Coordination of distributed/parallel multiple-grid domain decomposition

C.T.H. Everaars and F. Arbab

Computer Science/Department of Interactive Systems

CS-R9627 1996
Coordination of Distributed/Parallel Multiple-grid Domain Decomposition

C.T.H. Everaars and F. Arbab

ever@cwi.nl and farhad@cwi.nl

CWI

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

Abstract

A workable approach for the solution of many (numerical and non-numerical) problems is domain decomposition. If a problem can be divided into a number of sub-problems that can be solved in a distributed/parallel fashion, the overall performance can significantly improve. In this paper, we discuss one of our experiments using the new coordination language MANIFOLD to solve an instance of the classical optimization problem by domain decomposition. We demonstrate the applicability of MANIFOLD in expressing the solutions to domain decomposition problems in a generic way and its utility in producing executable code that can carry out such solutions in both distributed and parallel environments.

The multiple-grid domain decomposition method used in this paper is based on adaptive partitioning of the domain and results in highly irregular grids as shown in the examples. The implementation of the distributed/parallel approach presented in this paper looks very promising and its coordinator modules are generally applicable.


AMS Subject Classification (1991): 68N15. 68Q10.

Keywords and Phrases: distributed computing. parallel computing. coordination languages. models of communication. computational steering. domain decomposition.

1. Introduction

In sciences, engineering, and economics, decision problems are frequently modeled as optimizing the value of a function under some constraints. The problem in its generic form is formulated as follows:

\[
\min f(x) \quad \text{subject to } x \in D \subset \mathbb{R}^n.
\]  

(1.1)

There is an enormous amount and variety of literature about the theory and implementation of this problem. This variety is essentially due to different assumptions about the underlying problem structure. It is not our aim to offer the reader a tour through this literature, nor do we intend to present a very sophisticated algorithm for a certain class of global optimization problems. We consider solving an instance of (1.1) in a distributed/parallel fashion only as an example of the application of the domain decomposition method in the field of numerical computing. As a concrete example, we use a multi-extremal function in two variables, which is to be minimized in a certain domain. We use this function to illustrate the applicability of our generic domain decomposition coordinator module to implement irregular, adaptive multiple-grid methods. The same coordinators can be used without change for higher-dimensional functions as well as other non-numeric domain decomposition problems.

If a problem can be divided into a number of sub-problems that can be solved on a cluster of parallel computers and workstations, we may be able to significantly improve the performance of our solution. The new brand of coordination languages\cite{9} presents a viable approach to this
kind of problem decomposition. In this paper, we discuss one of our experiments using the new coordination language \texttt{MANIFOLD} to decompose an instance of the classical optimization problem and solve it using irregular grids in a distributed/parallel fashion.

\texttt{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. Although parallel computing and distributed computing are quite distinct in nature, they can both serve to improve performance. Distributed computing is related to the emergence of computer networks; computer applications move from single stand-alone mainframes to multiple communicating local workstations. Parallel computing arose from the quest to fundamentally improve the speed of sequential computation by using multiple processing units. From the language point of view, \texttt{MANIFOLD} does not make a distinction between a multiprocessor mainframe and a simple one-processor workstation. This feature makes \texttt{MANIFOLD} a very powerful tool for problem solving in heterogeneous computing environments.

To grow accustomed to the \texttt{MANIFOLD} language, in section 2 we give a brief introduction to the \texttt{MANIFOLD} language. In section 3 we start with the inevitable \texttt{"Hello World!"} program to show some of the syntax and semantics of \texttt{MANIFOLD}. It is beyond the scope of this paper to present the details of the syntax and semantics of the \texttt{MANIFOLD} language\footnote{For more information, refer to our html pages located at \url{http://www.cwi.nl/cwi/projects/manifold.html}}. In section 4 we present our optimization problem and describe the parallel/distributed domain decomposition. We close this paper with a short conclusion in section 5.

2. The Manifold Coordination Language

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

A \texttt{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 \texttt{MANIFOLD}, nor the fact that they are cooperating with other processes through \texttt{MANIFOLD} in a concurrent application.

The \texttt{MANIFOLD} system consists of a compiler, a run-time system library, a number of utility programs, libraries of built-in and predefined processes[2], a link file generator called \texttt{MLINK} and a run-time configurator called \texttt{CONFIG}. The system has been ported to several different platforms (e.g., SGI 5.3, SUN 4, Solaris 5.2, and IBM SP/1). \texttt{MLINK} uses the object files produced by the (\texttt{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, \texttt{CONFIG} determines the actual host(s) where the processes which are created in the \texttt{MANIFOLD} application will run.

The library routines that comprise the interface between \texttt{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

\texttt{MANIFOLD} is based on the \textit{Idealized Worker Idealized Manager} (IWIM) model of communication[3]. In this section we briefly describe this model and discuss its advantages over the \textit{Targeted-Send/Receive} (TSR) model on which object-oriented programming models and tools such as PVM[8], PARMACS[10], and MPI[11, 6] are based.

The basic concepts in the IWIM model (thus also in \texttt{MANIFOLD}) are \textit{processes}, \textit{events}, \textit{ports}, and \textit{channels} (in \texttt{MANIFOLD} called \textit{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 \( m_1 \) and \( m_2 \) 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.

```
1 ********************
2 * A TSR pseudo code *
3 ********************
4
5 process p
6 begin
7  compute m1
8  send m1 to q
9  compute m2
10 send m2 to q
11 do other things
12 receive m
13 do other computation using m
14 end
15
16 process q
17 begin
18 receive m1
19 receive m2
20 (let z be the sender of m1 and m2)
21 compute m using m1 and m2
22 send m to z
23 end
```

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

```
1 ********************
2 * A IWIM pseudo code *
3 ********************
4
5 process p
6 begin
7  compute m1
8  write m1 to output port o1
9  compute m2
10 write m2 to output port o2
11 do other things
12 read m from the input port i1
13 do other computation using m
14 end
15
16 process q
17 begin
18 receive m1 from input port i1
19 receive m2 from input port i2
20 compute m using m1 and m2
21 write m to output port o1
22 end
23
24 process c
25 begin
26 create the processes p and q
27 create the channel p.o1 -> q.i1
28 create the channel p.o2 -> q.i2
29 create the channel q.o1 -> p.i1
30 follow some termination protocol
31 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

\(^2\)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
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 $m_1$ and $m_2$), 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 $m_1$ and $m_2$, $q$ 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 $m_1$ from any source, which it will subsequently refer to as $z$; expects a second message $m_2$ (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 $m_1$ and $m_2$, 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, we demonstrate how easy it is with an IWIM based tool such as MANIFOLD, to adapt a parallel/distributed application to new requirements. Furthermore, the general applicability of the coordinator modules in our case study is something that is inherent in IWIM. This degree of flexibility and re-usability is not possible with TSR-based tools and languages such as PVM.

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.

---

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.
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
1 manifold printunits import.
2 3 auto process print is printunits
4 5 manifold Main
6 { 7 begin: "Hello World!" -> print.
8 }
```
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, “→”. 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 → to the input port of the process instance on its right-hand side. The process instance on the right-hand side of the → is, of course, print. What appears to be a character string constant on the left-hand side of the → 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.

![Diagram](https://via.placeholder.com/150)

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. **Domain Decomposition**

Consider the following general global optimization problem: given a bounded, \( D \subset \mathbb{R}^n \), and a continuous function \( f : D \to \mathbb{R} \), solve

\[
\min f(x) \quad \text{subject to } x \in D.
\]  

(4.2)

As an instance of (4.2) we take the Goldstein and Price function:

\[
\begin{align*}
\min z = & \ (1 + (x + y + 1)^2 (19 - 14 x + 3x^2 - 14 y + 6xy + 3y^2)) \\
& \ (30 + (2x - 3y)^2 (18 - 32x + 12x^2 + 48y - 36xy + 27y^2))
\end{align*}
\]

with \((x, y) \in [-2.0, 2.0]\)

(4.3)

Figure 2 shows the landscape formed by this function in its domain. Although at this scale the detailed “bumpiness” of this function cannot be seen, this figure still shows the potential difficulty of the general problem (4.2).

![Figure 2: The Goldstein and Price function](image)

Analytical solutions to such problems are, in general, non-existent and domain decomposition is a common search technique used to solve them through numerical methods. Domain decomposition imposes a grid on the domain of the function, splitting it into a number of sub-domains, as determined by the size of the grid. Next, we obtain a (number of) good rough estimate(s) for the lowest value of \( z \) in each sub-domain. Then, we either use the best obtained estimate for the optimum \( z \) value, or select the sub-domains with the most promising \( z \) values and decompose them into smaller sub-domains. In iterative refinement methods, new estimates for the lowest value of \( z \) in each of these sub-domains, recursively, narrow this search process further.
and further into smaller and smaller regions that (hopefully) tend towards the area with the real minimum $z$, while the estimates for the obtained minimum $z$ values become more and more accurate. In iterative single-grid domain decomposition, the same grid is imposed on all successive sub-domains. Multiple-grid adaptive domain decomposition techniques allow a different grid for each sub-domain, whose granularity and other properties may depend on the attributes of the sub-domain and those of the function within that region.

A simple domain decomposition program is presented in section 4.1. In section 4.2 we modify this program to handle iterative refinement and multiple, adaptive grids. In section 4.3 we visualize the numerical results of this program, and in section 4.4 we evolve our program into a simple computational steering application.

### 4.1 Single-grid Domain Decomposition

The following MANIFOLD program shows a non-iterative single-grid domain decomposition application.

```manifold
1 manifold PrintObjects atomic {internal.}.
2 manifold Split atomic {internal.}.
3 manifold AtomicEval(event, port in) atomic {internal.}.
4 manifold Eval forward.
5 manifold Merger port in a, b. atomic {internal.}.
6 /*---------------------------------------------*/
7 manifold Main
8 |
9 auto process split is Split.
10 auto process eval is Eval.
11 auto process print is PrintObjects.
12 |
13 begin: <<1, -2.0, -2.0, 2.0, 2.0, 5, 5>> -> split -> eval -> print.
14 |
15 | /*---------------------------------------------*/
16 manifold Eval()
17 |
18 event filled, flushed, finished.
19 |
20 process atomeval is AtomicEval(filled, 1000).
21 |
22 stream reconnect KB input -> *.
23 |
24 priority filled < finished.
25 |
26 begin:
27 (activate(atomeval), input -> atomeval,
28 guard (input, a_everdisconnected ! empty, finished) // no more input
29 ).
30 |
31 finished:
32 {
33 ignore filled. //possible event form atomeval
34 begin: atomeval -> output. //your output is only that of atomeval
35 }.
36 |
37 filled:
38 {
39 process merge<a, b | output> is Merger.
40 |
41 stream KK * -> (merge.a, merge.b).
42 stream KK merge -> output.
43 |
44 begin:
45 (activate(merge), input -> Eval -> merge.a,
46 atomeval -> merge.b, merge -> output
47 ).
48 |
49 finished:. //do nothing and leave this block
50 }.
51 |
52 end:
53 {
54 guard(output, a_disconnected, flushed), // ensure flushing
55 terminated(void) //wait for units to flush through output
56 ).
57 |
58 flushed: halt.
59 }
```

The `main` manifold in this application creates `split`, `eval`, and `print` as instances of manifold definitions `Split`, `Eval`, and `PrintObjects`, respectively (line 10-12). It then connects the output of a process instance which produces a unit that describes a domain and its decomposition (in our
case, $5 \times 5$), to the input port of split; the output port of split to the input port of eval; and the output port of eval to the input port of print. The process main terminates when all three connections are broken.

The code for Split is a C function. An instance of Split reads from its input port a unit that describes a (sub-)domain and the specification of a grid, produces units on its output port that describe the sub-domains obtained by imposing the grid on this input domain, and terminates.

An instance of Eval is expected to read all the sub-domains (in this case, there are 25) from its input port. It then finds the best estimate for the optimum $z$ value in each of its sub-domains, produces through its output port an ordered sequence of units describing the best solutions it has found, and terminates.

PrintObjects is implemented as a C function. An instance of PrintObjects simply prints the units it reads from its input, each of which describes a (sub-)domain and the $x$, $y$, and $z$ values for the estimated minimum $z$ value found at the point $(x,y)$ in that domain.

Up to now our program looks as easy as the “Hello World!” program in section 3. However, because we want to solve our optimization problem in a distributed/parallel fashion the Eval is more complex. We already mentioned what the manifold Eval is supposed to do; now we discuss how Eval does it. An instance of Eval coordinates the cooperation of instances of two other manifolds, namely AtomicEval and Merger.

AtomicEval is implemented as a C function. An instance of AtomicEval reads a bucket of $s > 0$ sub-domains (for simplicity, let $s = 1$) from its input port and raises a specific event, which it receives as a parameter, to inform other processes that it has filled up its input bucket with $s$ sub-domain descriptions. It then finds the best estimate for the optimum $z$ value in each of its sub-domains, producing an ordered sequence of units describing the best solutions it has found through its output port, and terminates. The algorithm used by AtomicEval to find the estimates for the optimum $z$ value in a sub-domain is completely internal to this computation module and is irrelevant for our purposes in this paper. In our example, we use sampling: we simply evaluate $z$ for a number of (say 1000) sample points in each sub-domain and consider the sample point with the minimum $z$ as the best estimate for that sub-domain.

Merger is also implemented as a C function. An instance of Merger reads from its ports a and b two ordered sequences of units describing sub-domains and their best estimates, and produces a sequence of one or more of its best sub-domains on its output port.

As noted above, an instance of Eval receives through its input port an unknown number of units that describe (sub-)domains. It is supposed to feed as many of its own input units to an atomic evaluator as the latter can take; feed the rest of its own input as the input to another copy of itself; merge the two output sequences (of the atomic evaluator and its new copy); and produce the resulting sequence through its output port. Let us follow the source code of the manifold Eval in more detail.

In its begin state, an instance of Eval connects its own input to an instance of the AtomicEval, it calls atomeval. It also installs a guard on of its own input port. This guard posts the event finished if it has an empty stream connected to its departure side, after the arrival side of this port has no more stream connections, following a first connection. This means that the event finished is posted in an instance of Eval after a first connection to the arrival side of its input is made, then all connections to the arrival side of its input are severed, and all units passed through this port are consumed. The connections in this state are shown in Figure 3.a.

Two events can preempt the begin state of an instance of Eval: (1) if the incoming stream connected to input is disconnected (no more incoming units) and atomeval reads all units available in its incoming stream, the guard on input posts the event finished; and (2) the process atomeval can read its fill and raise the event filled. Normally, only one of these events occurs; however, when the number of input units is exactly equal to the bucket size, $s$, of atomeval, both finished and filled can occur simultaneously. In this case, the priority statement makes sure that the
handling of finished takes precedence over filled.

Assume that the number of units in the input supplied to an instance of Eval is indeed less than or equal to the bucket size \( s \) of an atomic evaluator. In this case, the event finished will preempt the begin state and cause a transition to its corresponding state in Eval. In this state, we ignore the occurrence of filled that may have been raised by atomeval (if the number of input units is equal to the bucket size \( s \)) and deliver the output of atomeval as the output of the Eval. The connections in this state are shown in Figure 3.b.

Now suppose the number of units in the input supplied to an instance of Eval is greater than the bucket size \( s \) of an atomic evaluator. In this case, the event filled will preempt the begin state and cause a transition to its corresponding state in Eval. In this state we create an instance of the merger process, called merge. A new instance of the Eval is created in the begin state of the nested block. The rest of the input is passed on as the input to this new Eval, its output is merged with the output of the atomic evaluator, and the result is passed as the output of the Eval instance itself. The connections in this state are shown in Figure 3.c. An occurrence of finished in this state preempts the connected streams and causes a transition to the local finished state in this block. This preemption is necessary to inform the new instance of Eval (by breaking the stream that connects input to it) that it has no more input to receive, so that it can terminate. The empty body of the finished state means that it causes an exit from its containing block.

In the end state, an Eval instance installs a guard on its output port to post the event flushed after there is no stream connected to the arrival side of this port following its first connection. This means that the event flushed is posted in an instance of Eval after a connection is made to its arrival side, and all units arriving at this port have passed through. The Eval instance then waits for the termination of the special predefined process void, which will never happen (the special process void never terminates). This effectively causes the Eval instance to hang indefinitely. The only event that can terminate this indefinite wait is an occurrence of flushed which indicates there are no more units pending to go through the output port of the Eval instance.

The output of our program, below, shows the result produced by 25 instances of AtomicEval, each taking in the description of a single sub-domain. The top line shows the best estimate for its global minimum to be 3.126 at point \((0.006, -1.015)\).

```
domain = (-0.400, -1.200) ( 0.400, -0.400) point = (0.006, -1.015), z = 3.126
domain = (-1.200, -1.200) ( 0.400, -0.400) point = (-0.588, -0.406), z = 30.074
domain = (-1.200, -0.400) ( 0.400, 0.400) point = (-0.612, -0.381), z = 30.170
domain = (-0.400, -2.000) ( 0.400, -1.200) point = (-0.205, -1.201), z = 37.421
domain = ( 0.400, -1.200) ( 1.200, -0.400) point = (-0.612, -0.381), z = 30.170
domain = ( 0.400, -1.200) ( 1.200, -0.400) point = ( 0.431, -0.701), z = 40.373
domain = ( 1.200, 0.400) ( 2.000, 1.200) point = ( 0.890, -0.396), z = 89.792
domain = ( 1.200, 0.400) ( 1.200, 1.200) point = ( 1.982, 0.402), z = 840.567
domain = ( 0.400, 0.400) ( 1.200, 1.200) point = ( 1.097, 0.786), z = 849.547
```
The recursive way in which the coordinator process Eval creates and coordinates its atomic workers is interesting. These atomic workers (the numerical evaluators and the mergers, together with the atomic workers created in the coordinator process main) and their connections are shown in figure 4. Here $e_i$ ($m_i$) denotes the $i$th evaluator (merger), dashed lines represent the (re)connections of the same stream, and $n$ is the recursion depth of Eval. Note that all these processes run concurrently with each other. This means that, depending on the installation configuration of the MANIFOLD system, they can run truly in parallel with each other on a distributed and/or parallel platform.

![Figure 4: The atomic processes at work.](image-url)

Each MANIFOLD process runs as a separate thread (a light-weight process). In our installation of the MANIFOLD system on Sun and SGI machines, thirty or so of these threads are bundled together and comprise a MANIFOLD task. Each task instance is an operating-system-level (heavy-weight) process that runs somewhere on a distributed platform. The actual host(s) where these tasks run are specified in a configuration file which is read at runtime. If, e.g., we impose a $100 \times 100$ grid on the domain (so when the bucket size $s$ of AtomicEval is 10, the recursion depth of Eval is $n = 1000$) and the configuration file contains a number of SGIs, a number of SUNs, an IBM SP/2, and some Linux machines, etc., more than 67 MANIFOLD task instances are created and spread over this heterogeneous environment and run in a distributed/parallel fashion. For instance, on a multi-processor machine such as SGI, thirty or so threads in the same task can run concurrently. At most $k$ of these threads can actually be running (truly) in parallel with each other, where $k$ is the number of processors on the machine.

An interesting aspect of our application is the dynamic way in which Eval switches connections among the process instances it creates (see figure 3). Perhaps more interesting, is the fact that, in spite of its name, Eval knows nothing about evaluating functions! What Eval embodies is, not any computation, but only a protocol that describes how instances of two process definitions (e.g.,
AtomicEval and Merger in our case) should communicate with each other (see [5] for a detailed treatment of this phenomenon). A logical consequence of this clear separation of coordination and computation concerns into distinct modules is that we can use this same protocol for a completely different pair of process definitions than AtomicEval and Merger. In fact, this same module is used to implement a parallel/distributed bucket sort program as well [5]. An interesting use of this protocol would be to optimize a multi-extremal function \( f \) in \( n \) variables for large \( n \)’s in a distributed/parallel fashion. To accomplish this, we only need to change the atomic workers; no change to the coordinators (Eval and Main) is necessary, which means they do not even have to be recompiled for this new application.

### 4.2 Multiple-grid Domain Decomposition

In this section we discuss a multiple-grid domain decomposition method. We initially impose a 2 × 2 grid on the domain of the function and start on each sub-domain an evaluator (AtomicEval). The evaluator finds a rough estimate for the lowest value of \( z \) in its sub-domain and determines a suitable grid (in our example, either 4 × 2 or 2 × 4) for its further decomposition, should that become necessary later. Our version of the evaluator proposes a 4 × 2 grid for the sub-domain under its consideration if the function is more hilly in the \( x \)-direction than in the \( y \)-direction in this sub-domain; otherwise, it proposes a 2 × 4 grid. With these grids we always have eight evaluators which can (in principle) run concurrently. Of course the relationship between the domain, the function, and the splitting scheme may be considered more carefully to yield better grids, perhaps with more variety and more adaptively. Our particular choice of grids and the simple criterion we use to select between them are good enough for our demonstration purposes. Note, however, that the choice of grids, their sizes, their number, their degree of adaptivity, as well as the criteria used for selecting among them, are all details that are internal to AtomicEval and, thus, irrelevant to our coordination modules (Eval and Main).

Eval simply continues with selecting the sub-domain with the most promising \( z \) value and the splitter imposes the recommended grid on it for its decomposition. New estimates for the lowest value of \( z \) in each of these sub-domains, recursively, narrow this search process further and further into smaller and smaller regions that (hopefully) tend towards the area with the real minimum \( z \), while the estimates for the obtained minimum \( z \) values become more and more accurate. We stop this iterative decomposition algorithm when the relative improvement of the best solution found in two successive iteration steps falls below a certain threshold. The result is a highly irregular grid which shows the search path through the domain. The following MANIFOLD program shows the multiple-grid domain decomposition:

```manifold
1 manifold pass1 import.
2 manifold variable import.
3 manifold Eval import.
4 manifold PrintObjects atomic [internal.].
5 manifold Split atomic [internal.].
7 manifold AtomicEval(event, port in) atomic [internal.].
8 manifold Merger port in a, b. atomic [internal.].
9 manifold Checker(port in, port in, port in, event, event) atomic [internal.].
10 #define TOL 1.0e-5
11 #define IDLE terminated(void)
14 /*********************************************************************/
15 manifold Main
16 {
17 event checkit, goon, stop.
19 auto process best1 is variable.
20 auto process best2 is variable.
21 auto process pr is PrintObjects.
22 stream reconnect BK * -> pr.
24 begin:
26 auto process pl is pass1.
27 begin:
29 <<1, -2.0, -2.0, 2.0, 2.0, 2, 2>> -> Split -> Eval -> (-> pr, -> pl),
31 best2 = pl
32 ;
33 post (goon).
34 }
35 goon;
```
4. Domain Decomposition

Lines 1-3 declare the manifolds pass1 and variable and Eval from the previous section. The keyword import states that the real definition (i.e., the body) of these manifolds are given elsewhere (in a library or in another source file). An instance of the predefined manifold pass1 remains idle until its input is connected to a stream. Once this connection is made, it passes the unit it receives on its input through its output port and terminates. An instance of the predefined manifold variable repeatedly reads a unit from its input port. It remembers the unit it reads, and if the departure side of its output is connected, it passes the unit on through its output port. Lines 11-12 define some preprocessor macros, in the same syntax as that of the C preprocessor. These macros define our symbolic constants. The main manifold contains four states (line 24, 36, 49 and 55). In the begin state, the stream configuration on line 30 is constructed. The output of Eval is fanned out to the processes pr and p1, which are respectively instances of PrintObjects (line 21) and pass1 (line 26). Because we initially impose a 2 x 2 grid (line 30) on the domain, the first output of Eval consist of an ordered sequence of four units describing the best solutions found in the four sub-domains. The first unit of this sequence, containing the most promising sub-domain to find the minimum, is fed to p1, which delivers it to best2 (line 31) and terminates.

When all the connections set up in lines 30-11 are broken (this happens when the tuple producer <<1, -2.0, -2.0, 2.0, 2.0, 2, 2>>, Split and Eval are done with their jobs and die, and p1 delivers its value to best1) the goon event is posted (line 33) and we switch to the goon state. There, best2 delivers its value to best1 (line 37). On line 43, an unit is read from the output port of best1 (getunit(best1)) and fed back to a stream configuration similar to the one on the lines 30-31. When the connections set up on lines 43-44 are broken, the checklist event is posted (line 46) and we switch to the checklist state. In this state, an instance of the Checker manifold, which compares best1 and best2, is automatically created and activated and we wait (due to the word IDLE on line 52) until this process raises a goon or a stop event. The Checker instance raises the stop event when the relative improvement to the best solutions found in two successive iteration steps is below a certain threshold (TOL, line 11). This causes a state switch to the stop state (its body is empty) and stops the iterative domain decomposition. In the other case, a transition to the goon state sets up another iteration step (line 37-47). The output of this program is shown below.

| domain = (-2.000, -2.000) | 2.000, 2.000) s = (2, 2) |
| domain = (-2.000, -2.000) | 0.000, 0.000) point = (-0.035, -1.003), z = 3.303 a = (2, 4) |
| domain = (0.000, -2.000) | 2.000, 0.000) point = (0.054, -0.973), z = 3.708 a = (4, 2) |
| domain = (0.000, 0.000) | 2.000, 2.000) point = (1.778, 0.182), z = 84.152 a = (2, 4) |
| domain = (-2.000, 0.000) | 0.000, 2.000) point = (-0.902, 0.011), z = 313.979 a = (2, 4) |
| domain = (-1.000, -1.000) | 0.000, -0.500) point = (-0.010, -0.997), z = 3.039 a = (2, 4) |
| domain = (-1.000, -1.500) | 0.000, -1.000) point = (-0.044, -1.016), z = 3.462 a = (4, 2) |
| domain = (-1.000, -0.500) | 0.000, 0.000) point = (-0.605, -0.195), z = 30.039 a = (2, 4) |
| domain = (0.000, -2.000) | 1.000, -1.000) point = (0.000, -0.085), z = 256.380 a = (2, 4) |
| domain = (-1.000, -0.000) | 1.000, -1.500) point = (-0.001, -0.034), z = 496.128 a = (2, 4) |
| domain = (-2.000, -0.000) | 1.000, -1.000) point = (-1.000, -1.663), z = 647.851 a = (2, 4) |
| domain = (-2.000, -1.500) | -1.000, -1.500) point = (-1.006, -1.478), z = 1844.802 a = (2, 4) |
| domain = (-2.000, -1.500) | -1.000, -1.000) point = (-1.006, -1.478), z = 318.267 a = (2, 4) |
| domain = (-1.000, -1.500) | -0.500, -0.875) point = (-0.500, -0.881), z = 318.267 a = (2, 4) |
| domain = (-1.000, -0.875) | -0.500, -0.750) point = (-0.500, -0.750), z = 32.732 a = (2, 4) |
| domain = (-1.000, -0.625) | -0.500, -0.500) point = (-0.500, -0.500), z = 32.732 a = (2, 4) |
| domain = (-0.500, -0.625) | -0.500, -0.500) point = (-0.500, -0.500), z = 32.732 a = (2, 4) |
| domain = (-0.500, -0.750) | -0.500, -0.875) point = (-0.500, -0.881), z = 318.267 a = (2, 4) |
| domain = (-0.500, -0.750) | -0.500, -0.750) point = (-0.500, -0.750), z = 32.732 a = (2, 4) |
| domain = (-0.500, -0.625) | -0.500, -0.625) point = (-0.500, -0.625), z = 32.732 a = (2, 4) |
| domain = (-0.500, -0.750) | -0.500, -0.500) point = (-0.500, -0.500), z = 32.732 a = (2, 4) |
| domain = (-0.500, -0.625) | -0.500, -0.625) point = (-0.500, -0.625), z = 32.732 a = (2, 4) |
| domain = (-0.500, -0.750) | -0.500, -0.500) point = (-0.500, -0.500), z = 32.732 a = (2, 4) |
As shown in the output above, the description of a single sub-domain is extended with the recommended grid to be imposed on it if it is selected for further decomposition. The first line in this output is our initial input unit representing the whole domain and its desired splitting which is initially set to $2 \times 2$ ($s = (2, 2)$). Each succeeding group of eight lines then represents one iteration. The best sub-domain found in each iteration is fed as input to the next iteration. The first line of the last group (representing the 6th iteration) shows the best solution found ($z = 3.000$) to be at $(0.000, -1.000)$, which is much better than the best solution we found using our single $5 \times 5$ grid ($z = 3.1261$) in section 4.1.

It is quite common in global optimization to first apply a purely random search -- a very simple and popular “folklore” approach to global optimization -- as a preliminary search phase for reducing the initially chosen search domain. The current estimate of the global optimum found in this search can then form a starting point for a local search algorithm (e.g., the method of steepest descent, Newton, the conjugate gradient method, etc.). If we consider the work done in Eval as the preliminary search phase and define another manifold, LocalMinimizer, to implement our choice of a local search method, we can easily modify our coordinator module to accommodate such a hybrid scheme. All we need to do is change line 55 into stop: getunit(best2) -> LocalMinimizer -> pr.

### 4.3 Adding a Visualizer
Visualizing the results of our parallel/distributed application of the previous section can be very informative. With MANIFOLD, this can be done in a straightforward way. We simply make another atomic manifold (called Show) and make some drain cocks in the MANIFOLD code of section 4.2 with the stream constructor -> and the = operator (which in a hidden way uses the stream constructor). This results in the MANIFOLD program below, which is almost the same as the code in section 4.2.

```plaintext
1 manifold pass1 import. 2 manifold variable import. 3 manifold Eval import. 4 5 manifold PrintObjects atomic {internal.}. 6 manifold Split atomic {internal.}. 7 manifold AtomicEval(event, port in) atomic {internal.}. 8 manifold Checker(port in, port in, port in, event, event) atomic {internal.}. 9 manifold Merger port in a, b. atomic {internal.}. 10 manifold Eval import. 11 12 #define TOL 1.0e-6 13 #define IDLE terminated(void) 14 15 /*********************************************************************/ 16 manifold Main 17 { 18 event checkit, goon, stop. 19 20 auto process best1 is variable. 21 auto process best2 is variable. 22 auto process pr is PrintObjects. 23 process show is Show.
```
The irregular grid in Figure 5 shows the iterative search process in the domain towards the best solution \( z = 3.000 \) at point \( (0.000, -1.000) \). In this figure we can (partly) follow the splitting sequence which, as we know from the previous section, uses the grids \( 2 \times 2, 2 \times 4, 2 \times 4, 4 \times 2, 2 \times 4, \) and \( 4 \times 2 \). Due to the scale of this figure, only a part (the first three grids) of this irregular splitting can be seen clearly.

The Show manifold is simple to implement in C using a portable graphic library (e.g. Phigs, GKS, OpenGL) and a portable widget library.
The adding of the visualizer clearly shows the “plumbing” aspect of MANIFOLD programming: no explicit action is necessary to “move” information around in MANIFOLD — provide the pipes (with \(\rightarrow\) or/and \(=\)) and the units will flow.

4.4 Computational Steering Through a GUI

In this section we extend the Show manifold of section 4.3 with a simple graphical user interface (GUI) with some steering facilities. With this GUI we can select a domain by mouse (by drawing a rectangle) and start (by pressing mouse buttons) the iterative recursive domain decomposition of section 4.2 on that domain. We call the work which has to be done on such a selected domain a “cluster.” The processes contained in clusters are spread out over the computers specified in a configuration file (see section 4.2). This is completely transparent to the user. A user only needs to supply a list of his favorite machines in a configuration file. Once a cluster is started by a mouse click, it sends its identification back to the GUI, which is shown in the area above the “show” button. In figure 6, we see three clusters \((c_0, c_1 \text{ and } c_3)\). Selecting, e.g., \(c_0\) and \(c_1\) (whose identifications are highlighted) and pressing the show button results in the GUI as shown in figure 6. In this figure, in addition to the global minimum of 3.000 at \((0.000, -1.000)\) which we know from the previous section, a local minimum of 84.000 at \((1.800, 0.200)\) is also shown.

![Figure 6: The simple GUI](image)

Another facility of the GUI is its ability to deactivate a cluster (by selecting the cluster and clicking on the deactivate button). Consequently, the selected area of a deactivated cluster gets a black border to indicate its termination. Also, the clusters which terminate normally get the same black border.

With this simple GUI we can interactively explore the domain of the optimization problem in a distributed/parallel fashion. Below we give the manifold source code for this distributed/parallel computational steering example.

```manifold
1 manifold pass1 import.
```
4. Domain Decomposition

After the detailed explanation of the previous examples, it is sufficient to mention only the differences with the MANIFOLD code in section 4.3. To create a unique identification for the clusters we use the process index which is an instance of variable (line 19). Every time a cluster is created, it reads this variable and increments it by 1 to get the index for the next cluster (line 54). Because the clusters run in parallel, we must prevent the situation where two or more clusters read and increment index at the same time. For this protection we use semaphores (line 5-7). Of
course, we should also adapt the manifold Show to the new requirements (to handle mouse input, etc.) but that is just a straight-forward adaption of the C code. The remainder of this code is either already discussed in previous examples, or is just details.

5. Conclusions

MANIFOLD is a new coordination language for orchestration of the cooperation among large sets of concurrent processes that comprise parallel and/or distributed applications. One of the advantages of MANIFOLD is that it makes no distinction (that is visible to a programmer) between distributed and parallel environments: the same MANIFOLD code can run in both. A unique characteristic of MANIFOLD is its separation of computation concerns from communication concerns into distinct program modules. This leads to reusable pure-computation and reusable pure-coordination modules with little dependency on their application environments. All these features make MANIFOLD a suitable framework for the construction of modular software to solve irregular problems on parallel and/or distributed platforms.

Our experiment using MANIFOLD for this type of applications deals with an instance of the classical optimization problem. The emphasis of our work is on the construction and validation of the protocol modules necessary for this and other (numeric and non-numeric) applications. MANIFOLD allows such coordination modules to be compiled separately (and in isolation from any computation code), and stored in protocol libraries, whereby they can be subsequently linked with various separately compiled pure-computation modules to build running applications. Thus, the same coordinator modules described in this paper can be (and, indeed, are) used in various other domain decomposition applications as well as other non-numeric applications that use a similar splitting scheme.

Another important feature of MANIFOLD is its underlying plumbing paradigm which makes it easy – as we saw in our examples with the addition of Show – to compose and re-compose MANIFOLD applications and adapt them to new requirements. This has also lead to very promising results in the area of restructuring of existing sequential programs to run on distributed and parallel environments[7]. We are beginning new joint projects where the practical utility of MANIFOLD will be evaluated in the context of real commercial applications, many of which involve parallel and/or distributed solutions to irregular problems.

References


