ELSEVIER

Parallel Computing 24 (1998) 1081–1106

PARALLEL
COMPUTING

Using coordination to parallelize sparse-grid methods for 3-D CFD problems

Kees Everaars *, Barry Koren [1]

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

Received 21 May 1997; revised 24 September 1997

Abstract

In this paper, we investigate the good parallel computing properties of sparse-grid solution techniques. To this end, an existing sequential computational fluid dynamics (CFD) code for a standard 3-D problem from computational aerodynamics is restructured into a parallel application. The restructuring is organized according to a master/worker protocol. The coordinator modules developed thereby are implemented in the coordination language MANIFOLD and are applicable to other problems than the present CFD problem only. Performance results are given for both the sequential and the parallel versions of the code. The results are promising. The paper contributes to the state-of-the-art in improving the efficiency of large-scale computations. We also present a theoretical analysis of speed-up through parallelization in a multi-user single-machine environment. © 1998 Elsevier Science B.V. All rights reserved.

Keywords: Parallel computing; Coordination languages; Models of communication; Multigrid methods; Sparse-grid methods; Computational fluid dynamics; Three-dimensional flow problems

1. Introduction

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 - $O(N^d)$ 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 $O(N^d)$ algebraic equations with the aforementioned $O(N^d)$ real numbers as

* Corresponding author. E-mail: kees.everaars@cwi.nl

[1] E-mail: barry.koren@cwi.nl

0167-8191/98/$ - see front matter © 1998 Elsevier Science B.V. All rights reserved.
PII: S0167-8191(98)00043-X

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., Refs. [1–3], where Ref. [1] 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 Ref. [4] 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 [5] than a standard multigrid method (which in turn already has a much better performance in this sense than a single-grid method).

The efficiency of sparse-grid methods can still be improved further; an advantage of the methods is their good suitability for implementation on a parallel computer or a cluster of workstations. In this paper we present the parallel implementation of an existing sparse-grid solution method for the steady, 3-D Euler equations of gas dynamics [6,7]. Our starting point is a sequential Fortran 77 code describing this standard problem. If, for instance, entire subroutines of this code can be plugged into a new *parallel* structure, the resulting renovated software can take advantage of the improved performance offered by modern parallel computing environments, without rethinking or rewriting the bulk of the existing code [8]. The good parallel computing properties of sparse-grid solution techniques allow us to perform such a coarse-grain restructuring. The restructuring is organized according to a master/worker protocol and essentially consists of picking out the computation subroutines in the original Fortran 77 code, and glueing them together with coordination modules written in MANIFOLD. Hardly any rewriting or 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 they still manipulate the same global data. The MANIFOLD glue modules 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 computing environment. MANIFOLD is a coordination language developed at CWI (Centrum voor Wiskunde en Informatica) in the Netherlands. It is very well suited for managing complex, dynamically changing interconnections among sets of independent concurrent cooperating processes [9,10].

The rest of this paper is organized as follows. In Section 2, we introduce the discrete equations under consideration. In Section 3, we describe the concept of sparse-grid methods. For this, first standard multigrid methods are described. In Section 4, we briefly describe the sequential implementation of the 3-D CFD code and pay attention to its good parallel computing properties. Next, in Section 5 we show how we can restructure this sequential 3-D software into a parallel code, using the coordination language MANIFOLD. In Section 6, we give an analysis of the speed-up figures in a

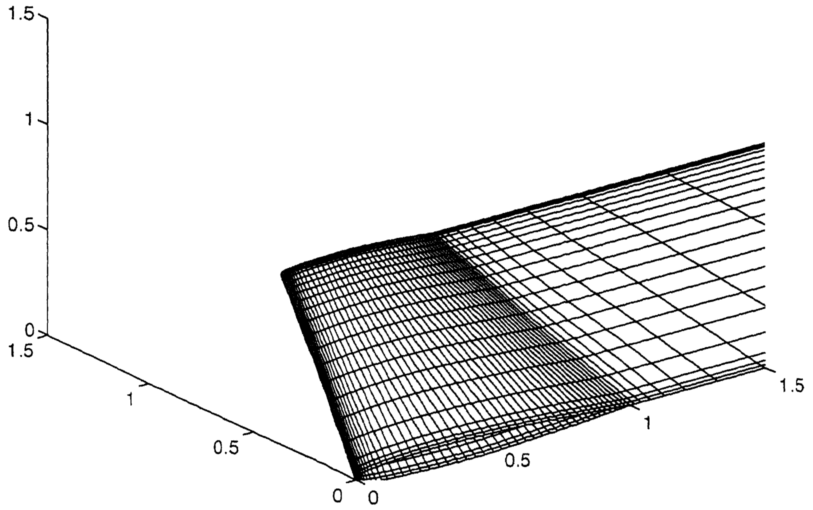


Fig. 1. View at ONERA-M6 wing (and at corresponding grid plane).

multi-user single-machine environment and show performance results for the test case of a half-wing in transonic flight: the standard test case of the ONERA M6 wing (Fig. 1) at a far-field Mach number of 0.84 and 3.06° angle of attack. Finally, the conclusion of the paper is in Section 7.

2. Equations

2.1. Continuous equations

In this paper, 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

$e = 1/(\gamma - 1)\frac{p}{\rho} + \frac{1}{2}(u^2 + v^2 + w^2)$, and in which $f(q)$, $g(q)$ and $h(q)$ are the so-called flux vectors

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

2.2. 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

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

where $\partial\Omega^*$ is the boundary of an arbitrary subdomain Ω^* 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 $\partial\Omega^*$. 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 Eq. (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

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

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 (Eq. (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 [11], 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 [12] of Osher's approximate Riemann solver [13]. For the left and right cell-face states, we take the first-order accurate approximations

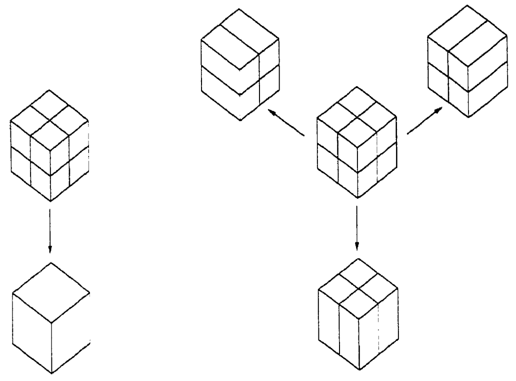
$$\begin{pmatrix} q^{\mathrm{l}}_{i+\frac{1}{2},j,k} \\ q^{\mathrm{r}}_{i+\frac{1}{2},j,k} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i+1,j,k} \end{pmatrix}, \; \begin{pmatrix} q^{\mathrm{l}}_{i,j+\frac{1}{2},k} \\ q^{\mathrm{r}}_{i,j+\frac{1}{2},k} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i,j+1,k} \end{pmatrix}, \; \begin{pmatrix} q^{\mathrm{l}}_{i,j,k+\frac{1}{2}} \\ q^{\mathrm{r}}_{i,j,k+\frac{1}{2}} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i,j,k+1} \end{pmatrix}. \tag{4}$$

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

3. 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



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

Fig. 2. Two types of 3-D coarsenings.

same. A more natural remedy is not to apply standard, i.e., full coarsening, but to use multiple semi-coarsening instead. Fig. 2a and b show standard coarsening and multiple semi-coarsening, respectively.

3.1. 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 Ref. [12].

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 [3], abbreviated as FAS), preceded by nested iteration (also called full multigrid [3], 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 × 2 × 2 volumes on the next finer grid (full coarsening, Fig. 2a). 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: $q_\lambda=q_\lambda+\tilde{I}_{\lambda-1}^{\lambda}\ (\tilde{q}_{\lambda-1}-q_{\lambda-1})$, where $\tilde{I}_{\lambda-1}^{\lambda}$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 $I_\lambda^{\lambda-1}$ and the prolongation operator $\tilde{I}_{\lambda-1}^{\lambda}$ are defined in Ref. [7].

3.2. 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 Ω_λ, $\lambda=0,1,\ldots,\lambda_{\max}$, by a partially ordered set of grids $\Omega_{l,m,n}$, $l=0,1,\ldots,l_{\max}$, $m=0,1,\ldots,m_{\max}$, $n=0,1,\ldots,n_{\max}$, 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 Refs. [14,15].

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 Ref. [4] (see Appendix A in Ref. [7] 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 operation, viz., the one described in Appendix B of Ref. [7]. (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 [16]. (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 Ref. [4]. (It is the correction–prolongation version of the solution prolongation described in Appendix A of Ref. [7], using fixed prolongation weights.) In Appendix C of Ref. [7], 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 $= l+m+n < l_{\max}+m_{\max}+n_{\max}$ contribute to the solution process, we speak of *full-grid-of-grids* semi-coarsening. A disadvantage of 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 [4,17,18]. 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 in Fig. 3a, the corresponding sparse grid-of-grids is the subset given in Fig. 3b. The reduction in the numbers of grid cells is enormous. The computational complexity of the sparse-grid-of-grids approach is $O(N\log^2N)$, 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 [5]. 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 [17]. This uses the family of grids $\Omega_{l,m,n}$, level $\leq 2l_{\max}$, $\max(l,m,n) \leq l_{\max}$ (see Fig. 3c), which (asymptotically) still has a computational complexity which is much smaller than that of the single-grid approach, viz., $O(N^2\log^2N)$, i.e., still almost the complexity of a 2-D problem only.

4. The 3-D CFD Fortran code

It is our experience that a parallel implementation is enhanced if first a sequential prototype is made available. In this way of working we can fully concentrate on the algorithmic aspects of our application and do not need to be occupied with all the ins and outs of parallel programming tools. For the present 3-D CFD algorithm, it becomes quickly clear which parts can run in parallel. The 3-D CFD code we consider in this section is sequential and is based on a data structure which is especially designed for the implementation of adaptive sparse-grid algorithms in three dimensions [15]. The full Fortran program consists of a data definition section, a main program and some 200 subroutines with a total length of some 8000 lines. In the following we give a small, but relevant part of the Fortran code, viz., a schematized version of the main program, the subroutine `fas` (Full Approximation Storage algorithm, also known as nonlinear multigrid algorithm, see Ref. [3], 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 the actual restructuring of it into a parallel application.

```
      program         oneram6

      include         'basis3.i'
      integer         level,levelmax,
     +                nmin,mmin,lmin,nmax,mmax,lmax
      logical         convergence
      external        fas,prolsolgr,scanlv
      common /gridset/ nmin,mmin,lmin,nmax,mmax,lmax

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c     main program
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

c     ------------------------------------------------------------
c     begin nested iteration

      do 20 level= 0,lmax

c        ---------------------------------------------------------
c        begin nonlinear multigrid iteration from all grids at
c        actual finest level

   10    call fas (level)

         if (convergence) then
            continue
         else
            goto 10
         endif

c        end nonlinear multigrid iteration from all grids at
c        actual finest level
c        ---------------------------------------------------------

c        ---------------------------------------------------------
c        begin solution prolongations from all grids at
c        actual finest level

         if (level.lt.lmax) then
            call scanlv (level+1,nmin,nmax,mmin,mmax,lmin,lmax,
     +                   prolsolgr)
         endif

c        end solution prolongations from all grids at
c        actual finest level
c        ---------------------------------------------------------

   20 continue

c     end nested iteration
c     ------------------------------------------------------------

c     ------------------------------------------------------------
c     begin solution prolongation to finest level

      do 30 level= lmax+1,levelmax
         call scanlv (level,nmin,nmax,mmin,mmax,lmin,lmax,prolsolgr)
   30 continue

c     end solution prolongation to finest level
c     ------------------------------------------------------------

      end
      subroutine       fas (level)

      integer          level,ilevel,
     +                 nmin,mmin,lmin,nmax,mmax,lmax
      logical          origrhs,plus
      external         copyrhsgr,copysolgr,pointgsgr,prolcorgr,
     +                 restrictgr,rhsgr,scanlv
      common /residu/  origrhs,plus
      common /gridset/ nmin,mmin,lmin,nmax,mmax,lmax
      external         copyrhsgr,copysolgr,pointgsgr,prolcorgr,
     +                 restrictgr,rhsgr,scanlv

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c     subroutine for nonlinear multigrid iteration
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      ilevel= level

c     pre-relaxations
   10 call scanlv (ilevel,nmin,nmax,mmin,mmax,lmin,lmax,pointgsgr)

      if (ilevel.eq.0) then
         goto 20
      endif

c     computation of defects
      call scanlv (ilevel,nmin,nmax,mmin,mmax,lmin,lmax,copyrhsgr)
      origrhs= .false.
      plus=    .false.
      call scanlv (ilevel,nmin,nmax,mmin,mmax,lmin,lmax,rhsgr)

c     computation of coarse-grid righthand sides
      call scanlv (ilevel-1,nmin,nmax,mmin,mmax,lmin,lmax,restrictgr)
      origrhs= .true.
      plus=    .true.
      call scanlv (ilevel-1,nmin,nmax,mmin,mmax,lmin,lmax,rhsgr)

c     back-up of coarse-grid solutions
      call scanlv (ilevel-1,nmin,nmax,mmin,mmax,lmin,lmax,copysolgr)

      ilevel= ilevel-1
      goto 10

c     post-relaxations
   20 call scanlv (ilevel,nmin,nmax,mmin,mmax,lmin,lmax,pointgsgr)

      if (ilevel.eq.level) then
         goto 40
      else
         goto 30
      endif

c     prolongation of corrections
   30 call scanlv (ilevel+1,nmin,nmax,mmin,mmax,lmin,lmax,prolcorgr)

      ilevel= ilevel+1
      goto 20

   40 return
      end
```

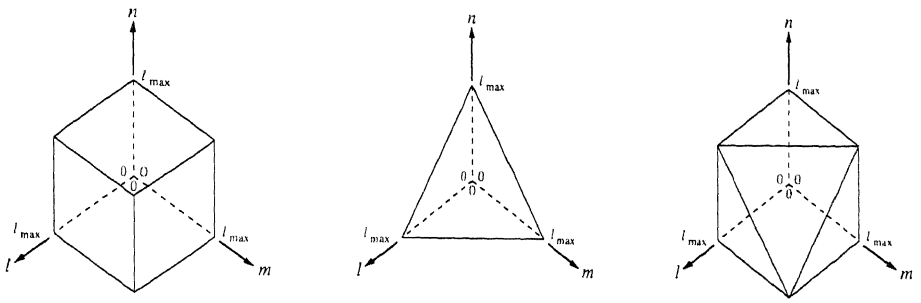
```
      subroutine scanlv (lev,nmin,nmax,mmin,mmax,lmin,lmax,tkgrid)

      integer    lev,nmin,nmax,mmin,mmax,lmin,lmax,n,m,l
      external   tkgrid

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c     subroutine for performing the user-defined operation tkgrid on
c     all grids at multigrid level lev
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      do 20 n= nmin,nmax
         do 10 m= mmin,mmax
            l= lev-m-n
            if ((l.le.lmax).and.(l.ge.lmin)) then
               call tkgrid (n,m,l)
            endif
   10    continue
   20 continue

      return
      end
```



a. Full. b. Sparse. c. Semi-sparse.

Fig. 3. Cubic, full grid-of-grids and the corresponding sparse and semi-sparse grid-of-grids.

In the pre- and post-relaxations (lines 20 and 45, respectively, in subroutine `fas`), the subroutine `scanlv` visits all the grids $\Omega_{l,m,n}$ at level $l+m+n$ and calls there the subroutine `pointgsgr` (which is the actual parameter of `tkgrid` on line 15 in subroutine `scanlv`). `pointgsgr` carries out a point Gauss–Seidel relaxation on all cells of grid $\Omega_{l,m,n}$ and because the subroutine `pointgsgr` only reads and writes data concerning its own grid, the relaxations can in principle be done in parallel for all the grids to be visited at a certain grid level. Given the fact that almost all computing time consumed by the full program, is used in the relaxations, parallel implementation is expected to pay off. This will be worked out in Section 5.

5. Restructuring the 3-D CFD code

In this section we describe the restructuring of the Fortran code, as presented in Section 4, into a parallel application. For the parallelization we use MANIFOLD. MANIFOLD is a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes [9]. MANIFOLD is based on the IWIM model of communication [10]. The basic concepts in the IWIM model are *processes*, *events*, *ports* and *channels*.

The crux of our restructuring is to allow the computations done in `pointgsgr` on every single grid visited with `scanlv`, to be carried out in separate processes. These processes can then run in parallel in MANIFOLD, as separate threads executed by different processors on a multi-processor hardware (e.g., a multi-processor SGI machine).

Separating this computation into a number of concurrent processes means that the information contained in the global data structures used in the `pointgsgr` subroutine must be supplied to each, and the results produced by each process must be collected. The obvious way to accomplish this is to arrange for the MANIFOLD coordinators to send and receive the (proper segments of the) global space through streams. This scheme is both easy to understand and easy to implement. However, at least in the special case of our application, it suffers from the burden of unnecessary communication overhead. Observe that several *pointgsgr* subroutine calls running as different MANIFOLD processes can run as threads (light-weight processes) in the same operating-system-level (heavy-weight) process, and thus can share the same global space. Thus, they do not need to receive their own individual copies of the space. This reduces the number of copies of the global space from one per MANIFOLD process to one per MANIFOLD task (where a MANIFOLD task is an operating-system-level process that runs somewhere on a parallel platform, and contains several MANIFOLD processes, each running as a separate thread).

For simplicity, in the restructuring presented in this paper we assume there is only one MANIFOLD task which contains several MANIFOLD processes. The restructured program we present here is thus not suitable for distributed memory computing. The additional book-keeping and extra communication necessary to run this example on a distributed platform is beyond the scope of this paper. Note that the restructured program we present here, nevertheless, *does* improve the performance of the application on a parallel platform (Section 6). For instance, in our configuration of MANIFOLD on a multi-processor SGI machine, some 30 threads in the same task can run `pointgsgr` concurrently, each on a different grid. With n the number of processors on the machine, at most n of these threads can run in parallel with each other.

5.1. The master / worker protocol

The restructuring of the Fortran code can be described in a kind of master/worker protocol. In a coordinator process (which is an instance of a protocol manifold named `ProtocolMW`) we create and activate a master process (named `oneram6`) that embodies the computations of the main program of the sequential version. When we arrive in master `oneram6` at a pre- or post-relaxation, the master delegates the computations done in `pointgsgr` to a separate worker process, for each single grid visited. Each time the master needs a worker, it raises an event to signal the coordinator to create the worker. In this way a pool of workers is working for the master, each worker performing the computations embodied in `pointgsgr`. The coordinator makes the identification of the worker known to the master `oneram6` by sending a reference of it to the master. With this information the master can activate the worker. Before the worker can really work, it must know on which grid (identified by the grid-of-grid coordinates n, m, l) it must perform the relaxation. The master has these coordinates available and writes them on its own output port. The coordinator takes care that the worker can read this information from its input port by setting up a stream between the output port of the master and its own input port. The master process continues its work and again requests the creation of another worker process. When all the workers are created and activated in this way, the master waits until the workers are done with the relaxation and are ready to die. After this rendezvous, the master `oneram6` proceeds with its sequential work until it again arrives at a point where it needs a pool of workers to delegate the relaxations to.

5.2. The manifold code of the protocol and its parameters

In MANIFOLD, we can easily realize the master/worker protocol described in Section 5.1 in a general way where the master and worker are parameters of the protocol. In this protocol we only describe how instances of the 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 worker. What is indeed important for the protocol is that the in/output and the event behavior of the master and worker comply with the protocol. E.g., the protocol manifold can create a worker only when the master requests for its creation by raising an event. Also, the master should write the data needed by the worker, on its own output port and the worker should read this information from its own input port, etc.

Below we give the MANIFOLD source code of the master/worker protocol and a stepwise description of the behavior interface of the master and worker manifold.

```
 1 // protocolMS.m
 2
 3 #define IDLE terminated(void)
 4
 5 export manifold ProtocolMS(manifold Master, manifold Worker,
                              event create_worker, event ready)
 6 {
 7   auto process master is Master.
 8
 9   begin:  (master, IDLE).
10
11   create_worker: {
12     process worker is Worker.
13
14     begin: (&worker -> master -> worker, IDLE).
15   }.
16
17   ready: halt.
18 }
```

The behavior interface of the master is as follows:
1. Perform some sequential work (optional).
2. Perform some work in parallel by creating a pool of workers and charge each with a computational job. Do this as follows:
   (a) Request a coordinator process (which is an instance of the protocol manifold `ProtocolMW`) to create a worker process by raising an event.
   (b) Wait, if necessary, for the availability of a unit (sent by the coordinator through the input port), which contains the identification (reference) of a worker process.
   (c) Now the master knows the identification of the worker, it can activate the worker.
   (d) Write the information, which the worker needs to do its job, on the output port. (The coordinator takes care that the worker can read this information.)
   (e) Repeat steps a, b, c and d for each worker that is needed. (In this way a pool of workers is created.)
   (f) Wait until all workers in the pool are ready to die (rendezvous).
3. Repeat 1 and 2 as many times as needed and raise an event to signal the coordinator process that the master is ready.

The behavior interface of the worker is as follows:
1. Read the information you need to know to do your job, from your own input port.
2. Do a computational job.

We now describe the MANIFOLD code of our protocol.

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.

Line 3 defines a pre-processor macro, in the same syntax as that of the C pre-processor.

Line 5 defines a manifold named ProtocolMW, which takes four arguments and states (through the keyword export) that this manifold can be used in other source files which import this MANIFOLD definition (as we will see later).

The first two arguments of ProtocolMW are the master and the worker manifolds, respectively. With these two parameters, ProtocolMW is independent of a particular master or worker, as long as they abide by the behavior interface described above. The third argument, with formal name create_worker, is an event the master raises to signal the ProtocolMW to create a worker. The fourth argument is an event, with formal name ready, which the master manifold raises to signal the ProtocolMW that it has completed its work.

Line 7 defines a process instance of the formal manifold argument Master, 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 in which it is defined. In this case the scope is defined by the '{' and '}' on lines 6 and 18, respectively.

The body of the manifold ProtocolMW is a block, i.e., the lines of code in between '{' and '}', containing at least one begin state. The present block has three states: the begin, create_worker and ready states (lines 9, 11 and 17). Activation of an instance of ProtocolMW automatically posts an occurrence of the special event begin in the event memory of that process instance. This makes the initial transition to the begin state possible.

In the body of the begin state (i.e., everything after the colon on line 9) we make the state sensitive for events from the master by taking the master up in the state body and we wait 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 3) of the IDLE macro (line 14). Because the special process void never terminates, this effectively causes the ProtocolMW instance to hang in the begin state until it detects an event in its event memory for which it has a state. Such an event will come soon, because an instance of master is expected to raise the event create_worker as soon as it wants a worker to delegate some work to. This event pre-empts the begin state and makes a state transition possible: the instance of ProtocolMW enters its second state - the create_worker state (lines 11–15). In this state we create a process named worker, which is an instance of the manifold Worker. Explicit creation of a process instance within a manifold is always done in the beginning of a block; in this case the block is formed by the braces on the lines 11 and 15, and the process creation takes place on line 12.

In the begin state of this block the stream configuration on line 14 is constructed and we wait for events (due to the word IDLE) from the master (create_worker and ready are possible events). In the stream configuration we see that the process identification of the worker (denoted by &worker) is sent through a stream (the first → on line 14) to the already active master, which sends the information the worker needs to do its job, through a stream (the second → on line 14) to the worker. When the master needs another worker, it again raises the create_worker event; this pre-empts the create_worker state and causes a state transition to the create_worker state. In this way, all workers are created and activated. Note that ProtocolMW knows nothing about the work pools in which the workers are housed. This is completely determined in the master (see points 2e and 2f in the master's behavior interface).

Finally, when the master has completed its work, it raises the ready event. This causes a state transition to the ready state on line 17, in which the primitive action halt effectively terminates the ProtocolMW instance.

5.3. A 'protocol' library

It is good practice to compile manifolds that embody general applicable coordinators (such as our ProtocolMW) separately and to archive them in what we can call a 'protocol' library. If we want to use, e.g., ProtocolMW we retrieve it from this library and use as its actual parameters a pair of user-supplied master and worker manifolds that behave according to the prescribed behavior, as documented in the reference manual of such a 'protocol' library. Such a pre-compiled library forms a powerful tool for the computing community.

The notion of a 'protocol' library can only exist when there is clear separation between computation modules (the master and the worker manifolds) and coordination modules (ProtocolMW). MANIFOLD, as a pure coordination language that encourages this separation of computation and communication concerns, is a perfect language for implementing such 'protocol' libraries [19,20].

Note that this way of working with a 'protocol' library is completely analogous to the use of, e.g., the *qsort* routine of the standard C library. This routine performs a quick sort algorithm on an array of *any* data type, and has a parameter which defines the sorting order. The user is free to implement this routine as long as it abides by the interface (behavior) prescribed by *qsort*. So *qsort* is not interested whether it is sorting apples or oranges but only expects that the user-supplied compare function returns a negative number if, e.g., apple A is considered to precede apple B because, e.g., it is bigger and red.

In Section 5.4, we describe the actual parameters which we use for the formal parameters in ProtocolMW.

5.4. The actual master and worker manifold

The master and the worker manifolds are easy to implement as atomic processes written in C. These C functions then call the original Fortran code to do the real work. The only changes we make are in the program oneram6 and in the subroutine fas (Section 4). The changes are the following.

- On line 1, program oneram6 is changed into subroutine oneram6.
- After line 61 (before the end statement), we add the line call raise_it.
- In the fas subroutine, on line 20 and line 45, we change the call to scanlv into a call to a new function named concurrent. Apart from the last formal parameter, tkgrid, this function has the same functionality as scanlv. It will be explained later.

The master and worker manifolds are contained in the file model.ato.c, listed below, where we code all the atomic processes and auxiliaries.

```
/* model.ato.c */

#include "AP_interface.h"
#include "debug.h"

AP_Event cp, fin;

/******************************************************************/
void w_oneram6(void)
{
  int err;

  extern void oneram6_(void);

  cp = AP_AllocateEvent();                                          P(cp)
  err = AP_InitHeaderEvent(cp, "create_pointgsgr");                 I(err)

  fin = AP_AllocateEvent();                                         P(fin)
  err = AP_InitHeaderEvent(fin, "finished");                        I(err)

  oneram6_();
}

/******************************************************************/
void concurrent_(int *pilevel,
                  int *pnmin, int *pnmax, int *pmmin, int *pmmax, int *plmin, int *plmax)
{
  AP_Process p = AP_AllocateProcess();
  int input = AP_PortIndex("input");
  int output = AP_PortIndex("output");
  AP_Unit u;
  AP_Event r = AP_AllocateEvent();
  AP_EventPatternSet eps = AP_AllocateEventPatternSet();
  AP_Process q = AP_AllocateProcess();
  int err, i;
  int ar[3];
  int now = 0;
  int ilevel = *pilevel, nmin = *pnmin, nmax = *pnmax,
                         mmin = *pmmin, mmax = *pmmax,
                         lmin = *plmin, lmax = *plmax;
  int n, m, l;
                                                                    P(p)
                                                                    I(input)
                                                                    I(output)
                                                                    P(r)
                                                                    P(eps)
                                                                    P(q)

  for (n = nmin; n <= nmax; n++) {
    for (m = mmin; m <= mmax; m++) {
      l = ilevel - m - n;
      if ((l <= lmax) && (l >= lmin) ) {

        err = AP_Raise(cp);                                         I(err)

        err = AP_PortRemoveUnit(input, &u, NULL);                   I(err) P(u)
        err = AP_DerefProcess(p, u, NULL, NULL);                    I(err)
        err = AP_Activate(p);                                       I(err)

        ar[0] = n; ar[1] = m; ar[2] = l;
        u = AP_FrameIntegerArray((int *) ar, 3);
        err = AP_PortPlaceUnit(output, u, NULL);                    I(err)

        now++;

        err = AP_EventPatternSetInsert(eps, AP_death, p);           I(err)
      }
    }
  }

  for (i = 1; i <= now; i++) {
    err = AP_DeleteWaitEvent(eps, r, q);                            I(err)
  }
}

/******************************************************************/
void raise_it_(void)
{
  int err;

  err = AP_Raise(fin);                                              I(err)
}

/******************************************************************/
void w_pointgsgr(void)
{
  int input = AP_PortIndex("input");
  int err;
  AP_Unit u;
  int ar[3];
  int n, m, l;

  extern void pointgsgr_(int* n, int* m, int* l);

  err = AP_PortRemoveUnit(input, &u, NULL);                         I(err) P(u)
  err = AP_FetchIntegerArray(u, ar, 3);                             I(err)
  err = AP_DeallocateUnit(u);                                       I(err)
  n = ar[0]; m = ar[1]; l = ar[2];
  pointgsgr_(&n, &m, &l);
}
```

The source code of the master consists of the following. We write a C function (lines 9–22) named w_oneram6 (an atomic process) in which we call the Fortran subroutine oneram6 (line 21) (the former main program in the sequential version). Thus w_oneram6 is in fact a C wrapper around the Fortran subroutine oneram6. Note the underscore behind oneram6 on this line. Here we use the fact that on many platforms, a Fortran subroutine X can be called from C, as a C function named X_.

To implement atomic processes we need the atomic process interface: a standard MANIFOLD library with many C functions, which allows access to the MANIFOLD world. All function calls in file model.ato.c starting with AP_refer to functions in this library.

As already mentioned, the master needs two events, one to request for using a worker, and one to signal ProtocolMW when it is ready with its work. These are the two global events, named cp and fin, on line 6 in model.ato.c. With the AP_ calls on lines 15 and 18, we allocate memory for these events. On lines 6 and 19 we couple the two events to create_pointgsgr and finished, respectively. These are the names under which the events are known outside model.ato.c.

Each time, after doing some sequential work, the Fortran routine oneram6_, called on line 21 of file model.ato.c, arrives at a (pre- or post-) relaxation. We have replaced the call to scanlv by a call to the new routine concurrent. In the routine concurrent we create a pool of workers (lines 48–68) and introduce a synchronization point by waiting until all the workers are ready to die (rendezvous). All the grids to be visited in scanlv are specified on lines 11–13 in subroutine scanlv (Section 4). In the master manifold, these grids are specified on the lines 48–51 in model.ato.c. Instead of a call to pointgsgr for each grid, as is done in scanlv, the master raises an event cp (line 53) to request ProtocolMW to create a worker, and waits (line 55) for the availability of a unit u (sent by ProtocolMW on line 14 of file protocolMW.m) at the input port of the master (set on line 28). This unit contains the identification of a worker process. On line 56, we read the process identification of the worker process from this unit and activate the worker. At that moment the first worker is in the pool and others will follow soon, as specified in the loop structure on lines 48–68.

On line 59, the information the worker needs to know to do its job (three integer coordinates specifying the grid to be visited) is assigned to an integer array. On line 61, it is packed as a unit and placed at the output port (set on line 29) of the master.

Because we want a synchronization point where we wait until all workers in the pool are ready to die, we must count the number of workers (denoted by now, line 63) and we add the death event (AP_death) of the worker process (p) (as an 'event pattern') to what is called an 'event pattern set' (eps). When the loop structure ends on line 68, the event pattern set eps contains the death events AP_death from all workers. This set is used to create the rendezvous.

When a worker is ready with its work, it raises a death event which is received by the master. This event is not raised explicitly (there is no AP_raise call in the worker), but is a part of the termination protocol of every manifold.

With the call on line 71 the master waits, if necessary, until an event occurrence is detected that matches one of the event patterns in the event pattern set eps. In this way, we scan all death events from the workers (lines 70–73) and the routine concurrent returns. After this, the master proceeds in a sequential way until it arrives again at a pre- or post-relaxation and the procedure is repeated. Finally, the master is done with its job, which is signalled to protocolMW by calling the new routine raise_it (lines 76–81, recall the second change we made in oneram6). On line 80 the event fin is raised.

The worker manifold (lines 84–99) is also implemented as a C wrapper, this time around the Fortran subroutine pointgsgr, callable from C as pointgsgr_. On line 94 the worker waits, if necessary, for the availability of a unit through its input port (set on line 86) that contains information about the grid to be visited (three integers). On line 95 this information is assigned to an array ar. The unit u is deallocated on line 96, because it is not needed any more and the three integers are used as parameters for the pointgsgr_ call, the Fortran routine from the sequential version (lines 97–98).

The I and P at the end of several lines in model.ato.c are C macros which check the return values of the AP_calls.

It is clear that the master and worker constructed in this way fully satisfy the behavior interface of the master/worker protocol given in Section 5.1.

5.5. The actual Manifold program

Using the manifold ProtocolMW together with the two actual parameters as described in Section 5.4, we can construct the following small MANIFOLD program, which finally changes our original sequential CFD application to a concurrent version.

```
 1 // model.m
 2
 3 event create_pointgsgr, finished.
 4
 5 manifold w_pointgsgr atomic {internal.}.
 6
 7 manifold w_oneram6 atomic {internal. event create_pointgsgr, finished.}.
 8
 9 manifold ProtocolMS(manifold Master, manifold Worker, event create_worker, event ready)
   import.
10
11 /********************************************************************/
12 manifold Main
13 {
14   begin: ProtocolMS(w_oneram6, w_pointgsgr, create_pointgsgr, finished).
15 }
```

On line 3 we declare two events, create_pointgsgr and finished. Because the declaration of these events appears outside of any blocks in this source file, they are global events, known in the entire source file.

Line 5 defines the worker manifold named w_pointgsgr, which takes no arguments, and states (through the keyword atomic) that it is not implemented in the MANIFOLD language, but in another programming language such as 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.

The same holds for the master manifold w_oneram6 (line 7). Because the events create_pointgsgr and finished are to be exchanged between the master and the rest of the MANIFOLD application, we also specify these two events between the brackets on this line.

Line 9 defines the manifold protocolMW which has four parameters: the master and worker manifolds and the two events create_pointgsgr and finished, respectively. The keyword import states that the real definition (i.e., the body) of this manifold is given elsewhere, e.g., in a library (as in our case) or in another source file.

Lines 12–15 define the manifold named Main which has only one state - the begin state. In this state a process instance of protocolMW is created and activated (this is done implicitly, by using the manifold name protocolMW). After this, the instance of Main (named main) terminates and the instances of protocolMW and w_oneram6 (with all the workers w_pointgsgr) run concurrently.

The object file obtained by compiling this MANIFOLD program must be linked with the object files obtained from the Fortran code and the C code (model.ato.c), to produce an executable file. The result of running this executable (on a single and/or multi-processor machine) is identical to the output produced by the original sequential Fortran code.

6. Performance analysis

6.1. Speed-up analysis

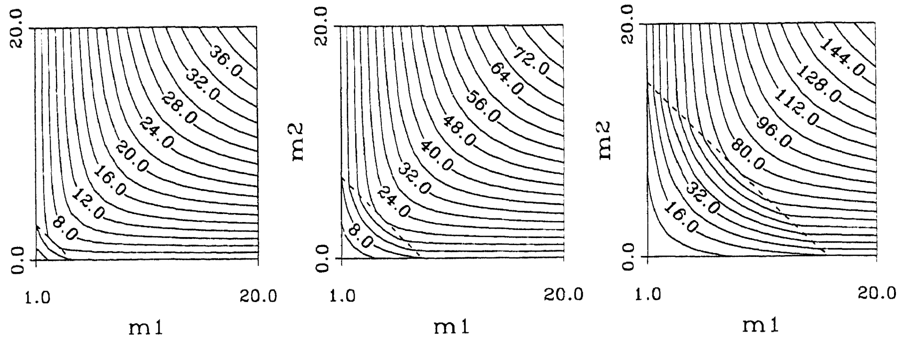
A number of experiments were conducted to obtain concrete numerical data to measure the effective speed-up of our parallelization. All experiments were run on a single multi-processor machine in a real contemporary computing environment, i.e., an environment in which it cannot be guaranteed that one is the only user. In such an environment, care should be taken in interpreting speed-up numbers. This is shown in the following multi-user, single-machine analysis, in which we make the following assumptions:

- the only processes which are significant with respect to the use of CPU time are computing processes,
- all computing processes get equal time slices from the scheduler of the machine and they totally consume their allotted time slices,
- the computational work embodied in a sequential program can be completely and equally distributed over parallel processes.

Then, with n the number of processors in a machine ($n \geq 1$), m_1 the number of processes in our own application ($m_1 \geq 1$; $m_1 = 1$ representing the sequential application) and m_2 the number of processes from other users ($m_2 \geq 0$), we can write an expression for the investment of CPU power p in our own application:

$$p = \begin{cases} n\dfrac{m_1}{m_1+m_2}, & m_1+m_2 > n. \\ m_1, & m_1+m_2 \leq n. \end{cases} \tag{5}$$

With the investment of CPU power inversely proportional to the elapsed computing times needed, from (5), expressions for various speed-up factors can be derived. As examples, we look at two of these factors.



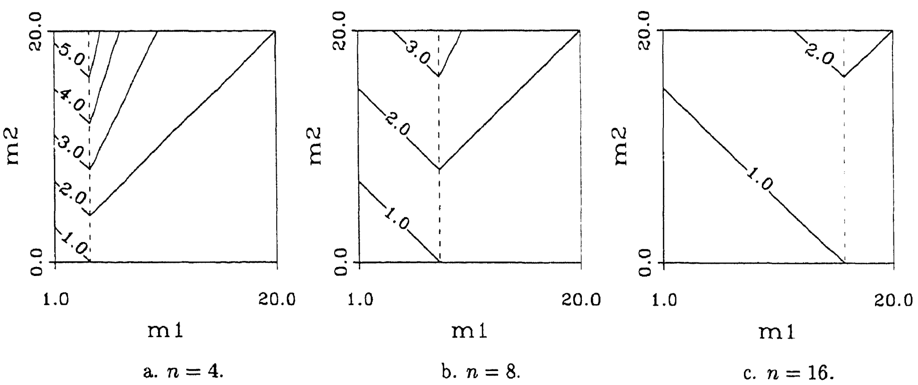
a. $n = 4$. b. $n = 8$. c. $n = 16$.

Fig. 4. Distributions of speed-up factors which can be expected when running a code containing m_1 parallel processes on a multi-processor machine ($n > 1$), instead of running the sequential version of that code on a single processor from that machine ($n = 1$, $m_1 = 1$), both applications with m_2 other processes running simultaneously.

The first speed-up factor relates the computing time of our parallel application run on a multi-processor machine ($n > 1$, $m_1 > 1$), to that of the sequential version run on a single-processor of the same machine ($n = 1$, $m_1 = 1$), in presence of the same number m_2 of other processes in both cases. We denote this speed-up factor as s_{n,m_1}:

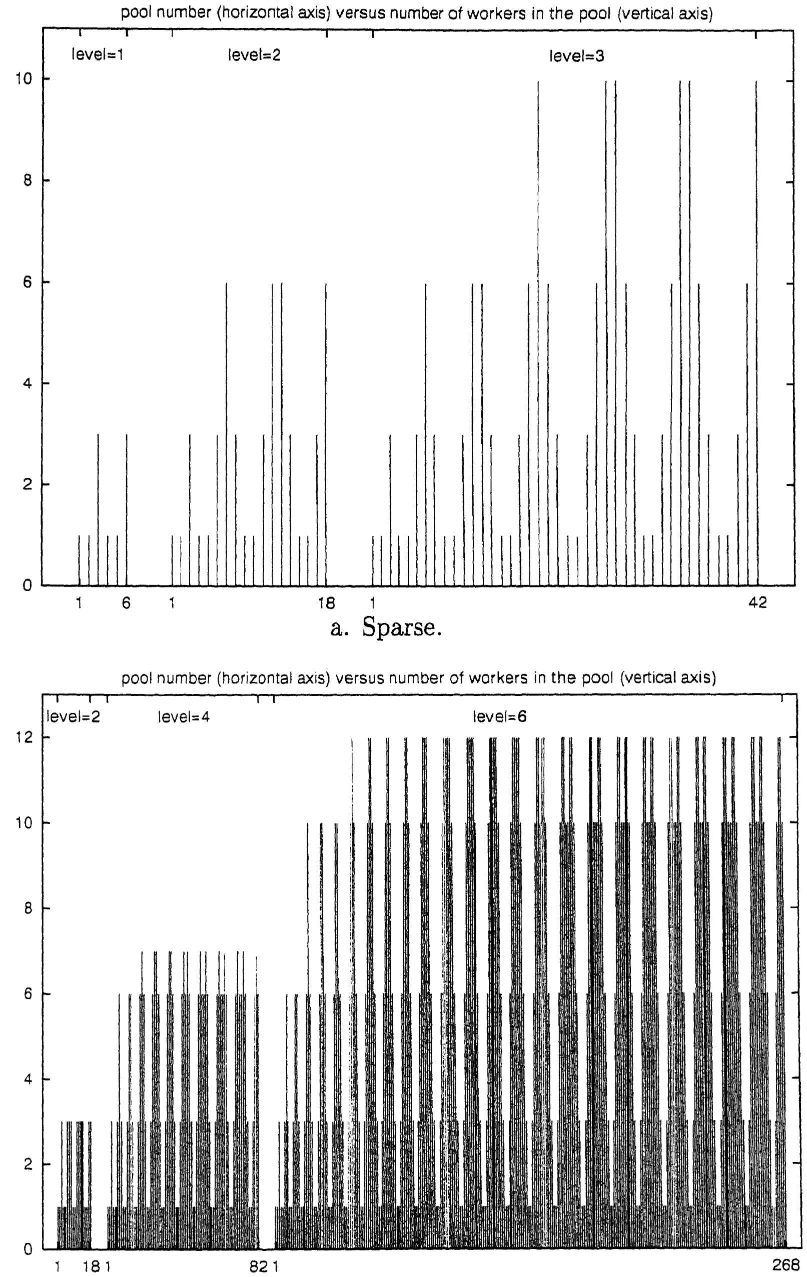
$$s_{n,m_1} \equiv \frac{p(n > 1, m_1 > 1, m_2)}{p(n = 1, m_1 = 1, m_2)} = \begin{cases} n(1 + m_2)\dfrac{m_1}{m_1 + m_2}, & m_1 + m_2 > n. \\ m_1(1 + m_2), & m_1 + m_2 \leq n. \end{cases} \tag{6}$$

Note that for the multi-user ($m_2 > 0$) situation, the speed-up s_{n,m_1} can be much larger than the number of processors n when $m_1 + m_2 > n$. Note also, that s_{n,m_1} *is* always larger than the number of processes m_1 when $m_1 + m_2 \leq n$. In Fig. 4, distributions of



a. $n = 4$. b. $n = 8$. c. $n = 16$.

Fig. 5. Distributions of speed-up factors which can be expected when running a code containing m_1 parallel processes together with m_2 other processes ($m_2 > 0$), instead of with no other processes ($m_2 = 0$), both runs on an n-processor machine.



a. Sparse.

b. Semi-sparse.

Fig. 6. Different pools of workers created during the parallel applications.

s_{n,m_1} are depicted for 4-, 8- and 16-processor machines, respectively. (Of course, the speed-up factors are defined at the integer points (m_1, m_2) only, the iso-lines as drawn in between these points are meant only to help in recognizing the discrete speed-up patterns.)

The second speed-up factor to be considered relates the computing time of our application when run on a machine with other processes running simultaneously, to that of a run on the same machine, but with no other processes. The corresponding speed-up factor, denoted as s_{m_2}, is:

$$s_{m_2} \equiv \frac{p(n,m_1,m_2=0)}{p(n,m_1,m_2>0)} = \begin{cases} \dfrac{m_1+m_2}{m_1}, & m_1 > n. \\ \dfrac{m_1+m_2}{n}, & m_1 \le n \text{ and } m_1+m_2 > n. \\ 1, & m_1+m_2 \le n. \end{cases} \tag{7}$$

In Fig. 5, distributions of s_{m_2} are depicted for 4-, 8- and 16-processor machines, respectively. Formula (7) may be practically relevant in comparative studies. With (7), from elapsed times measured in an environment in which a known number of other processes have been running simultaneously, one may approximately calculate the corresponding times in a hypothetical single-user $(m_2 = 0)$ environment. With the theoretical s_{m_2} computed from (7) and with the real (elapsed) time t_{real} $(m_2 > 0)$ measured, we may estimate the corresponding elapsed time in a single-user $(m_2 = 0)$ environment by

$$t(m_2=0) = s_{m_2} t_{real}(m_2>0). \tag{8}$$

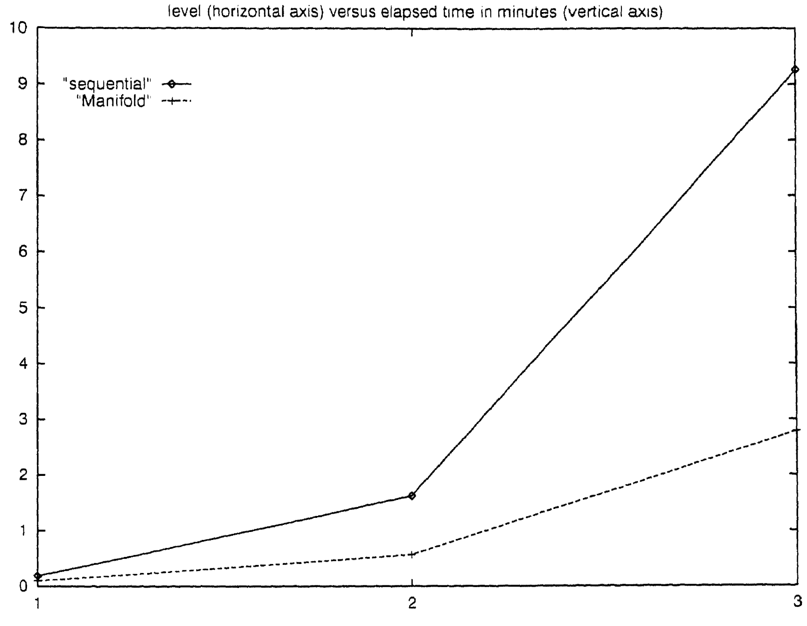
All our experiments were conducted during quiet periods of the system $(m_2 \approx 0)$. Therefore, since we mostly had $m_1 > n$, in our case $s_{m_2} \approx 1$ holds.

6.2. Performance results

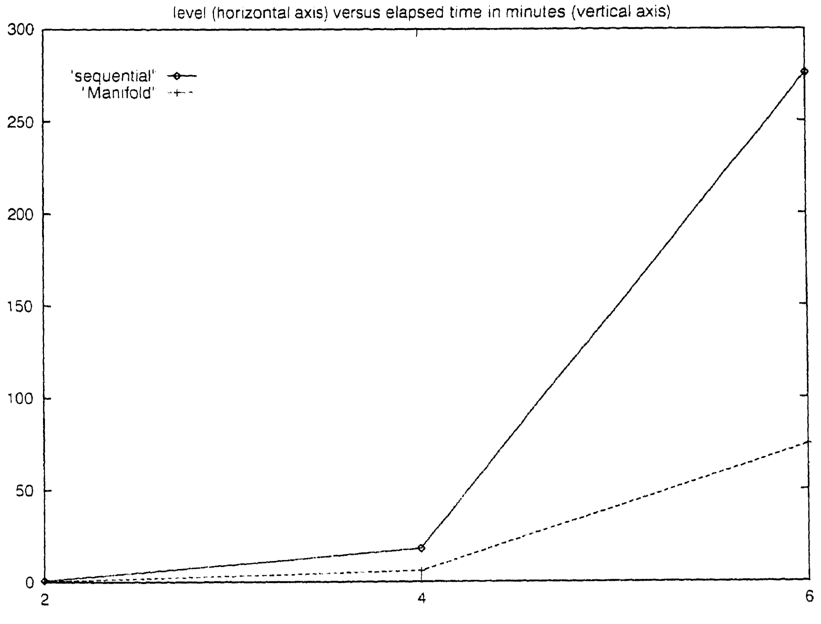
All experiments were run on an SGI Challenge L with four 200 MHz IP19 processors, each with a MIPS R4400 processor chip as CPU and a MIPS R4010 floating point chip for FPU. This 32-bit machine has 256 megabytes of main memory, 16 kilobytes of instruction cache, 16 kilobytes of data cache, and 4 megabytes of secondary

Table 1
Work pool and worker statistics

Application	Level	n_p	$(n_w)_{\max}$	$(n_w)_{\text{total}}$
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



a. Sparse.



b. Semi-sparse.

Fig. 7. Performances (average elapsed times) of the sequential version and the parallel version of the algorithm.

unified instruction/data cache. This machine runs under IRIX 5.3, is on a network, and is used as a server for computing and interactive jobs. Other SGI machines on this network function as file servers.

Computations were done for both the sparse- and the semi-sparse-grid approach. For the sparse-grid approach, the finest grid levels considered are: 1, 2 and 3, for the semi-sparse-grid approach, the finest grid levels are: 2, 4 and 6.

For both approaches, the dynamic creation of workers in different work pools is shown in Fig. 6a and b. From Fig. 6a we see that for level = 1, 6 pools of workers were created with their corresponding synchronization points and with 1, 1, 3, 1, 1 and 3 workers on board, respectively. This makes the total number of worker processes for this application equal to 10. For level = 2 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 1. Here, n_p denotes the number of pools, $(n_w)_{\max}$ the maximum number of workers in a pool and $(n_w)_{\text{total}}$ the total number of workers in the application. Note the enormous amount of processes involved in the sparse-grid and the semi-sparse-grid application and the big number of rendezvous created thereby.

The results of our performance measurements for both the sparse- and the semi-sparse-grid approaches are summarized in Fig. 7a and b, which show the elapsed times versus the grid level. All experiments were done during quiet periods of the system, but, as in any real contemporary computing environment, it could not be guaranteed that we were the only user. Furthermore, such unpredictable effects as network traffic and file server delays, etc., could not be eliminated and are reflected in our results. To even out such '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 these experiments are shown in Table 2a and b. In computing the average times given in the table, the best

Table 2
The elapsed times (in hours:minutes:seconds)

	Level	1st time	2nd time	3rd time	4th time	5th time	Average
a. Sparse							
Sequential	1	11.09	11.22	11.23	11.28	13.20	11.24
	2	1:35.54	1:35.87	1:36.56	1:39.82	1:41.22	1:37.42
	3	9:14.00	9:14.73	9:15.53	9:16.42	9:28.44	9:15.56
Parallel	1	5.73	5.78	5.81	5.94	7.02	5.84
	2	33.19	33.25	34.11	34.82	35.58	34.06
	3	2:45.52	2:46.28	2:47.62	2:48.29	2:51.30	2:47.40
b. Semi-sparse							
Sequential	2	50.07	50.26	50.47	50.55	52.20	50.43
	4	17:56.17	17:59.29	18:02.62	18:04.39	18:06.74	18:02.10
	6	4:33:03.59	4:33:07.66	4:37:07.13	4:38:10.83	4:51:59.52	4:36:08.54
Parallel	2	26.47	26.50	27.70	27.80	28.48	27.33
	4	5:42.72	5:53.15	5:59.63	6:03.39	6:12.96	5:58.72
	6	1:13:34.67	1:13:54.51	1:15:04.86	1:15:12.74	1:23:22.07	1:14:44.04

and the worst performances in each row were discarded. In Fig. 7a and b, these average times are depicted versus the grid level. From the results, it clearly appears that the MANIFOLD version takes good advantage of the parallelism offered by the four processors of the machine. The underlying thread facility in our implementation of MANIFOLD on the SGI IRIX operating system allows each thread to run on any available processor. For the sparse-grid and the semi-sparse-grid applications, the MANIFOLD-code times are about 3.25 and 3.75 times smaller, respectively, than the sequential-code times. So, in both cases we have obtained a nearly linear speed-up.

7. Conclusions

One of the promises of sparse-grid techniques, their good parallelization property, has been realized for the computation of a realistic and practically relevant test case from steady gas dynamics. The intrinsically low computational complexity of sparse- and semi-sparse-grid methods, plus the additional gains in computing time through parallelization, make both methods really appealing for very computing-intensive work. (As far as CFD applications are concerned, here one may think of, e.g., direct numerical simulation of turbulence or shape optimization problems.)

Our experiment of using MANIFOLD to restructure existing Fortran code (for a standard 3-D problem from computational aerodynamics) into a parallel application, indicates that this coordination language is well-suited for this kind of work. The highly modular structure of the resulting application and the ability to use existing computational subroutines of the sequential Fortran program are remarkable. The atomic manifold used in the parallel MANIFOLD version only calls C functions which are in fact (wrappers around) Fortran subroutines of the sequential program.

The unique property of MANIFOLD which enables such high degree of modularity is inherited from its underlying IWIM model. 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.

An added bonus of pure coordination modules is their re-usability: the same MANIFOLD modules developed for one application may be used in other parallel applications with the same or similar cooperation protocol, regardless of the fact that the two applications may perform different computations (the sparse-grid and semi-sparse-grid applications use the same protocol manifold, see also Ref. [20] for this notion of re-usability).

The performance evaluation of our test problem shows that MANIFOLD performs very well.

Acknowledgements

The authors want to thank Farhad Arbab for his suggestions to improve this paper.

References

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

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

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

[4] P.W. Hemker, Finite volume multigrid for 3D-problems, in: H. Deconinck, B. Koren (Eds.), Euler and Navier–Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration, Notes on Numerical Fluid Mechanics, 57, Vieweg, Braunschweig, 1997, pp. 393–417.

[5] M. Griebel, C. Zenger, S. Zimmer, Multilevel Gauss–Seidel-algorithms for full and sparse grid problems, Comput. 50 (1993) 127–148.

[6] P.W. Hemker, B. Koren, J. Noordmans, 3D multigrid on partially ordered sets of grids, Proceedings of the Fifth European Multigrid Conference, Birkhäuser, Basel, to appear.

[7] B. Koren, P.W. Hemker, P.M. de Zeeuw, Semi-coarsening in three directions for Euler-flow computations in three dimensions, in: H. Deconinck, B. Koren (Eds.), Euler and Navier–Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration, Notes on Numerical Fluid Mechanics, 57, Vieweg, Braunschweig, 1997, pp. 547–567.

[8] C.T.H. Everaars, F. Arbab, F.J. Burger, Restructuring sequential Fortran code into a parallel/distributed application, Proceedings of the International Conference on Software Maintenance '96, IEEE, 1996, pp. 13–22.

[9] F. Arbab, Coordination of massively concurrent activities, Report CS-R9565, CWI, Amsterdam, 1995. Available on-line at http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.

[10] F. Arbab, The IWIM model for coordination of concurrent activities, in: P. Ciancarini, C. Hankin (Eds.), Coordination Languages and Models, Proceedings of Coordination '96, Lecture Notes in Computer Science, 1061, Springer, Berlin, 1996, pp. 34–56.

[11] S.K. Godunov, Finite difference method for numerical computation of discontinuous solutions of the equations of fluid dynamics, Matematicheskii Sbornik 47 (1959) 271–306, Cornell Aeronautical Lab. Transl. from the Russian.

[12] P.W. Hemker, S.P. Spekreijse, Multiple grid and Osher's scheme for the efficient solution of the steady Euler equations, Appl. Numerical Math. 2 (1986) 475–493.

[13] S. Osher, F. Solomon, Upwind difference schemes for hyperbolic systems of conservation laws, Math. Comput. 38 (1982) 339–374.

[14] P.W. Hemker, C. Pflaum, Approximation on partially ordered sets of regular grids, Report NM-R9611, CWI, Amsterdam (1996).

[15] P.W. Hemker, P.M. de Zeeuw, BASIS3, a data structure for 3-dimensional sparse grids, in: H. Deconinck, B. Koren (Eds.), Euler and Navier–Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration, Notes on Numerical Fluid Mechanics, 57, Vieweg, Braunschweig, 1997, pp. 445–486.

[16] N.H. Naik, J. Van Rosendale, The improved robustness of multigrid elliptic solvers based on multiple semicoarsened grids, SIAM J. Numerical Anal. 30 (1993) 215–229.

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

[18] U. Rüde, Multilevel, extrapolation, and sparse grid methods, in: P.W. Hemker, P. Wesseling (Eds.), Multigrid Methods IV, International Series of Numerical Mathematics, 116, Birkhäuser, Basel, 1994, pp. 281–294.

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

[20] F. Arbab, C.L. Blom, F.J. Burger, C.T.H. Everaars, Reusable coordinator modules for massively concurrent applications, in: L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Proceedings of Euro-Par '96, Lecture Notes in Computer Science, 1123, Springer, Berlin, 1996, pp. 664–677.