Modernizing Existing Software: A Case Study *

C.T.H. Everaars
National Research Institute for Mathematics and Computer Science (CWI)
P.O. Box 94079, 1090 GB
Amsterdam, The Netherlands
Kees.Everaars@cwi.nl

F. Arbab
National Research Institute for Mathematics and Computer Science (CWI)
P.O. Box 94079, 1090 GB
Amsterdam, The Netherlands
Farhad.Arbab@cwi.nl

B. Koren
National Research Institute for Mathematics and Computer Science (CWI)
P.O. Box 94079, 1090 GB
Amsterdam, The Netherlands
Barry.Koren@cwi.nl

Categories and Subject Descriptors

D.1 Programming Techniques [**Concurrent Programming**]; D.2 Software Engineering [**Reusable Software**]; D.3 Programming Languages [**Languages Classifications**]: Concurrent, distributed, and parallel languages

General Terms

Design, experimentation, languages, performance

Keywords

Parallel and distributed computing, coordination languages, software renovation, software reusability, protocol library

ABSTRACT

In this paper, we discuss one of our experiments using the coordination language MANIFOLD to restructure an existing sequential numerical application into a concurrent application. The application was written in ANSI C and deals with a sparse-grid method for a transport problem. 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) with the help of a coordination language (the paste). We also give some performance results.

1. INTRODUCTION

A workable approach for modernizing 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 an existing sequential C code for computational fluid dynamics (CFD). This C source code deals with a time-dependent advection-diffusion problem solved with a sparse-grid technique and was developed at CWI by a group of researchers in the department of Modeling Analysis and Simulations, within the framework of the NWO-funded project "Sparse Grid Methods for Transport Problems". The developers of the program 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 the program results in *one* generally applicable coordinator module that can restructure the sequential program into a parallel application (which can run on a shared memory machine) as well as into a distributed application (which can run on a cluster of workstations).

The rest of this paper is organized as follows. In §2 we give a brief introduction to the MANIFOLD language. In §3 we present the simplified pseudo-program as distilled from the original ANSI C program, explore its structure and try to identify and isolate its software components. In §4, we describe the paste phase in the software renovation process and present our generic gluing modules written in the MANIFOLD coordination language. The actual restructuring of the original sequential program can be found in §5. In §6 we show how to run the concurrent version on a cluster of workstations and in §7 we give performance results. Finally, the conclusion of the paper is in §8.

2. THE MANIFOLD 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].

*Funding for this project was provided by the National Computing Facilities Foundation (NCF), under project number NRG 2001.06.

SC2004 Pittsburgh, Pennsylvania, USA

[1]For more information, refer to our html pages located at

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* [10]. MANIFOLD is a *complete* language (as opposed to a language extension, like Linda [9]) 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 the telephone call, processes in MANIFOLD (in this case *b* and *c*) do not explicitly send or receive messages to or 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 of the modules responsible for computation (the workers) from the modules responsible for coordination (the managers). This strengthens the modularity and enhances the re-usability of both types of modules (see [3, 1]).

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 languages. 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 the runtime of an application, CONFIG determines the actual host(s) where the processes that 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 [11], plus an inter-process communication facility roughly equivalent to a small subset of PVM [8].

For a performance comparison between MANIFOLD and the often used communication middleware PVM [8] we refer to [4]. For a general discussion about how a chosen coordination/communication tool (e.g., PVM) influences the structure of a computer program we refer to [3].

3. THE CUT

In this section we explore the structure of the ANSI C program of our sequential application . The program consists of a data definition section, a main program and some 33 subroutines with a total length of some 3500 lines. Instead of its full source code, we give only the relevant part of the C code for the sparse-grid method, viz., a schematized version of the main program, and the subroutine `subsolve`. With this small part of the C code we can explain the essential implementation aspects of the sparse-grid method, as well as its actual restructuring into a concurrent application.

```
 1 /* SeqSourceCode.c */
 2
 3 int root, level;
 4 double le_tol;
 5
 6 /* Declaration of the huge global data structure */
 7
 8 /*****************************************************/
 9 int main (int argc, char *argv[])
10 {
11   int i, j, lm, l;
12
13   /* Root level (i.e. refinement level
          of coarsest grid) */
14   root = atoi(argv[1]);
15   /* Additional refinement above the root level */
16   level = atoi(argv[2]);
17   /* The tolerance of the integrator. */
18   le_tol = atof(argv[3]);
19
20   /* Initialization data structure and
        some initial computations */
21
22   /* The heavy computational work  */
23   for (lm = level - 1; lm <= level; lm++) {
         /* loop over the grid level */
24     for (l = 0; l <= lm; l++) { /* loop over the
          grids belonging to a certain grid level */
25       subsolve(l, lm-l);
26     }
27   }
28
29   /* Prolongation work */
30   ...
31 }
32
33 /*****************************************************/
34 void subsolve (int l, int m)
35 {
36   /* Heavy computational work on grid (l, m) */
37   ...
38
39   /* The results are stored in
          the global data structure */
40   ...
41 }
```

http://www.cwi.nl/projects/manifold/manifold.html.

```
1 // protocolMW.m
2
3 #include "MBL.h"
4
5 #include "rdid.h"
6
7 #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) )
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).
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(
     process master <input, dataport | output, error>,
55   manifold Worker(event) )
56 {
57   save *.
58
59   begin: terminated(master).
60
61   create_pool: Create_Worker_Pool(master, Worker); post(begin).
62
63   finished: halt.
64 }
```

In the description of our protocol, we first discuss the manner (i.e., a parameterized subprogram) `ProtocolMW` (lines 54-64) followed by the manner `Create_Worker_Pool` (line 12-51) which is used by the former.

The actual manifold (named `Main`) that does the restructuring of the sequential source code invokes (as we see in §5) the `ProtocolMW` manner in its `begin` state. As a result, we enter the block of this manner (lines 56-64). Upon entering a block, the statements in its local declaration part are performed. In this case the only statement in this part is the `save` which states that we can switch only to states in this block (i.e., the `begin`, `create_pool` or `finished` states respectively on lines 59, 61 and 63). Other possible event occurrences are saved and can be handled (if necessary) outside this block.

After performing the local declaration part of the entered block the MANIFOLD run-time system automatically posts an occurrence of the predefined high-priority event `begin` in the event memory of the caller (as we will see this is `main` in §5) which causes a transition to the `begin` state. There must always be a `begin` state (i.e., a state with a single `begin` as its label) in every block. This insures that upon entering a block, at least this one state can be visited (i.e., the actions in its state body are performed), regardless of any other event occurrences that may or may not be present in the event memory.

In the `begin` state (line 59) we wait until the already active process instance `master` (received as a parameter on line 54) terminates. Because we have mentioned `master` (as an argument of the `terminate` primitive) in the state body, we also make 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 C code (excluding the `subsolve` work), arrives after some sequential computation work (initialization) at the point where it has to do the `subsolve` work, it raises an event named `create_pool` to signal that it needs a workers-pool to delegate that work to (master: 3(a)). This event pre-empts the `begin` state and causes a transition to the `create_pool` state (line 61). In this state the manner `Create_Worker_Pool` (lines 11-51) is called with the process instance `master`, and the manifold `Worker` (which the protocol manner `ProtocolMW` itself has received as a parameter on lines 54-55) as its actual parameters.

The manner `Create_Worker_Pool` conducts the workers in the pool and takes care that they can do their computational work properly. When the workers in the pool are done, they die and the manner returns. Afterwards (denoted by the semicolon on line 61) 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 soon because the `master` raises the event `finished` (master: 4) to denote that it does not need workers anymore. This causes a jump to the `finished` state (line 63), where the primitive action `halt` effectively returns the flow of control from the manner to its caller. The master is still running and is also done after performing the final prolongation computations.

The manner `Create_Worker_Pool` (lines 11-51) called on line 61 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` (defined in the MANIFOLD built-in library), 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 with `now` which is a mnemonic for Number Of Workers) and for counting the number of dead workers (by counting their `death_worker` events on line 42). Note that, MANIFOLD obviously only knows processes; there are no data structures in MANIFOLD, not even the simplest kind, a variable.

On line 21, a local event named `death_worker` is declared.

Because it can happen that both events `create_worker` and `rendezvous` are available in the event memory of the executing manifold instance that 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 pre-empt the `begin` state; and we wait (due to the word IDLE) for the termination of the special predefined process `void`. In the MANIFOLD language we express this by `terminated(void)` as can be seen from the meaning (line 9) of the IDLE macro (line 25). Because the special process `void` never terminates, this effectively causes a hang in the `begin` state

On lines 3-4, some global variables are declared followed on line 6 with the actual global data structure that contains the grid data. On lines 13-18 the global variables declared on lines 3-4 are set with values read from the command line at the time the program is executed. After that, the program continues with initializing the data structure and with some initial computation (line 20). After this a nested iteration starts (the nested for-loop on lines 22-27) in which the subroutine `subsolve` is called. In this nested loop a number of grids are visited and on each of these grids (a grid is specified by two integers; see the two integer arguments of `subsolve` on line 25) the subroutine `subsolve` is performed. `subsolve` is a very computation-intensive routine. In this routine, a linear system of equations ($Ax = b$) is solved for every time step. Moreover, this A matrix must be built up in the program which takes a lot of time. Also the adaptive time step in the time integrator (a so-called Rosenbrock solver) is something that must be computed again and again. After the nested loop, the coarse approximations on the visited grids are known and are prolongated on line 29 onto the finest grid used in the application to obtain a more accurate solution for it. With this the program comes to an end.

Because it is our aim to restructure this ANSI C program in a concurrent (parallel or distributed) structure we have special interest in which subroutines possess concurrent properties. In general, we can say that every grid subroutine with the property that it reads and writes data only from and to its own grid, can be restructured to run concurrently. In our program it turns out that `subsolve` has this property and because it is also very computing-intensive, it is a good candidate to run concurrently on all the grids to be visited in the nested for-loop.

A simple way to restructure our sequential ANSI C program into a concurrent one, is to introduce a workers-pool (containing a number of workers) when we arrive at the heavy computations that can be done concurrently. Each worker in the workers-pool performs the same operation `subsolve` on a different data segment of the global data structure independently of the others. In a program built according to this schema, 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 the concurrent region, only one single computational process continues (now it runs sequentially) in which the prolongation work is performed.

4. THE PASTE

The crux of our restructuring is to allow the computations done in `subsolve` on every single grid visited in the nested loop, 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, or in different tasks on a distributed platform (e.g., a network of workstations), or a in combination of the two.

We have organized the restructuring according to a master/worker protocol in which the master performs all the computation in the sequential source code except the work embodied in `subsolve`, which is 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 computation is performed in the master or 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. Furthermore, according to this protocol, the coordinator can create a worker only when the master raises an event to request for its creation.

Due to space limitation, we give only an informal description of the master/worker protocol in §4.1, a short description of its implementation in §4.2 and short stepwise description of the behavior interface of the master and worker in §4.3. For details we refer to the official report of the NCF project [6].

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 performs all computation in the main C program of the sequential version, except the computation to be carried out by `subsolve`. Each time the master arrives at the point where it has to do the `subsolve` work, it delegates this work to a worker in a workers-pool. The master makes its wish known to the coordinator by raising an event (`create_pool`)[2]. The coordinator reacts to 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 a worker reference, 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 `dead_worker` events) to acknowledge the successful rendezvous. After this rendezvous, the master reads from its input port the computational results of the workers. This is made possible 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 prolongation work and is done.

Note that in the master/worker protocol just described the master process passes all data to and from the workers. An alternative is for the master to introduce I/O workers. Of course this also involves extra coordination overhead. We have not tried this out because we were already content with the achieved results as given in §7.

4.2 Implementation of the Gluing Modules

The MANIFOLD source code of our master/worker protocol is given below. To clarify the way this protocol co-operates with the different steps in the master and workers behavior interface as given in §4.3, we provide references to those steps within parentheses. For the MANIFOLD terminology used here we refer to [2].

[2]We give the names of the events as used in the MANIFOLD source code (see §4.2) in parentheses.

until it detects an event in the event memory of the process instance where this manner is invoked and for which it has a state label. 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 (master: 3(b)). 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 (we intend to count their death_worker events in the rendezvous state on line 42); 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 and pass it the local event death_worker declared on line 21.

The death_worker event is an event the worker must raise to inform the manner Create_Worker_Pool, that it finished its job and is going to die (worker: 4).

The declarative statement on line 32 states that all stream connections between the output port of worker and the input port of the master (this input port is named dataport) must be of type KK (i.e., Keep-Keep). When streams of this type are used in a state they are not dismantled (i.e., disconnected from their sources and sinks) once the state is pre-empted. Normally, streams are BK (i.e., Break-Keep) streams which means that the stream is disconnected from its producer automatically, as soon as it is disconnected from its consumer, but disconnection from its producer does not disconnect the stream from its consumer.

In the begin state of the state create_worker, the stream configuration on line 36 is constructed and we wait for events (due to the word IDLE) 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 the already active master. The master receives this reference to worker and sends all the information worker requests through a stream (the second → on line 36) to worker. The worker process promptly reads the information it receives from master (worker: 1), does its job (worker: 2), and sends its computed results (worker: 3) through a stream (the third → on line 36) to the dataport port of master (denoted by master.dataport). The master process reads this and stores the results in the global master space (master: 3(f)). 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-empts this begin state (line 34) which dismantles the streams in this state and causes a transition to the create_worker state where the whole sequence starts 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 (master: 3(g)) after it reads the computed results of the remote workers (master: 3(f)) and causes a transition 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 transition 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.

Note that coordination schema in ProtocolMW can also handle a more demanding master. Just imagine that we have a master that instead of raising finished wants to introduce another workers-pool to delegate some work to. It could easily raise the event create_pool to denote that, in which case we jump again to the create_pool state and another pool is created. In [7, 5] we have exploited this facility in a slightly different master/worker protocol.

4.3 Behavior Interface of Master and Worker

The behavior interface of the master is given below. The line numbers in parentheses in this section refer to the MANIFOLD source code protocolMW.m in §4.2. Moreover, we refer to the process instance that invokes the protocolMW manner, as *the coordinator* (i.e., the instance of the manifold Main (line 13) in §5).

1. Make the extern events create_pool, create_worker, rendezvous, a_rendezvous, and finished available to the master so that it can communicate with the master/worker protocol.
2. Perform some initialization work (optional).
3. Perform some work concurrently by creating a pool of workers and charge each with a computational job. Do this as follows:
   (a) Request a coordinator process to create an empty pool of workers by raising the create_pool event (which is handled at line 61).
   (b) Request this coordinator process to create a worker in this pool by raising the event create_worker (which is handled at line 27).
   (c) Read a unit containing the process reference (identification) of a created worker from your own input port and activate it. (This unit, &worker, is sent through the first stream (->) on line 36 in protocolMW.m to the master).
   (d) Write the information, which the worker needs to do its job, on your own output port.
   (e) Repeat steps a, b, c and d for each worker as needed. (In this way a pool of workers is created and set to work.)
   (f) Collect the computational results from the workers (read those results from your own input port)
   (g) Raise the event rendezvous to request the coordinator to organize a rendezvous (which is handled at line 39).
   (h) Wait for the event a_rendezvous raised by the coordinator to acknowledge a successful rendezvous (line 50).
4. Repeat step 3 as many times as needed and raise at the end of this repetition the event finished (which is handled at line 63) to inform the coordinator process that the master does not need workers anymore.
5. Perform some final sequential computation (optional).

The behavior interface of the worker is described below. Here, the `death_worker` event is introduced via the first argument of the worker.

1. Read the information you need to do your job, from your own input port.
2. Do the computational job.
3. Write the computed results to your own output (master: 3(f)).
4. Raise the event `death_worker` (which is handled at line 42), to signal to the coordinator that you are done and are going to die.

5. THE CONCURRENT VERSION

The master and worker manifolds are easy to write as C wrappers around the original C subroutines of the sequential version. They are implemented according to the steps given in §4.3. Thereby we use a special ANSI C interface library where we find the routines for event handling, reading and writing data units from and to ports, etc. Due to space limitation we cannot show these wrappers. For this and other details we refer to [6]

Using the manifold `ProtocolMW` together with the two master and worker manifolds (i.e., those wrappers from above), we can construct the following small MANIFOLD program which finally changes our original sequential application into a concurrent version.

```
1  // mainprog.m
2
3  //pragma include "ResSourceCode.h"
4
5  #include "protocolMW.h"
6
7  manifold Worker(event) atomic.
8
9  manifold Master(port in p) port in input. port in dataport.
                              port out output. port out error.
10   atomic {internal. event create_pool, create_worker,
                        rendezvous, a_rendezvous, finished}.
11
12 /*************************************************/
13 manifold Main(process argv)
14 {
15   begin: ProtocolMW(Master(argv), Worker).
16 }
```

In this source code we define on lines 12-16 the manifold named `Main`, which in its `begin` state calls the `ProtocolMW` manner with the master and the worker manifolds as its actual arguments (line 15). After this, the instance of `Main`, the instance of the master `Master`, and all the necessary instances of the worker `Worker`, run concurrently.

6. RUNNING THE CONCURRENT VERSION

The source files that contain the MANIFOLD program (i.e., `mainprog.m` in §5 and `protocol.m` in §4.2) must be compiled with the MANIFOLD compiler, named MC. This compiler generates from each MANIFOLD source code a C source file which is subsequently compiled by a normal C compiler to an object file. These object files are linked with the object files obtained from the ANSI C files of the master and worker, the object files of the original sequential source code excluding the `main` and `subsolve` routines, and with some other C source files necessary to provide the inter-task information (these latter files are generated by the MANIFOLD linker named MLINK). In order to facilitate this whole procedure, the linker in the MANIFOLD system generates a *makefile*, which is meant to be used as a black-box by recursive make commands in programmer-defined makefiles that finally create the executable files suitable for the appropriate platforms.

Process instances in a MANIFOLD application always run as separate threads (light-weight processes [11]) within an operating-system level process. This latter heavy-weight process is called a task instance in MANIFOLD. Process instances are bundled in task instances either automatically or under user control. When all process instances of a MANIFOLD application run as threads in the same task instance, the application executes in parallel (i.e., not distributed). We can, however, also bundle the process instances in such a way that each worker is housed in a separate task instance. This mapping of process instances in task instances, which can be fully specified by the user, is considered to be a separate stage in the application construction and is described in a file which is input for the MANIFOLD linker MLINK. In the example below, we arrange it such that each worker is housed in a separate task instance (line numbers have been added).

```
 1 # mainprog.mlink
 2
 3 {task *
 4   {perpetual}
 5   {load 1}
 6   {weight Master 1}
 7   {weight Worker 1}
 8 }
 9 {task mainprog
10   {include mainprog.o}
11   {include protocolMW.o}
12 }
```

In this file, we specify that a task instance is considered to be "full" when its load exceeds 1 (line 5) and that the weight of an instance of the `Worker` or `Master` is also 1 (lines 6-7). The net result of this is that each task instance will house only one `Worker` or `Master` instance and thus instances of `Worker` or `Master` end up in different instances of the task named `mainprog` (line 9).

After this task composition stage the final stage in application construction can start: this is the runtime configuration stage. In that stage we define the mapping of tasks to hosts. This mapping too, is described in a file and is the input for the MANIFOLD runtime configurator named CONFIG. Suppose we need in our application in addition to the master, five workers; then we expect, with the above input file for MLINK, that at most (as we will see) six task instances come into existence during the run. Each of these task instances houses either a master or a worker instance. However, it can happen that a worker is already done before another worker is introduced by the master. In that case, the task instance (which is a heavy weight process) that has housed the freshly expired worker does not have a load of 1 anymore and is in principle ready to welcome a new worker. However, the standard behavior of a MANIFOLD task instance is that it dies when there are no thread processes running in it. To inhibit this task instance termination behavior, and to keep an empty task (i.e., task with load zero) alive for new workers, we use the keyword `perpetual` in the input file for the MANIFOLD linker MLINK (line 4). With this task termination behavior it can happen that we need less than six machines to run an application with five workers, which is more efficient. Therefore, when we start up the first task instance on the machine we are sitting behind (this so-called "start-up" machine is in our case `bumpa.sen.cwi.nl`), we have to organize five other machines for the possible other five task instances that are forked during the run. In the file below these additional machines are specified.

```
{host host1 diplice.sen.cwi.nl}
{host host2 alboka.sen.cwi.nl}
{host host3 altfluit.sen.cwi.nl}
{host host4 arghul.sen.cwi.nl}
{host host5 basfluit.sen.cwi.nl}
{locus mainprog  $host1 $host2 $host3 $host4 $host5}
```

Here, we define five variables `host1`, `host2`, up to `host5`, which we set to, respectively, `diplice.sen.cwi.nl`, `alboka.sen.cwi.nl` up to `basfluit.sen.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 of the task named `mainprog` can be started on any of these machines.

Running the restructured program, using the task composition stage and run-time configuration described above, the application executes in a *distributed* fashion and produces the following chronological output.

```
bumpa.sen.cwi.nl 262146 140 1048087412 175834
```

```
  mainprog Master(port in) ResSourceCode.c 136 -> Welcome
basfluit.sen.cwi.nl 1572865 79 1048087412 275851
  mainprog Worker(event) ResSourceCode.c 351 -> Welcome
basfluit.sen.cwi.nl 1572865 79 1048087412 366117
  mainprog Worker(event) ResSourceCode.c 370 -> Bye
arghul.sen.cwi.nl 1310721 79 1048087412 385644
  mainprog Worker(event) ResSourceCode.c 351 -> Welcome
basfluit.sen.cwi.nl 1572865 90 1048087412 414473
  mainprog Worker(event) ResSourceCode.c 351 -> Welcome
arghul.sen.cwi.nl 1310721 79 1048087412 483301
  mainprog Worker(event) ResSourceCode.c 370 -> Bye
basfluit.sen.cwi.nl 1572865 90 1048087412 511798
  mainprog Worker(event) ResSourceCode.c 370 -> Bye
altfluit.sen.cwi.nl 1048577 79 1048087412 520315
  mainprog Worker(event) ResSourceCode.c 351 -> Welcome
arghul.sen.cwi.nl 1310721 90 1048087412 552362
  mainprog Worker(event) ResSourceCode.c 351 -> Welcome
altfluit.sen.cwi.nl 1048577 79 1048087412 600215
  mainprog Worker(event) ResSourceCode.c 370 -> Bye
bumpa.sen.cwi.nl 262146 140 1048087412 637649
  mainprog Master(port in) ResSourceCode.c 337 -> Bye
arghul.sen.cwi.nl 1310721 90 1048087412 639482
  mainprog Worker(event) ResSourceCode.c 370 -> Bye
```

In this output we see messages from the master and workers when they start working ("Welcome") and when they are done and return ("Bye"). We don't show the computational results of this distributed run. These are written to a file and are exactly the same as in the sequential version.

Each of these messages has the following structure (one message consists of two lines here). 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, a time stamp that is expressed as two numbers (these numbers are the seconds and microseconds past since midnight (0 hour), January 1, 1970) 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 message, we always know *who* is printing, *what*, *where* and *when*.

When we look at the above output we see that not all the machines specified in the input file for the configurator are used. This is due to the *perpetual* termination behavior of a task instance and the fact that workers die before new ones are introduced in the workers-pool

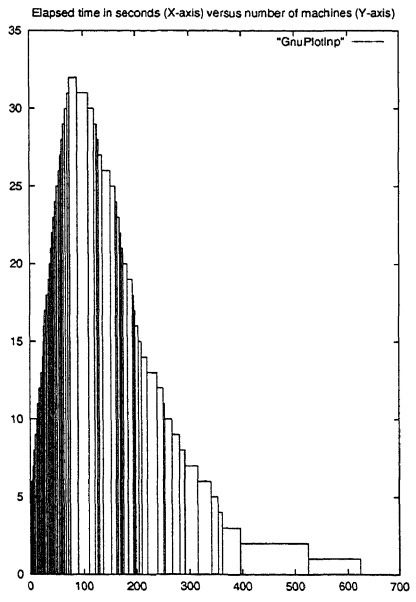

Figure 1: The ebb & flow during a run of our restructured application for level 15.

Because the number of task instances that are forked, varies during the run and each task instance runs on a separate machine, the number of machines vary in exactly the same way as the number of task instances do. From the output, like above, we can make a graph that shows the number of machines needed during the dynamic expansion and shrinking of our application run. In Figure 1 we show such a graph of an application that runs for 634 seconds and sometimes uses 32 machines. The weighted average of the machines used in this case is 11.

When we want to execute our application in a *parallel* way such that all the workers are in the same task instance, then we simply change the load on line 5 of `mainprog.mlink` to 6, and do the linking phase again.

7. PERFORMANCE RESULTS

We have carried out a number of experiments. Every run of our sequential and restructured application needs a number of parameters. These are (see lines 13-18 in §3), the refinement level of the coarsest grid (we have used 2), the additional refinement level (we have used 0 through 15), the tolerance in the integrator (we have used 1.0e-3 and 1.0e-4).

The relationship between the additional refinement level l and the number of workers w is that $w = 2l + 1$. Thus the total number of workers plus master is $2l + 2$. This latter number is an upper bound for the number of machines used during a distributed run.

We have run and compared the performance results of the sequential and the concurrent versions of our application on a cluster of 32 single processor workstations. Such a cluster is big enough to run the application with $l = 15$. Unfortunately, in our institute no homogeneous cluster of workstations of that size is available. All the machines in our cluster have an AMD Athlon Processor and a cache size of 256Kb. However 24 machines have a clock cycle of 1200Hz, 5 machines have a clock cycle of 1400Hz, and 3 machines have a clock cycle of 1466Hz. Although these machines have different CPU speeds, their speeds are of the same order of magnitude. The workstations in the cluster are connected to each other by a switched Ethernet (100 Mbps).

The experiments were done at night. However, even then, this means that we do not have a guarantee that we are the only user. There are always unpredictable effects such as network traffic and file server delays, etc. Furthermore some users of the machines in the cluster, run their own job(s) at night, run screen savers or have runaway Netscape jobs. All this causes differences in performance on identical hardware. These unknown effects cannot be eliminated and are always reflected in our computational results. To even out such "random" perturbations, we ran the two versions of the application five times and computed the average *elapsed* or *wall clock* times (i.e., the actual time the application program runs as it would be measured by a user sitting at the terminal with a stopwatch). Thereby we notice that the elapsed times for the five different runs were of the same order of magnitude. The timing measurements were obtained using the UNIX utility `/bin/time`. The results are given in Table 1. The weighted average of the number of machines used during a run and the average speedup are also given. Figures 2, 3, 4 and 5 graphically show the contents of Table 1. Because of the wide range of the average sequential and concurrent time we use the logarithmic scale in Figures 2 and 4.

For our analysis of the results, we distinguish the following categories of overhead introduced by our restructuring:

- The overhead introduced by the unpredictable effects of working in a multi-user environment. These effects are totally out of control in a multi-user environment without dedicated machines.
- The overhead introduced by the concurrency itself (i.e., the

overhead of making a sequential program run as a concurrent program).

- The overhead of the coordination layer (i.e., the actual implementation of the overhead of the concurrency).

Error Name:	level	st	ct	m	su
	0	0.03	9.27	1.9	0.0
Offending Command:			--rlineto--		
	2	0.06	13.09	2.8	0.0
	3	0.11	7,86	2.7	0.0
			11.45	2.9	0.0
Operand Stack:			17.40	3.6	0.0
	6	0.86	26.91	3.3	0.0
63	7	1.90	28.97	3.6	0.1
1.0e-3 run	8	4.27	30.06	3.7	0.1
	9	10.28	23.84	4.1	0.4
	10	24.14	21.82	5.5	1.1
	11	57.91	33.58	6.3	1.7
	12	145.47	50.79	7.6	2.9
	13	337.69	75.28	9.8	4.5
	14	818.62	124.20	11.7	6.6
	15	2019.02	259.69	12.2	7.8
	0	0.02	7.68	1.9	0.0
	1	0.05	13.04	2.4	0.0
	2	0.07	12.99	2.8	0.0
	3	0.15	7.44	2.6	0.0
	4	0.30	12.03	2.9	0.0
	5	0.68	16.39	3.3	0.0
	6	1.53	21.07	3.5	0.1
	7	3.53	28.68	3.7	0.1
1.0e-4 run	8	8.04	30.29	3.9	0.3
	9	21.00	26.24	4.8	0.8
	10	51.64	38.66	5.7	1.3
	11	124.17	46.30	7.6	2.7
	12	301.17	65.02	9.9	4.6
	13	724.92	129.28	11.4	5.6
	14	1751.02	227.18	13.1	7.7
	15	4118.08	519.15	13.3	7.9

Table 1: Average sequential time (st), average concurrent time (ct), weighted average of numbers of machines used (m), and average speedup ($su = st/ct$) for 1.0e-3 and 1.0e-4 runs for levels 0 through 15.

Because the differences between the five results were not so big we conclude that the effects of working in a multi-user environment are minimal in comparison with the other overhead. Looking at Table 1, we see that for the runs with $l < 10$ there is no gain in time for the 1.0e-3 and the 1.0e-4 runs (i.e., the speedup is less than 1.0). Probably the useful computational work done by the workers is too little in comparison with the overhead of the concurrency plus the overhead of the coordination layer. For the $l \geq 10$ runs we see a gain in time. For those levels the average speedup for the levels 10 through 15 ranges from 1.1 to 7.8 for the 1.0e-3 runs and from 1.3 to 7.9 for the 1.0e-4 runs. However, we also see that this time reduction is accomplished by a growing number of machines. Their averages range for the 1.0e-3 runs from 5.5 to 12.2 machines and for the 1.0e-4 runs from 5.7 to 13.3. Furthermore, we see that the average speedup in a run always lags behind the average number of machines it uses. For the levels 12 and higher the speedup is about half of the weighted number of machines used. From this we conclude that for those levels the overhead of the concurrency