Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Using Coordination to Parallelize Sparse-Grid Methods for 3D CFD Problems

C.T.H. Everaars, B. Koren

Software Engineering (SEN)

SEN-R9705 May 31, 1997

Report SEN-R9705
ISSN 1386-369X

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

Using Coordination to Parallelize Sparse-Grid Methods for 3D CFD Problems

C.T.H. Everaars and B. Koren
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Kees.Everaars@cwi.nl and Barry.Koren@cwi.nl

ABSTRACT
The good parallel computing properties of sparse-grid solution techniques are investigated. For this, an existing sequential CFD code for a standard 3D problem from computational aerodynamics is restructured into a parallel application. The restructuring is organized according to a master/slave protocol. The coordinator modules developed thereby are implemented in the coordination language MANIFOLD and are generally applicable. Performance results are given for both the sequential and parallel version of the code. The results are promising, the paper contributes to the state-of-the-art in improving the efficiency of large-scale computations. Also a theoretical analysis is made of speed-up through parallelization in a multi-user single-machine environment.

1991 Computing Reviews Classification System: D.1.3, D.3.2, D.3.3, F.1.2, I.1.3
1991 Mathematics Subject Classification: 65N50, 65N55, 65N99, 76M25, 76N15
Keywords and Phrases: parallel computing, coordination languages, models of communication, multigrid methods, sparse-grid methods, computational fluid dynamics, three-dimensional flow problems.
Note: Work carried out under the projects Coordination Languages (SEN3) and Industrial Processes (MAS2).

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 have to 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, d being the space dimension of the problem ($d = 1, 2$ or 3). So, 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 linearly proportional to the number of unknowns. For all other known solution methods, the amount of work grows faster than linearly proportional to the number of unknowns. For literature on multigrid techniques, see e.g. [6, 10, 21].

Novel multigrid techniques to speed up the solution of systems of discrete equations, are the so-called sparse-grid techniques, see [11] and the further references in there. A sparse-grid technique is very attractive from the viewpoint of computational efficiency, particularly for 3D problems. The gain in efficiency is achieved by a strong reduction of the numbers 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 [9] than a standard multigrid method (which latter performs much better in this sense already than a single-grid method).

Further efficiency improvement of sparse-grid methods is still possible; 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, 3D Euler equations of gas dynamics [12, 17]. Our starting point forms 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 [7]. 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/slave 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 [1, 3].

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 3D computational fluid dynamics (CFD) code and pay attention to its good parallel computing properties. In Section 5, we give a brief introduction to the MANIFOLD language. Next, in Section 6 we show how we can restructure the sequential 3D software discussed in Section 4 into a parallel code by using the coordination language MANIFOLD. In Section 7, we give an analysis of speed-up numbers in a 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 at a far-field Mach number of 0.84 and 3.06^o angle of attack). Finally, the conclusion of the paper is in Section 8.

2 Equations

2.1 Continuous equations

The steady, 3-D Euler equations are written as

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

with q the state vector

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

$f(q)$, $g(q)$ and $h(q)$ the flux vectors

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

and with e the sum of internal and kinetic energy, satisfying the perfect-gas relation

$$e = \frac{1}{\gamma - 1}\frac{p}{\rho} + \frac{1}{2}\left(u^2 + v^2 + w^2\right). \tag{1d}$$

2.2 Discretized equations

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 $\partial\Omega^\star$. A straightforward and simple discretization is obtained by subdividing the entire computational domain Ω, in a structured manner, into disjunct, non-overlapping subdomains $\Omega_{i,j,k}$, $i = 0, 1, \ldots, i_{\max}$, $j = 0, 1, \ldots, j_{\max}$, $k = 0, 1, \ldots, k_{\max}$ (finite volumes) and by requiring that

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

Using the rotational invariance of the Euler equations

$$f(q)n_x + g(q)n_y + h(q)n_z = T^{-1}(\theta,\phi) f(T(\theta,\phi)q), \tag{4}$$

where $T(\theta,\phi)$ is the rotation matrix

$$T(\theta,\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta\cos\phi & \sin\theta\sin\phi & 0 \\ 0 & -\sin\theta & \cos\theta\cos\phi & \cos\theta\sin\phi & 0 \\ 0 & 0 & -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \tag{5a}$$

$$\theta \equiv \frac{n_x}{\sqrt{n_x^2 + n_y^2 + n_z^2}}, \quad \phi \equiv \frac{n_y}{\sqrt{n_y^2 + n_z^2}}, \tag{5b}$$

(3) can be rewritten as

$$\oint_{\partial\Omega_{i,j,k}} T^{-1}(\theta,\phi) f(T(\theta,\phi)q)\mathrm{d}s = 0, \quad \forall i, j, k. \tag{6}$$

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 Godunov approach [8], along each cell face $\partial\Omega_{i,j,k}$, as in 2-D, 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 [14] of Osher's approximate Riemann solver [19]. For the left and right cell-face states, we take the first-order accurate approximations

$$\begin{pmatrix} q^l_{i+\frac{1}{2},j,k} \\ q^r_{i+\frac{1}{2},j,k} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i+1,j,k} \end{pmatrix}, \quad \begin{pmatrix} q^l_{i,j+\frac{1}{2},k} \\ q^r_{i,j+\frac{1}{2},k} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i,j+1,k} \end{pmatrix}, \quad \begin{pmatrix} q^l_{i,j,k+\frac{1}{2}} \\ q^r_{i,j,k+\frac{1}{2}} \end{pmatrix} = \begin{pmatrix} q_{i,j,k} \\ q_{i,j,k+1} \end{pmatrix}. \tag{7}$$

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

A significant difficulty of standard multigrid methods for 3-D problems, when compared to application to 2-D problems, is that the requirements to be imposed on the smoother are much more severe. When cells are used as grid elements, 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 satisfactory 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 numerical 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.

3.1 Standard multigrid

In this section we first describe the standard 3-D multigrid algorithm. We use the 3-D generalization of the optimal 2-D multigrid approach, that was originally described in [14].

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, the differential operator applied being $(\frac{\partial}{\partial u}, \frac{\partial}{\partial v}, \frac{\partial}{\partial w}, \frac{\partial}{\partial c}, \frac{\partial}{\partial z})^T$, where $c \equiv \sqrt{\gamma \frac{p}{\rho}}$, $z \equiv \ln\left(\frac{p}{\rho^\gamma}\right)$. This relaxation method is simple and robust.

As the standard multigrid method we apply the nonlinear version (FAS), preceded by nested iteration (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 iterate 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 application of 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 the application of 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 application of 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 by

$$\begin{aligned}(r_{\lambda-1})_{i,j,k} = (I_\lambda^{\lambda-1} r_\lambda)_{i,j,k} \quad \equiv \quad & (r_\lambda)_{2i,2j,2k} + \\ & (r_\lambda)_{2i-1,2j,2k} + (r_\lambda)_{2i,2j-1,2k} + (r_\lambda)_{2i,2j,2k-1} + \\ & (r_\lambda)_{2i-1,2j-1,2k} + (r_\lambda)_{2i-1,2j,2k-1} + (r_\lambda)_{2i,2j-1,2k-1} + \\ & (r_\lambda)_{2i-1,2j-1,2k-1}, \end{aligned} \tag{8a}$$

$$\begin{aligned}(\tilde{I}_{\lambda-1}^{\lambda} q_{\lambda-1})_{2i,2j,2k} = \\ (\tilde{I}_{\lambda-1}^{\lambda} q_{\lambda-1})_{2i-1,2j,2k} = (\tilde{I}_{\lambda-1}^{\lambda} q_{\lambda-1})_{2i,2j-1,2k} = (\tilde{I}_{\lambda-1}^{\lambda} q_{\lambda-1})_{2i,2j,2k-1} = \\ (\tilde{I}_{\lambda-1}^{\lambda} q_{\lambda-1})_{2i-1,2j-1,2k} = (\tilde{I}_{\lambda-1}^{\lambda} q_{\lambda-1})_{2i-1,2j,2k-1} = (\tilde{I}_{\lambda-1}^{\lambda} q_{\lambda-1})_{2i,2j-1,2k-1} = \\ (\tilde{I}_{\lambda-1}^{\lambda} q_{\lambda-1})_{2i-1,2j-1,2k-1} = \\ \equiv (q_{\lambda-1})_{i,j,k}. \end{aligned} \tag{8b}$$

3.2 Multiple semi-coarsened multigrid

Also in the case of the semi-coarsened multigrid method we 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

$\Omega_\lambda, \lambda = 0, \ldots, \lambda_{\max}$, 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 $l + m + n$ is called the level of grid $\Omega_{l,m,n}$. The nesting and the semi-coarsening relation between these grids is described in [13, 15].

Also here, nested iteration (FMG) 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, $\Omega_{0,0,0}$, which 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, with the 3-D prolongation according to formula (29) in [11] (see Appendix A in [17] 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. 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 on level $l + m + n$ is recurrently defined by the following steps:

1. Improve the solutions on level $l + m + n$ by the application of 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 standard multigrid, but use another restriction operator, viz. the one described in Appendix B of [17]. (The restriction of defects is still natural, i.e. by summation over all sub-cells.)

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

4. Correct the current solutions on 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 [18]. (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 [11]. (It is the correction-prolongation version of the solution prolongation described in Appendix A of [17], it uses fixed prolongation weights.) In Appendix C of [17], both correction prolongations are described explicitly.

5. Improve the solutions on level $l + m + n$ by the application of 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. 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 3D, 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 anymore, accurate approximations can be constructed by extrapolation [11, 16, 20]. 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 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 1D problem only! Theoretically, the sparse-grid-of-grids approach has the best ratio of discrete accuracy over number of grid points used [9]. 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 [16]. 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 2D problem only.

4 The 3D 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. In general, from

a given sequential algorithm, it becomes quickly clear which parts can run in parallel. The 3D 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 total 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 [21], p. 171 and further) 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,lmax,
     +                 nmin,mmin,lmin,nmax,mmax,lmax
      logical         convergence
      external        fas,prolsolgr,scanlv
        common /gridset/ nmin,mmin,lmin,nmax,mmax,lmax

cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c     main program
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

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

ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c     subroutine for nonlinear multigrid iteration
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      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
```

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 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 total program, is used in the relaxations, parallel implementation is expected to pay off. This will be worked out in Section 6.

5 The Manifold coordination language

In this section, we briefly introduce MANIFOLD: a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes [1]. MANIFOLD is based on the IWIM model of communication [3]. The basic concepts in the IWIM model are *processes*, *events*, *ports* and *channels* (Sections 5.1 through 5.3).

A MANIFOLD application consists of a (potentially very large) number of processes running 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 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, a run-time system library, a number of utility programs, libraries of built-in and pre-defined processes, a link file generator called MLINK and a run-time configurator called CONFIG. The system has been ported to several different platforms (e.g. SGI 5.3, SUN 4, Solaris 5.2, and IBM SP/1). MLINK uses the object files produced by the (MANIFOLD and other language) compilers to produce link files needed to compose the executable files for each required platform. At the run time of an application, CONFIG determines the actual host(s), where the processes (created in the MANIFOLD application) will run.

The library routines that comprise the interface between MANIFOLD and processes written in other languages (e.g. C), automatically perform the necessary data format conversions when data are routed between various different machines.

5.1 Processes

In MANIFOLD, the atomic workers of the IWIM model are called atomic processes. Any operating system-level process can be used as an atomic process in MANIFOLD. However, MANIFOLD also provides a library of functions that can be called from a regular C function running as an atomic process, to support a more appropriate interface between the atomic processes and the MANIFOLD world. Atomic processes can only produce and consume units through their ports, generate and receive events, and compute. In this way, the desired separation of computation and coordination is achieved.

Coordination processes are written in the MANIFOLD language and are called manifolds. The MANIFOLD language is a block-structured, declarative, event-driven language. A manifold definition consists of a header and a body. The header of a manifold gives its name, the number and types of its parameters, and the names of its input and output ports. The body of a manifold definition is a block. A block consists of a finite number of states. Each state has a label and a body. The label of a state defines the 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. The body of a simple state defines the set of actions that are to be performed upon transition to that state. The body of a compound state is either a (nested) block, or a call to a parameterized subprogram known as a *manner* in MANIFOLD. A manner consists of a header and a body. As for the subprograms in other languages, the header of a manner essentially defines its name and the types and the number of its parameters. A manner is either atomic or regular. The body of a regular manner is a block. The body of an atomic manner is a C function that can interface with the MANIFOLD world through the same interface library as for the compliant atomic processes.

5.2 Streams

All communication in MANIFOLD is asynchronous. In MANIFOLD, the asynchronous IWIM channels are called streams. A stream is a communication link that transports a sequence of bits, grouped into (variable length) *units*.

A stream represents a reliable and directed flow of information from its *source* to its *sink*. As in the IWIM model, the constructor of a stream between two processes is, in general, a third process. Once a stream is established between a producer process and a consumer process, it operates autonomously and transfers the units from its source to its sink. The sink of a stream requiring a unit is suspended only if no units are available in the stream. The suspended sink is resumed as soon as the next unit becomes available for its consumption. The source of a stream is never suspended because the infinite buffer capacity of a stream is never filled.

There are four basic stream types designated as `BB`, `BK`, `KB`, and `KK`, each behaving according to a slightly different protocol with regards to its automatic disconnection from its source or sink. Furthermore, in MANIFOLD, the `BK` and `KB` type streams can be declared to be *reconnectable*. See [3] for details.

5.3 Events and state transitions

In MANIFOLD, once an event is *raised* by a process, it continues with its processing, while the event occurrence propagates through the environment independently. Any receiver process that is interested in such an event occurrence will automatically receive it in its *event memory*. The observed event occurrences in the event memory of a process can be examined and reacted on by this process at its own leisure. In reaction to such an event occurrence, the observer process can make a transition from one labeled state to another.

The only control structure in the MANIFOLD language is an event-driven state transition mechanism. More familiar control structures, such as the sequential flow of control represented by the connective ";" (as in Pascal and C), conditional (i.e., "if") constructs, and loop constructs can be built out of this event mechanism, and are also available in the MANIFOLD language as convenience features.

Upon transition to a state, the primitive actions specified in its body are performed atomically in some non-deterministic order. Then, the state becomes *pre-emptable*: if the conditions for transition to

another state are satisfied, the current state is pre-empted, meaning that all streams that have been constructed are dismantled and a transition to a new state takes place. The most important primitive actions in a simple state body are (i) creating and activating processes, (ii) generating event occurrences, and (iii) connecting streams to the ports of various processes.

5.4 An example program

Consider a simple program to print a message on the standard output. The MANIFOLD source file for this program contains the following:

```
 1 manifold printunits import.
 2
 3 auto process print is printunits
 4
 5 manifold Main
 6 {
 7   begin: "Sparse grids applied to 3D CFD equations \
 9           will save you waiting frustrations. \
 9           Additional speed-up through Manifold \
10           will make your CFD-code, going for gold." -> print.
11 }
```

The first line of this code defines a manifold named `printunits` that takes no arguments, and states (through the keyword `import`) that the real definition of its body is contained in another source file. This defines the "interface" to a process type definition, whose actual "implementation" is given elsewhere. Whether the actual implementation of this process is an atomic process (e.g., a C function) or it is itself another manifold is indeed irrelevant in this source file. We assume that `printunits` waits to receive units through its standard input port and prints them. When `printunits` detects that there are no incoming streams left connected to its input port, and that it has printed the units it has received, it terminates.

The second line of code defines a new instance of the manifold `printunits`, calls it `print`, 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, this is the end of the application. Because the declaration of the process instance `print` appears outside of any blocks in this source file, it is a global process, known by every instance of every manifold whose body is defined in this source file.

The last lines of this code define a manifold named `Main`, which takes no parameters. Every manifold definition (and therefore every process instance) always has at least three default ports: `input`, `output` and `error`. The definition of these ports are not shown in this example, but for `Main` the ports are defined by default.

The body of this manifold is a block (enclosed in a pair of braces) and contains only a single state. The name `Main` is indeed special in MANIFOLD: there must be a manifold with that name in every MANIFOLD application and an automatically created instance of this manifold, called `main`, is the first process that is started up in an application. Activation of a manifold instance automatically posts an occurrence of the special event `begin` in the event memory of that process instance; in this case `main`. This makes the initial state transition possible: `main` enters its only state – the `begin` state.

The `begin` state contains only a single primitive action, represented by the stream construction symbol "→". Entering this state, `main` creates a stream instance (of the default `BK`-type) and connects the `output` port of the process instance on the left-hand side of the → to the `input` port of the process instance on its right-hand side. The process instance on the right-hand side of the → is `print`. What appears to be a character string constant on the left-hand side of the → is also a process instance: conceptually, a constant in MANIFOLD is a special process instance that produces its value as a unit on its `output` port and then dies.

Having made the stream connection between the two processes, `main` now waits for all stream connections made in this state to break up (on one of their ends). The stream breaks up, in this case on its source end, as soon as the string constant delivers its unit to the stream and dies. Since there are no other event occurrences in the event memory of `main`, the default transition for a state reaching its end (i.e., falling over its terminator period) now terminates the process `main`.

Meanwhile, `print` reads the unit and prints it. The stream type `BK` ensures that the connection between the stream and its sink is preserved even after a pre-emption, or its disconnection from its source. Once the stream is empty and disconnected from its source, it automatically disconnects from its sink. Now, `print` senses that it has no more incoming streams and dies. At this point, there are no other process instances left and the application terminates.

Note that our simple example consists of three process instances: two worker processes (a character string constant and `print`) and a coordinator process (`main`). Also note that the coordinator process

`main` only establishes the connection between the two worker processes. It does *not* transfer the units through the stream(s) it creates, nor does it interfere with the activities of the worker processes in other ways.

6 Restructuring the 3D CFD code

In this section we describe the restructuring of the Fortran code presented in the previous section into a parallel application. 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 it produces must be collected. The obvious way to accomplish this is to arrange for the MANIFOLD coordinators that the (proper segments of the) global space are sent and received 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 7). 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.

6.1 The master/slave protocol

The restructuring of the Fortran code can be described in a kind of master/slave protocol. In a coordinator process (which is an instance of a protocol manifold named `ProtocolMS`) 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 wants to delegate the computations done in `pointgsgr` to a separate slave process, for each single grid visited. Each time the master needs a slave, it raises an event to signal the coordinator to create the slave. In this way a pool of slaves is working for the master, each slave performing the computations embodied in `pointgsgr`. The coordinator makes the identification of the slave known to the master `oneram6` by sending a reference of it to the master. With this information the master can activate the slave. Before the slave can really work, it should know on which grid (identified by the grid-of-grid coordinates n, m, l) it should perform the relaxation. The master has these coordinates available and writes them on its own output port. The coordinator takes care that the slave 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 makes again a request for the creation of a slave process. When all the slaves are created and activated in this way, the master waits until the slaves are ready with the relaxation and are going to die. After this rendezvous, the master `oneram6` proceeds its sequential work until it again arrives at a point where it wants to use a pool of slaves to delegate the relaxations to.

6.2 The manifold code of the protocol and its parameters

In MANIFOLD we can easily realize the master/slave protocol described in Section 6.1 in a general way in which the master and slave are parameters of the protocol. In this protocol we only describe how instances of the master and slave 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 slave. What is indeed important for the protocol is that the in/output and the event behavior of the master and slave are tuned to the protocol. E.g., the protocol manifold is only able to create a slave when the master requests for its creation by raising an event. Also, the master should write the data needed by the slave, on its own output port and the slave should read this information from its own input port, etc.

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

```
 1 // protocolMS.m
 2
 3 #define IDLE terminated(void)
 4
 5 export manifold ProtocolMS(manifold Master, manifold Slave, event create_slave, event ready)
 6 {
 7   auto process master is Master.
 8
 9   begin:  (master, IDLE).
10
11   create_slave: {
12     process slave is Slave.
13
14     begin: (&slave -> master -> slave, 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 slaves and charge each with a computational job. Do this as follows:
   (a) Request a coordinator process (which is an instance of a protocol manifold named `ProtocolMS`) to create a slave 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 slave process.
   (c) Now the master knows the identification of the slave, he can activate him.
   (d) Write the information, which the slave needs to do this job, on the output port. (The coordinator takes care that the slave can read this information.)
   (e) Repeat steps a,b,c,d for each slave that is needed. (In this way a pool of slaves is comprised.)
   (f) Wait until all slaves in the pool are going to die (rendezvous).
3. Repeat 1 and 2 as much as needed and raise an event to signal the coordinator process that the master is ready.

The behavior interface of the slave 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.

Let us now give a MANIFOLD description of 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 `ProtocolMS` that takes four arguments and states (through the keyword `export`). This manifold can be used in other source files that import this MANIFOLD definition (as we will see).

The first two arguments of `ProtocolMS` are the master and the slave manifold, respectively. With these two parameters, `ProtocolMS` is independent of a particular master or slave, as long as they abide by the pre-described behavior interface. The third argument, with formal name `create_slave`, is an event the master raises to signal the `ProtocolMS` to create a slave. The fourth argument is an event, with formal name `ready`, which the master manifold raises to signal the `ProtocolMS` 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 manifold `ProtocolMS` is a block, i.e. some lines of software in between "{" and "}", containing at least the `begin` state. The present block has three states: the `begin`, `create_slave` and `ready` state (lines 9,11 and 17). Activation of a manifold instance of `ProtocolMS` automatically posts an occurrence of the special event `begin` in the event memory of that process instance, in this case `main`. This makes the initial state transition to the `begin` 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 `ProtocolMS` 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_slave` as soon as it wants a slave to delegate some work to. This event pre-empts the `begin` state and makes a state transition possible: the instance of `ProtocolMS` enters its second state – the `create_slave` state (lines 11-15). In this state we create a process instance named `slave` of manifold `Slave`. 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 at the lines 11 and 15, the process creation takes place at line 12.

In the `begin` state of this block the stream configuration at line 14 is constructed and we wait for events (due to the word IDLE) from the master (`create_slave` and `ready` are possible events). In the stream configuration we see that the process identification of the slave (denoted by `&slave`) is sent through a stream (the first $\rightarrow$ at line 14) to the already active master, which sends the information the slave needs to do its job, through a stream (the second $\rightarrow$ at line 14) to the slave. When the master is again at a point where it needs a slave it again raises the `create_slave` event, the `create_slave` state is pre-empted and again we get a state transition to the `create_slave` state. In this way all slaves are created and activated. Note that `ProtocolMS` knows nothing about the work pools in which the slaves 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 switch to the `ready` state at line 17, in which the primitive action `halt` effectively terminates the `ProtocolMS` instance.

6.3 A "protocol" library

It is good practice to compile manifolds that embody general applicable coordinators (such as our `ProtocolMS`) separately and to archive them in what we can call a "protocol" library. If we want to use e.g. `ProtocolMS` we retrieve it from this library and use as actual parameters for the protocol a user-supplied master and slave manifold that behaves 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 slave manifolds) and coordination modules (`ProtocolMS`). MANIFOLD, which is a pure coordination language that encourages this separation in computation and communication concerns, is a perfect language to implement such "protocol" libraries [4, 5].

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 it is bigger and red.

The subroutine `scanlv` which is used in the sequential version of our CFD application, has as the eighth parameter the subroutine `tkgrid`. This subroutine has three integer parameters, specifying the grid to be visited. Each user-supplied subroutine which is meant to be an actual parameter for the formal parameter `tkgrid`, should abide by the prescribed functionality. Note that `scanlv` knows nothing about what kind of computations it performs. It just visits grids and calls the actual subroutine it receives as a parameter on those grids.

In the next section we describe the actual parameters which we use for the formal parameters in `ProtocolMS`.

6.4 The actual master and slave manifold

The master and the slave manifold are easy to implement as atomic processes written in C. The only changes we make are in the program `oneram6` and in the subroutine `fas` (Section 4). The changes are the following:

- At 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, at 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 slave manifold are listed below in `model.ato.c`, the file in which we code all the atomic processes and auxiliaries.

```
 1 /* model.ato.c */
 2
 3 #include "AP_interface.h"
 4 #include "debug.h"
 5
 6 AP_Event cp, fin;
 7
 8 /*****************************************************************/
 9 void w_oneram6(void)
10 {
11   int err;
12
13   extern void oneram6_(void);
14
15   cp = AP_AllocateEvent();                                    P(cp)
16   err = AP_InitHeaderEvent(cp, "create_pointgsgr");           I(err)
17
18   fin = AP_AllocateEvent();                                   P(fin)
19   err = AP_InitHeaderEvent(fin, "finished");                  I(err)
20
21   oneram6_();
22 }
23
24 /*****************************************************************/
25 void concurrent_(int *pilevel, int *pnmin, int *pnmax, int *pmmin, int *pmmax, int *plmin, int *plmax)
26 {
27   AP_Process p = AP_AllocateProcess();
28   int input = AP_PortIndex("input");
29   int output = AP_PortIndex("output");
30   AP_Unit u;
31   AP_Event r = AP_AllocateEvent();
32   AP_EventPatternSet eps = AP_AllocateEventPatternSet();
33   AP_Process q = AP_AllocateProcess();
34   int err, i;
35   int ar[3];
36   int nos = 0;
37   int ilevel = *pilevel, nmin = *pnmin, nmax = *pnmax,
38                          mmin = *pmmin, mmax = *pmmax,
39                          lmin = *plmin, lmax = *plmax;
40   int n, m, l;
41                                                               P(p)
42                                                               I(input)
43                                                               I(output)
44                                                               P(r)
45                                                               P(eps)
46                                                               P(q)
47
48   for (n = nmin; n <= nmax; n++) {
49     for (m = mmin; m <= mmax; m++) {
50       l = ilevel - m - n;
51       if ((l <= lmax) && (l >= lmin) ) {
52
53         err = AP_Raise(cp);                                   I(err)
54
55         err = AP_PortRemoveUnit(input, &u, NULL);             I(err) P(u)
56         err = AP_DerefProcess(p, u, NULL, NULL);              I(err)
57         err = AP_Activate(p);                                 I(err)
58
59         ar[0] = n; ar[1] = m; ar[2] = l;
60         u = AP_FrameIntegerArray((int *) ar, 3);
61         err = AP_PortPlaceUnit(output, u, NULL);              I(err)
62
63         nos++;
64
65         err = AP_EventPatternSetInsert(eps, AP_death, p);     I(err)
66       }
67     }
68   }
69
70   for (i = 1; i <= nos; i++) {
71     err = AP_DeleteWaitEvent(eps, r, q);                      I(err)
72   }
73 }
74
75 /*****************************************************************/
76 void raise_it_(void)
77 {
78   int err;
79
80   err = AP_Raise(fin);                                        I(err)
81 }
```

```
82
83 /*****************************************************************/
84 void w_pointgsgr(void)
85 {
86   int input = AP_PortIndex("input");
87   int err;
88   AP_Unit u;
89   int ar[3];
90   int n, m, l;
91
92   extern void pointgsgr_(int* n, int* m, int* l);
93
94   err = AP_PortRemoveUnit(input, &u, NULL);                    I(err) P(u)
95   err = AP_FetchIntegerArray(u, ar, 3);                        I(err)
96   err = AP_DeallocateUnit(u);                                  I(err)
97   n = ar[0]; m = ar[1]; l = ar[2];
98   pointgsgr_(&n, &m, &l);
99 }
```

The source code of the master is made in the following way. 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` at this line. Here we make use of 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 a lot of C functions, which allows access to the MANIFOLD world. All function calls in file `model.ato.c` starting with `AP_` refer to functions from this library.

As already known, the master needs two events, one to request for using a slave, and one to signal `ProtocolMS` when it is ready with its work. Therefore we have at line 6 in `model.ato.c` two global events, named `cp` and `fin`, that correspond with this. With the `AP_` calls at lines 15 and 18 we allocate memory for these events. At 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 at 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 created a pool of slaves (lines 48-68) and introduced a synchronization point by waiting until all the slaves are going to die (rendezvous). All the grids to be visited in `scanlv` are specified at lines 11-13 in subroutine `scanlv` (Section 4). In the master manifold these grids are specified at 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 `ProtocolMS` to create a slave, and waits (line 55) for the availability of a unit `u` (sent by `ProtocolMS` at line 14 of file `protocolMS.m`) at the input port of the master (set at line 28). This unit contains the identification of a slave process. At line 56 we read the process identification of the slave process from this unit and activate the slave. At that moment the first slave is in the pool and other will follow soon, as is specified in the loop structure at lines 48-68.

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

Because we want a synchronization point where we wait until all slaves in the pool are going to die, we have to count the number of slaves (denoted by `nos`, line 63) and we add the death event (`AP_death`) of the slave process (`p`) (an event together with its broadcaster is called an "event pattern") to what is called an "event pattern set" (`eps`). When the loop structure ends at line 68, the event pattern set `eps` contains the death events `AP_death` from all slaves. This set is used to create the rendezvous.

When a slave 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 slave), but is a part of the termination protocol of every manifold.

With the call at 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 slaves (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 ready with its job, which is signalled to `protocolMS` by calling the new routine `raise_it` (lines 76-81, recall the second change we made in `oneram6`). At line 80 the event `fin` is raised.

The slave manifold (lines 84-99) is also implemented as a C wrapper, this time around the Fortran subroutine `pointgsgr`, callable from C as `pointgsgr_`. At line 94 the slave waits, if necessary, for the availability of a unit through its input port (set at line 86) that contains information about the grid to be visited (three integers). At line 95 this information is assigned to an array `ar`. Unit `u` is deallocated at line 96 because it is not needed anymore 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 slave constructed in this way fully satisfy the behavior interface of the master/slave protocol given in Section 6.1.

6.5 The actual Manifold program

Using the manifold `ProtcolMS` together with the two actual parameters as described in the previous section, 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 Slave, event create_slave, event ready) import.
10
11 /*****************************************************************/
12 manifold Main
13 {
14   begin: ProtocolMS(w_oneram6, w_pointgsgr, create_pointgsgr, finished).
15 }
```

At 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, it is a global event, known in the whole source file.

Line 5 defines the slave 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 a programming language such as C, C^{++}, or Fortran.

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 take up two events between the brackets at this line.

Line 9 defines the manifold `protocolMS` which has four parameters: the master and slave manifold 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 `protocolMS` is created and activated (this is done implicitly, by using the manifold name `protocolMS`). After this, the process instance of `Main` terminates and the instances of `protocolMS` and `w_oneram6` (with all the slaves `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.

7 Performance analysis

7.1 Speed-up analysis

All experiments have been 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 these 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 these,
- 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 our own application consists of ($m_1 \geq 1$; $m_1 = 1$ representing the sequential application) and m_2 the number of processes from other users ($m_2 \geq 0$), we can write as expression for the investment of CPU power p in our own application:

$$p = \begin{cases} n\frac{m_1}{m_1+m_2}, & \text{if } m_1 + m_2 > n \\ m_1, & \text{if } m_1 + m_2 \leq n \end{cases} . \tag{9}$$

With the investment of CPU power inversely proportional to the elapsed computing times needed, from (9), expressions for various speed-up factors can be derived. As examples, we look at two of these factors.

The first speed-up factor relates the computing times of our parallel application run on a multi-processor machine ($n > 1, m_1 > 1$), to those of the sequential version run on a single-processor from that machine ($n = 1, m_1 = 1$), both runs with the same number m_2 of other processes. For the corresponding speed-up factor, to be denoted by s_{n,m_1}, it follows

$$s_{n,m_1} \equiv \frac{p(n > 1, m_1 > 1, m_2)}{p(n = 1, m_1 = 1, m_2)} = \begin{cases} n(1+m_2)\frac{m_1}{m_1+m_2}, & \text{if } m_1 + m_2 > n \\ m_1(1+m_2), & \text{if } m_1 + m_2 \leq n \end{cases} . \tag{10}$$

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 for the case $m_1 + m_2 > n$, and *is* always larger than the number of processes m_1 for the case $m_1 + m_2 \leq n$. In Figure 3, distributions of s_{n,m_1} are depicted for a 4-,8- and 16-processor machine, 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 only meant to help in recognizing the discrete speed-up patterns.)

The second speed-up factor to be considered relates the computing times of our application when run on a machine with other processes running simultaneously, to those run on the same machine, but with no other processes. For the corresponding speed-up factor, to be denoted by s_{m_2}, it follows

$$s_{m_2} \equiv \frac{p(n, m_1, m_2 = 0)}{p(n, m_1, m_2 > 0)} = \begin{cases} \frac{m_1+m_2}{m_1}, & \text{if } m_1 > n \\ \frac{m_1+m_2}{n}, & \text{if } m_1 \leq n \text{ and } m_1 + m_2 > n \\ 1, & \text{if } m_1 + m_2 \leq n \end{cases} . \tag{11}$$

In Figure 4, distributions of s_{m_2} are depicted for a 4-,8- and 16-processor machine, respectively. Formula (11) may be practically relevant in comparative studies. With (11), 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 (11) and with the real (elapsed) time $t_{\text{real}}(m_2 > 0)$ measured, we may estimate the corresponding elapsed time in a single-user ($m_2 = 0$) environment by

$$t(m_2 = 0) = s_{m_2}\ t_{\text{real}}(m_2 > 0). \tag{12}$$

All our experiments have been done 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.

7.2 Performance results

All experiments have been 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 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 have been 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, these are: 2,4 and 6.

For both approaches, the dynamic creation of slaves in different work pools is shown in Figures 5a and 5b. From Figure 5a we see that for level=1, 6 pools of slaves have been created with their corresponding synchronization points and with 1,1,3,1,1 and 3 slaves on board, respectively. This makes the total number of slave processes for this application equal to 10. For level=2 there are 18 pools with

a total of 50 slaves, and for level=3 these numbers are 42 and 170, respectively. For both the sparse- and the semi-sparse-grid application, the numbers are summarized in Table 1. In here, n_p denotes the number of pools, $(n_s)_{\max}$ the maximum number of slaves in a pool and $(n_s)_{\text{total}}$ the total number of slaves 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. Between the rendezvous, the application runs in a sequential way.

The results of our performance measurements for both the sparse- and the semi-sparse-grid approach are summarized in Figures 6a and 6b, which show the elapsed times versus the grid level. All experiments have been run during quiet periods of the system, but, as in any real contemporary computing environment, it could not be guaranteed that we were the only users. 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 have run 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 the Tables 2a and 2b. In computing the average times given in both tables, the best and the worst performances in each row were discarded. In Figures 6a and 6b, these average times have been 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 application, 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.

8 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 by parallelization, make both methods really challenging 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 an existing Fortran code (for a standard 3D 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 application use the same protocol manifold, see also [5] for this notion of re-usability).

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

References

[1] 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.

[2] F. Arbab, Manifold version 2: Language reference manual. Report (preliminary version) CWI, Amsterdam, 1995.

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

[4] F. Arbab, The influence of coordination on program structure, in: *Proceedings of the 30th Hawaii International Conference on System Sciences* (IEEE, 1997).

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

[6] W.L. Briggs, *A Multigrid Tutorial* (SIAM, Philadelphia, 1987).

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

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

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

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

[11] P.W. Hemker, Finite volume multigrid for 3D-problems, in: H. Deconinck and 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) 393–417.

[12] P.W. Hemker, B. Koren and J. Noordmans, 3D multigrid on partially ordered sets of grids, in: *Proceedings of the Fifth European Multigrid Conference* (Birkhäuser, Basel, to appear).

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

[14] 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** (1986) 475-493.

[15] P.W. Hemker and P.M. de Zeeuw, BASIS3, a data structure for 3-dimensional sparse grids, in: H. Deconinck and 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) 445–486.

[16] B. Koren, P.W. Hemker and C.T.H. Everaars, Multiple semi-coarsened multigrid for 3D CFD, *AIAA-paper* 97-2029 (1997).

[17] 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, eds., *Euler and Navier-Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration, Notes on Numerical Fluid Mechanics*, **57** (Vieweg, Braunschweig, 1997) 547–567.

[18] 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** (1993) 215-229.

[19] S. Osher and F. Solomon, Upwind difference schemes for hyperbolic systems of conservation laws, *Mathematics of Computation*, **38** (1982) 339-374.

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

[21] P. Wesseling, *An Introduction to Multigrid Methods*, (Wiley, Chichester, 1992).

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 1: Work pool and slave statistics.

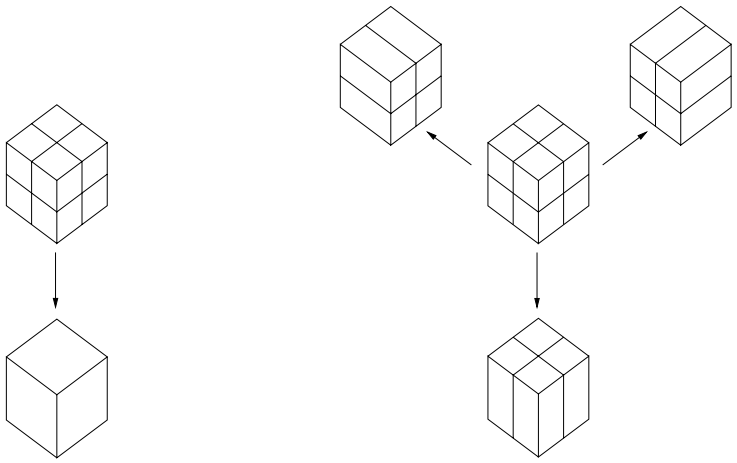
	level	1*st* time	2*nd* time	3*rd* time	4*th* time	5*th* time	average
sequential	1	11.09	11.22	11.23	11.28	13.20	11.24
	2	1:35.54	1:35.87	1:36.56	1:39.82	1:41.22	1:37.42
	3	9:14.00	9:14.73	9:15.53	9:16.42	9:28.44	9:15.56
parallel	1	5.73	5.78	5.81	5.94	7.02	5.84
	2	33.19	33.25	34.11	34.82	35.58	34.06
	3	2:45.52	2:46.28	2:47.62	2:48.29	2:51.30	2:47.40

a. Sparse.

	level	1*st* time	2*nd* time	3*rd* time	4*th* time	5*th* time	average
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

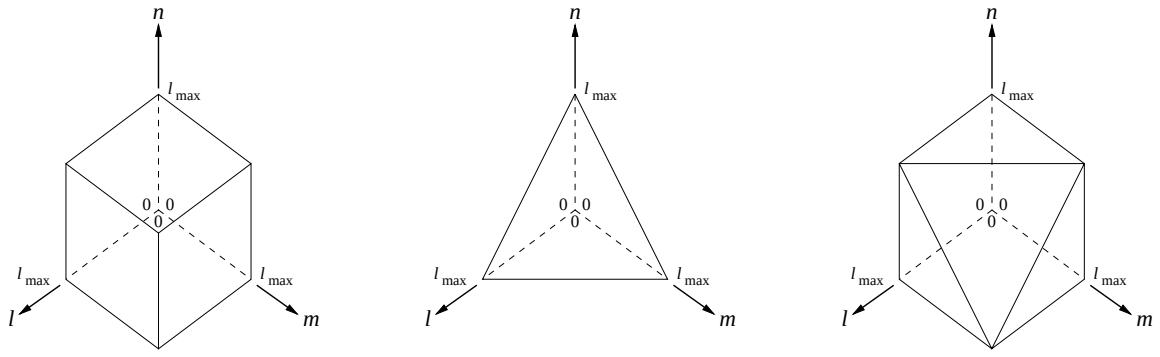
b. Semi-sparse.

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



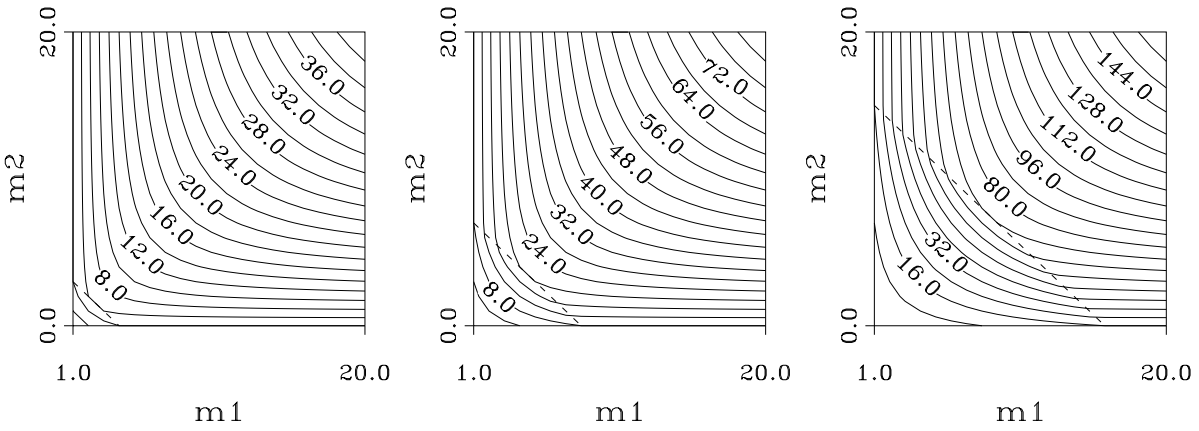
a. Full coarsening. b. Multiple semi-coarsening.

Figure 1: Two types of 3D coarsenings.



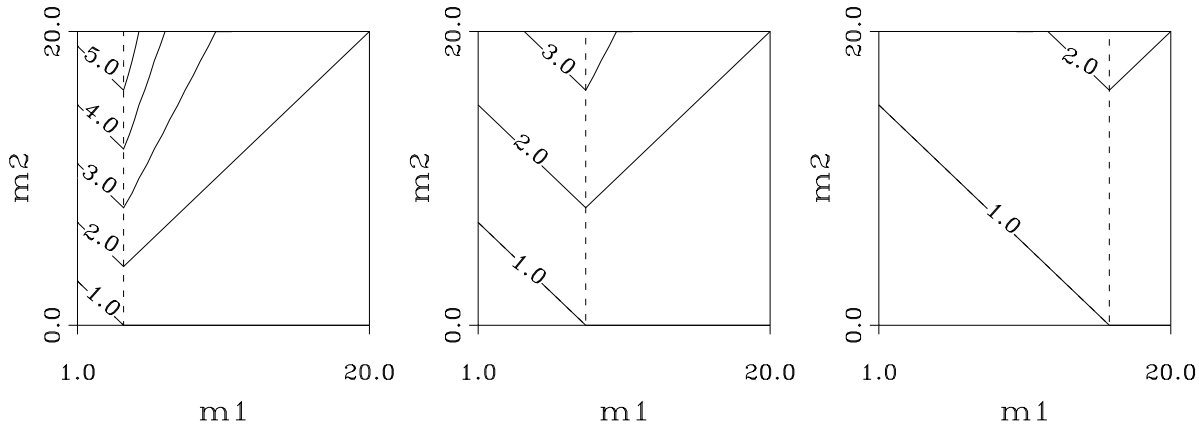
a. Full. b. Sparse. c. Semi-sparse.

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



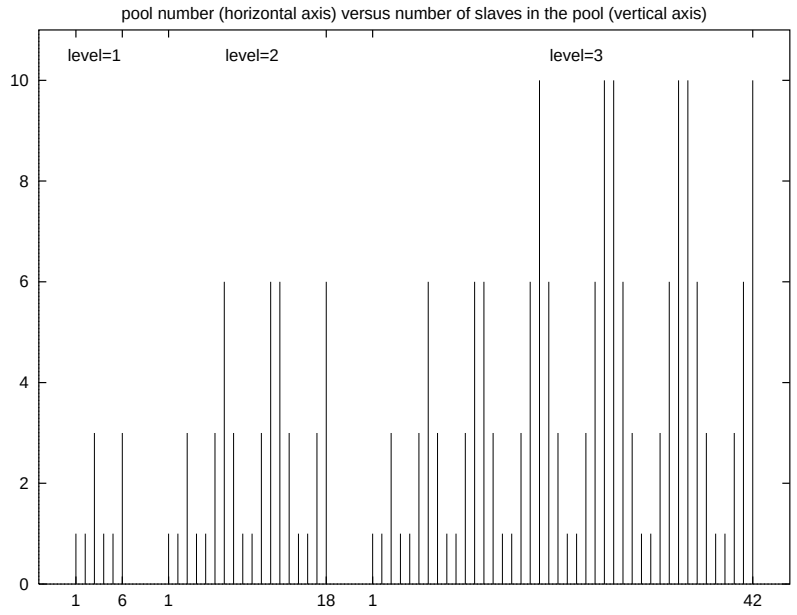
a. $n = 4$. b. $n = 8$. c. $n = 16$.

Figure 3: 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.

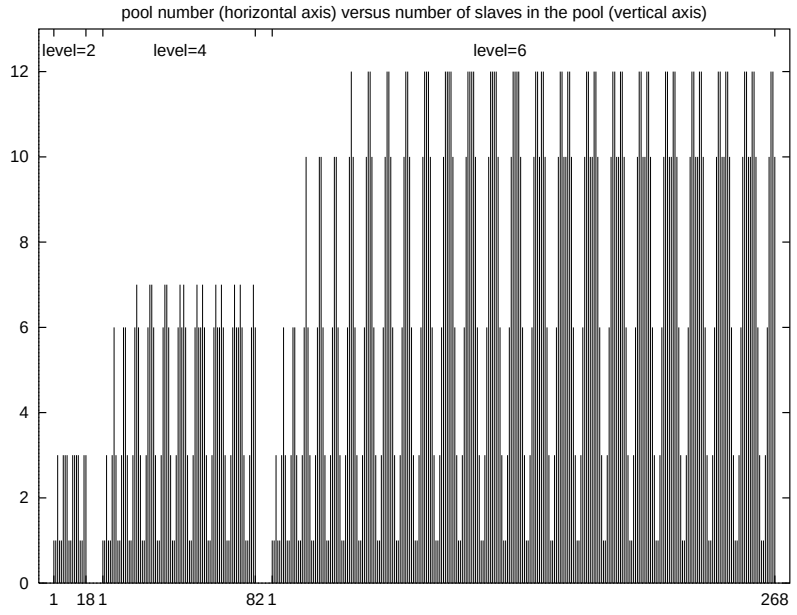


a. $n = 4$. b. $n = 8$. c. $n = 16$.

Figure 4: 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.

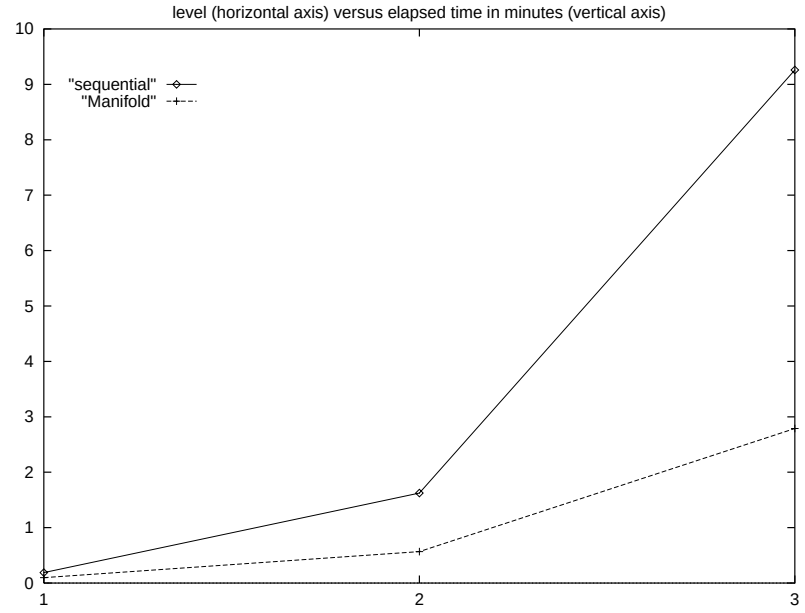


a. Sparse.

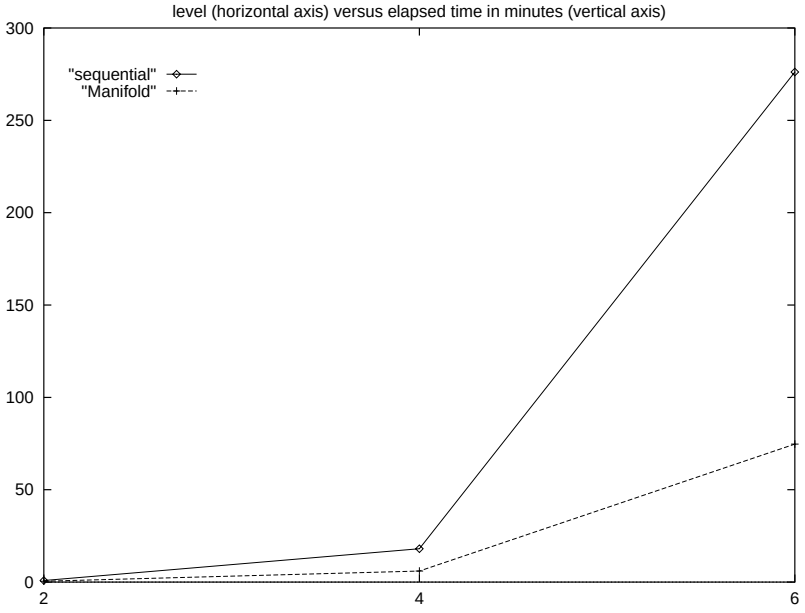


b. Semi-sparse.

Figure 5: Different pools of slaves created during the parallel applications.



a. Sparse.



b. Semi-sparse.

Figure 6: Performances (average elapsed times) of the sequential version and the parallel version of the algorithm.