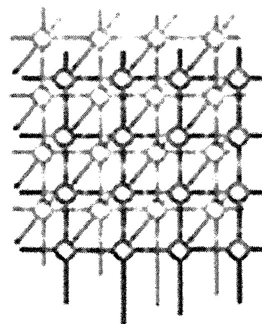
CONCURRENCY: PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **12**:157–174



Dynamic process composition and communication patterns in irregularly structured applications

C. T. H. Everaars*,†, F. Arbab‡ and B. Koren§

CWI, PO Box 94079, 1090 GB Amsterdam, The Netherlands

SUMMARY

In this paper we describe one experiment in which a new co-ordination language, called MANIFOLD, is used to restructure an existing sequential Fortran 77 code from computational fluid dynamics (CFD), into a parallel application. MANIFOLD is a co-ordination 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 co-operating processes. With a simple but generic master/worker protocol, written in the MANIFOLD language, we are able to reuse the existing code without rethinking or rewriting it. The performance evaluation of a standard 3D CFD problem shows that MANIFOLD performs very well. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: parallel languages; parallel computing; distributed computing; co-ordination languages; models of communication; irregular communication patterns; unstructured process composition; software renovation; multi-grid methods; sparse-grid methods; computational fluid dynamics; three-dimensional flow problems

1. INTRODUCTION

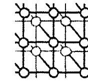
A workable approach for modernization of existing software into parallel/distributed applications is through coarse-grain restructuring. If, for instance, entire subroutines of legacy code can be plugged into the new structure, the investment required for the rediscovery of the details of what they do can

*Correspondence to: C. T. H. Everaars, CWI, PO Box 94079, 1090 GB Amsterdam, The Netherlands.
†E-mail: Kees.Everaars@cwi.nl
‡E-mail: Farhad.Arbab@cwi.nl
§E-mail: Barry.Koren@cwi.nl

Contract/grant sponsor: NCF; contract/grant number: NRG 98.04

Copyright © 2000 John Wiley & Sons, Ltd.

Received October 1998
Revised September 1999



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 patterns between the different partners, are regular, and the number 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 communication, representing very different approaches to parallel programming. Normally, languages such as Compositional C++, High Performance Fortran, Fortran M, Concurrent C(++) or tools (sometimes called *middleware*) such as MPI, PVM and PARMACS are used (see [1] for some critical notes on these languages and middleware). However, a promising novel approach is the application of *co-ordination* languages [2–4].

Co-ordination languages can be thought of as the linguistic counterpart of platforms which offer middleware support for software composition. Co-ordination languages, models and systems constitute a new field of study in programming and software systems, with the goal of finding solutions to the problem of managing the interaction among concurrent programs. Co-ordination can be defined as the study of the dynamic topologies of interactions among processes or agents, and the construction of protocols to realize such topologies that ensure well-behavedness. Analogous to the way in which topology abstracts away the metric details of geometry and focuses on the invariant properties of (seemingly very different) shapes, co-ordination abstracts away the details of computation in processes or agents, and focuses on the invariant properties of (seemingly very different) programs.

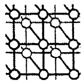
Co-ordination languages have been applied to the parallelization of computation-intensive sequential programs in the fields of simulation of fluid dynamics systems, matching of DNA strings, molecular synthesis, parallel and distributed simulation, monitoring of medical data, computer graphics, analysis of financial data integrated into decision support systems, and game playing (chess). See [3,5–7] for some concrete examples, and [4] for a survey.

In this paper we describe one experiment in which a new co-ordination language, called MANIFOLD, was used to restructure an existing Fortran 77 program into a parallel application. MANIFOLD is a co-ordination 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 co-operating processes [8,9].

The original Fortran 77 code in our experiment was developed at CWI in the Department of Numerical Mathematics, within the framework of the BRITE-EURAM Aeronautics R&D Programme of the European Union. The Fortran code consists of a number of subroutines (about 8000 lines) that manipulate a common data structure. It implements a multi-grid solution algorithm for the Euler equations representing three-dimensional, steady, compressible flows. The full-grid-of-grids approach was found to be effective (good convergence rates) but inefficient (long computing times). As a remedy, we looked for methods to restructure the code to run on multiprocessor machines and/or to distribute the 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,

Copyright © 2000 John Wiley & Sons, Ltd.



for a detailed description of the software we refer to the last four chapters of [10], 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 this paper, we first give, in Section 2, an introduction to the MANIFOLD language and discuss a 'toy' application. Apart from showing some of the syntax and semantics of MANIFOLD, we describe through this example most elements of the MANIFOLD system.¶

In Section 3, we present the simplified pseudo-program as distilled from the original Fortran 77 program and give its parallel counterpart. In Section 4, we describe how we implement our parallel version using the co-ordination language MANIFOLD. In Section 5, 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 6.

2. THE MANIFOLD CO-ORDINATION LANGUAGE

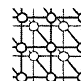
In this section, we briefly introduce MANIFOLD. MANIFOLD is used to develop concurrent software, regardless of whether it runs on a parallel or a distributed platform. MANIFOLD is not a parallel programming language; it is a *co-ordination language* as opposed to a *computation language* [2]. MANIFOLD is a *complete* language (as opposed to a language extension, like Linda [11]) for programming the co-operation protocols of concurrent systems. These protocols describe the routing of the information between various processes that comprise a concurrent application, and the dynamic changes that take place in such routing networks in reaction to events.

MANIFOLD is based on the IWIM (*Idealized Worker Idealized Manager*) model of communication [9]. The basic concepts in the IWIM model (and thus also in MANIFOLD) are *processes*, *events*, *ports* and *channels* (in MANIFOLD called streams) (see Sections 2.1 through 2.4). In IWIM a process can be regarded as a *worker* process or a *manager* (or co-ordinator) process. In IWIM an application is built as a (dynamic) hierarchy of worker and manager processes. Lowest in the hierarchy are pure worker processes that do not do any co-ordinating activities. Highest in the hierarchy is a pure co-ordinator. A process between the lowest and highest level may consider itself a worker doing a task for a manager higher in the hierarchy or a manager co-ordinating processes lower in the hierarchy.

Programming in MANIFOLD is a game of dynamically creating process instances and (re)connecting the ports of some processes via streams (asynchronous channels), in reaction to observed event occurrences. Its style reflects the way one programmer might discuss his interprocess communication application with another programmer on a telephone (let process a connect process b with process c so that c can get its input; when process b receives event e, broadcast by process c, react to that by doing this and that; etc.). As in this telephone call, processes in MANIFOLD (in this case b and c) do not explicitly send to or receive messages from other processes. Processes in MANIFOLD are treated as black-boxes that can only read or write through the openings (called ports) in their own bounding walls. It is the responsibility of a worker process to perform a (computational) task. A

¶For more information on MANIFOLD, we refer to our html pages located at http://www.cwi.nl/~farhad/manifold.html.

Copyright © 2000 John Wiley & Sons, Ltd.



worker process is not responsible for the communication that is necessary for it to obtain the proper input it requires to perform its task (it simply reads this information from its own input port), nor is it responsible for the communication that is necessary to deliver the results it produces to their proper recipients (it simply writes this information to its own output port). In general, *no process in IWIM is responsible for its own communication with other processes.* It is always the responsibility of a third party – a co-ordinator process called a *manager* – to arrange for and to co-ordinate the necessary communications among a set of worker processes. This third party sets up the communication channel between the output port of one process and the input port of another process, so that data can flow through it. This setting up of the communication links from the *outside* is very typical for MANIFOLD and has several advantages. One important advantage is that it results in a clear separation between the modules responsible for computation (the workers) and the modules responsible for co-ordination (the managers). This strengthens the modularity and enhances the reusability of both types of modules (see [1,9,12]).

A MANIFOLD application consists of a (potentially very large) number of (light- and/or heavy-weight) 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 co-operating with other processes through MANIFOLD in a concurrent application.

The MANIFOLD system consists of a compiler called MC, a runtime system library, a number of utility programs, libraries of built-in and predefined processes [13], a link file generator called MLINK and a runtime configurator called CONFIG. MLINK uses the object files produced by the (MANIFOLD and other language) compilers to produce link files needed to compose the application-executable files for each required platform. At runtime of an application, CONFIG determines the actual host(s) where the processes which are created in the MANIFOLD application will run.

The system has been ported to several different platforms (e.g. IBM RS60000 AIX, IBM SP1/2, Solaris, Linux, Cray and SGI). The system was developed with emphasis on portability and support for heterogeneity of the execution environment. It can be ported with little or no effort to any platform that supports a thread facility functionally equivalent to a small subset of the Posix threads, plus an inter-process communication facility roughly equivalent to a small subset of PVM [14].

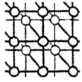
The MANIFOLD system automatically takes care of the data conversion necessary for communication in a heterogeneous environment. These conversions are only done when the receiving process really attempts to use the data. When data are simply to be passed on to another process on another machine, conversion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

In Sections 2.1 through 2.4 we briefly discuss the basic concepts in MANIFOLD, and we end our MANIFOLD introduction in Section 2.5 with a parallel/distributed 'toy' example.

2.1. 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. Furthermore, a regular C function running as an independent thread [15] can be used as an atomic process, too. Atomic processes can only produce and consume units through their ports, broadcast and receive events, and compute. They cannot set up streams between (ports of) processes. This is something only a

Copyright © 2000 John Wiley & Sons, Ltd.



co-ordinator process can do. In this way, the desired separation of computation and co-ordination is achieved.

Co-ordination processes are written in the MANIFOLD language and are called manifolds. A manifold definition defines a process type and 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 instance. 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 an interface library.

2.2. Ports

A *port* is a regulated opening at the boundary of a process, through which the information produced and/or consumed by the process is exchanged with other processes. Regulated means that the information can flow in only one direction through a port; it either flows into or out of the process. The information exchanged between a process and other processes through its ports is quantized in discrete bundles called units. A *unit* is a packet containing an arbitrary number of bits that are produced, transferred and consumed in an integral fashion, i.e. there are no partial units.

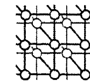
Ports through which units flow into a process are called the *input* ports of the process. Similarly, ports through which units flow out of a process are called the *output* ports of the process.

2.3. Streams

All communication in MANIFOLD is asynchronous. In MANIFOLD, the asynchronous IWIM channels are called streams. A stream is a communication link that transports 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 port of) a producer process and (a port of) a consumer process, it operates autonomously and transfers the units from its source to its sink. When the process at the sink of a stream requires a unit through its port connected to that stream, it is suspended only if no units are available in any of the streams connected to the arrival side of that port. The suspended process resumes 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 regard 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 [9] or [13] for details.

Copyright © 2000 John Wiley & Sons, Ltd.



2.4. Events and state transitions

In MANIFOLD, once an event is *raised* (broadcast) by a process, the latter continues, 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*. An observer process can react to an event occurrence in its event memory at its own leisure. In reaction to such an event occurrence, a manifold instance 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. Pre-empting a state can dismantle the streams (depending on their types) that were connected upon transition to that state. 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.

2.5. An artificial table tennis game

We now illustrate MANIFOLD through an example that implements a game of table tennis between *you* and *me*. We give the MANIFOLD source file of this example below (line numbers have been added):

```
1       #include "MBL.h"
2
3       #include "rdid.h"
4
5       event ping, pong.
6
7       /**************************************************/
8       manifold ping_pong_player(event e)
9       {
10        auto process v is variable.
11
12        begin: v = input; MES(v); raise(e); output = v; post(begin).
13      }
14
15      /**************************************************/
16      manifold umpire
17      {
18        process you is ping_pong_player(ping).
19        process me  is ping_pong_player(pong).
20
21        begin: (MES("Start ping-ponging"),
22               activate(you, me), "ball" -> you, terminated(void)).
23
```

Copyright © 2000 John Wiley & Sons, Ltd.

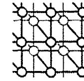
```
24          ping: you -> me.
25
26          pong: me -> you.
27        }
28
29        /**************************************************/
30        manifold Main
31        {
32          begin: umpire.
33        }
```

This code describes three manifolds (i.e. process types) named `ping_pong_player`, `umpire` and `Main` (respectively lines 8–13, 16–27, 30–33). A manifold is a template from which we can make process instances. A process instance always has an event memory into which the process instance itself or other process instances can put event occurrences. MANIFOLD is an event-driven language, which means that once a process instance detects an event occurrence in its event memory, the process instance makes a transition out of its current state to the state that is labeled with the name of that event occurrence. Switching to a state also means that the streams that were constructed in the former state are broken down (see [9] for the details). In MANIFOLD syntax, a state looks like '`event_name: actions to be executed.`' and its semantic is 'switch to this state when there is an event with this name in the event memory and execute the actions.' The most important (primitive) actions are (i) creating and activating process instances, (ii) broadcasting events (with the action `raise`) or putting them in a process' own event memory (with the action `post`), and (iii) connecting processes to each other by setting up streams between their ports (by the action denoted by the arrow `->`).

A manifold written in the MANIFOLD languages (we can also write them in a traditional programming language such as C) always has the following structure. After the word `manifold` a name is given (line 8: `ping_pong_player`) followed by some optional parameters (line 8: `event e`). After this first line comes the body of the manifold (lines 9–13). The body of a manifold is a block enclosed in a pair of braces (for the ping-pong player, lines 9 and 13) and contains some optional global declarations (line 10 for the ping-pong player, lines 18 and 19 for the umpire) followed by one or more states to which an instance of the manifold can jump when there is a suitable event in its event memory. In our case, we see that the `ping_pong_player` and `Main` manifolds (and thus their instances) have only the `begin` state. The umpire, on the other hand, has three different states: the `begin`, `ping` and `pong` states. A `begin` state in MANIFOLD is something special. Activation of a process instance automatically puts an occurrence of the special high priority `begin` event in the event memory of that process, which results in an initial state transition to the begin state.

In our artificial game of table tennis, we consider a ping-pong player as someone who sequentially performs in his begin state the following actions (sequential execution is syntactically denoted by the connective ';' between the actions):

- It reads (catches) a ball (line 12: `v=input`) from its input port and stores it in a variable (line 10).
- It shows the ball on the screen (line 12: MES(v)).
- It broadcasts the event it receives as a parameter (line 8) to signal the umpire that it intends to write (return) the ball to its output port (line 12: `raise(e)`).



- It puts the `begin` event in its own event memory (line 12: `post(begin)`) which results in another round of execution of the actions in the `begin` state.

The umpire manifold can be described as follows:

- In its local declaration part, the umpire uses the syntactic construct '`process x is y.`' to create two processes named `you` and `me` as instances of the ping-pong player manifold (line 18, 19). Note that the actual parameters in the process creation `you` and `me` are respectively `ping` and `pong` so that `you` always raises `ping` and `me` always raises `pong` (line 12: `raise(e)`).
- In the `begin` state, the umpire shows the message '`Start ping-ponging`' on the screen (line 21), activates the two process instances `you` and `me` (line 22), gives a ball to `you` (line 22: `'ball' -> you`) and waits (denoted by `terminated(void)` on line 22) until it detects one of the global events (declared on line 5).
- Because `you` starts returning the ball and raises (broadcasts) the `ping` event (line 12), the first event found in the event memory of the umpire is `ping`, which causes a state transition from the `begin` state to the `ping` state. In this state a stream is created between `you` and `me` so that the ball can flow through this stream to `me` (line 24: `you -> me`).
- The process instance `me` behaves the same way as `you` except that it raises the event `pong` to signal to the umpire that the ball is written to its output port. In reaction to this event, the umpire makes a state transition out of the `ping` state, which results in breaking the connection between `you` and `me`, and enters the `pong` state, where a new connection between `me` and `you` (line 26: `me -> you`) is created through which the ball can flow back to `you`. It is clear that this table tennis game never stops.

The third manifold is `Main`. 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. In our case the `Main` manifold has only the `begin` state in which it automatically creates and activates an instance of the umpire manifold, just by calling the umpire by name (line 32). This implicit process instantiation is an alternative to the explicit creation of a process (as we did on line 18) and explicit activation (as we did on line 22).

Our table tennis program is very artificial, and one might think of many other ways to implement it. For instance, we can set up the stream connections between the ping-pong players in the umpire in just one state, in which case there is no need to do state transitions and thus there is no need to set up and break the stream connections between the ping-pong players again and again. We chose to implement it as we did because this scheme clearly shows the dynamic changing of connections on the beat of event occurrences, which is an important aspect of MANIFOLD's event-driven nature.

Process instances in a MANIFOLD application always run as separate threads (light-weight processes [15]) within an operating-system level process. This latter heavy-weight process is called a task instance in MANIFOLD. The way process instances are bundled in task instances influences the mechanism that is actually used for the data transport in streams (the `->`). The bundling can be done automatically or under user control. When we ensure that all process instances of a MANIFOLD application run as threads in the same task instance, the effective data transport in the streams between these process instances is implemented in shared memory. In that case, we in fact play 'shared memory table tennis'. We can also bundle the process instances in such a way that each ping-pong player and

Copyright © 2000 John Wiley & Sons, Ltd.

the umpire is housed in a separate task instance. In that case the effective data transport in streams between the process instances is implemented using Unix sockets. The mapping of process instances in task instances is considered to be a separate stage in the application construction. This mapping is described in a file which is the input for the MANIFOLD linker MLINK. An example of such an input file is shown below (line numbers have been added):

```
1        {task *
2         {load 10}
3         {weight ping_pong_player 10}
4         {weight umpire 10}
5        }
6
7        {task pingpong
8          {include pingpong.o}
9        }
```

In this file we specify that a task instance is considered to be 'full' when its load exceeds 10 (line 2) and that the weight of an instance of the ping-pong player or umpire is also 10 (line 3, 4). The net result of this is that each task instance will house only one thread, and thus instances of ping-pong player and the umpire end up in different instances of the task named `pingpong` (line 7). The primary output of the manifold linker is a *makefile*, plus a number of C source files necessary to provide the inter-task information.

This *makefile* is meant to be used as a black-box by recursive make commands in programmer-defined makefiles that finally create the executable files suitable for the appropriate platforms. With this, the task composition stage comes to an end, and the final stage in application construction can start. This is the runtime configuration stage, in which we define the mapping of tasks to hosts. This mapping too is described in a file which is the input for the MANIFOLD runtime configurator CONFIG. An example of such an input file is shown below.

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

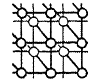
Here we define three variables `host1`, `host2` and `host3`, which we set to respectively `sampan.cwi.nl`, `pont.cwi.nl` and `opduwer.cwi.nl`. These are the names of computers located at different places and connected via a network. The last line in the file states that the instances (in our case three) of the `pingpong` task can be started on any of these three machines.

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

Running the ping-pong program using the task composition and runtime configuration described above forms a 'distributed table tennis game' and gives the following output:

```
sampan  262289 112 pingpong umpire() pingpong.m 21 -> Start ping-ponging
opduwer 786433 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
pont    524289 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
```

Copyright © 2000 John Wiley & Sons, Ltd.



```
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
```

Figure 1. The schema of the sequential code.

```
opduwer 786433 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
pont    524289 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
opduwer 786433 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
pont    524289 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
opduwer 786433 63 pingpong ping_pong_player(event) pingpong.m 12 -> ball
```

Each of these output lines has the following structure. It starts with a long label followed by a `->` before the actual message (the value of the argument of `MES`). The label shows respectively the machine on which the task instance runs, the identification of the task instance, the identification of the process instance, the name of the task, the name of the manifold, the name of the MANIFOLD source file, and the line number where `MES` is called. With such a label in front of the actual message, we always know *who* is printing *what* and *where*.

3. THE SIMPLIFIED PSEUDO-CODE AND ITS PARALLEL VERSION

The simplified pseudo-code as distilled from the original Fortran 77 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 [16]. 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 multiprocessor hardware, then we have the desired parallel structure.

Copyright © 2000 John Wiley & Sons, Ltd.

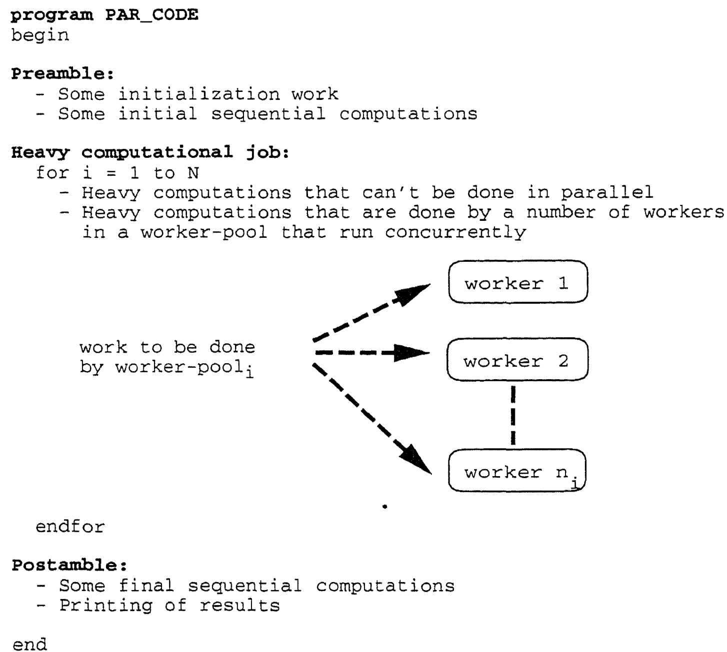
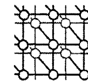


Figure 2. The schema of the parallel code

4. 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. In MANIFOLD we can do this in a general way, as shown in the code below (for details, see [16]).

```
 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        }
```

There, the master and the worker are parameters of the protocol implemented in the MANIFOLD language as a separate co-ordinator manifold named `ProtocolMW` (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.

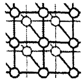
In the manifold `ProtocolMW`, we create and activate the master process that embodies all computation except the relaxations (line 7). On line 9 we make the `begin` state sensitive to events from the master by mentioning its name in the body of the state (what comes after the colon) and wait for events (due to the word IDLE). Each time the master arrives at the pre- or post-relaxation, it raises the event `create_worker` to signal to the co-ordinator to switch to the `create_worker` state where a worker is created (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 co-ordinator 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 co-ordinator 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 which it can delegate the relaxations.

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

Note that it is not necessary to have a stream 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) [15] 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.

Because we have implemented the master/worker protocol in MANIFOLD in a general way, where the master and the worker are parameters of the protocol, we can compile the protocol manifold `ProtocolMW` separately and archive it in a library. To use this protocol, the only thing we have to do is to retrieve it from this library and supply its formal parameters the proper actual values, which are

Copyright © 2000 John Wiley & Sons, Ltd.



processes. The actual master and worker manifolds are easy to implement as atomic processes written in C.

The C function for the master simply calls the original Fortran main program, which we changed a little bit. The changes have to do with the fact that the master must raise events (`create_worker` and `ready`) and must read from its input port and write to its output port. Also, it must wait to increment the loop index until all workers are done (see Figure 2).

The C function for the worker simply reads from its own input port and then calls the original Fortran code to do the real work.

Thus, the master and workers are in fact wrappers around the original Fortran code in which all the Fortran subroutines (8000 lines) are called without any change (see [16] for details).

Up to now we have spoken only about *one* computational Fortran code. In principle there were two Fortran codes, one for the so-called sparse-grid method and one for the so-called semi-sparse-grid method (see also the experiments in Section 5). Although both programs conform to the structure as given in Figure 1 and both methods use the same relaxation subroutines, their main programs were different. This means that we have for the sparse-grid application a sparse-grid master and for the semi-sparse-grid application a semi-sparse-grid master.

Using the manifold `ProtcolMW` we can construct the next two following MANIFOLD programs. This changes the original sequential code of our sparse-grid and semi-sparse-grid applications to their respective concurrent versions.

```
1    // sparse_model.m
2
3    event create_pointgsgr, finished.
4
5    manifold w_pointgsgr atomic {internal.}.
6
7    manifold w_oneram6_sparse atomic
       {internal. event create_pointgsgr, finished.}.
8
9    manifold ProtocolMS(manifold Master, manifold Worker,
                         event create_worker, event ready) import.
10
11   /*********************************************************/
12   manifold Main
13   {
14     begin: ProtocolMS(w_oneram6_sparse, w_pointgsgr,
                         create_pointgsgr, finished).
15   }
```

```
1    // semi_sparse_model.m
2
3    event create_pointgsgr, finished.
4
5    manifold w_pointgsgr atomic {internal.}.
6
```

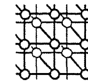


Table I. Average elapsed times (hours:minutes:seconds) for sparse- and semi-sparse-grid-of-grids approaches.

	Level	Sequential	Parallel
Sparse	1	11.24	5.84
	2	1:37.42	34.06
	3	9:15.56	2:47.40
Semi-sparse	2	50.43	27.33
	4	18:02.10	5:58.72
	6	4:36:08.54	1:14:44.04

```
 7   manifold w_oneram6_semi_sparse atomic
       {internal. event create_pointgsgr, finished.}.
 8
 9   manifold ProtocolMS(manifold Master, manifold Worker,
                         event create_worker, event ready) import.
10
11   /***********************************************************/
12   manifold Main
13   {
14     begin: ProtocolMS(w_oneram6_semi_sparse, w_pointgsgr,
                         create_pointgsgr, finished).
15   }
```

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

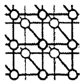
Line 5 defines the worker manifold named `w_pointgsgr`, which takes no arguments, and states (through the keyword `atomic`) that it is not implemented in the MANIFOLD language but in another programming language such as C, C^{++} or Fortran. The keyword `internal` states that the function that constitutes the body of this manifold is to run as a thread within an operating system level process.

The same holds for the master manifold `w_oneram6_sparse` (line 7). Because the events `create_pointgsgr` and `finished` are to be exchanged between the master and the rest of the MANIFOLD application, we also specify these two events between the braces on this line.

Line 9 defines the manifold `ProtocolMW`, which has four formal parameters: the master and worker manifolds and the two events `create_worker` and `ready`, 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 an instance of the `ProtocolMW` manifold is created and activated just by calling `ProtocolMW`, with the proper actual arguments, by name. After this, the instance of `Main` (named `main`) terminates,

Copyright © 2000 John Wiley & Sons, Ltd.



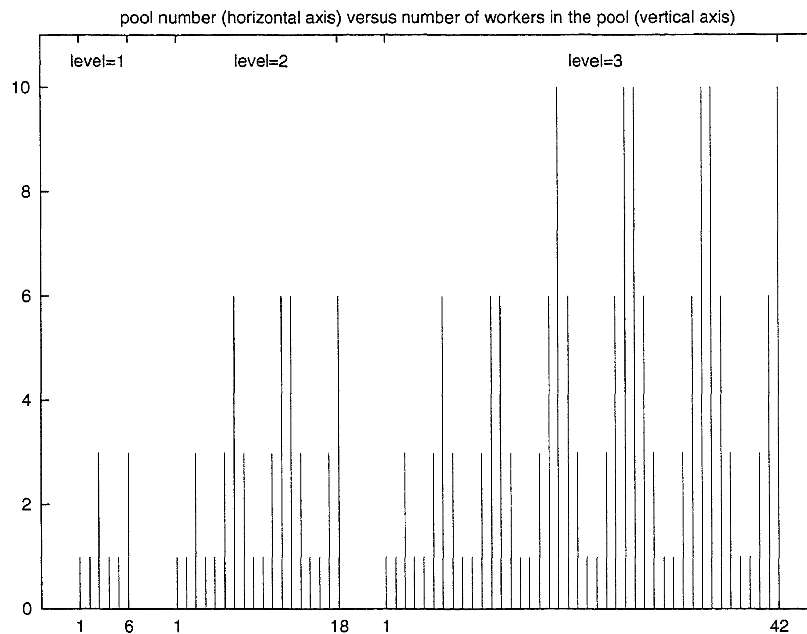


Figure 3. Different pools of workers created during the parallel sparse-grid applications.

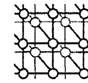
and the instances of `ProtocolMW` and `w_oneram6_sparse` (with all the workers `w_pointgsgr`) run concurrently.

5. 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, 2 and 3; for the semi-sparse-grid approach, the finest grid levels are 2, 4 and 6. The higher the grid level is, the heavier the computational work is. The results of our performance measurements for both the sparse- and the semi-sparse-grid approaches are summarized in Table I, 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

Copyright © 2000 John Wiley & Sons, Ltd.



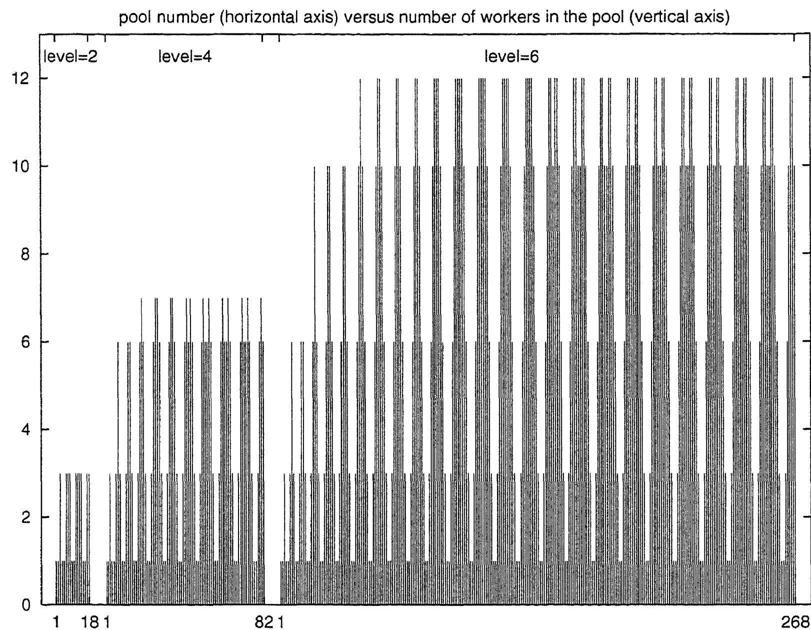


Figure 4. Different pools of workers created during the parallel semi-sparse-grid applications.

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 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 to obtain the results shown in Table I.

From these results, it is clear 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 shown in Figures 3 and 4, respectively. From Figure 3 we see that for level = 1, six 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.

Copyright © 2000 John Wiley & Sons, Ltd.

Table II. Work pool and worker statistics.

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

The numbers for both the sparse- and the semi-sparse-grid applications are summarized in Table II. Here, n_p denotes the number of pools, $(n_w)_{max}$, the maximum number of workers in a pool and $(n_w)_{total}$ 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 h 36 min down to 1 h 14 min by introducing 268 worker-pools in which a total of 1838 workers, as independently running processes, did their relaxation work.

6. CONCLUSIONS

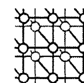
Parallelizing existing sequential applications is often considered to be a complicated and challenging task. Mostly it requires thorough knowledge about both the application to be parallelized and the development system to be used. Our experiment using MANIFOLD to restructure existing Fortran code into a parallel application indicates that this co-ordination language is well-suited for this kind of work.

The modular structure of the resulting application and the ability to use existing computational subroutines of the sequential Fortran program are remarkable. The property of MANIFOLD that enables such a high degree of modularity is inherited from its underlying IWIM (*Idealized Worker Idealized Manager*) model in which communication is set up from the *outside* [9]. The core relevant concept in the IWIM model of communication is isolation of the computational responsibilities from communication and co-ordination concerns into separate, pure computation modules and pure co-ordination modules. This is why the MANIFOLD modules in our example can co-ordinate the already existing computational Fortran subroutines without any change.

An added bonus of pure co-ordination modules is their reusability. The same MANIFOLD modules developed for one application may be used in other parallel applications with the same or similar co-operation 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 [12] for this notion of reusability). Another important advantage of MANIFOLD is that it makes no distinction (from the language point of view) between distributed and parallel environments; the same MANIFOLD code can run in both, as we show in our 'toy' application in Section 2.

All these features make MANIFOLD a suitable framework for construction of modular software on parallel and/or distributed platforms.

Copyright © 2000 John Wiley & Sons, Ltd.



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. The distributed restructuring essentially consists of picking out the computation subroutines in the original Fortran code and gluing them together with co-ordination 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 co-ordinators 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.

ACKNOWLEDGEMENT

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

REFERENCES

1. Arbab F. The influence of coordination on program structure. *Proceedings of the 30th Hawaii International Conference on System Sciences*, IEEE, January 1997.
2. Gelernter D, Carriero N. Coordination languages and their significance. *Commun. ACM* 1992; **35**(2):97–107.
3. Arbab F, Ciancarini P, Hankin C. Coordination languages for parallel programming. *Parallel Comput.* 1998; **24**(7):989–1004. Special issue on Coordination languages for parallel programming.
4. Papadopoulos GA, Arbab F. *Coordination Models and Languages*, volume 46 of *Advances in Computers*; Academic Press, 1998.
5. Andreoli J-M, Hankin C, Le Métayer D (eds). *Coordination Programming: Mechanisms, Models and Semantics*; Imperial College Press, 1996.
6. Ciancarini P, Hankin C (eds). *1st Int. Conf. on Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*; Springer-Verlag, April 1996.
7. Garlan D, Le Métayer D (eds). *2nd Int. Conf. on Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*; Springer-Verlag, September 1997.
8. Arbab F. 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. January 2000.
9. Arbab F. The IWIM model for coordination of concurrent activities. *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, Ciancarini P, Hankin C (eds); Springer-Verlag, April 1996; pp. 34–56.
10. Deconinck H, Koren B (eds). *Euler and Navier-Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration*, volume 57 of *Notes on Numerical Fluid Mechanics*; Vieweg: Braunschweig, 1997.
11. Gelernter D. Generative communication in Linda. *ACM Trans. Program. Lang. Syst* 1985; 7(1):80–112.
12. Arbab F, Blom CL, Burger FJ, Everaars CTH. Reusable coordinator modules for massively concurrent applications. *Software: Pract. Exp.* 1998; **28**(7):703–735. Extended version.
13. Arbab F. Manifold version 2: Language reference manual. *Technical Report*, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996. Available on-line http://www.cwi.nl/ftp/manifold/refman.ps.Z. January 2000.
14. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V. PVM 3 user's guide and reference manual. *Technical Report ORNL/TM-12187*, Oak Ridge National Laboratory, September 1994.
15. Nicols B, Buttlar D, Farrell JP. *Pthreads Programming*; O'Reilly & Associates, Inc.: Cebastopol, CA, 1996.
16. Everaars CTH, Koren B. Using coordination to parallelize sparse-grid methods for 3D CFD problems. *Parallel Comput.* 1998; **24**(7):1081–1106. Special issue on Coordination languages for parallel programming.

Copyright © 2000 John Wiley & Sons, Ltd.