Using Coordination to Restructure Sequential Source Code into a Concurrent Program *

C.T.H. Everaars, F. Arbab, and B. Koren
Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Kees.Everaars@cwi.nl, Farhad.Arbab@cwi.nl, and Barry.Koren@cwi.nl

Abstract

A workable approach for modernization of existing software into parallel/distributed applications is through coarse-grain restructuring. If, for instance, entire subroutines of legacy code can be plugged into a new structure, the investment required for the re-discovery of the details of what they do can be spared. The resulting renovated software can then take advantage of the improved performance offered by modern parallel/distributed computing environments, without rethinking or rewriting the bulk of their existing code.

In this paper, we discuss one of our experiments using the coordination language MANIFOLD *to restructure an existing sequential numerical application written in Fortran 77, into a concurrent application.*

1 Introduction

A key area in software modernization is renovating aging software systems to take advantage of today's parallel and distributed computing environments. Interestingly, not all "aging software" consists of the dusty decks of the so-called legacy systems inherited from the programming projects of the previous decades. A good deal of such software is still being produced today in on-going programming projects that, for one reason or another, prefer to use a tried and true language like Fortran 77 with which they have gained some expertise, rather than to struggle their way through uncharted territories of parallel and distributed programming tools and languages such as PVM, PARMACS, MPI, or even High-Performance Fortran. A good deal of both categories of such software can benefit from a restructuring that allows them to take advantage of the increased throughput offered by the modern parallel or distributed computing platforms.

A workable approach for modernization of such existing software into parallel/distributed applications is through coarse-grain restructuring. If, for instance, entire subroutines of legacy code can be plugged into a new structure, the investment required for the re-discovery of the details of what they do can be spared. The resulting renovated software can then take advantage of the improved performance offered by modern parallel/distributed computing environments, without rethinking or rewriting the bulk of their existing code. Our approach is simple and is in fact a cut-and-paste method. First, we try to identify and isolate components in the legacy source code (the cut). Second, we glue them together by writing coordinator modules (glue modules) in a coordination language (the paste). We have used Manifold as the glue language. Manifold is a general purpose coordination language especially designed to express cooperation protocols among components in component based systems.

Our point of departure is two different pieces of existing sequential Fortran code from computational fluid dynamics (CFD). These two pieces of code were developed at CWI by a group of researchers in the department of Numerical Mathematics, within the framework of the BRITE-EURAM Aeronautics R&D Programme of the European Union. Both implement a multi-grid solution algorithm [15, 12, 14] for the Euler equations representing three-dimensional, steady, compressible flows. In the first piece of code, the problem is solved using the so-called sparse-grid method, and the other uses the so-called semi-sparse-grid method [8]. The developers of these programs found their algorithms to be effective (good convergence rates) but inefficient (long computing times). As a remedy, they looked for methods to restructure their code to run on multi-processor machines and/or to distribute their computation over clusters of workstations.

Applying our cut-and-paste method to these two programs results in *one* generally applicable coordinator module that can restructure both sequential programs into paral-

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

lel applications (which run on a shared memory machine) as well as distributed applications (which now run on a cluster of workstations). We have reported earlier about the restructuring of these Fortran programs [6]. However, the coordinator modules developed there were only able to restructure the source code into a parallel application.

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

The rest of this paper is organized as follows. In section 2 we give a brief introduction to the MANIFOLD language. In section 3 we present the simplified pseudo-program as distilled from the original Fortran 77 program, explore its structure and try to identify and isolate software components in it. This leads us to a new concurrent scheme for the simplified pseudo-program. In section 4, we describe the paste phase in the software renovation process and present our generic gluing modules written in the MANIFOLD coordination language. In section 5 we test those generic gluing modules with a "toy" example that has the same structure as the original sequential Fortran code and we also give some performance results. The actual restructuring of the two original sequential programs can be found in section 6. Finally, the conclusion of the paper is in section 7.

2 The Manifold coordination language

In this section, we give a brief overview of MANIFOLD. It is beyond the scope of this paper to present all the details of the syntax and semantics of the MANIFOLD language[1].

MANIFOLD is used to develop concurrent software, regardless of whether it runs on a parallel or a distributed platforms. MANIFOLD is used to develop concurrent software, regardless of whether it runs on a parallel or a distributed platforms. MANIFOLD is not a parallel programming language; it is a *coordination language* as opposed to a *computation language* [11]. MANIFOLD is a *complete* language (as opposed to a language extension, like Linda [10]) for programming the cooperation protocols of concurrent systems. These protocols describe the routing of the information between various processes that comprise a concurrent application, and the dynamic changes that take place in such routing networks in reaction to events.

MANIFOLD is based on the IWIM (*Idealized Worker Idealized Manager*) model of communication [1]. The basic concepts in the IWIM model (and thus also in MANIFOLD) are *processes*, *events*, *ports*, and *channels* (in MANIFOLD called streams). In IWIM, a process can be regarded as a *worker* process or a *manager* (or coordinator) process. An application is built as a (dynamic) hierarchy of worker and manager processes. Lowest in the hierarchy are pure worker processes that do not do any coordinating activities. Highest in the hierarchy are pure coordinators. A process between the lowest and highest level may consider itself a worker doing a task for a manager higher in the hierarchy, or a manager coordinating processes lower in the hierarchy.

Programming in MANIFOLD is a game of dynamically creating process instances and (re)connecting the ports of some processes via streams (asynchronous channels), in reaction to observed event occurrences. Its style reflects the way one programmer might discuss his interprocess communication application with another programmer on a telephone (let process a connect process b with process c so that c can get its input; when process b receives event e, broadcast by process c, react to that by doing this and that; etc.). As in this telephone call, processes in MANIFOLD (in this case b and c) do not explicitly send to or receive messages from other processes. Processes in MANIFOLD are treated as black-boxes that can only read or write through the openings (called ports) in their own bounding walls. It is the responsibility of a worker process to perform a (computational) task. A worker process is not responsible for the communication that is necessary for it to obtain the proper input it requires to perform its task (it simply reads this information from its own input port), nor is it responsible for the communication that is necessary to deliver the results it produces to their proper recipients (it simply writes this information to its own output port). In general, *no process in IWIM is responsible for its own communication with other processes.* It is always the responsibility of a third party—a coordinator process or *manager*—to arrange for and to coordinate the necessary communications among a set of worker processes. This third party sets up the communication channel between the output port of one process and the input port of another process, so that data can flow through it. This setting up of the communication links from the *outside* (exogenous coordination) is very typical in MANIFOLD and has several advantages. One important advantage is that it results in a clear separation between the modules responsible for computation (the workers) and the modules responsible for coordination (the managers). This strengthens the modularity and enhances the re-usability of both types of modules (see [3, 1, 4]).

A MANIFOLD application consists of a (potentially very large) number of processes that run as threads bundled up (automatically or under user control) in one or more operating-system-level processes (called task instances in MANIFOLD). The different task instances in a MANIFOLD application can run on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming lan-

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

guages. Some of them (the so-called non-compliant atomic processes) may not know anything about MANIFOLD, nor the fact that they are cooperating with other processes through MANIFOLD in a concurrent application.

The MANIFOLD system consists of a compiler called MC, a runtime system library, a number of utility programs, libraries of built-in and predefined processes [2], 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 [13], plus an inter-process communication facility roughly equivalent to a small subset of PVM [9].

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

For an introduction to the MANIFOLD language see [5].

3 The Cut

```
program SEQ_CODE
begin

Preamble:
  - Some initialization work
  - Some initial sequential computations

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

Postamble:
  - Some final sequential computations
  - Printing of results

end
```

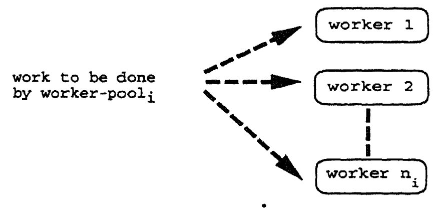
Figure 1. The schema of the sequential code

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 concurrently represent the original Fortran version's pre- or post-Gauss-Seidel relaxations on all the cells of a certain grid [8]. Because the relaxation subroutine reads and writes data concerning its own grid only, the relaxations can in principle be done concurrently for all the grids to be visited at a certain grid level. In figure 2, we show the concurrent version of the simplified pseudo-code. There, we create, inside a loop, a workers-pool consisting of a number of workers to which we delegate the relaxations of the different grids. Note that in figure 2 the number of workers in a workers-pool is not fixed, but depends on the index i of the loop.

```
program CONC_CODE
begin

Preamble:
  - Some initialization work
  - Some initial sequential computations

Heavy computational job:
  for i = 1 to N
    - Heavy computations that are done by a number of workers
      in a workers-pool that run concurrently
```



```
    - Heavy computations that cannot be done concurrently
  endfor

Postamble:
  - Some final sequential computations
  - Printing of results
```

Figure 2. The schema of the concurrent code

In a program built according to the schema in figure 2, none of the computational processes actually runs concurrently until it reaches a concurrent region. Then the multiple workers (i.e., the parallel or distributed threads) in the workers-pool begin, and the program runs concurrently. When the program exits a concurrent region, only one single computational process continues (now we run sequentially) until the process again enters a concurrent region and the process repeats. See figure 3 for this multiple-mode execution model.

4 The Paste

The crux of our restructuring is to allow the computations done in the relaxations on every single visited grid, be to carried out in separate processes. These processes can then run concurrently in MANIFOLD as separate threads executed by different processors on a multi-processor hardware (e.g., a multi-processor SGI machine), or in different tasks on a distributed platform (e.g., a network of workstations), or a combination of the two.

We have organized the restructuring according to a master/worker protocol in which the master performs all the

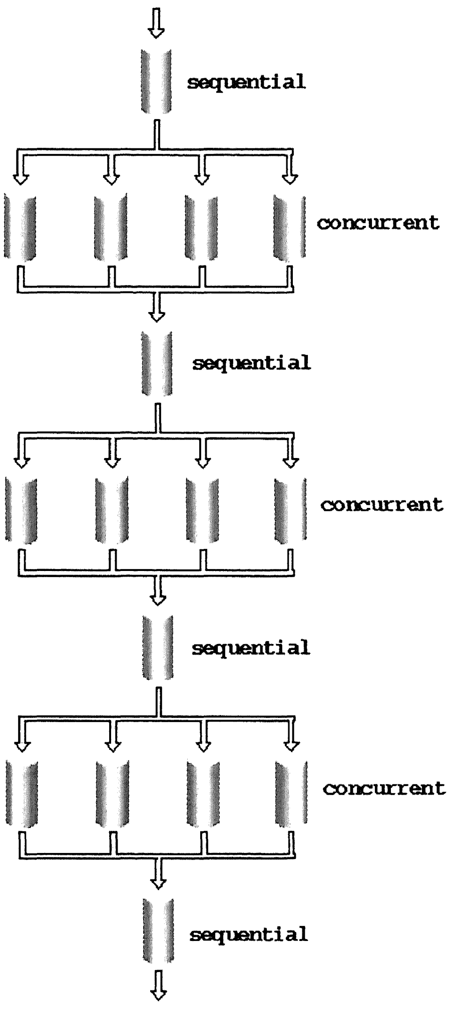


Figure 3. Execution of a typical program with sequential and concurrent parts.

computations of the sequential source code except the relaxations, which are done by the workers. In MANIFOLD, we can easily realize this master/worker protocol in a generic way, where the master and the worker are parameters of the protocol. In this protocol we describe only how instances of master and worker process definitions should communicate with each other. For the protocol, it is irrelevant to know what kind of computations are performed in the master and the worker. What is indeed important for the protocol is that the input/output and the event behavior of the master and the worker comply with the protocol. E.g., the master should write the data needed by the worker to its own output port and the worker, connected by a third party (a manager) to this port, should read this information from its own input port. Also, the coordinator can create a worker only when the master abides by the protocol and raises an event to request for its creation.

Due to space limitation, we give only an informal description of the master/worker protocol in section 4.1 and a short description of its implementation in section 4.2. For a detailed discussion of the behavior interface of the master and the worker and the way they are tuned to each other and to the protocol `ProtocolMW` we must refer to the official report of the NCF project [7].

4.1 The Glue

The master/worker protocol we use can be described as follows. In a coordinator process we create and activate a master process that embodies the computations, except the relaxations, of the main Fortran program of the sequential version. Each time the master arrives at a pre- or post-relaxation, it delegates this work to the workers in a workers-pool. The master makes its wish known to the coordinator by raising an event (`create_pool`)[2]. The coordinator reacts on this event by jumping to a state where it waits for requests coming from the master to create a worker for the workers-pool. Each time the master needs another worker for the workers-pool it raises an event (`create_worker`) to signal the coordinator to create one. Because the master wants to use the worker, it needs to know its identity. The coordinator makes this identity available to the master by sending its reference via a stream. The master waiting for its workers, receives this reference of the worker, activates it and takes care that the worker receives all necessary information so that it can do its job. The master writes this information on its output port which is connected by the coordinator to the input port of the worker, so that the latter can read it from this port. In this way, a pool of workers, created by the coordinator, is set to work by the master, each worker performing a relaxation computation. Before the master can continue its work, it must wait until all the workers are done with their relaxations and are ready to die, which they signal by raising an event (`dead_worker`). The master does not want to count those events by itself, but delegates the organization of this rendezvous (i.e., a synchronization point) by raising an event (`rendezvous`) to signal the coordinator to make the proper arrangements. In the meantime, the master takes a nap and waits for the event (`a_rendezvous`) raised by the coordinator (which is now responsible for counting the events (`dead_worker`)) to acknowledge the successful rendezvous. After this rendezvous, the master reads (if necessary, as we will see) from its input port the computational results of the workers. This is made possible

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

by the coordinator which has set up a stream between the output port of the worker and the input port of the master. Hereafter, the master proceeds with its sequential work (i.e., the index i of the loop in figure 2 is incremented by one) until it again arrives at a point where it needs a workers-pool to delegate the relaxations to.

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

- As noted before, most MANIFOLD processes run as threads bundled up in one or more MANIFOLD task instances (i.e., multi-threaded executables). It is a property of thread programming that threads, housed in the same multi-threaded executable, always share a global data space. For the communication between the master and a worker, this means that the latter does not need to receive its own individual copy of the space, as long as this worker runs in the same task instance where the master runs. In this case, it is sufficient for the worker to know the information that indicates on which grid (i.e., the indices that identify the grid) it must perform its operations. With this information, the actual data of the grid can be read from the shared global data space of the task instance. Also, there is no need in this case to send the computed results of the workers through streams back to the master. A worker can directly write its results into the shared global data space. We call workers of this type *local* workers. A local worker raises event x to inform the master that the communication must take place via shared memory as just described.

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

- The global data space used in the Fortran program essentially consist of two parts. One part contains all those data segments the workers use in their relaxation computations and which they can read and update (write) independently of each other. We call this part of the global data space the *non-fixed part*. The other part (containing grid connectivity data and geometric data) remains constant after the sequential computations in the preamble of figure 2, and is only read by the workers. We call this part of the data space the *fixed part*. The proper segment in the global master space that a remote worker needs in order to do its job consists of data from both the fixed part as well as the no-fixed part. Because the data from the fixed part needed by remote workers have a considerable overlap and because the fixed part part does not change after the sequential computations in the preamble, it is more efficient to communicated the complete fixed part of the global master space as one big chunk to remote task instances. We have arranged such that the *first* remote worker in a new remote instance is responsible for the initialization of the fixed part in its remote global space. Therefore, such a worker always raises an event xxx to inform the master to supply the fixed part. This is done in the usual MANIFOLD way: the master writes the data to its own output port which is connected, by a third party via a stream to the input port of the worker, which promptly reads it and does the initialization.

4.2 The Implementation

The MANIFOLD source code of our protocol is given below. See [5] for the MANIFOLD terminology we use.

```
// protocolMW.m

#include "MBL.h"

#include "rdid.h"

#include "protocolMW.h"
```

```
 8
 9  #define IDLE terminated(void)
10
11  /************************************************************/
12  manner Create_Worker_Pool(
      process master <input, dataport | output, error>,
13    manifold Worker(event, event, event, event) )
14  {
15    save *.
16    ignore death.
17
18    auto process now is variable(0).
19    auto process t is variable(0).
20
21    event death_worker.
22
23    priority create_worker > rendezvous.
24
25    begin: (MES("begin"), preemptall, IDLE).
26
27    create_worker: {
28      hold Worker.
29
30      process worker is Worker(death_worker, x, xx, xxx).
31
32      stream KK worker -> master.dataport.
33
34      begin: now = now + 1;
35        (MES("create_worker: begin"),
36        &worker -> master -> worker -> master.dataport, IDLE).
37    }.
38
39    rendezvous: {
40      begin: (preemptall, IDLE).
41
42      death_worker: t = t + 1;
43                    if (t < now) then (
44                      post(begin)
45                    ) else (
46                      post(end)
47                    ).
48    }.
49
50    end: (MES("rendezvous acknowledged"),
           raise(a_rendezvous)).
51  }
52
53  /************************************************************/
54  export manner ProtocolMW(
              manifold Master <input, dataport | output, error>,
55            manifold Worker(event, event, event, event) )
56  {
57    save *.
58
59    auto process master is Master.
60
61    begin: terminated(master).
62
63    create_pool: Create_Worker_Pool(master, Worker); post(begin).
64
65    finished: halt.
66  }
```

We first discuss the manner ProtocolMW (lines 54-66) followed by the manner Create_Worker_Pool (line 12-51) which is used by the first.

The actual manifold (named Main) that does the restructuring of the sequential source code invokes (as we will see in section 5) the ProtocolMW manner in its begin state. As a result, we enter the block of this manner (lines 56-66). Upon entering a block, first the statements in its local declaration part are performed (lines 57-59). Line 57 states that we can switch only to states in this block (i.e., the begin, create_pool or finished states respectively on lines 61, 63, and 65). Other possible event occurrences are saved. Line 59 defines a process instance of the formal manifold argument Master (line 54), calls it master, and states (through the keyword auto) that this process instance is to be automatically activated upon creation, and deactivated upon departure from the scope (i.e., departure from the block on line 66) in which it is defined (lines 56-66).

After performing the local declaration part of the entered block (lines 57-59) we automatically switch to the begin state. In the begin state (line 61) we wait until the already active process instance master terminates. Because we have mentioned master (as argument of the terminate primitive) in the state body, we also have made this state sensitive to events that are raised by master. Because master does not terminate, the net result of the action in the begin state is that we wait there until there is an event occurrence for which we have a matching event label. Because master, which is a process wrapper around the Fortran code (excluding the relaxations), after some sequential computation work arrives at the pre- and post-relaxations, it raises an event named create_pool to signal that it needs a workers-pool. This event pre-emptes the begin state and causes a state transition to the create_pool state (line 63). In this state the manner Create_Worker_Pool (lines 11-51) is called with the process instance master (created and activated on line 59), and the manifold Worker (which the protocol manner ProtocolMW itself has received as parameter on line 54) as its actual parameters. The manner Create_Worker_Pool conducts the workers in the pool and takes care that they can do their relaxation computations properly. When the workers in the pool are done, they die and the manner returns. Afterwards (denoted by the semicolon on line 63) we post the begin event so that we jump again to the begin state (line 61) where we wait for events. Another event will arrive because following some sequential computation, master either decides that it needs another workers-pool, (in which case, it raises the create_pool event, again), or it decides that it is done and raises the event finished. The finished event causes a state transition to the finished state (line 65), in which the primitive action halt effectively returns the flow of control from the manner to its caller.

The manner Create_Worker_Pool (lines 11-51) called on line 63 works as follows. Upon entering its block, first the statements in its local declaration part are performed (lines 15-23). Line 15 is a declarative statement which states that we can switch only to states specified in this block (lines 14-51). Line 16 is another declarative statement which states that death events can be removed from the event memory of the executing manifold instance, upon departure from the block (at line 51). On lines 18-19 we create and activate two process instances, respectively named now and t, of the predefined manifold variable, and initialize them with 0. We use these variables respectively for counting the number of created instances of the Worker manifold (we count them on line 34) and for counting the number of dead workers (by counting their death_worker events on line 42). On line 21, a local event named death_worker is declared. Because it can happen that both the events create_worker and rendezvous are available in the event memory of the executing manifold instance which calls this manner, we state with the priority declarative statement that jumping to the create_worker state has a higher priority than jumping to the rendezvous state.

The first state we visit in this manner is the begin state (line 25). There, we do the following: we print the message "begin" on the screen to indicate that we are in this state; we state by the primitive action preemptall that all events for which we have a handling state label can preempt the begin state; and we wait for events. An event will come soon, because master is expected to raise the event create_worker every time it wants another worker in the workers-pool. This event pre-empts the begin state and causes a state transition to the create_worker state.

In the create_worker state (lines 27-37) a number of workers are set to work in a workers-pool. The body of this state is a block. In its local declaration, we use the hold statement on line 28 so that we can handle events coming from Worker instances outside the scope in which those instances are known; otherwise, the instances of Worker are known only in the block in which they are defined (lines 27-37). On line 30, we create a process named worker. The four parameters used in the instantiation are respectively the local event death_worker (line 21) and the global events x, xx, and xxx, defined in the header file protocolMW.h (line 7). The declarative statement on line 32 states that all stream connections between the output port of worker and the input port the master (this input port is named dataport) must be of type KK (i.e., Keep-Keep). In the begin state of the state create_worker, the stream configuration on line 36 is constructed and we wait for events from the master (create_worker and rendezvous are possible events). In the stream configuration we see that the process identification of worker (denoted by &worker) is sent through a stream (the first → on line 36) to master. The master receives this reference to worker and sends all the information worker requests (by raising the x, xx, and xxx events) through a stream (the second → on line 36) to worker. The worker process promptly reads the information it receives from master, does its job, and if it is a remote worker, sends its the computed results through a stream (the third → on line 36) to the dataport port of master. The master process reads this and stores the results in the global master space. Due to the word IDLE (line 36) we stay in the state on line 34 until master again raises a create_worker event. This event pre-emptes this begin state (line 34) which dismantles the streams in this state and causes a state transition to the create_worker state where the whole sequence start again. Dismantling of the streams means, in this case, that all the streams on line 36 are broken at their sources (because they have the default type BK) with the exception of the stream for which the worker is the source; this stream is KK (see line 32) and must stay intact because when the worker is a remote worker this stream is used to transport its computed results to the master. This is how all workers are created and set to work in the pool.

The next event to be handled is the rendezvous event. This event is raised by master after it reads the computed results of the remote workers and causes a state switch to the rendezvous state which has two (sub)states: the begin state (line 40) and the death_worker state (line 42). In its begin state, we wait for the death_worker events. Each time a death_worker is detected, it is counted (line 42). As long as we have less death_worker events than the number of created workers (i.e., the value of now on line 34) we post the begin event (line 44) which causes a state switch back to the begin state (line 40) where we wait for other death_worker events. Otherwise, we post end (line 46) which causes a state switch to the end state (line 50). In this state we print a message on the screen, raise the event a_rendezvous, and the Create_Worker_Pool manner returns.

5 The Test

In this section we test the protocol with a toy application. We can arrange the computation in this application according to the schema in figure 2. Also for this application, we have chosen a global space that consist of two parts: a fixed part (an array of length three, initialized as 1, 2, and 3) and an non-fixed part (also an array of length three initialized in the same way). We have defined the following operations on the non-fixed array:

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

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

It is clear that the operation (a) cannot be done concurrently element-wise, whereas the operation (b) can easily be done element-wise by different workers in a concurrent fashion, each worker adding a fixed-part array element to its corresponding non-fixed part array element. With these two operations, it is simple to write a little program according to the schema given in figure 1. The initialization in the preamble (see figure 1) consists of the initializations of the fixed-part and non-fixed-part arrays. For the "heavy computations that cannot be done concurrently" (see again figure 1) we use operation (a). All other computations in that figure are (b) operations. In our test example we set the array length to three and the N in figure 1 is set to four (so we perform successively the operations (a, b, a, b, a, b, a, b, a, b, a, a) on the non-fixed-part array). Running this simple toy application and printing the initial values of the non-fixed array and its result after each operation gives the following output.

```
1      2      3
```

```
   4      6     9
   5      8    12
  17     25    37
  18     27    40
  58     85   125
  59     87   128
 187    274   402
 188    276   405
 593    869  1274
1867   2736  4010
```

Below, we give the **MANIFOLD** program in which we use the master/worker protocol `ProtocolMW` of section 4.2 in order to restructure the sequential version of our toy application into a concurrent one.

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

We briefly explain this source code. On lines 7 and 9 we declare respectively the worker manifold wo and the master manifold ma which are both written in ANSI C. We have implemented the master and the worker in such a way that they fully comply with `ProtocolMW`. Lines 14-17 define the manifold named `Main`, which has only one state: the `begin` state. In this state, an instance of the `ProtocolMW` manifold (its prototype is stored in the header file on line 3) is created and activated just by calling `ProtocolMW`, with the master and the worker as its actual arguments.

After this, the instance of `Main` terminates, and the instance of the protocol `ProtocolMW`, the instance of master ma and all the necessary instances of the worker wo, run concurrently.

The mapping of process instances into task instances (the so-called task composition stage) and the mapping of tasks to hosts (the so-called run-time configuration stage) are considered to be separate stages in the application construction. The mapping of process instances into task instances is described in a file which is the input for the **MANIFOLD** linker **MLINK**. For our toy application, we specify this file such that each worker is housed in a different task instance. The mapping of tasks to hosts is also described in a file which is the input for the **MANIFOLD** runtime configurator **CONFIG**. For our toy application, we have the following file:

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

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

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.

We have run this example on the above cluster of workerstations. The output is below[3].

```
sampan 262155 113 ptest ma ptest.ato.c 68 ->    1    2    3
sampan 262155 113 ptest ma ptest.ato.c 68 ->    4    6    9
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 25 -> begin
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
sampan 262155 263 ptest wo ptest.ato.c 215 ->
  I am a local worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
opduwer 786437 64 ptest wo ptest.ato.c 224 ->
  I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
pont 524289 64 ptest wo ptest.ato.c 224 ->
  I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW:
  Main protocolMW.m 50 -> rendezvous acknowledged
sampan 262155 113 ptest ma ptest.ato.c 68 ->    5    8   12
sampan 262155 113 ptest ma ptest.ato.c 68 ->   17   25   37
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 25 -> begin
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
sampan 262155 936 ptest wo ptest.ato.c 215 ->
  I am a local worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
opduwer 786437 83 ptest wo ptest.ato.c 224 ->
  I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
pont 524289 83 ptest wo ptest.ato.c 224 ->
  I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool: ProtocolMW:
  Main protocolMW.m 50 -> rendezvous acknowledged
sampan 262155 113 ptest ma ptest.ato.c 68 ->   18   27   40
sampan 262155 113 ptest ma ptest.ato.c 68 ->   58   85  125
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 25 -> begin
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
sampan 262155 1609 ptest wo ptest.ato.c 215 ->
  I am a local worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
opduwer 786437 102 ptest wo ptest.ato.c 224 ->
  I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
pont 524289 102 ptest wo ptest.ato.c 224 ->
  I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 50 -> rendezvous acknowledged
sampan 262155 113 ptest ma ptest.ato.c 68 ->   59   87  128
sampan 262155 113 ptest ma ptest.ato.c 68 ->  187  274  402
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 25 -> begin
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
sampan 262155 2282 ptest wo ptest.ato.c 215 ->
  I am a local worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
opduwer 786437 121 ptest wo ptest.ato.c 224 ->
  I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 35 -> create_worker: begin
pont 524289 121 ptest wo ptest.ato.c 224 ->
  I am a remote worker
sampan 262155 87 ptest Create_Worker_Pool:
  ProtocolMW: Main protocolMW.m 50 -> rendezvous acknowledged
sampan 262155 113 ptest ma ptest.ato.c 68 ->  188  276   405
sampan 262155 113 ptest ma ptest.ato.c 68 ->  593  869  1274
sampan 262155 113 ptest ma ptest.ato.c 68 -> 1867 2736  4010
```

Each of these output lines has the following structure. It starts with a long label followed by a -> before the actual message. The label shows, respectively, the machine on which the task instance runs, the identification of the task instance, the identification of the process instance, the name of the task, the name of the manifold, the name of the **MANIFOLD** source file, and the line number where the message is produced. With such a label in front of an actual

[3]Due to the two–column format of this paper, an indented line forms a whole with the previous line.

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

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

We ran our example on an SGI Origin 2000 multiprocessor machine with 32 processors, and also on a cluster of three SGI O2 single-processor machines. In the Origin 2000 we have 32 MIPS R12000 processors as CPUs plus MIPS R12010 and MIPS R12010 floating point coprocessors. In the O2 we have a MIPS R5000 processor as CPU plus a floating point co-processor. The performance results are in table 1. All experiments were done in quiet periods during normal working days. This means that we do not have a guaranty that we are the only user, which is a realistic assumption in any real comtempory computing environment. However, this also means that we should be careful to draw firm conclusions from these measurements. The different elapsed times for the sequential version on the multi-processor machine and on the cluster of workstations (in this case the cluster consist of a single machine) is due to the quicker hardware of the multi-processor machine. During the run on the multi-processor machine the weighted cpu percentages measured 270% which means that our application (with three workers processes) kept 2.7 of the 32 processors busy working. Because $32m * 2.7 \approx 86m$, this suggests that MANIFOLD can coordinate our toy application on the multi-processor machine without much overhead.

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

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

6 The Restructuring

Using the coordination module `ProtcolMW`, we can construct the following two MANIFOLD programs. These two programs change the original sequential code of our sparse-grid and semi-sparse-grid applications to their respective concurrent versions.

```
// sparse_model.m

//pragma include "aw.h"

#include "protocolMW.h"

manifold w_pointgsgr(event, event, event, event) atomic {internal.}.

manifold w_sparse() port in input. port in dataport.
                    port out output. port out error.
  atomic {internal. event create_pool, create_worker,
          rendezvous, a_rendezvous, finished, x, xx, xxx.}.

/*************************************************************/
manifold Main
{
  begin: ProtocolMW(w_sparse, w_pointgsgr).
}
```

```
// semi_sparse_model.m

//pragma include "aw.h"

#include "protocolMW.h"

manifold w_pointgsgr(event, event, event, event) atomic {internal.}.

manifold w_semi_sparse() port in input. port in dataport.
                         port out output. port out error.
  atomic {internal. event create_pool, create_worker,
          rendezvous, a_rendezvous, finished, x, xx, xxx.}.

/*************************************************************/
manifold Main
{
  begin: ProtocolMW(w_semi_sparse, w_pointgsgr).
}
```

The master and worker manifolds used as parameter of the protocol are both C functions (wrappers) that call the original Fortran subroutines (8000 lines) of the sequential program. The master and the worker behave in such a way that fully complies with the protocol `ProtocolMW`. For a stepwise description of their behavior see [7].

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 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.

Due to space limitation, for a detailed discussion of the performance of the restructured application we refer to [7] which is available online.

7 Conclusions

Our cut-and-paste restructuring essentially consists of picking out the computation subroutines in the original Fortran 77 code (the cut), and gluing them together with coordination modules written in MANIFOLD (the paste). No rewriting of, or other changes to, these subroutines is necessary: within the new structure, they have the same input/output and calling sequence conventions as they had in the old structure, and still manipulate the same global Fortran-common data arrays. The MANIFOLD glue modules, representing a master/worker protocol, are separately compiled programs that have no knowledge of the computation performed by the Fortran modules – they simply encapsulate the protocol necessary to coordinate the cooperation of the computation modules running in a parallel/distributed computing environment.

It is remarkable that we can realize the master/worker protocol in such a generic way where the master and the worker manifolds themselves are parameters of the protocol. With the possibility of using different manifolds as actual values for the formal manifold parameters of another manifold, we can easily build meta coordinators in MANIFOLD.

The unique property of MANIFOLD which enables such high degree of modularity is inherited from its underlying IWIM model in which the communication is set up from the *outside*. The core relevant concept in the IWIM model of communication is isolation of the computational responsibilities from communication and coordination concerns, into separate, pure computation modules and pure coordination modules. This is why the MANIFOLD modules in our example can coordinate the already existing computational Fortran subroutines, without any change. The master and worker manifolds used in the concurrent version only call C functions which are in fact (wrappers around) Fortran subroutines of the sequential program.

It is not so remarkable that sequential programs having a similar structure, but performing different algorithms (in our case the sparse-grid algorithm, semi-sparse-grid algorithm, and our toy algorithm) can be coordinated in a similar fashion. What is more interesting, as illustrated in our examples, is that we are able to abstract away the details of the computations; that it is possible to focus on the invariant (hidden) properties of seemingly very different programs, and that we can compile those invariant properties as coordination patterns in MANIFOLD. In fact, we compile structure. As in our examples, this same coordination structure (compiled MANIFOLD coordinators) can transparently run the same computation modules on parallel shared-memory or distributed cluster of workstation platforms. The nice thing in this distillation process is that we end up with *one* tangible piece of code that represents the common coodination structure. Such glue modules (coordinators) can then be compiled separately and stored in what we may call a "protocol library", ready for reuse.

References

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

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

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

[4] F. Arbab, C. Blom, F. Burger, and C. Everaars. Reusable coordinator modules for massively concurrent applications. *Software: Practice and Experience*, 28(7):703–735, June 1998. Extended version.

[5] C. Everaars and F. Arbab. *An Introduction into the Coordination Language Manifold.* CWI, Amsterdam, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 2001. to appear.

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

[7] C. Everaars, F. Arbab, and B. Koren. Parallel, distributed-memory implementation of sparse-grid methods for three-dimensional fluid-fbw computattions. Technical Report SEN-R0039, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, December 2000. Available online http://www.cwi.nl/static/publications/reports/SEN-2000.html.

[8] C. 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.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, September 1994.

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

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

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

[13] B. Nicols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Cebastopol, CA, 1996.

[14] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, San Diego, 2001.

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