Restructuring sequential Fortran code into a parallel/distributed application

C.T.H. Everaars, F. Arbab, F.J. Burger

Computer Science/Department of Interactive Systems

Report CS-R9628 July 1996

ISSN 0169-118X

Restructuring Sequential Fortran Code into a Parallel/Distributed Application

C.T.H. Everaars, F. Arbab, and F.J. Burger

ever@cwi.nl, farhad@cwi.nl, freek@cwi.nl

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

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 the new structure, the investment required for the re-discovery of the details of what they do can be spared. The resulting renovated software can then take advantage of the improved performance offered by modern parallel/distributed computing environments, without rethinking or rewriting the bulk of their existing code.

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

CR Subject Classification (1991): D3.3, D.1.3, D.3.2, F.1.2, I.1.3.
AMS Subject Classification (1991): 68N15, 68Q10.
Keywords and Phrases: distributed computing, parallel computing, coordination languages, models of communication, software renovation

1. Introduction

A key area in software modernization is renovating aging software systems to take advantage of the parallel and distributed computing environments of today. 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. Modernization of both types of software, however, is often subject to a common constraint: their owners or developers are unwilling to invest the effort required to use a different language, or even to do a fine-grain restructuring of their code.

A workable approach for modernization of such software is coarse-grain restructuring. If large sections, e.g., entire subroutines, of legacy code can be plugged into the new structure, the investment required for the re-discovery of the details of what they do can be spared. Analogously, if programmers are not forced to rethink and rewrite the details of the programs they currently produce, they would be more willing to use additional tools and techniques to parallelize and/or distribute their applications.

The new brand of coordination languages[10] presents a viable approach to this kind of software modernization. In this paper we describe one experiment in which a new coordination language, called MANIFOLD, was used to restructure an existing Fortran 77 program. The original Fortran 77 code was developed at CWI by a group of researchers in the department of Numerical Mathematics, within the framework of the BRITE-EURAM Aeronautics R&D Programme of the European Union. It implements their multi-grid solution algorithm for the Euler equations representing three-dimensional, steady, compressible flows[12]. They found their full-grid-of-grids approach to be effective (good convergence rates) but inefficient (long computing times). As a remedy, they looked for methods to restructure their code to run on multi-processor machines and/or to distribute their computation over clusters of workstations. Not all software that can (or is re-structured to) run on a multi-processor parallel platform is a good candidate for distributed computing. As it turns out, this program is a good candidate for distributed computing because its relaxation on the separate highest-level grids can proceed on different hosts with virtually no communication at all. Naturally, we intended to preserve and take advantage of this desirable characteristic in our restructuring.

In this paper we describe the restructuring method we used for this program. 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 Fortran program here that has the same logical design and structure as the original program, but whose computation is reduced to a few trivial arithmetic operations. Our restructuring essentially consists of picking out the computation subroutines in the original Fortran 77 code, and gluing them together with coordination modules written in MANIFOLD. 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 common data arrays. The MANIFOLD glue modules are separately compiled programs that have no knowledge of the computation performed by the Fortran modules – they simply encapsulate the protocol necessary to coordinate the cooperation of the computation modules running in a parallel/distributed computing environment. MANIFOLD is a coordination language developed at CWI (Centrum voor Wiskunde en Informatica) in the Netherlands. It is very well suited for managing complex, dynamically changing interconnections among sets of independent concurrent cooperating processes.

The rest of this paper is organized as follows. In section 2 we give a brief introduction to the MANIFOLD language and its underlying communication model. In section 3 we start with the inevitable "Hello World!" program to show some of the syntax and semantics of MANIFOLD. It is beyond the scope of this paper to present the details of the syntax and semantics of the MANIFOLD language[1]. In section 4 we present the sequential Fortran program and in section 5 we describe our restructuring using MANIFOLD. The interface between the MANIFOLD modules and the Fortran subroutines is a number of C functions which from the MANIFOLD point of view, represent the *atomic* processes and the atomic manners of the application. This interface is described in section 6. Finally, the conclusion of the paper is in section 7.

2. The Manifold Coordination Language

In this section, we briefly introduce MANIFOLD: a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes[1].

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 cooperating with other processes through MANIFOLD in a concurrent application.

The MANIFOLD system consists of a compiler, a run-time system library, a number of utility programs, libraries of builtin and predefined processes[2], a link file generator called MLINK and a run-time configurator called CONFIG. The system has been ported to several different platforms (e.g., SGI 5.3, SUN 4, Solaris 5.2, and IBM SP/1). MLINK uses the object files produced by the

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

(MANIFOLD and other language) compilers to produce link files needed to compose the application executable files for each required platform. At the run time of an application, CONFIG determines the actual host(s) where the processes which are created in the MANIFOLD application will run.

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

2.1 Conceptual Model

MANIFOLD is based on the *Idealized Worker Idealized Manager* (IWIM) model of communication[3]. In this section we briefly describe this model and discuss its advantages over the *Targeted-Send/Receive* (TSR) model on which object-oriented programming models and tools such as PVM[9], PARMACS[11], and MPI[13, 7] are based.

The basic concepts in the IWIM model (thus also in MANIFOLD) are *processes*, *events*, *ports*, and *channels* (in MANIFOLD called *streams*). We discuss these concepts in sections 2.2 through 2.4. Unlike the TSR model, there is no way in the IWIM model for a process to explicitly send a message to or receive a message from another process; (normal) worker processes can only produce their output, consume their input, and broadcast events. It is the job of special manager processes to coordinate the communication among their worker processes by establishing a dynamically changing data-flow network of point-to-point connections. We can illustrate the differences between the TSR and the IWIM models through the following simple example.

Consider an application that consists of the two processes p and q. The partial results $m1$ and $m2$ produced by p are needed by q, which in turn uses them to compute another result, m, to be used by p.

In the TSR model this abstract communication scenario results in the following TSR pseudo code.

```
*********************
* A TSR pseudo code *
*********************

process p
begin
  compute m1
  send m1 to q

  compute m2
  send m2 to q

  do other things

  receive m
  do other computation using m
end

process q
begin
  receive m1

  receive m2

  (let z be the sender of m1 and m2)

  compute m using m1 and m2
  send m to z
end
```

In the IWIM model this scenario is expressed as the following IWIM pseudo code.

```
**********************
* A IWIM pseudo code *
**********************

process p
begin
  compute m1
  write m1 to output port o1

  compute m2
  write m2 to output port o2

  do other things

  read m from the input port i1
  do other computation using m
```

```
17 end
18
19 process q
20 begin
21   read m1 from input port i1
22
23   read m2 from input port i2
24
25   compute m using m1 and m2
26
27   write m to output port o1
28 end
29
30 process c
31 begin
32   create the processes p and q
33
34   create the channel p.o1 -> q.i1
35   create the channel p.o2 -> q.i2
36   create the channel q.o1 -> p.i1
37
38   follow some termination protocol
39 end
```

Some of the significant differences between the above two pieces of pseudo code are summarized below:

- The cooperation model in the TSR pseudo code is implicit whereas in the IWIM pseudo code it is explicit.

  This TSR pseudo code is simultaneously both a description of what computation is performed by p and q, and a description of how they cooperate with each other. The communication concerns (lines 8, 11, 15, 21, 23, 28) are mixed and interspersed with computation (lines 7, 10, 16, 27). Thus, in the final source code of the application, there will be no isolated piece of code that can be considered as the realization of its cooperation model.

  In the IWIM pseudo code we see that all the communication concerns are moved out of p and q into an isolated piece of code that is the process c. Note that in this code p and q do not explicitly communicate with each other as is the case in the TSR version. Here p and q are treated as black-box workers that can only read or write through the openings (called ports) in their own bounding walls. It is a third manager or coordinator process, c, that is responsible for setting up the communication channels between the different ports of p and q. On the lines 34-36 we use the notation $p.i$ to refer to the port i of the process instance p; e.g., line 34 states that a channel is created between the *o1* output port of p and the *i1* input port of q.

- The separation of computational concerns and communication concerns in the IWIM model, leads to two types of processes in this model: worker processes and manager (or coordinator) processes. In the TSR model all processes have the same hybrid form.

  In the IWIM pseudo code, p and q can be regarded as "ideal" workers. They do not know and do not care where their input comes from, nor where their output goes to. They know nothing about the pattern of cooperation in this application; they can just as easily be incorporated in any other application, and will do their job provided that they receive "the right" input at the right time.

  The process c is an "ideal" manager. It knows nothing about the details of the tasks performed by p and q. Its only concern is to ensure that they are created at the right time, receive the right input from the right sources, and deliver their results to the right sinks. It also knows when additional new process instances are supposed to be created, how the network of communication channels among processes must change in reaction to significant event occurrences, etc. (none of which is actually a concern in this simple example).

- The separation of computation and coordination responsibilities into distinct worker and manager processes in the IWIM model enhances their re-usability.

  The fact that an ideal worker does not know and does not care where its input comes from, nor where its output goes to, weakens its implicit dependence on its environment, strengthens its modularity, and enhances its re-usability. Also, the fact that an ideal manager process knows nothing about the computation performed by the workers it coordinates, makes it generic and re-usable. In the IWIM model, the cooperation protocols for a concurrent application can be developed modularly as a set of coordinator processes. It is likely that some of such ideal managers, individually or collectively, can be used in other applications, coordinating very different worker processes, producing very different results; as long as their cooperation follows the same protocol, the same coordinator processes can be used (see [5] for more details and a concrete example). Modularity and re-usability of the coordinator processes also enhance the re-usability of the resulting software.

- The IWIM pseudo code is easier to adapt to new requirements than the TSR pseudo code.

  The "Targeted Send" in the TSR pseudo code creates a stronger coupling between the processes than is really necessary. Because of this, the TSR pseudo code is less easy to adapt to new requirements than the IWIM pseudo code. This becomes clear when we notice the asymmetry between send and receive operations in the TSR model. Every send must specify a target for its message, whereas a receive can receive a message from any anonymous source.[2] In our example, p must know q, otherwise, it cannot send a message to it. The proper functioning of p depends on the availability of another process in its environment that (1) must behave as p expects (i.e., be prepared to receive $m1$ and $m2$), and (2) must be accessible to p through the name q. On the other hand, p does not (need to) know the source of the message it receives as m. And this ignorance is a blessing. If after receiving $m1$ and $m2$, q decides that the final result it must send back to p is to be produced by yet another process, x, p need not be bothered by this "delegation" of responsibility from q to x.

  We can better appreciate the significance of the asymmetry between send and receive in a tangible form when we compare the processes p and q with each other. The assumptions hard-wired into q about its environment (i.e., availability and accessibility of other processes in the concurrent application) are weaker than those in p. The process q waits to receive a message $m1$ from any source, which it will subsequently refer to as z; expects a second message $m2$ (which it can verify to be from the same source, z, if necessary); computes some result, m; and sends it to z. The behavior of the process p, on the other hand, cannot be described without reference to q. The weaker dependence of q on its environment, as compared with p, makes it a more reusable process that can perform its service for other processes in the same or other applications.

  Note, however, that q is not as flexible as we may want it to be: the fact that the result of its computation is sent back to the source of its input messages is something that is hard-wired in its source code, due to its final targeted send. If, perhaps in a different application environment, we decide that the result produced by q is needed by another process, y, instead of the same process, z, that provides it with $m1$ and $m2$, we have no choice but to modify the source code for q. This is a change only to the cooperation model in the application, not a change to the substance of what q does. The unfortunate necessity of modification to the source code of q, in this case, is only a consequence of its targeted send.

The IWIM model avoids the negative influence of the TSR model on the program structure[4]. Specifically, in our case study reported here, it allows us to re-use the bulk of the existing Fortran code without *any* modification: as we demonstrate in this paper, nothing (e.g., send/receive primitives) is added to or modified in the re-used Fortran code. This degree of re-usability is not possible with TSR-based tools and languages such as PVM.

[2]In some message passing models, an optional source can be specified in a receive. Although this makes receive look symmetric to send in its appearance, semantically, they are still very different. A send is semantically meaningless without a target. On the other hand, a receive without a source is always meaningful. The function of the optional source specified in a receive is to filter incoming messages based on their sources. This is only a convenience feature – the same effect can also be achieved using an unrestricted receive followed by an explicit filtering.

2.2 Processes

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

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

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 a sequence of bits, grouped into (variable length) *units*.

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

There are four basic stream types designated as BB, BK, KB, and KK, each behaving according to a slightly different protocol with regards to its automatic disconnection from its source or sink. Furthermore, in MANIFOLD, the BK and KB type streams can be declared to be *reconnectable*. See [2] or [3] for details.

2.4 Events and State Transitions

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

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

Upon transition to a state, the primitive actions specified in its body are performed atomically in some non-deterministic order. Then, the state becomes *preemptable*: if the conditions for transition to another state are satisfied, the current state is preempted, meaning that all streams that have been constructed are dismantled and a transition to a new state takes place. The most important primitive actions in a simple state body are (1) creating and activating processes, (2) generating event occurrences, and (3) connecting streams to the ports of various processes.

3. Hello World!

Consider a simple program to print a message such as "Hello World!" on the standard output. The MANIFOLD source file for this program contains the following:

```
manifold printunits import.

auto process print is printunits

manifold Main
{
  begin: "Hello World!" -> print.
}
```

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

The second line of code defines a new instance of the manifold `printunits`, calls it `print`, and states (through the keyword `auto`) that this process instance is to be automatically activated upon creation, and deactivated upon departure from the scope wherein it is defined; in this case, this is the end of the application. Because the declaration of the process instance `print` appears outside of any blocks in this source file, it is a global process, known by every instance of every manifold whose body is defined in this source file.

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

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

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

Having made the stream connection between the two processes, `main` now waits for all stream connection made in this state to break up (on at least one of their ends). The stream breaks up, in this case, on its source end as soon as the string constant delivers its unit to the stream and dies.

Since there are no other event occurrences in the event memory of main, the default transition for a state reaching its end (i.e., falling over its terminator period) now terminates the process main.

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

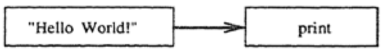

Figure 1: "Hello World" in Manifold

Note that our simple example, here, consists of three process instances: two worker processes, a character string constant and print, and a coordinator process, main. Figure 1 shows the relationship between the constant and print, as established by main. Note also that the coordinator process main only establishes the connection between the two worker processes. It does *not* transfer the units through the stream(s) it creates, nor does it interfere with the activities of the worker processes in other ways.

4. The Simplified Fortran Code

In this section we present our simplified Fortran code as distilled from the original program to solve three-dimensional, steady, compressible Euler equations described in [12]. The Fortran code consists of a number of subroutines that manipulate a common data structure. Our goal is to show the restructuring of the sequential application to a parallel/distributed one. For our purpose, it is sufficient to use a very simple global data structure, e.g., an array with ten integers. Of course, the original global data structure is more complex than our trivial array, but dealing with larger numbers of global variables and larger array sizes is too distracting here. In fact, our restructuring imposes no limit on the complexity of the data structures used in the application.

The Fortran program consists of a data definition section, a main program, and five subroutines. One of the subroutines, priglo, simply prints out our global array and is used for tracing and to report the results. The doseqa, dopar, and doseqb subroutines contain three steps of the computation performed on the global data array. The scan subroutine is used to visit selected segments of the global array on which these computations are to be performed. It receives, as its arguments, the array indices identifying the selected array segment and the subroutine that performs the appropriate computation on this segment. This Fortran program is contained in a source file, called seq_model.f, which is shown in the following listing:

```
c_____________________________________________________________
      block data aname
      include 'global.i'
      data ar /1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
      end
c_____________________________________________________________
      program main
      include 'global.i'
      external doseqa, doseqb, dopar
      call priglo
      call scan(1, 10, doseqa)
      call priglo
      call scan(3, 8, dopar)
      call priglo
      call scan(1, 10, doseqb)
      call priglo
      end
c_____________________________________________________________
      subroutine priglo
      include 'global.i'
      integer i
      write(*, '(10i3)') (ar(i), i = 1, 10)
      end
c_____________________________________________________________
      subroutine doseqa(i)
      integer i
      include 'global.i'
      ar(i) = ar(i) - ar(5)
```

```
29          end
30 c
31          subroutine dopar(i)
32          integer i
33          include 'global.i'
34          ar(i) = ar(i) ** 2
35          end
36 c
37          subroutine doseqb(i)
38          integer i
39          include 'global.i'
40          ar(i) = ar(i) + ar(4)
41          end
42 c
43          subroutine scan(from, to, doIt)
44          integer from, to
45          external doit
46          integer i
47          do 10 i = from, to
48            call doit(i)
49 10       continue
50          end
```

The include file global.i referred to in the above listing contains the following:

```
integer ar(10)
common /glodat/ ar
```

Although the include file facility is not part of the Fortran 77 language and thus, in principle, is not portable, most Fortran 77 compilers support this facility.

Following is the result produced by running the executable produced by the f77 compiler on the seq_model.f source file:

```
 1  2  3  4  5  6  7  8  9 10

-4 -3 -2 -1  0  6  7  8  9 10

-4 -3  4  1  0 36 49 64  9 10

-3 -2  5  2  2 38 51 66 11 12
```

The global data structure is first initialized (line 4 in the listing, above) in a Fortran block data program. Once the main program starts, it first calls priglo to print out the contents of the global data structure (line 10). This produces the first line of our output. On line 11, the global array is visited sequentially from index 1 up to and including index 10. The computation performed on the array elements with these indices is contained in the subroutine doseqa. Another call to priglo shows the result of this computation in the second line of our output.

The computation performed by doseqa is shown on line 28. Observe that there is a dependency between ar(i) and some other element in the global data structure, namely ar(5). The final result depends on the order in which each of the computations in doseqa is performed, and thus, the computations done in line 11 are not suitable for parallelization/distribution. This is not the case for the computations done by the call made to dopar on line 13. Squaring of the array elements 3 up to and including 8 (line 34 in dopar) can in principle be done concurrently. A call to priglo on line 14 shows the results in the third line of our output. Line 15 performs another sequence of computations which must be carried out in a sequential order, because of their dependency on ar(4) (see line 40 in doseqb). Another call to priglo on line 16 shows the final results on the last line of the our output.

5. Restructuring of the Fortran Code

In this section we describe the restructuring of the Fortran code presented in the previous section into a parallel and distributed application. The crux of our restructuring is to allow the computations done on every single array element in the call to scan on line 13 in the above listing, to be carried out in a separate process. 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.

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

Observe also that if the computation represented by doseqa is trivial or non-existent, it is not necessary to send copies of the initial global array to each task at all: the initial global array can be directly initialized at link time, such that each task instance would have the array "on-board" at its start-up. The computation done in doseqa can then be repeated in every task instance separately.

For simplicity, in the restructuring presented in this paper we assume there is only one MANIFOLD task which contains several MANIFOLD processes. The restructured program we present here is thus not suitable, as it is, for distributed computation. However, the additional book-keeping and the simple extra communication necessary to run this example on a distributed platform is straight-forward and (due to space limitation) is beyond the scope of this paper. Note that the restructured program we present here, nevertheless, *does* improve the performance of the application on a parallel platform. For instance, on a multi-processor SGI machine, thirty or so threads in the same task can concurrently run dopar, each on a different array element. At most n of these threads can actually be running (truly) in parallel with each other, where n is the number of processors on the machine.

The MANIFOLD source code that coordinates our restructured application is contained in the file `model.m`, listed below:

```
manner priglo atomic.
manner wdoseqa atomic.
manifold dopar(port in, event, event) atomic (internal.).
manner wdoseqb atomic.

manifold printunits import.
manifold variable(port in) import.
manifold variable import.

#define IDLE terminated(void)
#define FROM 3
#define TO 8
#define NOD (TO - FROM + 1)

/*****************************************************/
manner doit(port in p, event ready)
hold dopar.
{
  event init.
  hold dopar.

  begin: (dopar(p, init, ready), IDLE).
  init:.
}

/*****************************************************/
manifold Main
{
  event e, wait, goon.
  auto process i is variable.
  auto process t is variable(0).

  begin: priglo;
         wdoseqa;
         priglo;
         for i = FROM while i <= TO
           step i = i + 1 do doit(i, e);
         post(wait).

   wait: (begin: preemptall.
             e.*: t = t + 1;
                  if (t == NOD) then post(goon)
                  else post(begin).
44          }.
45
46    goon: priglo;
47          wdoseqb;
48          priglo.
49 }
```

The object file obtained by compiling this MANIFOLD program must be linked with the object files obtained from a Fortran and a C compilation to produce an executable file. The contents of the necessary Fortran (`fmodel.f`) and C (`model.ato.c`) files are presented in section 6. 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, shown in section 4.

Lines 1-4 define the atomic manners `priglo`, `wdoseqa` and `wdoseqb`, which take no arguments, and the atomic manifold `dopar` which takes three arguments. These definitions simply represent the stubs for three C functions that will be called as subroutines, and a C function that will execute concurrently as a MANIFOLD process. This process (manifold `dopar`) takes a `port in` argument, and two `event` arguments.

The manners `priglo`, `wdoseqa` and `wdoseqb` correspond, respectively, to the Fortran subroutines `priglo`, `doseqa` and `doseqb` in the sequential Fortran program. The actual C code for these functions is contained in the source file `model.ato.c`, in section 6. The manifold `dopar` performs the same computation that the `dopar` subroutine in the sequential Fortran program performs, except that it does it on a single array element. The arguments of `dopar` are explained later.

Lines 6-8 declare the manifolds `printunits`, `variable` with one argument, and `variable` with no arguments. The keyword `import` states that the real definitions (i.e., the bodies) of these manifolds are contained in another source file (indeed, these manifolds are predefined in the MANIFOLD library).

The lines 10-13 define some preprocessor macros, in the same syntax as that of the C preprocessor. These macros define our symbolic constants.

Lines 27-49 define the `Main` manifold of our application. The body of `Main` (enclosed in the pair of braces on lines 28 and 49) contains three states: `begin`, `wait` and `goon`.

The body of the `begin` state (i.e. everything after the colon on line 33 up to the terminator period on line 38) sequentially calls `priglo`, `wdoseqa`, and `priglo`, again, before executing a loop, followed by posting an event (`wait`). The two `priglo` calls produce the first two output lines, before and after the sequential computation done in `wdoseqa`. As we will see in section 6, `priglo` really calls the original Fortran `priglo` subroutine, and `wdoseqa` is only a wrapper around the `scan` call on line 11 in the original sequential Fortran code.

The for-loop on the lines 36-37 calls the `doit` manner `NOD` times for the values of `i` ranging from `FROM` to `TO`. With our definitions for `FROM` (3) and `TO` (8), this means 6 calls to `doit` with `i` values ranging from 3 to 8. Each call to the manner `doit` (lines 16-24) creates a new instance of the manifold `dopar` as a separate process and passes it the value of `i` as its first parameter. An instance of `dopar` also receives two events, `init` and `ready` (whose corresponding actual parameter is the event `e` in the call to `doit` on line 37). An instance of `dopar` is expected to raise the event `init` as soon as it is done with its initialization, so that other instances of `doit` can be started up without running into conflicts with each other. Raising `ready` signals that the `doit` instance is done its job and is about to terminate.

After the creation and activation of the atomic process `dopar` on line 22, the `doit` manner waits (due to IDLE) until it detects the event `init`. This signals that `dopar` has completed its initialization (in our case, it has obtained the value of `i`) and other instances can be created without conflicts. In reaction to this event, `doit` makes a transition to a state with no body: the event is considered handled and the manner call returns. Thus, we proceed with the next iteration of the loop on the lines 36-37, which makes another call to `doit` with the next `i` value. Meanwhile, the instance(s) of `dopar` started up in the previous iteration(s) of this loop all proceed, in parallel, with their own computation.

When an instance of dopar completes its computation, it raises the event e (the actual value for ready) and terminates. In its wait state, main waits to receive exactly NOD occurrences of e before posting an event to make a transition to its goon state. Once in the goon state, main sequentially calls priglo before and after performing the computation done in wdoseqb, which produces the last two lines of our output. As we will see in section 6, wdoseqb is simply a wrapper for the scan call performed on line 15 of the original sequential Fortran code in section 4.

6. The Computation Modules

There must be a C function corresponding to every atomic manner and atomic manifold defined in a MANIFOLD program. The C functions for the atomic manifold dopar and the atomic manners priglo, wdoseqa, and wdoseqb used in our restructured program are defined in the source file model.ato.c, shown below:

```
 1 #include "AP_interface.h"
 2 #include "debug.h"
 3 /*******************************************************/
 4 int IntFromPort(AP_Port port)
 5 {
 6   int i, err;
 7   AP_Unit u;
 8   err  = AP_PortGetUnit(port, &u);              I(err) P(u)
 9   err = AP_FetchInteger(u, &i);                 I(err)
10   err = AP_DeallocateUnit(u);                   I(err)
11   return i;
12 }
13 /*******************************************************/
14 void wdoseqa(void)
15 {
16   extern void doseqa_(void);
17   wdoseqa_();
18 }
19 /*******************************************************/
20 void dopar(AP_Port pm, AP_Event init, AP_Event ready)
21 {
22   int input = AP_PortIndex("input");
23   int PlaceInDataStructure;
24   extern void scan_(int* to, int* from,
25                     void (*doit)(int* i) );
26   extern void dopar_(int* i);
27   PlaceInDataStructure = IntFromPort(pm);
28   AP_Raise(init);
29   scan_(&PlaceInDataStructure,
30         &PlaceInDataStructure, dopar_);
31   AP_Raise(ready);
32 }
33 /*******************************************************/
34 void wdoseqb(void)
35 {
36   extern void doseqb_(void);
37   wdoseqb_();
38 }
39 /*******************************************************/
40 void priglo(void)
41 {
42   extern void priglo_(void);
43   priglo_();
44 }
```

Observe that the C functions priglo, wdoseqa, and wdoseqb each directly calls the Fortran subroutine with its corresponding name. (On many platforms, a Fortran subroutine X can be called from C, as a C function named X_.)

The C function dopar needs to do a bit more because of parameter passing, and to comply with the initialization/termination protocol it is expected to abide by. It first obtains the value of its index from its first parameter (line 27) and then raises what it knows as the init event to signal that it is done with its initialization. Next (on the lines 29-30), it calls the Fortran scan subroutine, passing it the same index value as its first two parameters (to make it work on only one array element) and the Fortran subroutine dopar as its third. Upon return from scan, the C function dopar raises what it knows as the ready event before it terminates.

The source file fmodel.f, shown below, is the same as the original sequential Fortran program in section 4, except that the main program is deleted here, and two new subroutines are added. The new subroutines wdoseqa and wdoseqb, respectively, perform the scan calls on lines 11 and 15 in the main program of the original sequential Fortran code.

```
c________________________________________________
        block data aname
```

```
      include 'global.i'
      data ar /1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
      end
c_____________________________________________________________
      subroutine priglo
      include 'global.i'
      integer i
      write(*, '(10i3)') (ar(i), i = 1, 10)
      end
c_____________________________________________________________
      subroutine doseqa(i)
      integer i
      include 'global.i'
      ar(i) = ar(i) - ar(5)
      end
c_____________________________________________________________
      subroutine wdoseqa
      external doseqa
      call scan(1, 10, doseqa)
      end
c_____________________________________________________________
      subroutine dopar(i)
      integer i
      include 'global.i'
      ar(i) = ar(i) ** 2
      end
c_____________________________________________________________
      subroutine doseqb(i)
      integer i
      include 'global.i'
      ar(i) = ar(i) + ar(4)
      end
c_____________________________________________________________
      subroutine wdoseqb
      external doseqb
      call scan(1, 10, doseqb)
      end
c_____________________________________________________________
      subroutine scan(from, to, doit)
      integer from, to
      external doit
      integer i
      do 10 i = from, to
        call doit(i)
 10   continue
      end
```

7. Conclusions

Our experiment using MANIFOLD to restructure existing Fortran code into a parallel/distributed application indicates that this coordination language is well suited for this kind of tasks. The highly modular structure of the resulting application and its ability to use existing computational subroutines of the sequential Fortran program is remarkable. The atomic manners and manifold used in the parallel MANIFOLD version only call C functions which are in fact (wrappers around) Fortran subroutines of the sequential program.

The unique property of MANIFOLD that enables such high degree of modularity is inherited from its underlying IWIM model. The core relevant concept in the IWIM model of communication is isolation of computation responsibilities from communication and coordination concerns, into separate pure computation and pure coordination modules. This is why the MANIFOLD modules in our example can coordinate the activity of the set of computation worker processes which run the same Fortran subroutines without any change.

The modularity of MANIFOLD and the fact that a coordinator module cannot make any distinction between a computing process and another coordination module, allows application development or software renovation to proceed in a stepwise manner over a period of time. For example, a block of code can initially be plugged into a concurrent structure as a monolithic computing process, to obtain a running parallel/distributed application. As more experience is gained through running the new application, computation bottlenecks may be identified. This may lead to replacing some such monolithic blocks of code with more MANIFOLD modules that coordinate the activity of smaller blocks of computation code, in a new concurrent sub-structure.

This form of software renovation spares the burden of understanding the intricate details of existing computation code, which is often developed by very specialized researchers. For instance, in our example, we do not need to understand the internal working of the numerical algorithm used in **doseqa**, **doseqb**, or **dopar**, which in fact perform a collective symmetric point Gauss-Seidel relaxation for smoothing in the sparse-grid method[12]; nor do we need to understand in what fashion their subroutine **scan** really visits each point.

An added bonus of pure coordination modules is their re-usability: the same MANIFOLD modules developed for one application can often be used in other parallel/distributed applications with the same or similar cooperation protocol, regardless of the fact that the two applications may perform entirely different computations. A concrete example of this notion of re-usability is discussed in [5]. The usefulness of the IWIM model and, in particular, the MANIFOLD language in these and other applications ([8], [6]) has been very encouraging. The plumbing paradigm inherent in IWIM makes it easy to compose and recompose a MANIFOLD application and adapt it to new requirements. To enhance the effectiveness of this coordination language, we are presently developing a visual programming environment around MANIFOLD which takes advantage of its underlying plumbing paradigm[6].

REFERENCES

1. F. Arbab. Coordination of massively concurrent activities. Technical Report CS–R9565, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, November 1995. Available on-line http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.
2. F. Arbab. Manifold version 2: Language reference manual. Technical Report preliminary version, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1995.
3. F. Arbab. The IWIM model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Model*, volume 1061 of *Lecture Notes in Computer Science*, pages 34–56. Springer-Verlag, April 1996.
4. F. Arbab. The influence of coordination on program structure. In *submitted to HICSS-30*. IEEE, January 1997.
5. F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaars. Reusable coordinator modules for massively concurrent applications. In *Euro-Par '96*, Lecture Notes in Computer Science. Springer-Verlag, August 1996.
6. P. Bouvry and F. Arbab. Visifold: A visual environment for a coordination language. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *Lecture Notes in Computer Science*, pages 403–406. Springer-Verlag, April 1996.
7. Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An introduction to the MPI standard. Technical Report CS-95-274, University of Tennessee, January 1995.
8. C. T. H. Everaars and F. Arbab. Coordination of distributed/parallel multiple-grid domain decomposition. In *Proceedings of Irregular '96*, Lecture Notes in Computer Science. Springer-Verlag, August 1996.
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 and N. Carriero. Coordination languages and their significance. *Communication of the ACM*, 35(2):97–107, February 1992.
11. R. Hempel, HC. Hoppe, U. Keller, and W. Krotz. PARMACS v6.1 specification. Technical report, PALLAS GmbH, Hermulheimer Strasse 10, D-50321, August 1995.
12. B. Koren, P. W. Hemker, and P. M. de Zeeuw. Semi-coarsening in three directions for Euler flow computations in three dimensions. In H. Deconinck and B. Koren, editors, *Euler and Navier-Stokes Solvers Using Multi-Dimensional Upwind Schemes and Multigrid Acceleration*, Notes on Numerical Fluid Mechanics. Vieweg, Braunschweig, 1996.
13. The Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, May 1994. Available on-line http://www.mcs.anl.gov/mpi/mpi-report.ps.