious in this respect is the sparse-grid-of-grids approach, where only grids $\Omega_{l,m,n}$, level$\leq l_{\max}$ contribute. With the full grid-of-grids depicted as a cube

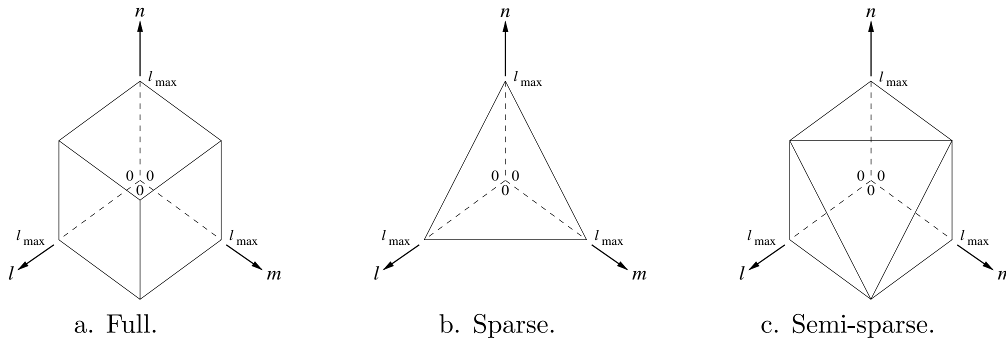


Figure 2: Cubic, full grid-of-grids and the corresponding sparse and semi-sparse grid-of-grids.

in figure 2a, the corresponding sparse grid-of-grids is the subset given in figure 2b. The reduction in the numbers of grid cells is enormous. The computational complexity of the sparse-grid-of-grids approach is $\mathcal{O}(N\log^2 N)$, i.e., almost the complexity of a 1-D problem only! Theoretically, the sparse-grid-of-grids approach has the best ratio of discrete accuracy over number of grid points used [6]. In the ideal case, the full grid-of-grids should be completely replaced by a sparse grid-of-grids. In practice, although very fast, the accuracy of the sparse-grid approximations is slightly disappointing. It appears that more accurate approximations are obtained *not* by only increasing the number of levels, but also by dropping the cells with extreme aspect ratios. This leads to the compromise of the semi-sparse grid-of-grids [15]. This uses the family of grids $\Omega_{l,m,n}$, level$\leq 2l_{\max}$, $\max(l,m,n)\leq l_{\max}$ (see figure 2c), which (asymptotically) still has a computational complexity which is much smaller than that of the single-grid approach, viz., $\mathcal{O}(N^2\log^2 N)$, i.e., still almost the complexity of a 2-D problem only.

3 The Manifold Coordination Language

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.

parallel programming language; it is a *coordination language* as opposed to a *computation language* [17]. MANIFOLD is a *complete* language (as opposed to a language extension, like Linda [18]) 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 networks in reaction to events.

MANIFOLD is based on the IWIM (*Idealized Worker Idealized Manager*) model of communication [19]. 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 to or receive messages 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 between the modules responsible for computation (the workers) and the modules responsible for coordination (the managers). This strengthens the modularity and enhances the re-usability of both types of modules (see [20, 19, 21]).

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 [22], 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 runtime of an application, CONFIG determines the actual host(s) where the

processes which 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 [23].

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 is 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 §5.2.1 just before we use it in the description of MANIFOLD source code of the coordinators in §5.1. For a general introduction to the MANIFOLD language see [24].

4 The Cut

In this section we explore the structure of two Fortran programs. The structure of the implementation of sequential Fortran programs for the sparse-grid method and the semi-sparse-grid method are very similar and are based on a data structure which is especially designed for the implementation of adaptive sparse-grid algorithms in three dimensions [13]. Both Fortran programs consist of a data definition section, a main program and some 200 subroutines with a total length of some 8000 lines. Because the Fortran source codes are so similar we give only the source code for the sparse-grid method. Instead of its full source code, we give only the relevant part of the Fortran code for the sparse-grid method, viz., a schematized version of the main program, the subroutine `fas` (Full Approximation Storage algorithm, also known as nonlinear multigrid algorithm, see [4], p. 171 and on) and the subroutine `scanlv` (a subroutine for performing a user-defined operation on all grids at some multigrid level). With this small part of the Fortran code we can explain the essential implementation aspects of the sparse-grid method, as well as its actual restructuring into a concurrent application.

```
1         program          oneram6
2
3         include         'basis3.i'
4         integer          level,levelmax,
5        +                 nmin,mmin,lmin,nmax,mmax,lmax
6         logical          convergence
7         external         fas,prolsolgr,scanlv
8         common /gridset/ nmin,mmin,lmin,nmax,mmax,lmax
9
10 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
11 c      main program
12 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
13
14 c      ----------------------------------------------------------------
15 c      Initialization data structure and some initial computations
16        .....
17        .....
18
19 c      ----------------------------------------------------------------
20 c      begin nested iteration
21
22        do 20 level= 0,lmax
23
24 c         -------------------------------------------------------------
25 c         begin nonlinear multigrid iteration from all grids at
26 c         actual finest level
27
28   10      call fas (level)
29
30           if (convergence) then
31              continue
32           else
33              goto 10
34           endif
35
36 c         end nonlinear multigrid iteration from all grids at
37 c         actual finest level
38 c         -------------------------------------------------------------
39
```

```
40 c        ---------------------------------------------------------------
41 c        begin solution prolongations from all grids at
42 c        actual finest level
43
44          if (level.lt.lmax) then
45             call scanlv (level+1,nmin,nmax,mmin,mmax,lmin,lmax,
46      +                   prolsolgr)
47          endif
48
49 c        end solution prolongations from all grids at
50 c        actual finest level
51 c        ---------------------------------------------------------------
52
53    20  continue
54
55 c     end nested iteration
56 c     ------------------------------------------------------------------
57
58 c     ------------------------------------------------------------------
59 c     begin solution prolongation to finest level
60
61       do 30 level= lmax+1,levelmax
62          call scanlv (level,nmin,nmax,mmin,mmax,lmin,lmax,prolsolgr)
63    30  continue
64
65 c     end solution prolongation to finest level
66 c     ------------------------------------------------------------------
67
68       end
69
70       subroutine       fas (level)
71
72       integer          level,ilevel,
73      +                 nmin,mmin,lmin,nmax,mmax,lmax
74       logical          origrhs,plus
75       external         copyrhsgr,copysolgr,pointgsgr,prolcorgr,
76      +                 restrictgr,rhsgr,scanlv
77       common /residu/  origrhs,plus
78       common /gridset/ nmin,mmin,lmin,nmax,mmax,lmax
79       external         copyrhsgr,copysolgr,pointgsgr,prolcorgr,
80      +                 restrictgr,rhsgr,scanlv
81
82 ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
83 c     subroutine for nonlinear multigrid iteration
84 ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
85
86       ilevel= level
87
88 c     pre-relaxations
89    10  call scanlv (ilevel,nmin,nmax,mmin,mmax,lmin,lmax,pointgsgr)
90
91       if (ilevel.eq.0) then
92          goto 20
93       endif
94
95 c     computation of defects
96       call scanlv (ilevel,nmin,nmax,mmin,mmax,lmin,lmax,copyrhsgr)
97       origrhs= .false.
98       plus=    .false.
99       call scanlv (ilevel,nmin,nmax,mmin,mmax,lmin,lmax,rhsgr)
100
101 c     computation of coarse-grid righthand sides
102       call scanlv (ilevel-1,nmin,nmax,mmin,mmax,lmin,lmax,restrictgr)
103       origrhs= .true.
104       plus=    .true.
105       call scanlv (ilevel-1,nmin,nmax,mmin,mmax,lmin,lmax,rhsgr)
106
107 c     back-up of coarse-grid solutions
108       call scanlv (ilevel-1,nmin,nmax,mmin,mmax,lmin,lmax,copysolgr)
109
110       ilevel= ilevel-1
111       goto 10
112
113 c     post-relaxations
114    20  call scanlv (ilevel,nmin,nmax,mmin,mmax,lmin,lmax,pointgsgr)
115
116       if (ilevel.eq.level) then
117          goto 40
118       else
119          goto 30
120       endif
121
122 c     prolongation of corrections
123    30  call scanlv (ilevel+1,nmin,nmax,mmin,mmax,lmin,lmax,prolcorgr)
124
125       ilevel= ilevel+1
126       goto 20
127
128    40  return
129       end
130
```

```
131
132       subroutine scanlv (lev,nmin,nmax,mmin,mmax,lmin,lmax,tkgrid)
133
134       integer    lev,nmin,nmax,mmin,mmax,lmin,lmax,n,m,l
135       external   tkgrid
136
137 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
138 c     subroutine for performing the user-defined operation tkgrid on
139 c     all grids at multigrid level lev
140 cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
141
142       do 20 n= nmin,nmax
143          do 10 m= mmin,mmax
144             l= lev-m-n
145             if ((l.le.lmax).and.(l.ge.lmin)) then
146                call tkgrid (n,m,l)
147             endif
148   10     continue
149   20  continue
150
151       return
152       end
```

The program starts with initializing the data structure and with some initial computations (lines 14-17). After this a nested iteration starts (the do 20 loop on lines 22-53) and the program ends with some final computations (lines 61-63). A closer look at the nested iteration shows us that the routine `fas` (lines 70-129) is called (line 28) and that it is followed by some more computations (lines 44-47).

A quick look at the source code (1-152) shows that the routine `scanlv` (lines 132-152) plays an important role in our Fortran program. From its source code we see that this routine visits a number of grids (denoted by the first seven arguments on line 132) and performs a user-defined operation named `tkgrid` (denoted by the last formal argument on line 132) on all the cells of those grids. For example, on line 45, `scanlv` visits a number of grids and performs a `prolsolgr` operation (which is the actual argument for the formal argument `tkgrid` of `scanlv`) on all the cells of those grids.

In the `fas` routine we see that `scanlv` is called with various user-defined operations (lines 89, 96, 99, 102, 105, 108, 114 and 123). Because it is our aim to restructure this Fortran program in a concurrent (parallel or distributed) structure we have special interest in which operations possess concurrent properties. In general we can say that every grid operation with the property that it only reads and writes data from and to its own grid, can be restructured to run concurrently. In our program it turns out that the `pointgsgr` operation that takes care of the pre- and post-relaxations (lines 89 and 114, respectively) has this property. `pointgsgr` carries out a Gauss-Seidel relaxation on all cells of a particular grid and because it is very computing-intensive, it is a good candidate to run it concurrently on all the grids to be visited in a `scanlv` call.

Most other operations performed in the `scanlv` calls do not have this property. They perform computations on the cells of a grid (and change their data) by using data from cells of other grids. Therefore, we must abide by the order in which the grids are visited in `scanlv` and for this reason those operations cannot run concurrently.

The structure of the Fortran source code as just described, is in fact similar to the simplified pseudo-code in figure 3. Here too, after some initialization work and some initial sequential computations, we get to a loop in which a number of operations are performed. Some of these operations cannot be done concurrently, whereas others can. (In the pseudo-code in figure 3 we have only two kinds of operations, which is different than the original Fortran code. However, this difference is not essential.) The simplified pseudo-code also ends with some sequential computations. A simple way to restructure the schema of figure 3 into a concurrent one is to introduce a workers-pool (containing a number of workers) every time we arrive at heavy computations that can be done concurrently. We show this new schema in figure 4. Each worker in the workers-pool performs the same operation on a different data segment independently of the others. In a program built according this schema, none of the computational processes actually runs concurrently until it reaches a concurrent region. Then the multiple workers (i.e., the parallel or

```
program SEQ_CODE
begin

Preamble:
  - Some initialization work
  - Some initial sequential computations

Heavy computational job:
  for i = 1 to N
    - Heavy computations that can in principle be done concurrently
    - Heavy computations that cannot be done concurrently
  endfor

Postamble:
  - Some final sequential computations
  - Printing of results

end
```

Figure 3: The schema of the sequential code

distributed threads) in the workers-pool begin, and the program runs concurrently. When the program exits a concurrent region, only one single computational process continues (now we run sequentially) until the process again enters a concurrent region and the process repeats. See figure 5 for this multiple-mode execution model.

In the case of our Fortran program, the workers in the workers-pools are Gauss-Seidel workers, performing relaxations on all the cells of a certain grid. Note that in figure 4 the number of workers in a workers-pool is not fixed, but depends on the index i of the loop.

In the next section we explain how we have organized the schema in figure 4 as 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 relaxation computations done in `pointgsgr` on every single grid visited with `scanlv`, be to 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 of the sequential source code except the relaxations, which are 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 computations are performed in the master and 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.

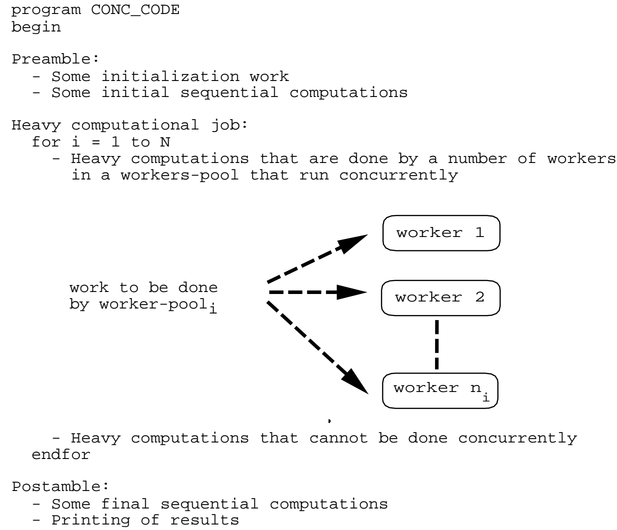


Figure 4: The schema of the concurrent code

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 embodies the computations, except the relaxations, of the main Fortran program of the sequential version. Each time the master arrives at a pre- or post-relaxation, it delegates this work to the workers in a workers-pool. The master makes its wish known to the coordinator by raising an event (`create_pool`)[2]. The coordinator reacts on 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 (if necessary, as we will see) from its input port the computational results of the workers. This is made possible by the

[2] We give the names of the events used in the **MANIFOLD** source code in parentheses.

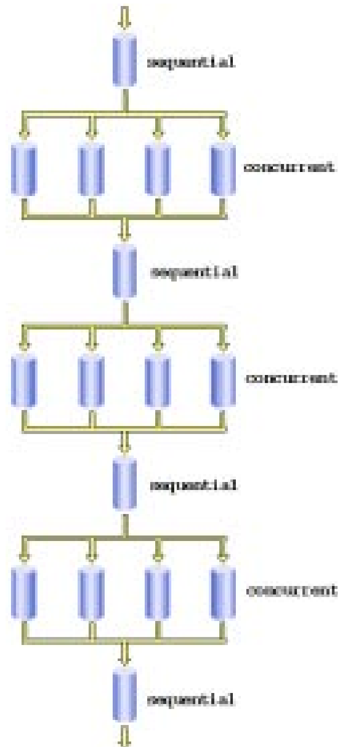


Figure 5: Execution of a typical program with sequential and concurrent parts.

coordinator which has set up a stream between the output port of the worker and the input port of the master. Hereafter, the master proceeds with its sequential work (i.e., the index i of the loop in figure 4 is incremented by one) until it again arrives at a point where it needs a workers-pool to delegate the relaxations to.

With this description we have covered the most important part of the master/worker protocol. There are, however, some other things we must consider too, which lead to the introduction of some more events (`x`, `xx` and `xxx`). This has to do with the following. Separating the computation into a number of concurrent processes means that the information contained in the global data structure used in the relaxation subroutine must be supplied to each process, and that the results produced by each process must be collected. The simple way to accomplish this is to arrange for the MANIFOLD coordinators to send and receive the (proper segments of the) global space through streams. However, there are more efficient ways to do this wherein we exploit the way shared memory is used in multi-threaded executables and the fact that we can divide the data structure of our Fortran application in two parts. We clarify this by the following two points.

- As noted before, most MANIFOLD processes run as threads bundled up in one or more MANIFOLD task instances (i.e., multi-threaded executables). It is a property of thread programming that threads, housed in the same multi-threaded executable, always share a global data space. For the communication between the master and a worker, this means that the latter does not need to receive

its own individual copy of the space, as long as this worker runs in the same task instance where the master runs. In this case, it is sufficient for the worker to know the information that indicates on which grid (i.e., the indices that identify the grid) it must perform its operations. With this information, the actual data of the grid can be read from the shared global data space of the task instance. Also, there is no need in this case to send the computed results of the workers through streams back to the master. A worker can directly write its results into the shared global data space. We call workers of this type *local* workers. A local worker raises event `x` to inform the master that the communication must take place via shared memory as just described.

We refer to the task instance in which the master runs as *the master task instance* and to the other task instances as *remote task instances*. Furthermore, we refer to the global data spaces in these task instances as, respectively, the *global master space* and *global remote spaces*. It is clear now that, when a worker is performing its computations in a remote task (this task instance has its own uninitialized global space and knows nothing of the global master space) it is not sufficient to send it the indices that identify the location of a grid in the global master space. In this case we must send the complete data segment of the grid from the global master space to that remote worker and communicate the results of that worker back to the master. We call workers of this type *remote* workers. A worker can determine whether it is a local or a remote worker by calling a function (or checking a variable) that indicates whether or not it runs in the master task instance. A remote worker always raises event `xx` to inform the master that the communication must take place via distributed memory as described. This inter-task communication is, of course, more expensive that the intra-task communications in shared memory.

- The global data space used in the Fortran program essentially consist of two parts. One part contains all those data segments the workers use in their relaxation computations and which they can read and update (write) independently of each other. We call this part of the global data space the *non-fixed part*. The other part (containing grid connectivity data and geometric data) remains constant after the sequential computations in the preamble of figure 4, and is only read by the workers. We call this part of the data space the *fixed part*. The proper segment in the global master space that a remote worker needs in order to do its job consists of data from both the fixed part as well as the no-fixed part. Because the data from the fixed part needed by remote workers have a considerable overlap and because the fixed part does not change after the sequential computations in the preamble, it is more efficient to communicated the complete fixed part of the global master space as one big chunk to remote task instances. We have arranged such that the *first* remote worker in a new remote instance is responsible for the initialization of the fixed part in its global remote space. Therefore, such a worker always raises an event `xxx` to inform the master to supply the fixed part. This is done in the usual MANIFOLD way: the master writes the data to its own output port which is connected, by a third party via a stream to the input port of the worker, which promptly reads it and does the initialization.

5.2 The Implementation

In this section we discuss the actual implementation of the master/worker protocol. First, in §5.2.1 we introduce some MANIFOLD terminology so that we can use it in our description of the source code. For the details about MANIFOLD we refer to [22]. In §5.2.2, we give the source code and explain its syntax. In §5.2.3, we explain the internal working of the state machines of the master/worker protocol. The master/worker protocol is implemented in a generic way where the master and the worker manifolds

themselves are parameters of the protocol. In §5.2.4, we describe those parameters (i.e., we describe the behavior of the master and worker manifolds). Every manifold pair whose behavior conforms with this description can be used as parameters in the master/worker protocol `protocolMW`.

5.2.1 MANIFOLD 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.

Syntactical Structure of a Block

Syntactically a block consists of a optional *local declaration part* followed by a finite number of *states* (at least the `begin` state should be present). It is easy to recognized 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 syntactical 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 sub state 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 construct are made with 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 and enclosed in a pair of parentheses forming a so called *group*.

An expression is a sequence of actions that, optionally, yield 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 between 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 use 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.

Semantic 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) is clearly given in its description in the reference manual [22]. 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 only cause a transition out of the current state 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 parameter 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 [22] for the details about when a stream connection breaks) and (2) all processes mentioned as parameter 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 semantic 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.

5.2.2 The Gluing Modules

The **MANIFOLD** source code of our protocol is given below.

```
 1 // protocolMW.m
 2
 3 #include "MBL.h"
 4
 5 #include "rdid.h"
 6
 7 #include "protocolMW.h"
 8
 9 #define IDLE terminated(void)
10
11 /**********************************************************************/
12 manner Create_Worker_Pool(process master <input, dataport | output, error>,
13                           manifold Worker(event, event, event, event) )
14 {
15   save *.
16   ignore death.
17
18   auto process now is variable(0).
19   auto process t is variable(0).
20
21   event death_worker.
22
23   priority create_worker > rendezvous.
24
25   begin: (MES("begin"), preemptall, IDLE).
26
27   create_worker: {
28     hold Worker.
29
30     process worker is Worker(death_worker, x, xx, xxx).
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(manifold Master <input, dataport | output, error>,
55                          manifold Worker(event, event, event, event) )
56 {
57   save *.
58
59   auto process master is Master.
60
61   begin: terminated(master).
62
63   create_pool: Create_Worker_Pool(master, Worker); post(begin).
64
65   finished: halt.
66 }
```

Looking syntactically to the source code and using the **Manifold** terminology explained in 5.2.1 we observe the following:

- The source code describes two regular manners named respectively `Create_Worker_Pool` (lines 11-51) and `ProtocolMW` (lines 53-66). 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 four event parameters.

  In the header of `ProtocolMW` (lines 54-55) we also see two parameters. The first parameter is now a manifold, named `Master` which has besides the normal standard ports (`input`, `output`, `error`) also a port named `dataport`. The second parameter is again a manifold. It is named `Worker` that has four event 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 `rendez_vous` state (line 39-48) and the `end` state (line 50).

  In the local declaration part, we see the two declarative statements `save` and `ignore` on line 15 and 16, and the creation of process instances `now` and `t` (line 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 only has 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 `rendez_vous` 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 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 (lines 6-10) we use in the file `ProtocolMW.m`.

```
 1 // protocolMW.h
 2
 3 manner ProtocolMW(manifold Master <input, dataport | output, error>,
 4                  manifold Worker(event, event, event, event) ) elsewhere.
 5
 6 extern event create_pool,
 7        create_worker,
 8        rendezvous, a_rendezvous,
 9        finished,
10        x, xx, xxx.
```

- 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 9) states that this manner can be used in other source files which import this MANIFOLD definition.

5.2.3 How it Runs

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 made 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 54-66) followed by the manner `Create_Worker_Pool` (line 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) the `ProtocolMW` manner in its `begin` state. As a result, we enter the block of this manner (lines 56-66). Upon entering a block, first the statements in its local declaration part are performed (lines 57-59).

Line 57 states that we can switch only to states in this block (i.e., the `begin`, `create_pool` or `finished` states respectively on lines 61, 63 and 65). Other possible event occurrences are saved and can be handled (if necessary) outside this block.

Line 59 defines a process instance of the formal manifold argument `Master` (line 54), calls it `master`, and states (through the keyword `auto`) that this process instance is to be automatically activated upon creation, and deactivated upon departure from the scope (i.e., departure from the block on line 66) in which it is defined (lines 56-66).

After performing the local declaration part of the entered block (lines 57-59) the MANIFOLD run-time system automatically posts an occurrence of the predefined high-priority event `begin` in the event memory of the caller (i.e., `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` 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 Fortran code (excluding the relaxations), after some sequential computation work arrives at the pre- and post-relaxations, it raises an event named `create_pool` to signal that it needs a workers-pool (master: 4(a)). This event pre-emptes the `begin` state and causes a transition to the `create_pool` state (line 63). In this state the manner `Create_Worker_Pool` (lines 11-51) is called with the process instance `master` (created and activated on line 59), and the manifold `Worker` (which the protocol manner `ProtocolMW` itself has received as a parameter on line 54) as its actual parameters. The manner `Create_Worker_Pool` conducts the workers in the pool and takes care that they can do their relaxation computations properly. When the workers in the pool are done, they die and the manner returns. Afterwards (denoted by the semicolon on line 63) 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 because following some sequential computation, `master` either decides that it needs another workers-pool, in which case it again raises the `create_pool` event (master: 3(a)), or it decides that it is done and raises the event `finished` (master: 5). In the first case we jump again to the `create_pool` state and the whole sequence starts again (i.e., the index i of the loop in figure 4 is incremented by one). In the other case (i.e., the index i of the loop in figure 4 is `N` and the postamble has been performed), we jump to the `finished` state (line 65), where the primitive action `halt` effectively returns the flow of control from the manner to its caller.

The manner `Create_Worker_Pool` (lines 11-51) called on line 63 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 only knows 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 the events `create_worker` and `rendezvous` are available in the

event memory of the executing manifold instance which 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`. The four parameters used in the instantiation are respectively the local event `death_worker` (line 21) and the global events `x`, `xx` and `xxx`, defined in the header file `protocolMW.h` (line 7).

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 `x`, `xx` and `xxx` events are events the worker can raise to signal the master what kind of information it needs to do its job (1(a)i, 1(b)iiB, 1(b)iii and 1(b)iiA). The reason for those events are already described at the end of §5.1.

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 disconnected 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 (by raising the `x`, `xx` and `xxx` events; worker: 1(a)i, i(b)iiA, i(b)iiB and i(b)iii) through a stream (the second $\rightarrow$ on line 36) to `worker`. The `worker` process promptly reads the information it receives from `master` (worker: 1(b)iiC and 1(b)iv), does its job (worker: 2), and if it is a remote worker, 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-emptes 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 start 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 less `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.

5.2.4 The Behavior Interface of the Master and Worker

The behavior interface 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. Further 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 step 5 of §6.2).

1. Make the extern events defined in the header file `protocolMW.h` available for the master so that it can communicate with `protocolMW` in `protocolMW.m`. Also, introduce some extra global variables to be used by workers to find out if they are local or remote (see the description of the variables `fpi` and `imt` and their use in the file `ptest.ato.c` in §6.2).
2. Perform some initialization work (optional) and some initial sequential computations (optional) (i.e., the preamble in figure 4).
3. Perform some work concurrently by creating a pool of workers and charge each with a computational job (i.e., the heavy computations in figure 4 that can in principle be done concurrently). 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 for this pool by raising an event `create_worker` (line 27).
   (c) Read a unit containing the process reference (identification) of an created worker from your own input port and activate it. (This unit, `&worker`, is send 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. Do this as follows:
      i. Wait (with a blocking wait) for one of the events `x` or `xx` raised by a worker (worker: 1(a)i, 1(b)iiB and 1(b)iiiB).
      ii. If `x` has been received, write the location where to work in the non fixed part of the global master space on your own output port (worker: 1(a)ii).

iii. If `xx` has been received do the following:

A. Check (with a non-blocking wait) if event `xxx` also has been received (worker: 1(b)iiA). Remark: We must wait for the event `xxx` in a non-blocking way because we do not know if it comes (worker: 1(b)ii). What we do know is that if `xxx` has been raised by the worker it is always raised before `xx` and thus is observable earlier in the event memory of the master and thus can always be found with the non-blocking wait (worker:1(b)ii).

B. If `xxx` has been received, write the whole fixed part of the global master space on your own output port (worker: 1(b)iiC).

C. Write the location where to work in the non-fixed part of the global master space, together with the corresponding segment of the non-fixed part, on your own output port (worker: 1(b)iv).

(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 remote workers (i.e., read those results from your own input port named `dataport` (worker: 3) and update with those results the non-fixed part in the global master space.

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

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

4. Perform some sequential work (optional) (i.e., the heavy computations in figure 4 that cannot be done concurrently).

5. Repeat 3 and 4 as many times as needed and raise at the end of this repetition the event `finished` (line 65) to inform the coordinator process that the master is done.

6. Perform some final sequential computations (optional) and print the results (optional) (i.e., the postamble in figure 4).

The behavior interface of the worker is described below. Here, the `death_worker` event is introduced via the first argument of the worker. The events `x`, `xx` and `xxx` are available, respectively, via its second, third and fourth arguments.

1. Read the information you need to know to do your job, from your own input port. Do this as follows:

(a) If the worker is a local worker do the following:

i. Raise `x` to signal the master to write the location where to work in the non fixed part of the global master space on its output port (master: 3(d)i).

ii. Read the location where to work in the non-fixed part of the global master space from your own input port (master: 3(d)ii).

(b) If the worker is a remote worker do the following:

i. Find out if it is the first worker in a remote task instance.

ii. If it is the first worker in a remote task instance do the following:

A. Raise `xxx` to signal the coordinator to write the whole fixed-part of the global master space on its own output port (master: 3(d)i).

B. Raise `xx` to signal the master to write the location where to work in the non fixed part of the global master space, together with its corresponding segment, on its output port (master: 3(d)i).

C. Read a unit containing the fixed part of the global master space and initialize with it the fixed part in the global remote space (master: 3(d)iiiB).

iii. If it is not the first worker in a remote task instance, just do step 1(b)iiB of the worker.

iv. Read the location where to work in the non-fixed part of the global data space of the master, together with the corresponding segment in it, from your own input port (master: 3(d)iiiC).

2. Do the computational job.

3. If the worker is a remote worker write the computed results on your own output (master: 3(f)).

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

6 The Test

In this section we illustrate the cut-and-paste method with a toy application. The source code of the sequential version of our toy application follows exactly the scheme as given in figure 3. As in the original Fortran application, here too the global space consists of two parts: a fixed part and a non-fixed part and some operations are to be performed on the non-fixed part. Some of these operations can go concurrently whereas others can only be performed sequentially. In §6.1 we give the source code of the toy example and define two types of operation on the non-fixed part of the global space. In §6.2 we restructure this application, according to the scheme given in figure 4, into a concurrent version using the glue modules of §5.2.2. In §6.3 we run this concurrent version and shows its output. Finally we demonstrate its performance in §6.4.

6.1 The Sequential Source Code of Toy Example

The global space in the toy application consists of two parts: a fixed part and an non-fixed part. Both arrays have the same length and are initialized in the same way (the first element is `1`, the second is `2`, etc.). We have defined the following operations on the non-fixed array:

(a) Add to each element of the non-fixed part array its previous element of the same array. For the first element, add the last element.

(b) Element-wise add the fixed-part array to the non-fixed part array.

It is clear that the operation (a) cannot be done concurrently element-wise, whereas the operation (b) can easily be done element-wise by different workers in a concurrent fashion, each worker adding a fixed-part array element to its corresponding non-fixed part array element. With these two operations, it is simple to write a little program according to the schema given in figure 3. The initialization in the preamble (see figure 3) consists of the initializations of the fixed-part and non-fixed-part arrays. For the "heavy

computations that cannot be done concurrently" (see again figure 3) we use operation (a). All other computations in that figure are (b) operations. In our test example we set the array length to three and the `N` in figure 3 is set to four (so we perform successively the operations (a, b, a, b, a, b, a, b, a, b, a, a) on the non-fixed-part array).

The sequential ANSI C source code (stored in file `seq.c`) of the "toy" example is given below and does not need an explanation.

```
 1 /* seq.c */
 2
 3 #include <stdio.h>
 4
 5 #define BOUND 3
 6 #define N     4
 7
 8 int fixed_part[BOUND];
 9 int non_fixed_part[BOUND];
10
11 /*******************************************************************/
12 void add_a_previous_element(int index)
13 {
14   /* auxilary for add_previous_element */
15
16   if (index == 0) {
17     non_fixed_part[0] = non_fixed_part[0] + non_fixed_part[BOUND - 1];
18   } else {
19     non_fixed_part[index] = non_fixed_part[index] + non_fixed_part[index - 1];
20   };
21 }
22
23 /*******************************************************************/
24 void add_previous_elements(void)
25 {
26   /* operation (a) */
27
28   int i;
29
30   for (i = 0; i < BOUND; i++) {
31     add_a_previous_element(i);
32   }
33 }
34
35 /*******************************************************************/
36 void add_a_fixed_part_element(int index)
37 {
38   /* auxilary for add_fixed_part_elements */
39
40   non_fixed_part[index] = non_fixed_part[index] + fixed_part[index];
41 }
42
43 /*******************************************************************/
44 void add_fixed_part_elements(void)
45 {
46   /* operation (b) */
47
48   int i;
49
50   for (i = 0; i < BOUND; i++) {
51     add_a_fixed_part_element(i);
52   }
53 }
54
55 /*******************************************************************/
56 void print_it(int line)
57 {
58   int i, tn = 0;
59   char buf[150];
60
61   for (i = 0; i < BOUND;  i++) {
62     sprintf(buf + tn * 8, "%8d", non_fixed_part[i]);
63     tn++;
64   }
65   printf("line %d: %s\n", line, buf);
66 }
67
68 /*******************************************************************/
69 void some_initialization_work(void)
70 {
71   int i;
72
73   for (i = 0; i < BOUND; i++) {
74     fixed_part[i] = i + 1;
75     non_fixed_part[i] = i + 1;
76   }
77 }
78
79 /*******************************************************************/
```

```
 80 void some_initial_sequential_computations(void)
 81 {
 82   add_previous_elements();
 83 }
 84
 85 /********************************************************************/
 86 void computations_that_can_be_done_concurrently(void)
 87 {
 88   add_fixed_part_elements();
 89 }
 90
 91 /********************************************************************/
 92 void computations_that_can_not_be_done_concurrently(void)
 93 {
 94   add_previous_elements();
 95 }
 96
 97 /********************************************************************/
 98 void some_final_sequential_computations(void)
 99 {
100   add_previous_elements();
101 }
102
103 /********************************************************************/
104 void main(void)
105 {
106   int i;
107
108   /* Preamble */
109   some_initialization_work();
110   print_it(__LINE__);
111   some_initial_sequential_computations();
112   print_it(__LINE__);
113
114   /* Heavy_computational_job */
115   for (i = 0; i < N; i++) {
116     computations_that_can_be_done_concurrently();
117     print_it(__LINE__);
118     computations_that_can_not_be_done_concurrently();
119     print_it(__LINE__);
120   };
121
122   /* Postamble */
123   some_final_sequential_computations();
124   print_it(__LINE__);
125 }
```

Comparing lines 109-124 of the source code with the scheme in figure 3, we see that the source code exactly follows the schema. The only thing we have added is that we also print the initial values of the non-fixed array and its result after each operation.

The (a) and (b) operations are represented in the source code by, respectively, the routines `add_previous_elements` (lines 24-33) and `add_fix_part_elements` (line 44-53).

Running this simple sequential toy application gives the following output.

```
line 110:       1       2       3
line 112:       4       6       9
line 117:       5       8      12
line 119:      17      25      37
line 117:      18      27      40
line 119:      58      85     125
line 117:      59      87     128
line 119:     187     274     402
line 117:     188     276     405
line 119:     593     869    1274
line 124:    1867    2736    4010
```

6.2 The Concurrent Version of the Toy Example

The concurrent version of the toy example is derived in the next five simple steps.

1. Disable the `main` subroutine in the original source code.

   Because the concurrent version of our application has its own `main` subroutine and because we want to reuse the source code of `seq.c`, we have to change the name of this subroutine into something

else, so that we do not have two captains on one ship[3]. This slightly adapted but original source code named `Aseq.c` is the first file that we use for the concurrent version.

2. Store the prototypes of the routines in `Aseq.c` that we need directly in the ANSI C version of the master and the worker (see file `ptest.ato.c` in step 4 below), in a separate header file. This produces our next header file.

```
1 /* Aseq.h */
2
3 extern void add_a_fixed_part_element(int index);
4 extern void some_initialization_work(void);
5 extern void some_initial_sequential_computations(void);
6 extern void computations_that_can_not_be_done_concurrently(void);
7 extern void some_final_sequential_computations(void);
```

Because The ANSI C print routine `printf` used in routine `print_it` (on line 65) cannot be used in a concurrent environment (using it, the output of different processes mix up on the screen), we rewrite this routine and that is why it does not appear up in the above header file. Its new name, as we see in step 4 below, is the old one preceded with the `N` for new.

Also the routines `computations_that_can_be_done_concurrently` (line 86-89) and `main` (lines 104-125) do not appear in the header file. We must rewrite the first according to step 3 in the behavior interface of the master (see §5.2.4). Its new name is again the old one preceded with an `N`. The second routine becomes the master (its name is `ma`, see step 4 below) and is rewritten according to the steps 1-6 in the behavior interface of the master (see again §5.2.4).

3. Make an interface to the data defined in `Aseq.c`, so that we can exchange data with the original source code.
In the file containing the implementation of the master and worker manifolds (i.e., the file `ptest.ato.c` in step 4 below) some of the data defined in `Aseq.c` must be used. Therefore, we specify the following interface for this data.

```
1 /* aux.c */
2
3 #include "AP_interface.h"
4
5 #include "MBL_interface.h"
6
7 #include "ptest.ato.h"
8
9 #include "adid.h"
10
11 #include "aux.h"
12
13 #define BOUND 3
14 #define N     4
15
16 extern int fixed_part[BOUND];
17 extern int non_fixed_part[BOUND];
18
19 /*********************************************************************/
20 AP_Unit get_fixed_part_data(void)
21 {
22   AP_Unit u;
23
24   u = AP_FrameIntegerArray((int *) fixed_part, BOUND);
25   return u;
26 }
27
28 /*********************************************************************/
29 void put_fixed_part_data(AP_Unit u)
30 {
31    int err;
32
33    err = AP_FetchIntegerArray(u, fixed_part, BOUND);     I(err)
34    err = AP_DeallocateUnit(u);                           I(err)
```

[3]In principle the name change of the subroutine `main` is not necessary. It only prevents warnings from the loader about the multiple defined `main`'s.

```
35 }
36
37 /*******************************************************************/
38 int get_non_fixed_part(int index)
39 {
40   return non_fixed_part[index];
41 }
42
43 /*******************************************************************/
44 void put_non_fixed_part(int index, int value)
45 {
46   non_fixed_part[index] = value;
47 }
48
49 /*******************************************************************/
50 int get_BOUND(void)
51 {
52   return BOUND;
53 }
54
55 /*******************************************************************/
56 int get_N(void)
57 {
58   return N;
59 }
60
```

The prototypes of those routines are stored in next header file.

```
 1 /* aux.h */
 2
 3 extern AP_Unit get_fixed_part_data(void);
 4 extern void put_fixed_part_data(AP_Unit u);
 5
 6 extern int get_non_fixed_part(int index);
 7 extern void put_non_fixed_part(int index, int value);
 8
 9 extern int get_BOUND(void);
10 extern int get_N(void);
```

4. Implement the master and worker, according to the behavior interface of §5.2.4, in ANSI C. The source code of the master and workers is below.

```
 1 /* ptest.ato.c */
 2
 3 #include "AP_interface.h"
 4
 5 #include "MBL_interface.h"
 6
 7 #include "ptest.ato.h"
 8
 9 #include "adid.h"
10
11 #include "aux.h"
12
13 #include "Aseq.h"
14
15 AP_Event create_pool, create_worker,  rendezvous, a_rendezvous, finished, x, xx, xxx;
16
17 #define TRUE 1
18 #define FALSE 0
19
20 /* fpi = Fixed Part Initialized
21    This variable is in every task instance initialized with FALSE.
22    The fpi variable in the master task instance is set to TRUE by the
23    the master to indicate that the fixed part in the master task has
24    been initialized.
25    The fpi variable in a remote task instance is set to TRUE by
26    the (first) remote worker to indicate that the fixed part in the
27    remote task instance has been initialized.
28    The fpi variable is used by a remote worker to find out if it
29    is the FIRST remote worker in task instance.
30    See its use on the lines 31, 270, and 282 */
31 int fpi = FALSE;
32
33 /* imt = In Master Task
34    This variable is in every task instance initialized with FALSE.
35    The imt variable in the master task instance is set to TRUE by the
36    master.  By inquiring this variable, a worker can always found out
37    if it is a local or remote worker.
38    See its use on the lines 39, 75, and 256 */
39 int imt = FALSE;
40
41 /*******************************************************************/
42 void global_init_stuff(void)
43 {
```

```
 44   int err;
 45
 46   /* step 1 in the behavior interface of the master in section 5.2.4 */
 47
 48   create_pool = AP_AllocateEvent();                              P(create_pool)
 49   err = AP_InitHeaderEvent(create_pool, "create_pool");          I(err)
 50
 51   create_worker = AP_AllocateEvent();                            P(create_worker)
 52   err = AP_InitHeaderEvent(create_worker, "create_worker");      I(err)
 53
 54   rendezvous = AP_AllocateEvent();                               P(rendezvous)
 55   err = AP_InitHeaderEvent(rendezvous, "rendezvous");            I(err)
 56
 57   a_rendezvous = AP_AllocateEvent();                             P(a_rendezvous)
 58   err = AP_InitHeaderEvent(a_rendezvous, "a_rendezvous");        I(err)
 59
 60   finished = AP_AllocateEvent();                                 P(finished)
 61   err = AP_InitHeaderEvent(finished, "finished");                I(err)
 62
 63   x = AP_AllocateEvent();                                        P(x)
 64   err = AP_InitHeaderEvent(x, "x");                              I(err)
 65
 66   xx = AP_AllocateEvent();                                       P(x)
 67   err = AP_InitHeaderEvent(xx, "xx");                            I(err)
 68
 69   xxx = AP_AllocateEvent();                                      P(xxx)
 70   err = AP_InitHeaderEvent(xxx, "xxx");                          I(err)
 71
 72   /* Some initializations.
 73      See description fpi and imt variable above on lines 20-39. */
 74   fpi = TRUE;
 75   imt = TRUE;
 76 }
 77
 78 /*********************************************************************/
 79 void Nprint_it(void)
 80 {
 81   int i, tn = 0;
 82   char buf[150];
 83
 84   for (i = 0; i < get_BOUND();  i++) {
 85     sprintf(buf + tn * 8, "%8d", get_non_fixed_part(i) );
 86     tn++;
 87   }
 88                                                                  M(buf)
 89 }
 90
 91 /*********************************************************************/
 92 void Ncomputations_that_can_be_done_concurrently(void)
 93 {
 94   /* All the steps mentioned in this routine refer to the behavior
 95      interface of the master in section 5.2.4 */
 96
 97   /* This routine is the complete implementation of step 3 */
 98
 99   int dataport = AP_LocalPortId("dataport");
100
101   AP_Unit u;
102   AP_Process p = AP_AllocateProcess();
103   AP_Event r = AP_AllocateEvent();
104   AP_EventPatternSet eps = AP_AllocateEventPatternSet();
105   AP_Process q = AP_AllocateProcess();
106
107   int err, i, ar[2];
108   int noxxevents = 0;
109
110   char mesg[300];
111
112      S("computations_that_can_be_done_concurrently: begins")
113                                                          P(p)
114                                                          P(r)
115                                                          P(eps)
116                                                          P(q)
117                                                          I(dataport)
118   /* step 3(a) */
119   err = AP_Raise(create_pool);                           I(err)
120
121   for (i = 0; i < get_BOUND(); i++) {
122
123     /* step 3(b) */
124     err = AP_Raise(create_worker);                       I(err)
125
126     /* step 3(c) */                                      W
127     err = AP_PortRemoveUnit(AP_INPUT, &u, NULL);         I(err) P(u) W
128                                                          W
129     err = AP_DerefProcess(p, u, NULL, NULL);             I(err) W
130     err = AP_Activate(p);                                I(err)
131
132     /* step 3(d)i */
133     err = AP_EventPatternSetInsert(eps, x, p);           I(err)
134     err = AP_EventPatternSetInsert(eps, xx, p);          I(err)
```

```
135                                                        W
136     err = AP_DeleteWaitEvent(eps, r, q);               I(err) W
137
138     if (err = AP_EqualEvent(x, r)) {                   I(err)
139       /* step 3(d)ii */
140       u = AP_FrameInteger(i);
141                                                        W
142       err = AP_PortPlaceUnit(AP_OUTPUT, u, NULL);      I(err) W
143       err = AP_DeallocateUnit(u);                      I(err)
144     } else {
145       /* step 3(d)iii */
146       /* The detected event is now xx */
147       noxxevents++;
148
149       /* step 3(d)iiiA */
150       err = AP_EventPatternSetDelete(eps, NULL, NULL);  I(err)
151       err = AP_EventPatternSetInsert(eps, xxx, p);     I(err)
152
153       if (err = AP_DeleteCheckEvent(eps, r, q) ) {     I(err)
154         /* step 3(d)iiiB */
155         /* The detected event is now xxx */
156         /* first send up the fixed part */
157         u = get_fixed_part_data();
158                                                          W
159         err = AP_PortPlaceUnit(AP_OUTPUT, u, NULL);    I(err) W
160         err = AP_DeallocateUnit(u);                    I(err)
161       }
162
163       /* step 3(d)iiiC */
164       /* send up the non fixed part */
165       ar[0] = i; ar[1] = get_non_fixed_part(i);
166       u = AP_FrameIntegerArray((int *) ar, 2);
167                                                        W
168       err = AP_PortPlaceUnit(AP_OUTPUT, u, NULL);      I(err) W
169       err = AP_DeallocateUnit(u);                      I(err)
170     }
171
172     err = AP_EventPatternSetDelete(eps, NULL, NULL);   I(err)
173     /* step 3(e) */
174   }
175
176   /* step 3(f) */                                      PI(noxxevents)
177   for (i = 0; i < noxxevents; i++) {                   W
178     err = AP_PortRemoveUnit(dataport, &u, NULL);       I(err) P(u) W
179     err = AP_FetchIntegerArray(u, ar, 2);              I(err)
180     err = AP_DeallocateUnit(u);                        I(err)
181     put_non_fixed_part(ar[0], ar[1]);
182   }
183
184   /* step 3(g) */
185   err = AP_Raise(rendezvous);                                  I(err) W
186
187   /* step 3(h) */
188   err = AP_EventPatternSetInsert(eps, a_rendezvous, NULL);     I(err)
189                                                                W
190   err = AP_DeleteWaitEvent(eps, r, p);                         I(err) W
191
192   err = AP_SPrintEvent(mesg, 100, r);                          S(mesg) I(err)
193
194   do {
195     err = AP_DeleteCheckEvent(NULL, r, p);                     PI(err) I(err) W
196     if (err == 1) {
197                                        S("event delete from E.M. of master")
198       err = AP_SPrintEvent(mesg, 100, r);                      S(mesg) I(err)
199       err = AP_SPrintProcess(mesg, 300, p);                    S(mesg) I(err)
200     }
201   } while (err == 1);
202
203   err = AP_DeallocateProcess(p);                               I(err)
204   err = AP_DeallocateProcess(q);                               I(err)
205   err = AP_DeallocateEvent(r);                                 I(err)
206   err = AP_DeallocateEventPatternSet(eps);                     I(err)
207
208                 S("Scomputations_that_can_be_done_concurrently: ends")
209 }
210
211 /*********************************************************************/
212 void ma(void)
213 {
214   /* All the steps mentioned in this routine refer to the behavior
215      interface of the master in section 5.2.4 */
216
217   int i, err;
218
219   /* step 1 */                                         S("ma starts")
220   global_init_stuff();
221
222   /* step 2 */
223   /* Preamble */
224   some_initialization_work();                          S("Nprint_it")
225   Nprint_it();
```

```
226   some_initial_sequential_computations();              S("Nprint_it")
227   Nprint_it();
228
229   /* Heavy_computational_job */
230   for (i = 0; i < get_N(); i++) {
231     /* step 3 */
232     Ncomputations_that_can_be_done_concurrently();      S("Nprint_it")
233     Nprint_it();
234     /* step 4 */
235     computations_that_can_not_be_done_concurrently();   S("Nprint_it")
236     Nprint_it();
237     /* step 5 */
238   };
239   err = AP_Raise(finished);                            I(err) W
240
241   /* step 6 */
242   /* Postamble */
243   some_final_sequential_computations();                S("Nprint_it")
244   Nprint_it();
245 }
246
247 /*********************************************************************/
248 void wo(AP_Event e, AP_Event x, AP_Event xx, AP_Event xxx)
249 {
250   int err, where;
251   AP_Unit u;
252   int ar[2];
253   int send_results = FALSE;
254
255                                                                W
256   if (imt) {
257                                         M("I am a local worker")
258     /* step 1(a)i */
259     err = AP_Raise(x);                                         I(err)
260
261     /* step 1(a)ii */
262     /* Receive index of array element that should be changed */ W
263     err = AP_PortRemoveUnit(AP_INPUT, &u, NULL);               I(err) P(u) W
264     err = AP_FetchInteger(u, &where);                          I(err)
265     err = AP_DeallocateUnit(u);                                I(err)
266
267   } else {
268                                         M("I am a remote worker")
269     /* step 1(b)i */
270     if (!fpi) {
271       /* step 1(b)ii */
272       /* I am the first remote worker in the task instance */
273       /* step 1(b)iiA */
274       err = AP_Raise(xxx);                                     I(err) W
275       /* step 1(b)iiB */
276       err = AP_Raise(xx);                                      I(err)
277
278       /* step 1(b)iiC */
279       /* Receive fixed_part */                                 W
280       err = AP_PortRemoveUnit(AP_INPUT, &u, NULL);             I(err) P(u) W
281       put_fixed_part_data(u);
282       fpi = TRUE;
283     } else {
284       /* step 1(b)iii */
285       err = AP_Raise(xx);                                      I(err)
286     };
287
288     send_results = TRUE;
289
290     /* step 1(b)iv */
291     /* Receive index of array element and the array element itself */  W
292     err = AP_PortRemoveUnit(AP_INPUT, &u, NULL);               I(err) P(u) W
293     err = AP_FetchIntegerArray(u, ar, 2);                      I(err)
294     err = AP_DeallocateUnit(u);                                I(err)
295     where = ar[0];
296     put_non_fixed_part(where, ar[1]);
297   }
298
299   /* step 2 */
300   /* The heavy computational job in the worker */
301   add_a_fixed_part_element(where);
302
303   /* step 3 */
304   if (send_results) {
305     /* Update non_fixed_part in the task that contains the master */
306     ar[1] = get_non_fixed_part(ar[0]);
307     u = AP_FrameIntegerArray((int *) ar, 2);
308                                                                W
309     err = AP_PortPlaceUnit(AP_OUTPUT, u, NULL);                I(err) W
310     err = AP_DeallocateUnit(u);                                I(err)
311   }
312
313   /* step 4 */
314   err = AP_Raise(e);                                           I(err) W
315 }
```

The global_init_stuff (lines 42-76) is the implementation of step 1 of the behavior interface of the master (see §5.2.4).

Nprint_it (lines 79-89) is the new version of the routine print_it and prints the non-fixed part of the global space in an ordered fashion on the screen.

The routine Ncomputations_that_can_be_done_concurrently (lines 92-209) follows exactly step 3 in the behavior interface of the master (see §5.2.4).

The master manifold named ma (lines 212-245) is a rewriting of the routine main of the sequential version. It follows exactly the steps 1-6 of the behavior interface of the master (see §5.2.4).

The worker manifold named wo (lines 248-315) is the new version of the routine add_a_fixed_part_element(int index), which is the routine used by computations_that_can_be_done_concurrently in the sequential version. It follows exactly the steps 1-4 of the behavior interface of the worker (see §5.2.4).

The master and worker manifolds are in fact C wrappers around the original C subroutines of the sequential version.

In the source code of the master and worker we can easily recognize the different steps in their behavior interfaces as descripted in §5.2.4 (references appear as comments in the source code). Therefore, we do not give a long description of this source code but only 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.

Events in atomic manifolds (i.e., manifolds written in a conventional programming language) are raised with the C routine AP_Raise (e.g., line 124). To handle events, we first must insert them in a so-called *event pattern set* (with AP_EventPatternSetInsert; e.g., line 133). After this, we can search, with AP_DeleteCheckEvent (blocking; e.g., 153) or with AP_DeleteWaitEvent (blocking; e.g., 136) 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. Of course, we also need a routine to delete an event out of the event pattern set (AP_EventPatternSetDelete; e.g., line 150) and we need a routine to check if an event is equal to another event (AP_EqualEvent; line 138).

Reading and writing units from and to ports are done, respectively, with AP_PortRemoveUnit (e.g., line 127) and AP_PortPlaceUnit (e.g., line 168). Other routines in ptest.ato.c are already explained, or reveal their purposes by their names. The ones that are not discussed are not critical for the understanding of the main activities in the source code. For the details we refer to [22].

In the ANSI C source code, we use a number of macro's for error checking (see e.g., the macro I on line 49) or for printing to the screen in an ordered fashion (see e.g., the macro M in the lines 88, 257 and 268). Their definitions are given in the header file adid.h which is included in ptest.ato.c on line 9.

Note that the header files aux.h (see step 3) and Aseq.h (see step 2) are included in this source (line 11 and 13).

The file with the prototypes of the master and the worker is below.

```
1 /* ptest.ato.h */
2
3 #include "AP_interface.h"
4
5 extern void wo(AP_Event, AP_Event, AP_Event, AP_Event);
6
7 extern void ma(void);
```

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

5. Use the master/worker protocol `ProtocolMW` of §5.2 together with the actual master and worker parameters as defined in step 4.

   Below, we give the **Manifold** program in which we use the master/worker protocol `ProtocolMW` to restructure the sequential version of our toy application into a concurrent one.

```
 1 // ptest.m
 2
 3 //pragma include "ptest.ato.h"
 4
 5 #include "protocolMW.h"
 6
 7 manifold wo(event, event, event, event) atomic {internal.}.
 8
 9 manifold ma() port in input. port in dataport. port out output. port out error.
10   atomic {internal. event create_pool, create_worker, rendezvous, a_rendezvous, finished, x, xx, xxx.}.
11
12 /********************************************************************/
13 manifold Main
14 {
15   begin: ProtocolMW(ma, wo).
16 }
```

   We briefly explain this source code.

   On line 3 we include the header file `ptest.ato.h` (see step 4) 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 §5.2.2). Note that we also use this header file in `ptest.m` as well as in `ProtocolMW.m` 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 `wo`, which takes four event arguments, 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 `ma` (lines 9-10), except that it has no arguments 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 13-16 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 as actual arguments. After this, the instance of `Main`, the instance of the master `ma`, and all the necessary instances of the worker `wo`, run concurrently.

6.3 Running the Toy Example

The source files that contain the **Manifold** program (i.e., `ptest.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., `ptest.ato.c` and `aux.c`) 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 [25]) 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 (in our case this means that we have only local workers), the application executes in parallel (i.e., not distributed). In that case, as explained in §5.1, the communication is arranged via the shared global space of the multi-threaded executable. 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 houses in a separate task instance (line numbers have been added).

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

In this file, we specify that a task instance is considered to be "full" when its load exceeds 10 (line 5) and that the weight of an instance of the `Worker` is also 10 (line 6). The net result of this is that each task instance will house only one `Worker` instance and thus instances of `Worker` end up in different instances of the task named `ptest` (line 8). Because the default weight of a manifold is always zero this means that in the master task instance always one worker is doing its job. 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. Because we have introduced in the toy application three different workers, we expect, with the above input file for MLINK, three task instances and that is why we have arranged this mapping in such a way that there are also three different machines available for them.

```
{host host1 pont.cwi.nl}
{host host2 opduwer.cwi.nl
{host host3 sampan.cwi.nl}
{locus ptest $host1 $host2 $host3}
```

In the above file, we define three variables `host1`, `host2`, and `host3`, which we set to, respectively, `pont.cwi.nl`, `opduwer.cwi.nl` and `sampan.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 (in our case three) of the task named `ptest` can be started on any of these three machines.

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

```
sampan 262155 113 ptest ma ptest.ato.c 88 ->        1       2       3
sampan 262155 113 ptest ma ptest.ato.c 88 ->        4       6       9
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 25 -> begin
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
sampan 262155 263 ptest wo ptest.ato.c 257 -> I am a local worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
```

```
opduwer 786437 64 ptest wo ptest.ato.c 268 -> I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
pont 524289 64 ptest wo ptest.ato.c 268 -> I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 50 -> rendezvous acknowledged
sampan 262155 113 ptest ma ptest.ato.c 88 ->        5       8      12
sampan 262155 113 ptest ma ptest.ato.c 88 ->       17      25      37
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 25 -> begin
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
sampan 262155 936 ptest wo ptest.ato.c 257 -> I am a local worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
opduwer 786437 83 ptest wo ptest.ato.c 268 -> I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
pont 524289 83 ptest wo ptest.ato.c 268 -> I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 50 -> rendezvous acknowledged
sampan 262155 113 ptest ma ptest.ato.c 88 ->       18      27      40
sampan 262155 113 ptest ma ptest.ato.c 88 ->       58      85     125
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 25 -> begin
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
sampan 262155 1609 ptest wo ptest.ato.c 257 -> I am a local worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
opduwer 786437 102 ptest wo ptest.ato.c 268 -> I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
pont 524289 102 ptest wo ptest.ato.c 268 -> I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 50 -> rendezvous acknowledged
sampan 262155 113 ptest ma ptest.ato.c 88 ->       59      87     128
sampan 262155 113 ptest ma ptest.ato.c 88 ->      187     274     402
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 25 -> begin
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
sampan 262155 2282 ptest wo ptest.ato.c 257 -> I am a local worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
opduwer 786437 121 ptest wo ptest.ato.c 268 -> I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
pont 524289 121 ptest wo ptest.ato.c 268 -> I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW: Main protocolMW.m 50 -> rendezvous acknowledged
sampan 262155 113 ptest ma ptest.ato.c 88 ->      188     276     405
sampan 262155 113 ptest ma ptest.ato.c 88 ->      593     869    1274
sampan 262155 113 ptest ma ptest.ato.c 88 ->     1867    2736    4010
```

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, 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* and *where*. In the MANIFOLD source code an actual message is given as the argument of a `MES` call. In the source code of `protocolMW` (5.2.2), we use `MES` to make the state transitions visible (see lines 25, 35 and 50). In the ANSI C code of the master and worker (stored in a file named `ptest.ato.c`) we also produce some messages. The worker produces a message in which it tells if it is a local (line 257) or a remote worker (line 268), and the master informs us about the values in the non-fixed part array (line 88). As we can verify, the computational results of this distributed run are the same as in the sequential version.

When we want to execute the toy program in a *parallel* way, we simply change the load on line 5 of `ptest.mlink` to `30`, and do the linking phase again. This can also be accomplished by not using a map file for MLINK.

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.

6.4 The Performance

It is clear that when the time spent executing a parallel algorithm is long compared to the time required to coordinate, the cost of the coordination is no problem. But if the time required for the computation is not so long, then the time spend on coordination becomes very important. Because in our example the work to be done is exactly *one* floating point operation, we cannot expect the concurrent version to be faster than the sequential one. To give some meaningful performance results, we increased the number of floating point operation in the workers to a more realistic level (10^{10}).

We ran and compared the performance results of the sequential and the concurrent versions of our

Table 1: The elapsed times (in minutes) for the different versions on different machine types.

machine type	*sequential*	*concurrent*
multi-processor machine	86m	32m
cluster of workstations	115m	41m

example on an SGI Origin 2000 multi-processor machine and also on a cluster of three SGI O2 single processor machines. The hardware of the Origin 2000 has 32 processors. 16 CPU'S of this machine are MIPS R12000 processors each with its own MIPS R12010 floating point co-processor. Its other 16 CPU's are MIPS R10000 processors each combined a MIPS R10010 floating point co-processor. In the SGI O2 we have a MIPS R5000 processor as CPU plus a floating point co-processor. The workstations in the cluster are connected to each other by a shared Ethernet (10 Mbps).

The experiments were done in quiet periods during normal working days. This means that we do not have a guarantee that we are the only user, which is a realistic assumption in any real comtempory computing environment. Furthermore, such unpredictable effects as network traffic and file server delays, etc., could not be eliminated and are always reflected in the computational results. To even out such "random" perturbations, we ran the two versions of the application several times close to each other in real time for the different machine types and collected their *elapsed* or *wallclock* 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 utiliy `/bin/time`. Some typical results are given in table 1. Although we must be careful to draw firm conclusions from these measurements we remark that during the run on the multi-processor machine the weighted cpu percentages measured 270%, which means that our application (with three worker processes) kept 2.7 of the 32 processors busy working. Because $32m * 2.7 \approx 86m$, this suggests that MANIFOLD can coordinate our toy application on the multi-processor machine without much overhead. Furthermore, the difference in the elapsed times for the sequential version runs on the multi-processor machine and on the cluster of workstations (in this case the cluster consist of a single machine) is due to the faster hardware of the multi-processor machine.

7 The Restructuring

After this detailed discussion of the restructuring of the toy program, we can be very brief about the actual restructuring of the sparse-grid and the semi-sparse-grid programs. All we need to do is to follow the five steps as given in §6.2.

The master and worker manifolds that are used as parameters of the protocol `protocolMW` are both implemented as C functions, only now, because the original source code is in Fortran, those C functions call the proper parts of the original *Fortran* subroutines (about 8000 lines). For the C function that implements the master, this means that it calls that part of the sequential Fortran source code that embodies the computations excluding the relaxation computations. For the C function that implements the worker, this means that it calls that part of the sequential Fortran source code that embodies the relaxation computations (i.e., the subroutine `pointgsgr`). The bridge between C and Fortran is through the so called underscore method: a Fortran subroutine `X` can be called from C as a C function named `X_`. This scheme works on most platforms.

Below, we give the two MANIFOLD programs that change the original sequential code of our sparse-grid and semi-sparse-grid applications to their respective concurrent versions.

Note that their **MANIFOLD** sources are exactly the same as in the toy application, only now the master is named `w_sparse` (for the sparse-grid program) or `w_semi_sparse` (for the semi-sparse-grid program) and the worker is named, in both cases, `w_pointgsgr`. The `w_` prefixes used in the masters' and the worker's names emphasize that they are wrappers around the original source code.

```
 1 // sparse_model.m
 2
 3 //pragma include "aw.h"
 4
 5 #include "protocolMW.h"
 6
 7 manifold w_pointgsgr(event, event, event, event) atomic {internal.}.
 8
 9 manifold w_sparse() port in input. port in dataport. port out output. port out error.
10                     atomic {internal. event create_pool, create_worker,
11                             rendezvous, a_rendezvous, finished, x, xx, xxx.}.
12
13 /**********************************************************************/
14 manifold Main
15 {
16   begin: ProtocolMW(w_sparse, w_pointgsgr).
17 }
```

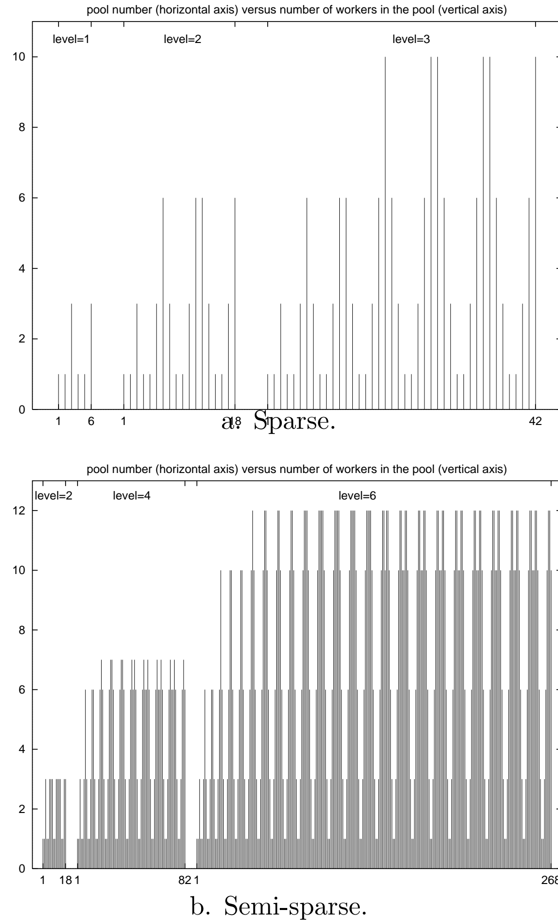
```
 1 // semi_sparse_model.m
 2
 3 //pragma include "aw.h"
 4
 5 #include "protocolMW.h"
 6
 7 manifold w_pointgsgr(event, event, event, event) atomic {internal.}.
 8
 9 manifold w_semi_sparse() port in input. port in dataport. port out output. port out error.
10                          atomic {internal. event create_pool, create_worker,
11                                  rendezvous, a_rendezvous, finished, x, xx, xxx.}.
12
13 /**********************************************************************/
14 manifold Main
15 {
16   begin: ProtocolMW(w_semi_sparse, w_pointgsgr).
17 }
```

8 The Performance Analysis

In this section, we discuss, in §8.1, the dynamic expansion and shrinking of our application during its run and we compare in §8.2, the performance results before and after the restructuring of the sparse and semi-sparse grid programs. We give performance results for our shared memory runs, in §8.2.1, as well as for our distributed memory runs, in §8.2.2.

8.1 Ebb & Flow

In §4, we already remarked that in the sparse-grid and semi-sparse-grid applications the number of workers varies in the different workers-pools created in a run. The reason is that sometimes there is more relaxation work to perform, requiring more worker processes in a workers-pool, and at other times there is hardly any relaxation work to do, requiring only a few worker processes. This dynamic expansion and shrinking of our applications during their runs—the ebb & flow in the application—is shown in figures 6a and 6b. In figure 6a we see that for level=1, 6 pools of workers were created and the number of workers in each pool is respectively 1, 1, 3, 1, 1 and 3. This adds up to the total of 10 worker processes in this application. Note that when the workers in a particular workers-pool are done the application reaches a synchronization point (a rendezvous) after which, the application proceeds sequentially until another workers-pool is created and the work proceeds concurrently. For level=2, we see that there are 18 pools with a total of 50 workers, and for level=3, these numbers are 42 and 170, respectively. For both the sparse- and the semi-sparse-grid applications, the numbers are summarized in table 2. Here, n_p denotes the number of pools, $(n_s)_{\max}$ the maximum number of workers in a pool and $(n_s)_{\text{total}}$ the total number



a. Sparse.

b. Semi-sparse.

Figure 6: Different pools of workers created during the parallel applications.

of workers in the application. Note the enormous amount of processes involved in the sparse-grid and the semi-sparse-grid applications and the big number of resulting rendezvous. The ebb & flow effect in the restructured application, brought about by the **Manifold** coordinators, is in principle a pleasant quality for an application to have. Applications written in such a style are very kind towards other users of a computer system because they ask for more resources only when there is a computational need for them and their resource requirements are more modest when they have less to do. In figure 7, we show the actual ebb & flow in the CPU usage during a run of the semi-sparse-grid application for level 6 on a multi-processor machine. During this run, which lasted for about 1200 seconds, we show after each second, the number of assigned processors to the application. These 1200 data points (given as percentages) are shown on the vertical axis and are set against time on the horizontal axis. A percentage on this axis, e.g. 800%, means that at that particular moment 8 processors are each 100% busy working on the application. Note the correspondence between figure 7 and the level 6 part of figure 6b. Probably

application	*level*	n_p	$(n_s)_{\max}$	$(n_s)_{\text{total}}$
sparse	1	6	3	10
	2	18	6	50
	3	42	10	170
semi-sparse	2	18	3	38
	4	82	7	336
	6	268	12	1838

Table 2: Workers-pool and worker statistics.

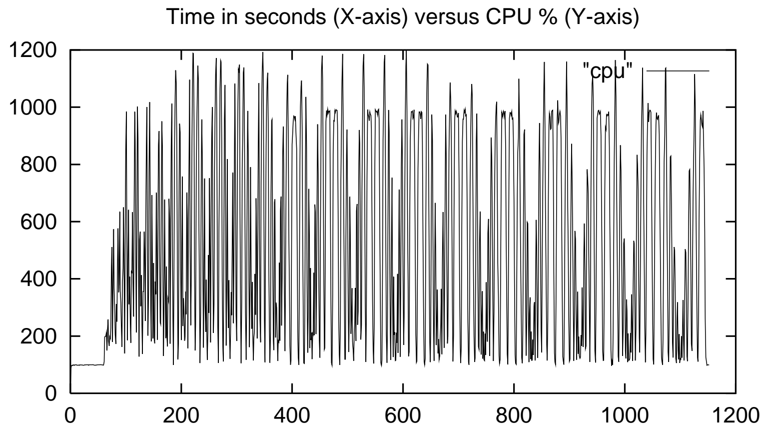


Figure 7: CPU usage in the semi-sparse application during the run on level 6.

the application actually gets all the resources it askes for.

8.2 Performance Results

In both experiments (i.e., the shared memory and the distributed runs) the grid levels considered in the sparse-grid application are 1, 2 and 3, and for the semi-sparse-grid application, the grid levels are 2, 4 and 6. The wing we used in our tests is a half-wing in transonic flight, more specificly, the standard test case of the ONERA M6 wing (see figure 8) at a far-field Mach number of 0.84 and 3.06^o angle of attack.

Both experiments were done during normal working days under the same conditions (multiple users, network traffic and file server delays, etc.) as in §6.4. To even out these "random" perturbations, we ran the two versions of the application on each of the three levels close to each other in real time. This has been done for each version of the application, five times on each level. The raw numbers obtained from both experiments are shown in some tables. In computing the average times given in these tables, the best and the worst performances in each row were discarded.

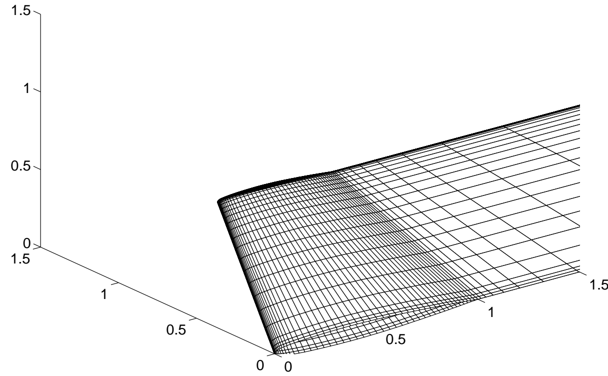


Figure 8: View at ONERA-M6 wing (and at corresponding grid plane).

8.2.1 Shared Memory Performance Results

In our first experiment, we used the same SGI Origin 2000 multi-processor machine with 32 processors as described in §6.4. The elepased times (in seconds) in this experiment are shown in tables 3a and 3b and were obtained using the UNIX utiliy `/bin/time`.

	level	*result 1*	*result 2*	*result 3*	*result 4*	*result 5*	*average*
sequential	1	3.74	3.76	3.80	3.97	4.16	3.85
	2	28.94	30.09	30.50	30.71	31.45	30.43
	3	184.13	187.11	187.79	188.27	191.96	187.73
shared	1	4.18	4.18	4.19	4.19	4.22	4.19
	2	31.19	31.38	31.75	32.01	32.86	31.71
	3	104.25	104.73	105.20	107.53	111.22	105.82

a. Sparse.

From this table we conclude the following:

- The restructuring does not pay off for the applications that run sequentially with an average of less than 31 seconds. In these cases (i.e., sparse at levels 1 and 2 and semi-sparse at level 2) the coordination introduced by the restructuring is too much overhead in comparison with the amount of computations performed in the computational processes.
- The restructuring pays off in the other cases (i.e., sparse at level 3 and semi-sparse at levels 2 and 4). The averaged realized speedups in these cases are respectively 1.77 (i.e., 187.73/105.82), 2.81 (i.e., 361.33/128.75) and 4.68 (i.e., 5342.51/1141.53).

Interpreting the results produced on this "non-dedicated" machine (recall the multiple users, network traffic and file server delays, etc.) and coming up with conclusions regarding the effectiveness of our

	level	*result 1*	*result 2*	*result 3*	*result 4*	*result 5*	*average*
sequential	2	16.52	16.58	16.59	17.06	17.71	16.74
	4	351.56	352.21	354.15	377.63	379.92	361.33
	6	5337.36	5339.77	5339.98	5347.79	5713.25	5342.51
shared	2	18.68	18.82	18.85	18.98	18.99	18.88
	4	116.02	124.08	128.79	133.39	135.49	128.75
	6	1116.14	1135.05	1136.34	1153.22	1155.59	1141.53

b. Semi-sparse.

Table 3: The SGI Origin elapsed times (in seconds).

restructuring is not so straight forward. Consider for example, the sequential time of result 3 of the semi-sparse run at level 6 (i.e., 5339,98 seconds) and compare this with its corresponding shared memory run (i.e., 1136.34 seconds). Is the gained speedup of 4.70 (i.e., 5339.98/1136.34) on a "non-dedicated" 32 processor machine a good result or not? Note that a full utilization of the 32 processors is certainly not possible because this application never asks for more than 13 computational processes (i.e., the master and 12 workers) of which at most 12 are working at any time. But, to what extent were the application's resource requests actually granted by the system? What we would like to know is how many of the total number of processors were actually working for the application and what is the amount of the overhead introduced by the restructuring.

To explore this, we insert in our computational processes (i.e., the master process instance and the process instances of the workers) timing calls just before and after the real computational work. In this way, we exclude the work involved in reading and writing data to and from ports, raising and handling of events, etc. This kind of work can be considered as overhead introduced by the restructuring of the application. In the timing routines we use the UNIX timer `times`. This routine returns the elapsed real time, in clock ticks, from an arbitrary point in the past (e.g., the system start-up time) and has a high resolution (100 HZ on our machines, i.e., 100 ticks per second). Running an application with the added timing routines yields a (huge) set of high-resolution timestamps of the following form: "'process m starts doing real computational work at clock tick x and is done with these computations at clock tick y". This information can easily be transformed into "During m% of the total elapsed clock ticks, n computational processes are running". For the "result 3" runs in tables 3a and 3b, these percentages are given in table 4. Almost at the end of this table, under the heading "average", we give the weighted average of the number of computational processes (in the table abbreviated as cp) at work on a tick (e.g., for the third column this is: $0 * 0.0231 + 1 * 0.7144 + 2 * 0.0372 + 3 * 0.0709 + 4 * 0.0511 + 5 * 0.0300 + 6 * 0.0456 + 7 * 0.0198 + 8 * 0.0064 + 9 * 0.0015 = 1.83$). On the last line of this table we also give the realized speedup for the "result 3" run as computed from tables 3a and 3b (e.g., $3.80/4.19 = 0.91$ etc.).

From this table we conclude the following:

- It is remarkable that for the sparse application at levels 1 and 2, and for the semisparse application at level 2, the applications are unable to take advantage of the parallel platform (observe their speedups, or rather speeddowns). For example, for the sparse application at level 1, we know from figure 6a that the theoretical demand for workers in a workers-pool can be three, however we do not see in table 4 more than one computational process running. The reason for this is that the computational work performed in these workers are so light that once a worker is told what to

	sparse			*semi-sparse*		
level	*1*	*2*	*3*	*2*	*4*	*6*
0 cp	3.02%	2.31%	2.31%	2.09%	5.91%	10.36%
1 cp	96.98%	97.69%	71.44%	97.86%	39.95%	22.20%
2 cp	0%	0%	3.72%	0.05%	6.84%	6.64%
3 cp	0%	0%	7.09%	0%	8.90%	6.47%
4 cp	0%	0%	5.11%	0%	5.63%	6.52%
5 cp	0%	0%	3.00%	0%	6.18%	6.72%
6 cp	0%	0%	4.56%	0%	22.47%	5.72%
7 cp	0%	0%	1.98%	0%	4.12%	4.60%
8 cp	0%	0%	0.64%	0%	0%	4.12%
9 cp	0%	0%	0.15%	0%	0%	3.37%
10 cp	0%	0%	0%	0%	0%	13.63%
11 cp	0%	0%	0%	0%	0%	2.71%
12 cp	0%	0%	0%	0%	0%	6.95%
average	0.97	0.98	1.83	0.98	2.97	4.94
speedup	0.91	0.96	1.79	0.88	2.75	4.70

Table 4: Actual demand for computational processes (cp) in the "result 3" runs.

do it does it before the master has a chance to initiate another worker (the master delegates the computational jobs one after the other to the workers, after which they run independently). Thus, workers performing such light-weight computations never really run concurrently.

- The overhead introduced due to the restructuring (and timing) is given on the first line of the table. The percentages given there, denote the number of clock-ticks during which no real computational work is performed (`cp = 0`). These ticks are spent on reading and writing data from and to ports, raising and handling of events, network traffic, file server delays, other delays caused by running in a multi-user environment, and the added timing primitives. In fact, we can consider these percentages as upperbounds for the overhead of the restructuring.

- The realized speedup figures for the "result 3" runs are in good correspondence with the average number of computational processes busy working during a clock tick.

Of course, we realize that adding timing calls to source code of an application distorts the program's performance. However, in most cases the elapsed times hardly differ from each other with and without the added timing calls. Sometimes the application ran even faster with the timing calls, which is not surprising on an multi-processor machine where we are not the only user. For the sparse application at levels 2 and 3, and for the semi-sparse application at all levels the difference was less the 4.9%. Only the semi_sparse application at level 1 shows a greater difference (sometimes even 13.3%). This is also is no surprise considering its low averaged elapsed time of 3.85, which makes it more sensitive to disturbances.

We return to the question of whether the gained speedup of 4.70 in the "result 3" run for the semi-sparse application at level 6 on the "non-dedicated" 32 processor machine, is a good result or not. We see that by spending 10.36% (see table 4) of the total elapsed clock ticks on non-computational issues (i.e., the restructuring overhead) an average of 4.94 computational processes run per clock tick in this run. Comparing this run with its corresponding sequential run gives a speedup of 4.70 which in this context we

consider as good. Because the figures as given in tables 3a and 3b are derived from executables wherein the timing calls are added in the shared memory runs, the speedup figures as given in table 4 are slightly pessimistic.

8.2.2 Distributed Memory Performance Results

In our second experiment we used the same cluster of SGI O2 single processor workstations, as described in §6.4.

It is a property of the (semi)sparse grid application that the computational work performed by a worker is heavier when that worker is in a bigger workers-pool. Thus, it obviously makes sense to spread the workers in bigger workers-pools over a cluster of workstations (three in our case). In the distribution we chose, the worker process instances are bundled in such a way that each task instance contains no more than $\lfloor (n_s)_{\max}/3 \rfloor$ worker process instances, where $(n_s)_{\max}$ is the maximum number of workers in a pool (see table 2). In §6.4, we explained that this can be done by choosing the right load and weight values in the input file for the MANIFOLD linker MLINK. The task instances that dynamically come into existence during the run are spread over the cluster of workstations specified in the input file for the MANIFOLD configurator CONFIG (see §6.4 for the details).

The elapsed times (in seconds) for this experiment are shown in tables 3a and 3b and were again obtained using the UNIX utility `/bin/time`. It was not possible to run the sparse application at level 3 nor the semi-sparse application at level 6, because their executables do not fit in the memory of the workstations.

	level	*result 1*	*result 2*	*result 3*	*result 4*	*result 5*	*average*
sequential	1	11.57	12.21	12.56	14.02	14.54	12.93
	2	97.65	100.85	101.68	103.11	103.40	101.88
	3	583.65	588.04	592.59	610.46	650.35	597.03
distributed	1	20.54	21.30	21.62	21.66	30.03	21.53
	2	159.19	163.2	168.58	168.98	172.01	166.93
	3	-	-	-	-	-	-

a. Sparse.

	level	*result 1*	*result 2*	*result 3*	*result 4*	*result 5*	*average*
sequential	2	54.82	55.92	57.23	59.65	63.97	57.60
	4	1168.49	1178.47	1188.88	1258.97	1350.50	1208.77
	6	18153.39	18639.26	18642.08	19027.86	19247.72	18769.73
distributed	2	84.30	85.00	85.61	85.89	86.98	85.50
	4	1242.87	1272.78	1286.23	1302.94	1314.51	1287.32
	6	-	-	-	-	-	-

b. Semi-sparse.

Table 5: The Cluster elapsed times (in seconds).

From this table we conclude that the distribution of the workers over the cluster of workstation does not pay off. The introduced coordination overhead of restructuring is simply too much in comparison with the amount of computations in worker process instances.

9 Conclusions

Our cut-and-paste restructuring essentially consists of picking out the computation subroutines in the original Fortran 77 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, and still manipulate the same global Fortran-common data arrays. The **MANIFOLD** glue modules, representing a master/worker protocol, are separately compiled programs that have no knowledge of the computation performed by the Fortran 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 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 Fortran subroutines, without any change. The master and worker manifolds used in the concurrent version only call C functions which are in fact (wrappers around) Fortran subroutines of the sequential program.

It is not so remarkable that sequential programs having a similar structure, but performing different algorithms (in our case the sparse-grid algorithm, semi-sparse-grid algorithm, and our toy algorithm) can be coordinated in a similar fashion. What is more interesting, as illustrated in our examples, is that we are able to abstract away the details of the computations; that it is possible to focus on the invariant (hidden) properties of seemingly very different programs, and that we can compile those invariant properties as coordination patterns in **MANIFOLD**. In fact, we compile structure. As in our examples, this same coordination structure (compiled **MANIFOLD** coordinators) can transparently run the same computation modules on parallel shared-memory or distributed cluster 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.

References

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

[2] W.L. Briggs. *A Multigrid Tutorial*. SIAM, Philadelphia, 1987.

[3] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer, Berlin, 1985.

[4] P. Wesseling. *An Introduction to Multigrid Methods*. Wiley, Chichester, 1992.

[5] P.W. Hemker. Finite volume multigrid for 3d-problems. In H. Deconinck and B. Koren, editors, *Euler and Navier-Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration*, volume 57 of *Notes on Numerical Fluid Mechanics*, pages 393–417. Vieweg, Braunschweig, 1997.

[6] M. Griebel, C. Zenger, and S. Zimmer. Multilevel gauss-seidel-algorithms for full and sparse grid problems. *Computing*, 50:127–148, 1993.

[7] P.W. Hemker, B. Koren, and J. Noordmans. 3d multigrid on partially ordered sets of grids. In W. Hackbusch and G. Wittum, editors, *Proceedings of the Fifth European Multigrid Conference*, volume 3 of *Lecture Notes in Computational Science and Engineering*, pages 105–124. Springer, Berlin, 1998.

[8] B. Koren, P.W. Hemker, and P.M. de Zeeuw. Semi-coarsening in three directions for euler-flow computations in three dimensions. In H. Deconinck and B. Koren, editors, *Euler and Navier-Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration*, volume 57 of *Notes on Numerical Fluid Mechanics*, pages 547–567. Vieweg, Braunschweig, 1997.

[9] S.K. Godunov. *Finite difference method for numerical computation of discontinuous solutions of the equations of fluid dynamics*. Matematicheskii Sbornik, 47, 1959. Cornell Aeronautical Lab. Transl. from the Russian.

[10] P.W. Hemker and S.P. Spekreijse. Multiple grid and osher's scheme for the efficient solution of the steady euler equations. *Applied Numerical Mathematics*, 2:475–493, 1986.

[11] S. Osher and F. Solomon. Upwind difference schemes for hyperbolic systems of conservation laws. *Mathematics of Computation*, 38:339–374, 1982.

[12] P.W. Hemker and C. Pflaum. Approximation on partially ordered sets of regular grids. Technical Report NW-R9611, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996.

[13] P.W. Hemker and P.M. de Zeeuw. A data structure for 3-dimensional sparse grids. In H. Deconinck and B. Koren, editors, *Euler and Navier-Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration*, volume 57 of *Notes on Numerical Fluid Mechanics*, pages 445–486. Vieweg, Braunschweig, 1997.

[14] N.H. Naik and J. Van Rosendale. The improved robustness of multigrid elliptic solvers based on multiple semicoarsened grids. *SIAM Journal on Numerical Analysis*, 30:215–229, 1993.

[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] U. Rüde. Multilevel, extrapolation, and sparse grid methods. In P.W. Hemker and P. Wesseling, editors, *Multigrid Methods IV*, volume 116 of *International Series of Numerical Mathematics*, pages 281–294. Birkhäuser, Basel, 1994.

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

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

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

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

[21] 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, June 1998. Extended version.

[22] F. Arbab. Manifold version 2: Language reference manual. Technical report, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996. Available on-line http://www.cwi.nl/ftp/manifold/refman.ps.Z.

[23] 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, September 1994.

[24] C.T.H. Everaars and F. Arbab. *An Introduction into the Coordination Language Manifold*. CWI, Amsterdam, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 2000. to appear.

[25] Bradford Nicols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Cebastopol, CA, 1996.