Dynamic Process Composition and Communication Patterns in Irregularly Structured Applications *

C.T.H. Everaars, B. Koren and F. Arbab
email: Kees.Everaars@cwi.nl, Barry.Koren@cwi.nl and Farhad.Arbab@cwi.nl

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, Telefax 020-5924199 (+31 20 5924199)

Abstract. In this paper we describe one experiment in which a new coordination language, called **MANIFOLD**, is used to restructure an existing sequential Fortran code from computational fluid dynamic (CFD), into a parallel application. **MANIFOLD** is a coordination language developed at CWI (Centrum voor Wiskunde en Informatica) in the Netherlands. It is very well suited for applications involving dynamic process creation and dynamically changing (ir)regular communication patterns among sets of independent concurrent cooperating processes. With a simple, but generic, master/worker protocol, written in the **MANIFOLD** language, we are able to reuse the existing code again, without rethinking or rewriting it. The performance evaluation of a standard 3D CFD problem shows that **MANIFOLD** performs very well.

1 Introduction

A workable approach for modernization of existing software into parallel/distributed applications is through coarse-grain restructuring. If, for instance, entire subroutines of legacy code can be plugged into the new structure, the investment required for the re-discovery of the details of what they do can be spared. The resulting renovated software can then take advantage of the improved performance offered by modern parallel/distributed computing environments, without rethinking or rewriting the bulk of their existing code. The necessary communications between the different partners in such a new concurrent system can have different forms. In some cases, the channel structures representing the communication patters between the different partners, are regular and the numbers of

the partners is fixed (structured static communication). In other cases, the communication patterns are irregular and the number of partners changes over time (unstructured dynamic communication). There are many different languages and programming tools available that can be used to implement this kind of communications, representing very different approaches to parallel programming. Normally, languages like Compositional C++, High Performance Fortran, Fortran M, Concurrent C(++) or tools like MPI, PVM, and PARMACS are used (see [1] for some critical notes on these languages and tools). There is, however, a promising novel approach: the application of *coordination* languages [2–4].

In this paper we describe one experiment in which a new coordination language, called **MANIFOLD**, was used to restructure an existing Fortran 77 program into a parallel application. **MANIFOLD** is a coordination language developed at CWI (Centrum voor Wiskunde en Informatica) in the Netherlands. It is very well suited for applications involving dynamic process creation and dynamically changing (ir)regular communication patterns among sets of independent concurrent cooperating processes [5, 6]. Programming in **MANIFOLD** is a game of dynamically creating process instances and (re)connecting the ports of some processes via streams (asynchronous channels), in reaction to observed event occurrences. This style reflects the way one programmer might discuss his interprocess communication application with another programmer on telephone (let process a connect process b with process c so that c can get its input; when process b receives event e, broadcast by process c, react on that by doing this and that; etc.). As in this telephone analogy, processes in **MANIFOLD** do not explicitly send to or receive messages from other processes. Processes in **MANIFOLD** are treated as black-box workers that can only read or write through the openings (called ports) in their own bounding walls. It is always a third party - a coordinator process called a manager - that is responsible for setting up the communication channel (in **MANIFOLD** called a stream) between the output port of one process and the input port of another process, so that data can flow through it. This setting up of the communication links from the *outside* is very typical for **MANIFOLD** and has several advantages. An important advantage is that it results in a clear separation between the modules responsible for computation and the modules responsible for coordination, and thus strengthens the modularity and enhances the re-usability of both types of modules (see [1, 6, 7]).

The **MANIFOLD** system runs on multiple platforms and consists of a compiler, a run-time system library, a number of utility programs, and libraries of built-in and predefined processes of general interest. Presently, it runs on IBM RS60000 AIX, IBM SP1/2, Solaris, Linux, Cray, and SGI IRIX [1].

The original Fortran 77 code in our experiment was developed at CWI by a group of researchers in the department of Numerical Mathematics, within the framework of the BRITE-EURAM Aeronautics R&D Programme of the European Union. The Fortran code consist of a number of subroutines (about 8000 lines) that manipulate a common date structure. It implements their multi-

---

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

```
program SEQ_CODE
begin

Preamble:
  - Some initialization work
  - Some initial sequential computations

Heavy computational job:
  for i = 1 to N
    - Heavy computations that can't be done in parallel
    - Heavy computations that can in principle be done in parallel
  endfor

Postamble:
  - Some final sequential computations
  - Printing of results

end
```

Fig. 1. The schema of the sequential code

grid solution algorithm for the Euler equations representing three-dimensional, steady, compressible flows. They found their full-grid-of-grids approach to be effective (good convergence rates) but inefficient (long computing times). As a remedy, they looked for methods to restructure their code to run on multi-processor machines and/or to distribute their computation over clusters of workstations.

Clearly, the details of the computational algorithms used in the original program are too voluminous to reproduce here, and such computational detail is essentially irrelevant for our restructuring. Thus we refer for a detailed description of the software to the last four chapters of reference [8], the official report on the BRITE-EURAM project, and instead use a simplified pseudo program in this paper that has the same logical design and structure as the original program. In section 2, we present this simplified pseudo program and give its parallel counterpart. Next, in section 3, we describe how we implement our parallel version using the coordination language MANIFOLD. In section 4, we show performance results for the standard test case of an ONERA M6 half-wing in transonic flight. Finally, the conclusion of the paper is in section 5.

2   The Simplified Pseudo Code and its Parallel Version

The simplified pseudo code as distilled from the original program is shown in figure 1. The heavy computations that, in principle, can be done in parallel represent the original Fortran version's pre- or post- Gauss-Seidel relaxations on all the cells of a certain grid [9]. Because the relaxation subroutine reads and writes data concerning its own grid only, the relaxations can in principle be done in parallel for all the grids to be visited at a certain grid level. In figure 2, we show the parallel version of the simplified pseudo code. There, we create inside a loop a worker-pool consisting of a number of workers to which we delegate the relaxations of the different grids. Note that the number of workers is dependent on the index i of the loop. When the workers can run as separate processes using different processors on a multi-processor hardware, then we have the desired parallel structure.
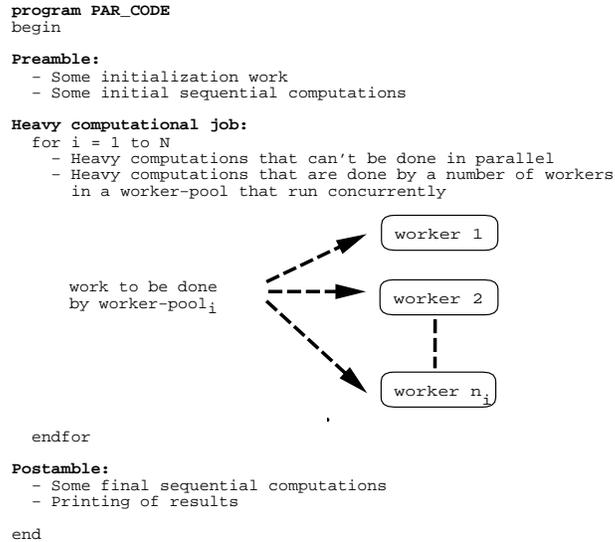
```
program PAR_CODE
begin

Preamble:
   - Some initialization work
   - Some initial sequential computations

Heavy computational job:
   for i = 1 to N
      - Heavy computations that can't be done in parallel
      - Heavy computations that are done by a number of workers
        in a worker-pool that run concurrently
```



```
   endfor

Postamble:
   - Some final sequential computations
   - Printing of results

end
```

Fig. 2. The schema of the parallel code

3 The Parallel Implementation using Manifold

We can describe the parallel schema of figure 2 in a kind of master/worker protocol in which the master performs all the computations of the sequential code except the relaxations which are done by the workers.

```
1  // protocolMW.m
2
3  #define IDLE terminated(void)
4
5  export manifold ProtocolMW(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 }
```

In **Manifold** we can do this in a general way, as shown in the code above (for details see reference [9]). There, the master and the worker are parameters of the protocol implemented in the **Manifold** language as a separate coordinator process (line 5). In this protocol we describe only how instances of the master and worker process definitions communicate with each other. For the protocol it is irrelevant to know what kind of computations are performed in the master and the worker.

In the coordinator process we create and activate the master process that embodies all computation except the relaxations (line 7). Each time the master

arrives at the pre- or post-relaxation, he raises an event to signal the coordinator to create a worker (line 12). In this way, a pool of workers is set to work for the master whereby each pool contains its own number of workers (see figure 2). Before a worker can really work, it should know on which grid it must perform the relaxation. Because the master has this information available, the coordinator sets up a communication channel (called a stream in **Manifold**) between the master and the worker so that the master can send this information to the worker (see the arrow between the master and the worker on line 14 which represents a stream). Also, the coordinator must inform the master of the identification of the worker so that it can activate the worker (see the arrow between the reference of the worker denoted by `&worker` and the master). When all the workers of a certain worker-pool are created and activated in this way, the master waits until the workers are done with the relaxation and are prepared to terminate. After this rendezvous, the master continues its work (i.e., the index i of the loop in figure 2 is incremented by one) until it again arrives at a point where it wants to use a pool of workers to delegate the relaxations to.

Note that it is not necessary to have a streams from workers to the master through which the workers send the results of their relaxations back to the master. The reason for this is that the relaxation work of the different workers, running as different **Manifold** processes, can run as threads (light-weight processes) [10]. in the same operating-system-level (heavy weight) process, and thus can share the same global data space. Therefore, the restructuring we present here is not suitable for distributed memory computing. Nevertheless, the restructured program we present here *does* improve the performance of the application as we will see in the next section. For a description of the distributed memory restructuring see http://www.cwi.nl/~farhad/CWICoordina.html.

Having implemented the master/worker protocol in **Manifold** in a general way, the only thing we need to do is to use this protocol by replacing its formal parameters by actual values which are processes. The actual master and worker manifolds are easy to implement as atomic processes written in C. These C functions then call the original Fortran code (8000 lines) to do the real work [9].

4   Performance Results

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 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 were done for both the sparse- and the semi-sparse-grid approaches. For the sparse-grid approach, the finest grid levels considered are: 1,

Table 1. Average elapsed times (hours:minutes:seconds) for sparse- and semi-sparse-grid-of-grids approaches.

	level	*sequential*	*parallel*
	1	11.24	5.84
sparse	2	1:37.42	34.06
	3	9:15.56	2:47.40
	2	50.43	27.33
semi-sparse	4	18:02.10	5:58.72
	6	4:36:08.54	1:14:44.04

2 and 3; for the semi-sparse-grid approach, the finest grid levels are: 2, 4 and 6. The higher the grid level the heavier the computational work. The results of our performance measurements for both the sparse- and the semi-sparse-grid approaches are summarized in table 1, which shows 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. From the raw numbers obtained from these experiments we discarded the best and the worst performances and computed the averages of the other three which are shown in table 1.

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.

The dynamic creation of workers in different work pools for the sparse- and the semi-sparse-grid versions, are respectively shown in Figures 3 and 4. From

Table 2. Work pool and worker statistics.

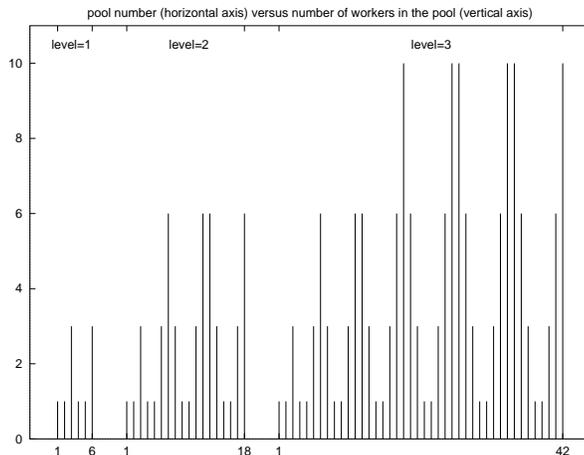| *application*	*level*	n_p	$(n_w)_{\max}$	$(n_w)_{\text{total}}$
	1	6	3	10
sparse	2	18	6	50
	3	42	10	170
	2	18	3	38
semi-sparse	4	82	7	336
	6	268	12	1838

Fig. 3. Different pools of workers created during the parallel sparse-grid applications.

Figure 3 we see that for level=1, 6 pools of workers were created with their corresponding synchronization points each with 1, 1, 3, 1, 1 and 3 workers on board, respectively. This makes the total number of worker processes for this application 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. The numbers for both the sparse- and the semi-sparse-grid applications are summarized in Table 2. Here, n_p denotes the number of pools, $(n_w)_{\max}$ the maximum number of workers in a pool and $(n_w)_{\mathrm{total}}$ the total number of workers in the application. Note that in the semi-sparse-grid application in level 6 the average elapsed time went from 4 hours and 36 minutes down to 1 hour and 14 minutes by introducing 268 worker-pools in which a total of 1838 workers, as independently running processes, did their relaxation work.

5 Conclusions

Parallelizing existing sequential applications is often considered to be a complicated and challenging task. Mostly it requires thorough knowledge about both the application to be parallelized and the development system to be used. Our experiment using **MANIFOLD** to restructure existing Fortran code 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 unique property of **MANIFOLD** that enables such high degree of modularity is inherited from its underlying IWIM (*Idealized Worker Idealized Manager*) model in which communication is set up from the *outside* [6]. The core relevant concept in the IWIM model of communication is isolation of the
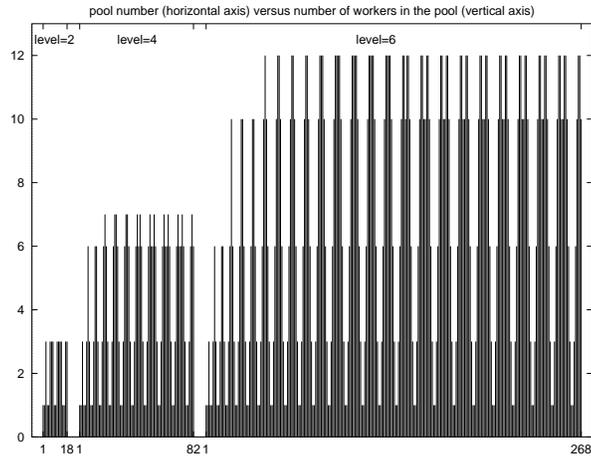
Fig. 4. Different pools of workers created during the parallel semi-sparse-grid applications.

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 completely different computations (the sparse-grid and semi-sparse-grid application use the same protocol manifold, see also reference [7] for this notion of re-usability).

The performance evaluation of our test problem shows that **MANIFOLD** performs very well. Encouraged by the good results of this pilot study and the practical value it has for the partners who already use the sequential sparse-grid software, we now intend to develop a distributed restructuring of this sequential application. Also the distributed restructuring essentially consists of picking out the computation subroutines in the original Fortran 77 code, and gluing them together with coordination modules written in **MANIFOLD**. Again no rewriting of, or other changes to, these subroutines are necessary and we can reorganize according to a master/worker protocol. The additional work we have to do now, is to arrange for the **MANIFOLD** coordinators to send and receive the proper segments of the global data structure, which in the parallel version were available through shared memory, via streams. We can thereby implement the **MANIFOLD** glue modules (as separately compiled programs!) in such a way that their **MANIFOLD** code can run in distributed, as well as parallel environments.

References

1. F. Arbab. The influence of coordination on program structure. In *Proceedings of the 30th Hawaii International Conference on System Sciences*. IEEE, January 1997.
2. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communication of the ACM*, 35(2):97–107, February 1992.
3. F. Arbab, P. Ciancarini, and C. Hankin. Coordination languages for parallel programming. *Parallel Computing*, 24(7):989–1004, July 1998. special issue on Coordination languages for parallel programming.
4. G.A. Papadopoulos and F. Arbab. *Coordination Models and Languages*, volume 46 of *Advances in Computers*. Academic Press, 1998.
5. F. Arbab. Coordination of massively concurrent activities. Technical Report CS–R9565, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, November 1995. Available on-line http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.
6. F. Arbab. The IWIM model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag, April 1996.
7. F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaars. Reusable coordinator modules for massively concurrent applications. *Software: Practice and Experience*, 28(7):703–735, June 1998. Extended version.
8. H. Deconinck and eds. B. Koren. *Euler and Navier-Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration*. Notes on Numerical Fluid Mechanics 57. Vieweg, Braunschweig, 1997.
9. C.T.H. Everaars and B. Koren. Using coordination to parallelize sparse-grid methods for 3D CFD problems. *Parallel Computing*, 24(7):1081–1106, July 1998. special issue on Coordination languages for parallel programming.
10. Bradford Nicols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Cebastopol, CA, 1996.