A Runtime Framework for Parallel Programs

Joy Mukherjee

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science & Application

Dr. Srinidhi Varadarajan (Chair)

Dr. Naren Ramakrishnan

Dr. James D. Arthur

Dr. Calvin J. Ribbens

Dr. Scott F. Midkiff

August 16, 2006

Blacksburg, Virginia

Keywords: Parallel Programs, Legacy Procedural codes, Lightweight Threads, Component Composition, Runtime Linking and Loading, Dynamic Adaptation.

A Runtime Framework for Parallel Programs

Joy Mukherjee

ABSTRACT

This dissertation proposes the Weaves runtime framework for the execution of large scale parallel programs over lightweight intra-process threads. The goal of the Weaves framework is to help process-based legacy parallel programs exploit the scalability of threads *without any modifications*. The framework separates global variables used by identical, but independent, threads of legacy parallel programs without resorting to thread-based re-programming. At the same time, it also facilitates low-overhead collaboration among threads of a legacy parallel program through multi-granular *selective* sharing of global variables.

Applications that follow the tenets of the Weaves framework can load multiple identical, but independent, copies of arbitrary object files within a single process. They can compose the runtime images of these object files in graph-like ways and run intra-process threads through them to realize various degrees of multi-granular selective sharing or separation of global variables among the threads. Using direct runtime control over the resolution of individual references to functions and variables, they can also manipulate program composition at fine granularities. Most importantly, the Weaves framework does not entail any modifications to either the source codes or the native codes of the object files. The framework is completely transparent.

Results from experiments with a real-world process-based parallel application depict that

the framework can correctly execute a thousand parallel threads containing non-threadsafe global variables on a single machine—nearly twice as many as the traditional process-based approach can—without any code modifications. On increasing the number of machines, the application experiences super-linear speedup, which illustrates scalability. Results from another similar application, chosen from a different software area to emphasize the breadth of this research, show that the framework's facilities for low-overhead collaboration among parallel threads allows for *significantly* greater scales of achievable parallelism than technologies for inter-process collaboration allow. Ultimately, larger scales of parallelism enable more accurate software modeling of real-world parallel systems, such as computer networks and multi-physics natural phenomena.

"I'd rather be a could-be if I cannot be an are;
because a could-be is a maybe who is reaching for a star.
I'd rather be a has-been than a might-have-been, by far;
for a might have-been has never been, but a has was once an are."

--Milton Berle

To Dadun.

To my grandparents.

To my parents.

To Mona.

Acknowledgements

I take this opportunity to thank the people who helped me in various ways throughout the time-span of this work:

- My advisor, Dr. Srinidhi Varadarajan, who has been a friend, a philosopher and a guide. He helped me at all stages of this work, from conceptualization to implementation.
- My Ph.D. committee—Dr. Naren Ramakrishnan, Dr. James D. Arthur, Dr. Calvin J. Ribbens, and Dr. Scottt F. Midkiff—for their suggestions, support and encouragement.
- Dr. Naren Ramakrishnan, for helping me apply this work to various practical software problems and for guiding me while publishing related results.
- Dr. Calvin J. Ribbens, for his help with innumerable official issues and for his technical inputs.
- Dr. Godmar Back, for his technical suggestions.
- The GNU Compiler Collection (GCC) mailing lists.
- The staff at the Department of Computer Science. In particular, Ginger Clayton and Melanie Darden .
- Sara Thorne-Thomsen for her help with reviews of this dissertation.

- My friends who made my stay a mix of more fun and less work: Omprakash Seresta, Bharath Ramesh, Arvind Kumar Sharma, Ankit Singhal, Anil Bazaz, Akbar Rizvi, Deepak Bhojwani, Sayed Ali Yawar, Navrag B. Singh, Sameer Mulani, Dhaval P. Makecha, Jon Bernard, Karthik Channakeshava, Veena Basavraj, Ayush Gupta.

- My colleagues at the Computing Systems Research Laboratory (CSRL): Bharath Ramesh, Craig Bergstrom, Patrick Liesveld, Vedvyas Duggirala, Hari Krishna Pyla, Pilsung Kang, Lee B. Smith, Joe Ruscio, Chris Knestrick.

- Special thanks to Craig, Hari and Chris for allowing me to use extracts from their work.

- My family for their support and love: (Late) Dolgobinda Mukherjee and Umarani Mukherjee; (Late) Satyendra Nath Banerjee and Sumita Banerjee; Swadhin Mukherjee and Anjali Mukherjee; Swaraj Mukherjee, Gargi Mukherjee and Puja Mukherjee; Anjan Banerjee and Mamata Banerjee; Arpan Banerjee.

- The staff at The Cellar Restaurant in downtown Blacksburg.

- Lastly, my fiancée Monalisa Chatterjee, for her patience and support during the toughest stages of this work.

Joy Mukherjee

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel computing systems are becoming pervasive. At one end are parallel and distributed systems such as clusters and distributed supercomputers and at the other are low-cost multi-core personal computers. The demand for such a range of parallel computing has fueled the development and adoption of parallel programs at all levels. As a result, software developers are incorporating parallelism into all sorts of software applications, ranging from compute-intensive programs such as scientific simulations to day-to-day utility programs such as browsers.

An important effect of the increased adoption of parallel systems is the programming of many contemporary applications explicitly for parallel and distributed platforms. Software modeling of common physical phenomena such as the weather, real-world networks such as the Internet, and so on comprise an inherent degree of parallelism. Weather modeling

consists of the joint effect of simultaneous, or parallel, software modeling of factors such as winds, ocean currents, the Sun, and so forth. Modeling the Internet consists of joint effects of simultaneous, or parallel, software modeling of network applications, protocol stacks, and so forth. Applications that model such real-world phenomena need to account for the parallelism inherent in the corresponding physical manifestations. It is therefore, natural and intuitive to program such applications with explicit support for parallelism.

1.1 Problem Synthesis

Many contemporary applications require support for large-scale parallelism. In general, applications such as weather simulations and computer network emulation produce better results and more accurate models when they can exploit larger scales of parallelism. To a certain extent, the logic behind this general observation is intuitive. For instance, consider network emulation. Larger scales of parallelism can help network emulators model a greater number of simultaneous network entities such as computers (or nodes), network applications, protocol stacks and so forth. These increased numbers, in turn, facilitate more accurate characterization of network dynamics such as traffic, load and topology. Ultimately, such characterization helps stringently test new protocols.

Such applications benefit from various mechanisms that aid large-scale parallelism. Multi-machine clusters and supercomputers facilitate large-scale parallelism through increased hardware resources. At the same time, mechanisms for better exploitation of an individual

cluster node or low-end shared-memory multi-processor (SMP) machine can also contribute to greater parallelism. This research focuses on the increased exploitation of an individual multi-core computer node or a single SMP machine for larger scales of parallelism.

Another element of this research is that it deals with applications that use *legacy procedural codes*. Many software applications that require support for large-scale parallelism also need to reuse legacy procedural codes developed, validated and verified through decades of research and usage. Network emulation and parallel scientific computing are two software areas that consist of many such applications.

- Network Emulation: As mentioned earlier, network emulation benefits from large-scale parallel modeling of simultaneous network nodes and applications. At the same time, network emulation entails the use of codes for TCP/IP[1] stacks, telnet, and so forth. To model the exact behavior of real-world networks, emulators must use operationally correct versions of these codes. However, the real-world versions have evolved through decades of regular use, and it is extremely difficult to develop, verify and validate new versions that can comprehensively replace their real-world counterparts. Therefore, network emulators reuse *unmodified* legacy real-world versions of these codes, which are written in procedural programming languages such as C.

- Scientific Computing: Scientific modeling of natural phenomena is another instance of large-scale parallel applications that use legacy procedural codes. Consider, for instance, simulating a gas turbine. Mathematical modeling of such a multi-physics

[1]TCP stands for Transmission Control Protocol [Ste97]. IP stands for Internet Protocol [Kne04].

scientific problem replaces the main problem by a set of smaller problems (on simple geometries) that need to be solved simultaneously. Corresponding software modeling instantiates a parallel solver program for each of the simpler problems and multiple parallel mediator programs to facilitate collaboration between the solvers. A typical problem is a complex composition of thousands of solvers and mediators. At the same time, solvers and mediators use scientific codes from different problem solving environment (PSE) toolboxes. Typical PSE toolboxes contain a legacy of computational routines verified and validated over decades of research. Most of these codes are written in procedural languages such as C and FORTRAN.

These two radically different software areas emphasize the widespread use of legacy procedural codes. From the perspective of this research, the three most important properties of legacy procedural codes are:

- They *cannot be modified without serious consequences*. The sheer size of some of these codes (such as the IP stack), the tremendous investments associated with them (such as decades of investment in scientific solvers) and the enormous software engineering task of revalidating modified versions are some important reasons behind this property.

- They often use *global variables*. For instance, telnet and IP stack codes are rich in global variables [Var02]. So are several scientific solvers [NAG06a].

- They are programmed in procedural languages such as C (telnet and IP stack [Var02]) and/or FORTRAN (scientific solvers [NAG06a]).

To summarize, the problem domain of this research consists of large-scale parallel applications that use legacy procedural codes.

1.1.1 Overall Goal

Before proceeding, it is important to clarify certain terms that are used repeatedly throughout the rest of this document:

- The phrase *Legacy Parallel Programs* is used to identify the problem domain, that is, large-scale parallel applications that use legacy procedural codes.
- The word *threads* is used to identify lightweight intra-process threads such as POSIX threads [NBF96] and GNU's user-level threads [Eng06].
- The word *transparent* is used to indicate "no code modifications".

Broadly, contemporary parallel programs follow one of two execution paradigms, threads or processes. Most legacy parallel programs use processes to realize parallel flows of execution. However, threads are lightweight and facilitate larger scales of parallelism [NBF96] than processes. Typical SMP workstations and multi-core processors can support more concurrent threads than processes. Furthermore, creation and runtime management of threads require less operating system (OS) action, which reduces overall overhead.

Most importantly, almost all parallel programs require some sort of collaboration between parallel flows of execution. For instance, a case of network emulation can require running

multiple parallel applications, such as telnet, over one IP stack (modeling multiple telnets on one virtual node). Here, parallel flows of execution for each telnet must collaborate within the IP stack. Also, as mentioned earlier, parallel mediators facilitate collaboration among solvers in multi-physics scientific simulations.

Processes use inter process communication (IPC) for such collaboration. However, IPC requires operating system (OS) action [Ram04], which adds extra overhead, thereby limiting the scalability of parallelism. In contrast, because threads run within a single process, inter-thread collaboration can be realized through low-overhead sharing of global variables.

Therefore, threads can help legacy parallel programs exploit larger scales of parallelism over individual nodes of a cluster and over individual SMP machines. Such exploitation leads to better utilization of the overall resources and ultimately facilitates more accurate modeling of large-scale parallel phenomena such as networks and multi-physics problems.

The overall goal of this research is to help "unmodified" legacy parallel programs exploit the scalability provided by threads.

1.1.2 Research Challenges

The fundamental problem obstructing progress towards this goal is best explained by an example. Suppose that a legacy parallel program comprises two identical, but independent, parallel flows of execution, each with its own copy of all global variables. The processes paradigm that is traditionally used for running such programs automatically creates virtual

machine abstractions to encapsulate each parallel flow of execution and the associated global variables. However, under the threads paradigm, the two flows of execution (intra-process threads) end up sharing all global variables and inadvertently interfere with each other thereby leading to erroneous behavior. For instance, in a case of network emulation, running two independent telnets over threads results in the inadvertent sharing of telnet's global variables among the two supposedly independent threads. Such sharing leads to erroneous modeling.

To the best of our knowledge, no technology exists that can alleviate this problem without code modifications. However, as mentioned earlier, modification of legacy parallel programs is not preferable. This observation leads to the first challenge encountered in this research:

the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

Separation of global variables conflicts with low-overhead sharing of global variables for collaboration, one of the main benefits of using threads in the first place. This conflict can only be addressed through *selective* sharing of global variables among threads of legacy parallel programs. Furthermore, such selective sharing needs to be exercised at arbitrary granularities, from a single variable to entire components. For example, emulation of multiple telnets on one virtual node requires the sharing of all global variables in an entire IP stack component among multiple telnet threads. Again, certain parallel scientific applications require the sharing of individual solution variables and boundary condition variables among different solver and mediator threads. The next chapter illustrates these examples in detail.

To the best of our knowledge, at present, multi-granular selective sharing of global variables among threads can only be realized through explicit programming according to the tenets of the threads paradigm. Because legacy parallel programs traditionally use processes for execution, most of their codes do not subscribe to such constructs. However, reprogramming legacy procedural codes according to thread-programming constructs is taboo in this research's problem domain. This observation leads to the second challenge encountered in this research:

the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

1.2 Solution Approach

Threads are runtime entities. They exist only at the runtime stage of a program's life cycle. Hence, this research takes an *entirely* runtime approach to address the challenges it encounters. At runtime, legacy procedural codes are available as native-code objects only. Therefore, this research approaches the encountered challenges at the level of native-code objects. At no point does the solution approach entail any modifications, either to source codes or to native codes. The approach is completely transparent.

1.2.1 Proposed Work

This research proposes the Weaves runtime framework for execution of unmodified legacy parallel programs over intra-process threads. The term "Weaved applications" is used for programs that follow the tenets of the Weaves framework. Weaved applications can load encapsulated modules from native-code object files. They can load multiple identical, but independent modules within a single process without any modifications to the concerned object file. Every module encapsulates all its global variables within itself.

Just as the compile-time linking of object files creates an executable program, the runtime composition of a set of modules creates a weave. A weave is, therefore, an intra-process subprogram that can support a flow of execution. Weaves composed from disjunctive sets of modules are completely independent subprograms within a single process. Additionally, the framework allows a single module to be shared among multiple weaves, which can be leveraged to realize arbitrary graph-like sharing of modules among different weaves. By allowing direct runtime control over the resolution of individual references to functions and variables in a module's code, the Weaves framework empowers programs with the ability to manipulate weave composition at fine granularities. When references from different weaves resolve to a single function or a single variable, the concerned function or variable is shared among the different weaves. Thus, direct control over the resolution of individual references can extend selective sharing to finer granularities.

All the components of Weaved applications, including the fundamental module, are intra-

process runtime entities. Also, the Weaves framework does not entail any code modifications either to the source codes of modules to the native codes of relocatable objects. The framework is completely transparent.

The Weaved version of a legacy parallel program can run threads through any weave, because a weave is an intra-process subprogram that can support a flow of execution. Threads running through independent weaves do not share any global variables. Thus, the Weaves framework transparently separates global variables used by identical, but independent, threads of a legacy parallel program. Furthermore, threads running through weaves that selectively share modules or variables, experience all the elements of this sharing. Therefore, the Weaves framework transparently realizes low-overhead collaboration through multi-granular selective sharing of global variables among threads of a legacy parallel program.

Together, the facilities of the Weaves framework help "unmodified" legacy parallel programs exploit the scalability provided by threads. Threads allow for larger scales of parallelism through better utilization of resources on a single multi-core node or SMP workstation. Ultimately, larger scales of parallelism aid accurate modeling of large-scale parallel phenomena such as networks and multi-physics problems.

1.2.2 Usability

At the basic level, the Weaves framework offers its services as a library. It supports simple Application Programming Interfaces (APIs) for loading modules, composing weaves, resolving

individual references and executing threads. Users can explicitly program the composition of Weaved applications using these APIs. The framework also provides a meta-language for specifying the composition of a Weaved application in a configuration file and a script that automatically creates and runs the application from the meta-description.

Essentially, configuration files encode the *composition* of modules in a Weaved application, where modules are runtime images of object files. This property makes them very similar to Makefiles, which encode the *composition* of object files in an executable program or a shared library. Consequently, from the usability perspective, composing Weaved applications is comparable to writing Makefiles.

1.2.3 Portability

The Weaves framework is currently implemented on GNU/Linux over three architectures: x86, x86_64, and ia64. The implementation is heavily dependent on the Executable and Linkable File Format (ELF) [TIS95], the format of native relocatable objects used by most GNU/Linux systems. Weaves' runtime loader and linker called Load and Let Link (LLL) implements the core aspects of the framework, which include loading modules, composing weaves and direct control over the resolution of individual references.

Since most Linux-based systems subscribe to ELF and related semantics, the implementation is fairly architecture-neutral. A port to the Power PC architecture is currently underway. Porting to different operating systems such as Windows and OS-X poses some problems,

because they do not comply with the ELF standard. However, most regular operating systems, including OS-X and Windows, allow the object file based decoupling of applications. Even though they have different formats, these object files are similar in structure and content to ELF relocatable objects. Theoretically, therefore, the Weaves framework is portable across a wide range of operating systems and architectures.

1.3 Lateral Technological Advances

The Weaves framework institutes lateral advances in software areas that frequently encounter legacy parallel programs. This research demonstrates these advances through experiments with large real-world network emulations and parallel scientific applications. These experiments in two radically different domains of parallel systems emphasize the broad impact of the Weaves framework.

The results from the experiments with network emulation show that Weaved realizations transparently exploit overall resources for larger scales of parallelism. A Weaved emulation can emulate a thousand virtual nodes over a single physical machine. On increasing the number of machines, the emulation speeds up super-linearly i.e. on increasing the number of machines by a factor of N, the time taken to run a certain emulation drops by a factor greater than N. In effect, such super-linear speedup exemplifies the scalability of Weaved network emulations.

The results of the experiments with parallel scientific applications show that Weaved realiza-

tions can support nearly twice the number of unmodified and non-threadsafe parallel solvers as the corresponding process-based realizations can. The results also show that Weaved applications can transparently exploit low-overhead collaboration among parallel solver threads through user-level sharing of global variables, which allows for *significantly* greater scales of achievable parallelism than corresponding IPC-based collaboration does.

These results together demonstrate that the Weaves framework is instrumental in helping legacy parallel programs transparently exploit overall resources for larger scales of parallelism. Ultimately, larger scales of parallelism enable more accurate testing of network protocols and more accurate modeling of multi-physics phenomena. These lateral advances, therefore, substantiate the efficacy of the Weaves framework.

1.4 Organization

The rest of this thesis is organized as follows. The next chapter, chapter 2 details on motivating applications from the areas of network emulation and parallel scientific computing. Chapter 3 discusses related work. Chapter 4 presents the elements of the Weaves framework. Chapter 5 illustrates Weaves-based approaches to network emulation and parallel scientific computing through various case studies. It also discusses the experiments and the associated results that elucidate the benefits of the Weaves framework. Chapter 6 contains concluding remarks. Finally, chapter 7 briefs on ongoing work.

Chapter 2

Motivating Applications

This chapter presents in detail motivating applications from the areas of network emulation and parallel scientific computing, that illustrate the specific requirements motivating the current work. This thesis resorts to these applications at various points to elucidate key concepts and insights. These applications, chosen for the purposes of illustration only, provide examples of two radically different areas of parallel software systems to emphasize the impact of this research on the broad range of parallel computing systems.

2.1 Network Emulation

Network emulation requires support for large-scale parallelism to model real-world networks such as the Internet. Furthermore, it requires real-world network applications and protocol codes that are typically legacy procedural codes. Therefore, many instances of network

emulation fall within the domain of legacy parallel programs that can benefit from the Weaves framework. The following background of network emulation highlights in greater detail its pertinence to this research.

The last several years have seen the deployment of network protocol development environments that allow users to create complex controlled experimental test-beds to verify and validate network protocols. SOftware researchers broadly classify protocol development environments into (a) Network simulators and (b) Direct code execution environments (DCEE). While simulators such as NS [NsN06, HK97, FV06], OPNET [OPN05, OPN06, Yuj01], REAL [Kes97], x-kernel [HP88, BP96], PARSEC [Ter01] and dummynet [Riz97] offer efficient event-driven execution models, they require that the protocol under test be written in their event driven model. Designers can refine the simulated protocol and then convert it to a real-world implementation. The basic problem is that there is no easy way to ensure the equivalence of the simulated protocol and its real-world version. A secondary issue in network simulators stems from their clean-room implementation. Because the TCP/IP protocol stack in simulators is written from scratch, it does not exactly emulate real-world protocol stacks. However, the idiosyncrasies of real-world TCP/IP can significantly affect performance [AP99, Pax99].

DCEEs such as ENTRAPID [HSK99, EII00], NIST network emulator [CS03], ModelNet [VYW+02] and MARS [AS94] solve the verification and validation problems by directly executing unmodified real-world code in a network test-bed environment. Network emulation naturally benefits from large-scale parallel modeling of simultaneous network nodes, and applications. Larger scales of parallelism can help network emulators model a greater number

of simultaneous network entities such as nodes, network applications, protocol stacks and so forth. The increased numbers, in turn, facilitate more accurate characterization of network dynamics such as traffic, load, and topology. Ultimately, accurate characterization of networks helps stringently test new protocols.

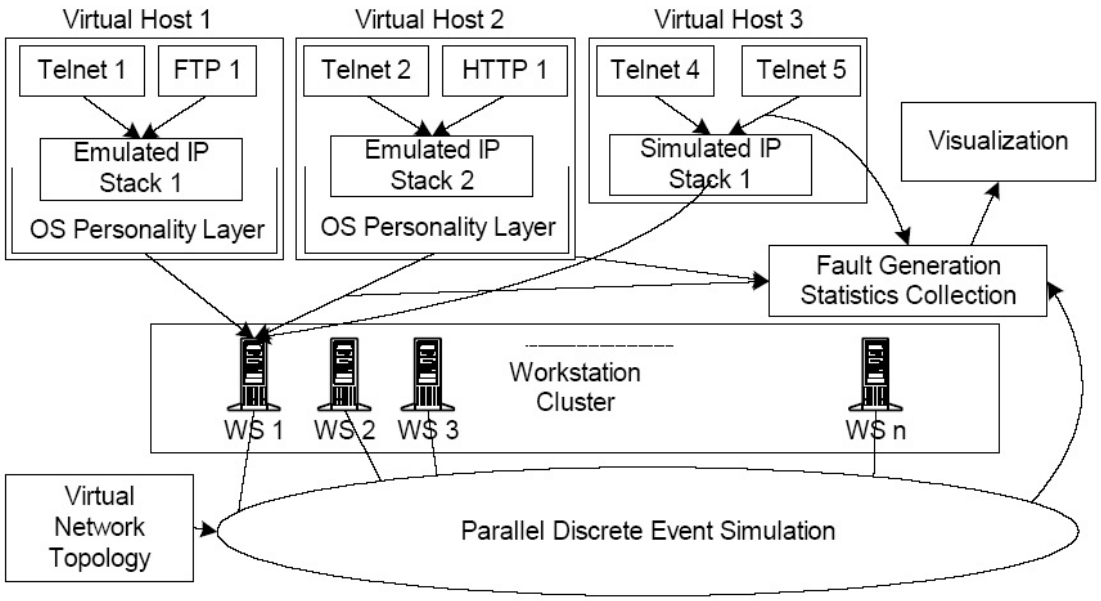


Figure 2.1: The DCEE test-bed called the Open Network Emulator (ONE). The ONE needs to model thousands of simultaneously (parallelly) running real-world network applications on a single workstation.

As Figure 2.1 shows, the DCEE testbed called the Open Network Emulator (ONE)[1] [Var02] needs to model thousands of simultaneously (parallel) running real-world network applications on a single machine. The goals of the ONE project are:

1. To provide a protocol development environment that models large-scale real-world

[1]Varadarajan envisioned the ONE for large-scale network emulations.

networks (hundreds of thousands of virtual network nodes).

2. To support direct execution of unmodified protocol and application code.

2.1.1 Network Emulation and Threads

These goals put the ONE squarely within the domain of large-scale parallel applications that use legacy procedural codes. The first goal, a development environment for modeling large-scale networks, requires support for large-scale parallelism. Larger scales of parallelism can help the ONE model a greater number of simultaneous network entities such as nodes, network applications, protocol stacks and so forth. The increased numbers, in turn, facilitate more accurate characterization of network dynamics such as traffic, load and topology. Ultimately, such characterization helps stringently test new protocols.

The second goal, direct execution of *unmodified* protocol and application code, implies use of traditional procedural codes. Specifically, it entails the use of codes for TCP/IP protocol stacks, telnet, ftp, and so on. To model the exact behavior of real-world networks, the ONE must use operationally correct versions of these codes. However, because the real-world versions have evolved through decades of regular use, it is extremely difficult to develop, verify and validate new versions that can comprehensively replace their real-world counterparts. For this reason, the ONE reuse *unmodified* legacy real-world versions of these codes that are written in procedural programming languages such as C.

Currently, DCEEs model all virtual network nodes and applications as parallel OS processes.

The large context switch time of processes and OS limits on the maximum number of processes (of the order of hundreds on typical workstations of today) inherently restrict the scalability of process-based solutions.

Even on a cluster supercomputer with hundreds of nodes, the ONE needs to model thousands of parallel network applications on a single physical machine (Figure 2.1). We argue that intra-process threads can help the ONE model larger networks than current process-based schemes [Var02]. Typical workstations of today can support more concurrent threads than processes [NBF96]. Therefore, the ONE can model a greater number of simultaneous network applications as parallel threads than as parallel processes. Furthermore, creation and runtime management of threads incur less OS action, which reduces overall overhead.

Lastly, but most importantly, threads run within the same process. This property can facilitate low-overhead collaboration among parallel network applications being modeled by the ONE through sharing of global variables. Such low-overhead collaboration, in turn, can be instrumental in lowering the overhead of modeling large-scale networks. The following discussion illustrates this idea.

Figure 2.2 shows a network scenario where two telnet applications run over a single IP stack. To model this scenario using processes, the ONE must link the telnet application against the IP stack and run two instances of the resultant program. As Figure 2.3(a) shows, such a process-based approach completely separates each instance of telnet. However, this model is erroneous, because network traffic from various applications can interfere with each other within the IP stack. Consider, for instance, a network node (machine) that supports both

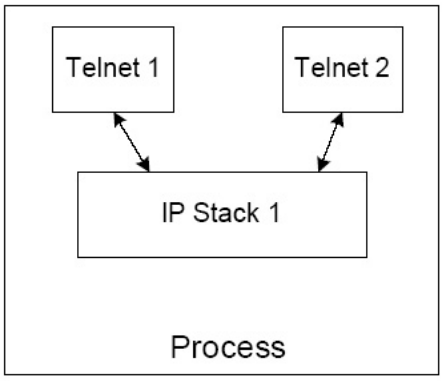


Figure 2.2: A simple network model with 2 telnets running over a single IP stack. The composition emulates a single virtual host.

real time applications such as videoconferencing as well as best effort applications such as file transfer progams (FTP). The traffic from the real time application interferes with the best effort FTP application within the IP stack, resulting in less than adequate performance for the real-time application.

For this reason, the ONE cannot model the joint effect of two telnets running over the same IP stack through this approach. While it is possible to synchronize the IP stacks on the two telnet processes through IPC mechanisms, such an approach entails either code modifications or extra overhead or both. Hence, IPC either violates transparent reuse of legacy codes or, at best, limits the scalability of network emulations, because IPC consists of high-overhead OS actions [Ram04]).

Figure 2.3(b) shows a model based on threads. In this case, the ONE must link the telnet application against the IP stack as before, and initiate two threads through the resultant program. Both threads run through the same instance of the IP stack. Thus, the model

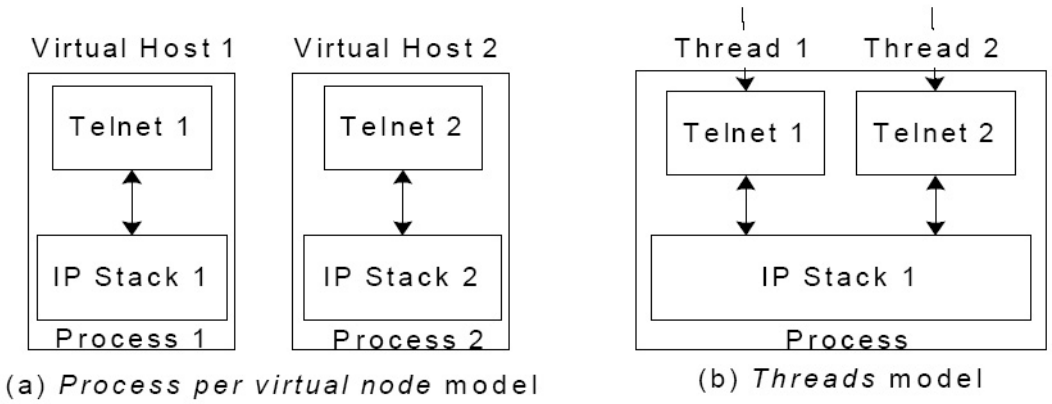


Figure 2.3: The composition shown in Figure 2.2 modeled under (a) process per virtual node model and (b) threads model. Neither of these models can emulate the desired real-world behavior without significant changes to telnet/IP codes and/or extra overhead.

benefits from low-overhead collaboration among the telnet threads within the *shared* IP stack. The actual real-world semantics of the two telnets interfering within the IP stack is closely captured without entailing any code modification or extra overhead.

2.1.2 Challenges

A major problem with the thread-based approach arises from updates to global variables. Consider the Figure 2.3 scenario once again. Telnet contains global variables and is non-threadsafe. Because threads share all global variables, a telnet thread modifying a global variable can inadvertently change the state of the other unrelated telnet thread causing erroneous behavior. The ideal solution to this problem requires two copies of all the global variables used in the telnet application. In programs that explicitly follow the threads

paradigm, the sharing of global variables is intentional. In the scenario discussed here, this sharing is neither intentional nor necessarily desirable.

The two conflicting needs at the crux of the ONE's problems when using threads are:

- The need to avoid sharing global variables between two telnet threads when they run through the telnet code.
- The need to share global variables between the same two threads when they run through the IP stack code.

Currently, there exists no technology that can fulfill the first need without modifications or additions to the telnet codes. However, as mentioned earlier, being a DCEE test-bed that runs real-world codes, the ONE does not prefer these modifications. Therefore, the first need of the ONE leads to the first research challenge, the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

The second need of the ONE sharing a single IP stack's global variables between two otherwise independent telnet threads. Furthermore, emulating advanced network scenarios, such as one depicted in Figure 2.4, entails *selectively* sharing different independent IP stacks among different sets of application threads.

To our knowledge, selectively sharing IP stacks among application threads can only be realized by explicitly reprogramming the IP stack code following constructs of the threads paradigm. However, since telnet and the IP stack are legacy procedural codes, such repro-

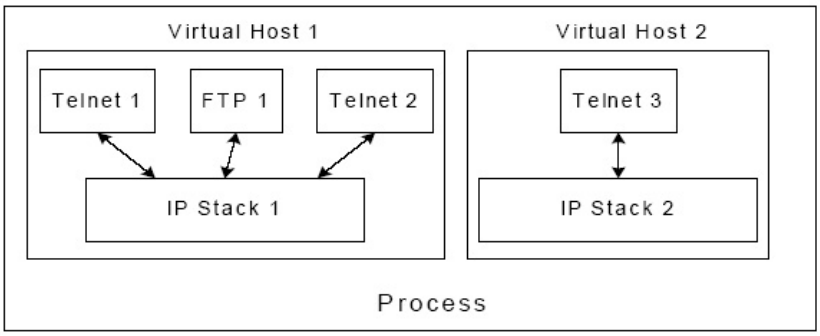


Figure 2.4: Advanced network scenarios entail *selectively* sharing different independent IP stacks among different sets of application (telnet, ftp) threads.

gramming is not preferable. The second need of the ONE, therefore, leads to the second research challenge, the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

2.2 Parallel Scientific Computing

Parallel scientific computing requires support for large-scale parallelism to model physical phenomena such as the weather. Furthermore, it requires using scientific solver programs that are typically legacy procedural codes. Therefore, many instances of parallel scientific computing fall within the domain of legacy parallel programs that can benefit from the Weaves framework. The following background of parallel scientific applications highlights in greater detail its pertinence to this research. Compared to network emulation, parallel scientific computing comprises more illustrative instances of multi-granular selective sharing of global variables among threads of legacy parallel programs.

Parallel scientific applications frequently use collaborating partial differential equation (PDE) solver programs [DHRR99] to model heterogeneous multi-physics problems. Consider modeling physical phenomena within a gas turbine. This software application requires combining simultaneous (parallel) models for heat flows (throughout the engine), stresses (in the moving parts), fluid flows (for gases in the combustion chamber), and combustion (in the engine cylinder). The mathematics of the problem describes each constituent model by a PDE with various formulations for the geometry, operator, right side, and boundary conditions. The basic idea here is to replace the multi-physics problem by a set of smaller problems (on simple geometries) that need to be solved simultaneously while satisfying a set of interface conditions.

2.2.1 Computational Background

Figure 2.5 shows how a set of smaller problems can replace a multi-physics problem. The computational basis of this idea consists of the interface relaxation approach to support a network of interacting PDE solvers [MR92]. Computational modeling of the multi-physics problem distinguishes between solvers and mediators. A PDE solver is instantiated for each of the simpler problems and a mediator is instantiated for every interface, to facilitate collaboration between the solvers. The mediators are responsible for ensuring that the solutions of the solvers match at the interfaces. The term "match" is defined mathematically (e.g., the solutions should join smoothly at the interface and have continuous derivatives) or by the physics of a specific problem (e.g., conservation constraints).

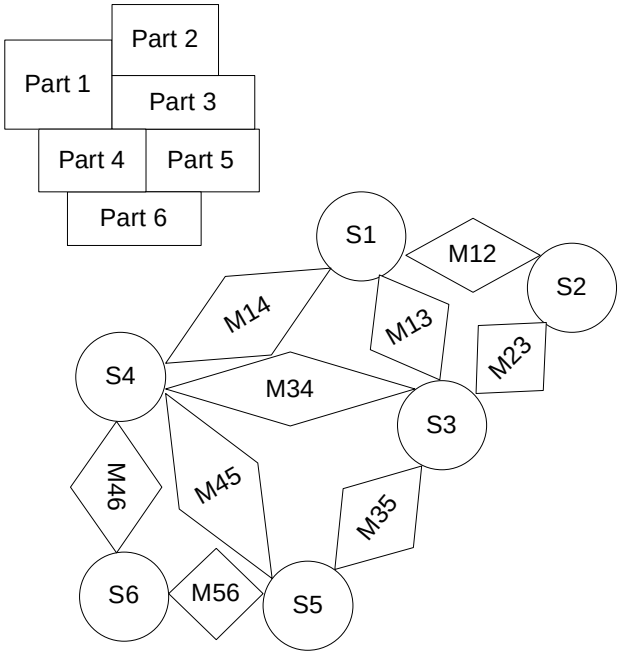


Figure 2.5: (Above) Composite multi-physics problem with six sub-domains. (Below) A network of collaborating solvers (S) and mediators (M) to solve the composite PDE problem. Each mediator is responsible for agreement along one of the interfaces.

Since the interface relaxation approach is often confused with classical domain decomposition, it is helpful to highlight the differences. In domain decomposition, one PDE problem is split into sub-domain problems of the same type that the (unknown) solution values at the interface points interconnect, thereby creating a single underlying PDE for all sub-domains. In interface relaxation, each sub-domain can have its own PDE and the interface conditions are generally derived from the underlying physics of the problem. Schwartz splitting [Mu99] is a meeting point between these two extremes where the sub-domains have the same PDE, but are coupled through continuity of solution and its derivatives. Interface relaxation is, therefore, the most general approach to modeling complex physical phenomena.

$$UU_{xx} + (1 + U)U_{yy} + aU(1 + U) = b(x^2 + y^2 - 2) \text{ Domains 1,4}$$

$$U_{xx}/(1 + (x - y)^2) + U_{yy}/(1 + (4x - 5y)^2) + cU/(101 + U) = 0 \text{ Domains 2,3}$$

$$U_{xx} + U_{yy} - d(U_x + U_y) + cU = 0 \text{ Domains 5,6}$$

Figure 2.6: PDEs defined over six sub-domains of the boiling mechanism shown in Figure 2.5.

As an example, Figure 2.6 shows the model of a boiling mechanism formulated by adapting ideas from [Ric98]. The formulation comprises six sub-domains and the following PDEs defined over them (). Here, U_{xx} and U_{yy} denote second order partial derivatives; U_x and U_y denote first order derivatives of the unknown function U(x,y). The PDEs are not common across the sub-domains. The PDEs in each sub-domain must be solved in an inner loop of the interface relaxation routine which then applies an 'averaging' or 'smoothing' formula to ensure that the solutions agree across sub-domain boundaries. The simplest approaches

simply exchange solution values across the boundaries, but there are more complex relaxation formulas as well [RTV99].

Figure 2.7 illustrates typical solver and mediator codes. A solver takes as input a PDE structure identifying the operator, geometry, right side and boundary conditions, and computes solutions using some computational routine (`PDE_solve`), such as finite difference or finite element approximation. The PDE problem characteristics determine the choice of the `PDE_solve` routine from a problem solving environment (PSE) [VR05] toolbox. Typical PSE toolboxes contain a legacy of computational routines verified and validated over decades of research. Even if `PDE_solve` routines implement different algorithms, they could use identical global symbols and signatures. Further, a certain problem instance might entail the same solver on multiple sub-domains or require different solvers or both. After computing solutions, a solver passes the results to mediators and waits until the mediators report back fresh boundary conditions. Upon the receipt of new conditions, it re-computes the solutions and repeats the whole process till a satisfactory state is reached. From the solver perspective, the semantics of interaction with the mediators are not much different from a functional interface.

The computational definition of a mediator states that it should be "capable of accepting values from solvers, apply relaxation routines (`Relax_soln`) and return improved values to the solvers" [Ric98]. A mediator should, therefore, be able to collaborate with multiple solvers through exchange of solution structures and boundary condition variables. As in the case of the solvers, the problem characteristics dictates the choice of the `Relax_soln` routine

from legacy PSE toolboxes. Once again, multiple Relax_soln routines can expose identical names and signatures, and a certain problem instance might entail identical or different mediators or both. Another important property is that different mediators and associated relaxation routines can either (1) require all solutions at once, or (2) use solutions as they are needed and become available.

The properties of parallel scientific applications that are relevent to this research are summarized as follows:

1. They use thousands of parallel solvers and mediators to model realistic problem instances such as turbines and heat engines.

2. They reuse unmodified legacy computational routines and programs from scientific PSE toolboxes that have been verified and validated over decades of research.

2.2.2 PDE Solvers and Threads

These properties put PDE solver programs within the domain of large-scale parallel applications that use legacy procedural codes. The first property, requires support for large-scale parallel solvers and mediators. Larger scales of parallelism can help realize a greater number of simultaneous solvers and mediators. The increased numbers, in turn, facilitate fine-grain decomposition of multi-physics problems. Ultimately, fine-grain decomposition aids accurate characterization of physical phenomena. The second property implies use of traditional procedural codes written in C and FORTRAN (languages that have found traditional favor in

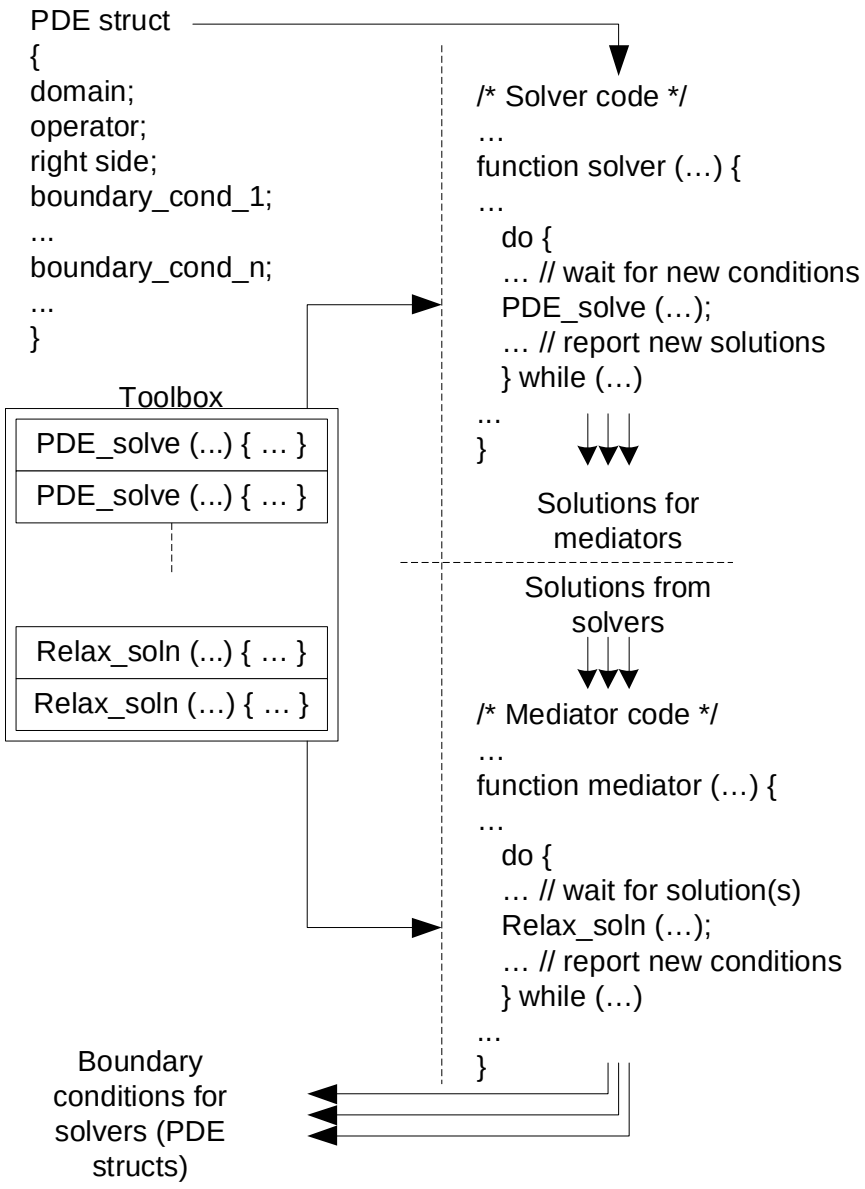


Figure 2.7: Typical solver and mediator codes. A solver takes as input a PDE structure identifying the domain, operator, right side, boundary conditions, and computes solutions. A mediator accepts values from solvers, applies relaxation formulas, and returns improved boundary condition estimates to the solvers. PDE_solve and Relax_soln routines are chosen from a PSE toolbox.

scientific computing due to generally better performance and lower overhead). Typical PSE toolboxes contain a legacy of computational routines verified and validated over decades of research.

The traditional software approach to collaborating PDE solvers consists of agent technology [DHRR99] in highly distributed environments such as clusters and distributed memory machines. Agent-based schemes model simultaneous solvers as parallel processes running on different machines in a distributed computing system. They also model mediators in the same manner. Agent-based solutions use message passing for exchange of solutions and boundary conditions among solvers and mediators. In highly distributed environments, with solvers running on different physical machines, message passing is often the only means of exchanging solution or boundary condition information. Message passing also facilitates flexible composition of varied multi-physics problems through transparent decoupling of solvers and mediators.

Intra-process threads can help model larger PDE solver instances than current agent-based schemes. Typical SMP workstations of today can support more concurrent intra-process lightweight threads than processes. A thread-based approach, therefore, can realize a greater number of simultaneous solvers and parallel and mediators than process-based agents. Furthermore, the creation and runtime management of threads incur less OS action, which reduces overall overhead [NBF96].

More importantly, intra-process threads run within the same process. This property can be used to facilitate low-overhead collaboration among solvers and mediators through user-

level sharing of solution structures and boundary condition variables. Compared to message passing among process-based agents, which entails OS-level IPC actions, such user-level sharing of solutions and boundary variables incur lower overhead.

2.2.3 Challenges

The first issue with a thread-based approach arises when PDE solver instances comprise multiple identical solvers and mediators. Figure 2.8 illustrates this issue. It depicts a simple parallel scientific application, comprising two solvers (S1 and S2) and one mediator (M12). If S1 and S2 are identical, they cannot be run over threads within the same process unless they do not contain global variables. However, many legacy scientific solvers contain global variables and are non-threadsafe [NAG06a]. If such solvers are used, a thread-based realization leads to an inadvertent sharing of the global variables between S1 and S2, thereby resulting in erroneous behavior.

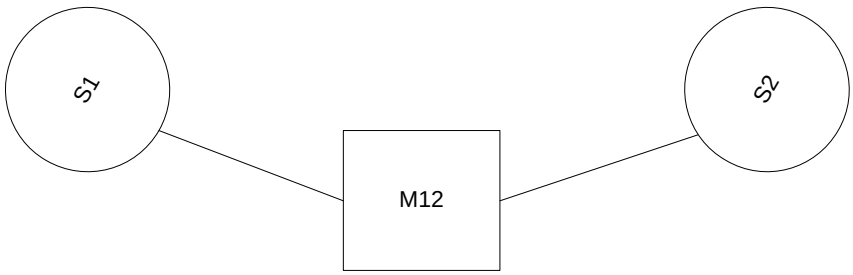


Figure 2.8: Simple instance of collaborating PDE solvers. Mediator M12 relaxes solutions from solvers S1 and S2.

Currently, no technology exists that can address this issue without modifications or additions

to the solver and mediator codes. However, modifications or additions to scientific programs are taboo due to decade long investments in verification and validation of legacy solver codes. Essentially, this issue leads to the first research challenge, the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

At the same time, realistic multi-physics scenarios require exchange of solutions and boundary conditions among solvers and mediators. To exploit low-overhead collaboration, the solver and mediator threads need to share solution *data structures* and boundary condition *variables*. Due to potentially random scenarios, such sharing needs to be *selectively* instituted at the *fine granularity* of individual data structures and variables. For instance, in the scenario depicted in Figure 2.5, solver thread S4 needs to share different boundary variables with four possibly identical mediator threads M14, M34, M45, and M46.

Currently, fine-grain selective sharing of boundary variables among solver and mediator threads can only be realized by explicitly reprogramming the solver and mediator codes following constructs of the threads paradigm. However, because solvers and mediators comprise legacy procedural codes, such reprogramming is not preferable. This observation leads to the second research challenge, the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

2.3 Summary

This chapter has provided detailed descriptions of examples from the areas of network emulation and parallel scientific computing. These example applications have illustrated the specific requirements that motivate the current work. The radically different software areas of these motivating applications emphasize the broad impact of this work.

These descriptions have demonstrated that each of the motivating applications fall within the problem domain of this work, large-scale parallel applications that use legacy procedural codes. Larger scales of parallelism can help network emulators model a greater number of simultaneous network entities such as nodes, network applications, protocol stacks and so forth. The increased numbers, in turn, facilitate more accurate characterization of network dynamics such as traffic, load and topology. Ultimately, accurate characterization of networks helps stringently test new protocols.

In the area of parallel scientific computing, larger scales of parallelism can help realize a greater number of simultaneous solvers and mediators. The increased numbers, in turn, facilitate fine-grain decomposition of multi-physics problems. Ultimately, fine-grain decomposition aids accurate characterization of physical phenomena.

This chapter has illustrated the manner in which our motivating applications benefit from the overall goal of this research. Even on a cluster supercomputer with hundreds of nodes, large-scale network emulation needs to model thousands of parallel virtual nodes and applications on a single physical machine. Because SMP workstations of today can support

more threads than processes, thread-based approaches can help model larger networks than current process-based schemes. Furthermore, creation and runtime management of threads incur less OS action, which reduces overall overhead. Most importantly, intra-process threads run within the same process. This property facilitates low-overhead collaboration through user-level sharing of global variables among emulated parallel network applications.

Similarly, in scientific computing, thread-based approaches help realize larger parallel scientific solver instances than current agent-based schemes. Here also, threads facilitate low-overhead collaboration among parallel solvers and mediators through sharing of global boundary condition variables, which lowers the overhead of modeling complex multi-physics problem scenarios.

Finally, this chapter has illustrated the manner in which the motivating applications lead to the research challenges. Certain cases of network emulation require running multiple identical, but independent telnet threads within a single process. Certain scientific applications require running multiple identical, but independent, PDE solver threads within a single process. Since telnet and PDE solvers are legacy procedural codes that frequently contain global variables, these requirements lead to the first research challenge, the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

Again, certain cases of network emulation require selectively sharing different IP stacks among multiple independent telnet threads. At the same time, certain scientific applications require arbitrary sharing of individual boundary variables among various independent solver

and mediator threads. These requirements together lead directly to the second research challenge, the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

Chapter 3

Related Work

This chapter presents a comparison of the Weaves framework, and its benefits for legacy parallel programs, to related work. Currently, no technology exists that can address either of the challenges outlined in this research without instrumenting modifications on legacy procedural codes. At a more fundamental level, to the best of our knowledge, thread-based execution of *unmodified* legacy parallel programs that contain global variables and are non-threadsafe has never been attempted. Additionally, most related attempts resort to explicit programming based on the threads paradigm to facilitate selective sharing of global variables among threads.

Comparing the Weaves framework to some powerful approaches to parallel programming underscores its uniqeuness and highlights its capability of running process-based legacy parallel programs over threads without any modifications. For simplicity and clarity, scientific

applications comprising collaborating PDE solvers (see Chapter 2) are used to illuminate the strengths and weaknesses of different approaches. Finally, for conciseness, each of these related approachs has been analyzed using a single representative work from the literature on it.

3.1 Concurrent Approaches (Linda)

Intra-process threads naturally lead to concurrent programming techniques, where multiple control flows execute within one process. Concurrent programming [GB82, CG89, DGTY95, Fos96, Sat02, OAR04, BKdSH01] also offers a way around IPC overhead—global in-memory data structures—for inter-flow collaboration. Linda [GB82, CG89] is an exemplary representative of general concurrent programming techniques, and therefore, the strengths and weaknesses of a Linda-based approach are broadly valid across all concurrent approaches.

In a Linda-based scheme, each solver and mediator instance is a tuple implemented as a lightweight thread inside a single process (tuple space). Boundary conditions and solution structures are manifested as data tuples that are written to and read from the same global tuple space. This scheme is scalable and allows low-overhead collaboration among solvers and mediators through multi-granular selective sharing of global variables.

The biggest problem with a Linda-based approach is "lack of encapsulation in a global tuple space" [Fos96] within a single process. Certain scientific applications comprise multiple identical, but independent, legacy PDE solvers that contain global variables. Concurrent

techniques, as Linda exemplifies, offer no mechanism to transparently separate these solvers in intra-process environments.

In contrast, a Weaves-based approach facilitates true encapsulation of legacy procedural codes in intra-process environments. It allows legacy parallel programs to run multiple identical, but completely independent, threads within a single process. Each thread has its own independent set of global variables. Most importantly, facilitating encapsulation does not entail any modifications to the concerned codes either at the source-level or at the native-code level.

Another problem with Linda is its radically different programming methodology of the tuple space, which has hindered adoption. In contrast, the Weaves framework stays within the broad realms of procedural programming. The current work can, however, easily merge with a tuple space model[1].

3.2 Compositional Approaches (PCOM2)

Most parallel frameworks are targeted at increasing efficiency of applications through parallelization. Darlington, Guo, To and Yang [DGTY95] use special source-level constructs called skeletons to provide compositional parallelism and reusability of existing sequential code. Jade [RL98], ORCA [BBH+98] and Strand (PCN) [FOT92, FT89] address issues pertaining to parallelism, sharing of program elements, and code reuse. All of them, however,

[1]See Mukhrejee [Muk02] for a discussion on the tuple-space aspects of the Weaves framework.

approach the problem from the source language perspective, use high-level constructs and are ultimately based on using multiple processes.

Certain compositional parallel programming models [JL05, CCA04, Fos96, MDB03, SteJr90], address componentization issues in concurrent languages. They use coordination languages to componentized existing procedural codes and compiler-based techniques to compose a parallel program. PCOM2 [MDB03] illustrates the general pros and cons of approaches in this category. Consider a simple PCOM2 approach, a start component spawns solver and mediator components that run over parallel threads and exchange information through global data structures.

The setup is similar to a Linda-based approach as are the issues. The problem is that none of these compositional schemes enable true encapsulation because all components still inhabit the same namespace. Essentially, compositional parallel programming models cannot encapsulate multiple identical, but independent, copies of traditional procedural code components. Consequently, they cannot prevent inadvertent interference of among threads running identical, but independent, legacy PDE solvers containing global variables.

One advantage of some of these techniques is that being compiler-based they can automatically modify codes at compile time. However, even though automatic, such schemes entail code modification (patching), which leads back to verification/validation issues within the problem domain of legacy parallel programs. Moreover, developing such an automatic patching tool that is generic across various parallel software systems is an enormous software engineering task.

In contrast, a Weaves-based approach facilitates true encapsulation and multiple instantiation of legacy procedural codes in intra-process environments. Most importantly, in facilitating encapsulation, it does not entail any modifications to the concerned codes either at the source-level or at the native-code level.

3.3 Component-based Approaches (OOP)

It is possible to encapsulate and modularize traditional procedural codes at runtime using component-based technologies such as CORBA [OMG06], COM/DCOM [MS06], and Object Oriented Programming (OOP). Technologies such as CORBA and DCOM are mainly meant for distributed environments. OOP, on the other hand, is more pertinent to intra-process environments. Hence, OOP is used as a general representative of other technologies in this category.

In an OOP-based [CK01, PCB94, LM01] approach, solvers and mediators are classes that are instantiated and composed depending on a particular problem instance. Encapsulation and multiple instantiation primitives of OOP help separate identical solver and mediator components along with their global variables. Multiple parallel intra-process threads are fired off at entry functions of each solver and mediator to execute a PDE solver application. The solvers and mediators can communicate through user-level shared data structures at various granularities. Alternatively, mediator components (objects) can be shared between multiple solver threads by passing them as parameters to the associated solver objects. Both

these mechanisms allow low-overhead selective access to shared global variables from within solver and mediator components.

The problem with this approach is that solver and mediator codes must be available as classes, a source-level construct specific to object oriented languages (OOL). To get around this, solvers and mediators must be reprogrammed or the traditional procedural codes must be wrapped within classes written in an OOL. However, while recoding violates the transparency requirement, wrapping is complex and potentially expensive in terms of performance. Finally, even if solver and mediator classes are available, OOP creates a gap between the graphical network-like description of solvers and mediators in a scientific application and its actual runtime image. To bridge this gap, users much program the application structure in an OOL to instantiate and compose objects from classes, a non-trivial task because automatically translating a high-level meta-description or a graphical problem representation to OOP code requires sophisticated tools specific to collaborating PDE solver applications. The OOP approach, therefore, lacks flexibility and domain neutrality.

In contrast, a Weaves-based approach facilitates *transparent* encapsulation and multiple instantiation of legacy procedural codes in intra-process environments. It does not entail any source-level intervention, such as wrapping. A Weaves-based approach, therefore, maintains the simplicity and efficiency of normal procedural programming. Furthermore, a Weaves-based approach *transparently* facilitates multi-granular selective sharing of global variables among parallel threads of legacy parallel programs. Compared to programming complex program structures in OOP, writing a Weaves-based configuration file is simpler and more

intuitive.

3.4 Summary

This chapter has presented a comparison of the Weaves framework, and its benefits for legacy parallel programs, to related work. This comparison shows that currently, no technology exists for transparent separation, or transparent multi-granular selective sharing, of global variables among threads of a legacy procedural program. It also shows that, at a fundamental level thread-based execution of *unmodified* legacy parallel programs that contain global variables and are non-threadsafe has never been attempted. Most related attempts resort to explicit programming based on the threads paradigm to facilitate selective sharing of global variables among threads.

Comparing the Weaves framework to some powerful approaches to parallel programming highlights its strenghts. Existing concurrent programming techniques such as Linda [GB82, CG89] offer no mechanism to transparently global variables of identical, but independent, threads legacy parallel programs. Existing compositional parallel programming models such as PCOM2 [MDB03] suffer from similar constraints. Existing component-based technologies such as OOP [CK01] need to wrap legacy procedural codes within classes, which adds extra overhead.

On the other hand, a Weaves-based approach facilitates transparent runtime (intra-process) encapsulation and multiple instantiation of legacy procedural codes. It does not entail source-

level intervention, such as wrapping, or code modification. The Weaves framework maintains the simplicity and efficiency of normal procedural programming. Furthermore, a Weaves-based approach facilitates transparent multi-granular selective sharing of global variables among parallel threads of legacy parallel programs. Writing a configuration file for a Weaves-based parallel program is simpler and more intuitive than programming the structure of a parallel program in OOP.

Chapter 4

The Weaves Framework

This chapter provides a description of the elements of the Weaves runtime framework for parallel programs. "Weaved applications" is the terminology used for programs that subscribe to the tenets of the Weaves framework. The chapter consists of four parts: the definition of the components of a Weaved application, the developmental aspects of Weaved applications, the details of the current implementation including preliminary evaluation, and a discussion of some of the properties of Weaved application that illustrate the manner in which the framework helps address the research challenges outlined in Chapter 1.

Because threads are runtime entities, both the challenges involve runtime aspects. Therefore, the Weaves framework takes a runtime approach to address them. At runtime, legacy procedural codes are available as native code objects only. Hence, the framework approaches the challenges at the level of native code objects. To a great extent, a native code based

approach helps avoid modifications to legacy procedural codes.

To facilitate transparent separation of global variables between identical threads, the first challenge, the framework takes a component-based approach. In effect, this approach helps load encapsulated native code object components into the runtime environment of programs. Furthermore, the framework treats relocatable objects (.o files) as loadable components. Relocatable objects can be flexibly created at multiple granularities without concerns such as referential completeness. They also maintain information on individual *references* (to functions and variables) in their code. These properties of relocatable objects help the framework facilitate transparent multi-granular selective sharing of global variables among threads, the second challenge.

Since they are referentially incomplete, relocatable object components are not runnable by themselves. One or more of these components must be composed (linked together) into referentially complete program images. The Weaves framework takes a runtime approach to the composition of relocatable object components. Along with the multi-granular decoupling inherent relocatable object components, this approach to runtime composition empowers the framework with the ability to transparently realize arbitrary selective sharing of global variables among threads of legacy parallel programs. Finally, at no point does the Weaves framework resort to any code modifications either at the source level or at the level of relocatable objects (native code patches). The framework is completely transparent.

4.1 Component Definitions

The Weaves framework defines the following components of a Weaved application. The terminology that follows was chosen (a) to avoid overloading established terms; and (b) to make it clear that the framework is not bound to any established paradigm:

- Module: A module is the intra-process runtime image of a native code relocatable object (.o file) [TIS95]. It is the main unit of encapsulation in the framework. A module in the Weaves framework corresponds to an object in OOP. It can be programmed in any compiled language (such as C, C++, and FORTRAN). Each module defines its own namespace and encapsulates all its global variables. Weaved applications can load multiple identical modules from a single relocatable object file (multiple instantiation) without requiring any modifications to the concerned object. Encapsulation allows each copy to have its own independent namespace and global data within the address space of a single process. The Weaves framework allows modules to be dynamically loaded from relocatable objects without requiring referential completeness. Consequently, a module's code can contain undefined references to external program elements (in short, external references). Weaved applications have direct access to most[1] *global references*[2] contained in a module's code. They can control the resolution of each individual global reference.

- Weave: A weave is at the core of the Weaves framework. It consists of a collection of one

[1] Aliased references such as dynamically assigned pointer variables are not handled at this time.

[2] "Global references" must resolve to global functions and variables. They subsume external references.

or more modules composed (linked) together at runtime. A weave can support a flow of execution. It is an intra-process subprogram that unifies namespaces of constituent modules. Therefore, identical modules should not be included within a single weave (just as two copies of the same object file cannot be linked into the same executable program). However, different weaves can comprise similar, but independent, modules. Going a step further, the Weaves framework allows a single module to be part of multiple weaves. This property lays the foundation for selective sharing within the Weaves framework.

Just as the compile-time linking of object files creates an executable, the dynamic composition of modules creates a weave. Both comprise resolution of external references among constituent objects/modules [TIS95]. However, there are certain differences:

- Weave composition is an intra-process runtime activity (where modules are intra-process runtime entities).
- Weave composition does not necessitate the resolution of all references within constituent modules.
- Weaved applications can exercise direct control over the resolution of individual global references (to definitions such as functions and variables) to fulfill referential completeness or to transcend weave boundaries/limitations.
- External references from a shared module might need to resolve to different definitions in different weaves. Using late binding mechanisms, the Weaves framework performs weave-dependent resolution of such external references at runtime.

- String: A string is the unit of execution in the framework. It is an intra-process thread that can be dynamically initiated and managed. A string can be either a kernel-level (OS-level) thread or a user-level thread. A string executing within a referentially complete weave is similar to a process running through the runtime image of an executable program. Multiple strings can simultaneously (parallely) run through the same or different weaves.

- Co-strings: Co-strings are strings that execute within the same weave.

- Monitor: Because a Weaved application must load modules and compose weaves at runtime, it requires a minimal bootstrapping module, which runs on its main process thread. This bootstrapping module is responsible for setting up the Weaved application, that is, loading modules, composing weaves and starting strings. The framework permits an application to customize the main processs functionality after it has started strings. The term 'Monitor' is used for the main process, because it can be customized to *monitor* and externally/asynchronously modify a Weaved application's constitution/functionality at runtime. Such rewiring capabilities of Weaves are, however, not pertinent to the central theme of this thesis. Interested readers can consult Mukherjee and Varadarajan [MV05b] for more information.

- Tapestry: A tapestry is a single Weaved application comprising an entire composition of modules, weaves, strings and a monitor. The physical manifestation of a tapestry is a single operating system (OS) process.

Figure 4.1 depicts the components of a Weaved application. A Weaved application (tapestry), including all strings and the monitor, runs as a single OS process.

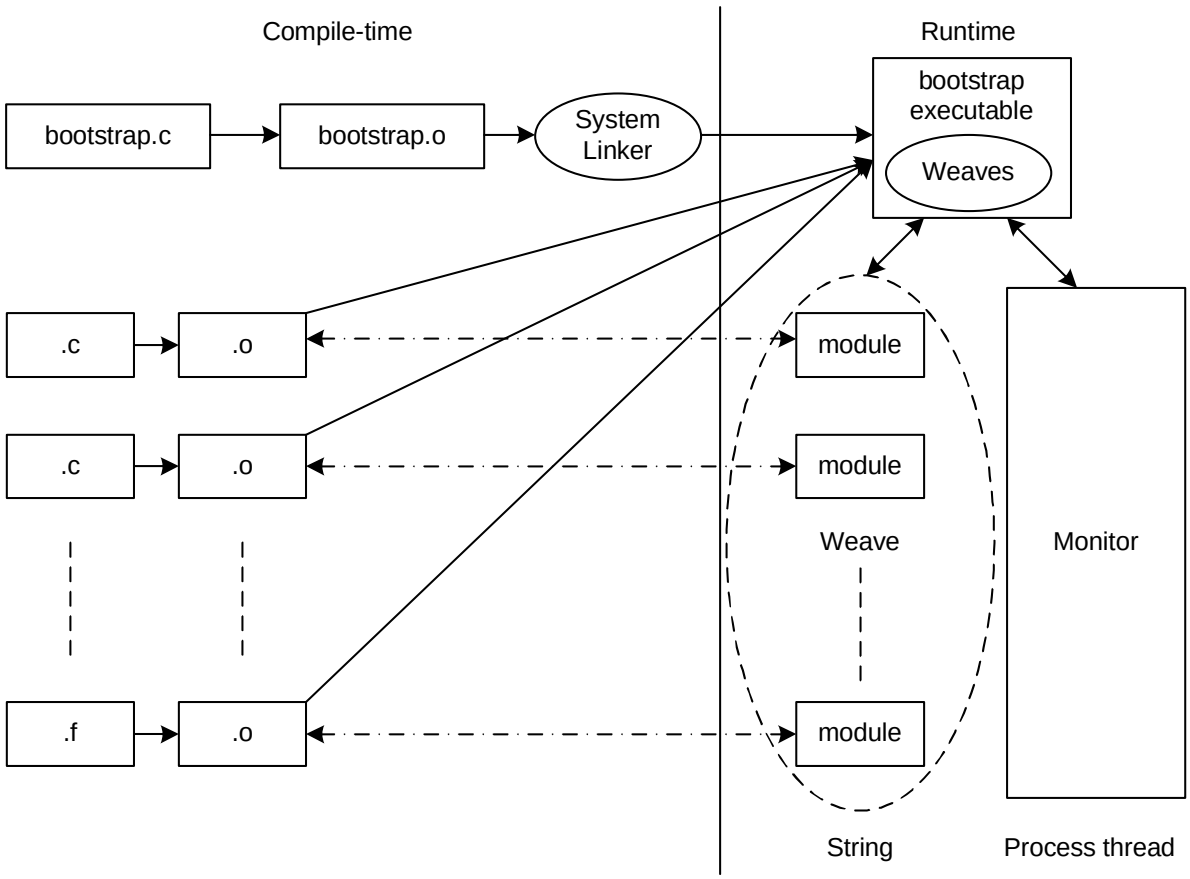


Figure 4.1: Components of a Weaved application: modules, weaves, strings, and the monitor. All components are intra-process runtime entities.

4.2 Developmental Aspects of Weaved Applications

At the basic level, the Weaves framework offers its services as a library. It supports 5 simple Application Programming Interfaces (APIs). Table 4.1 describes actions, inputs,

and compositional issues associated with each API. Further detials are mentioned under Implementation.

Users can program a bootstrap module to load modules, compose weaves and initiate strings using these APIs. They must then compile the boostrap module, link it to the Weaves library and run it as a normal executable program (Figure 4.1). Figure 4.2(b) depicts bootstrap pseudo code for the generic Weaved application shown in Figure 4.2(a). The framework also provides a meta-language for specifying application tapestries in a configuration file and a script that automatically generates a bootstrap module, builds it and runs the resultant executable program. Figure 4.2(c) depicts the configuration file for the Weaved application shown in Figure 4.2(a). One direction of current research on Weaves aims at an integrated Graphic User Interface (GUI) for tapestry specification and automatic execution.

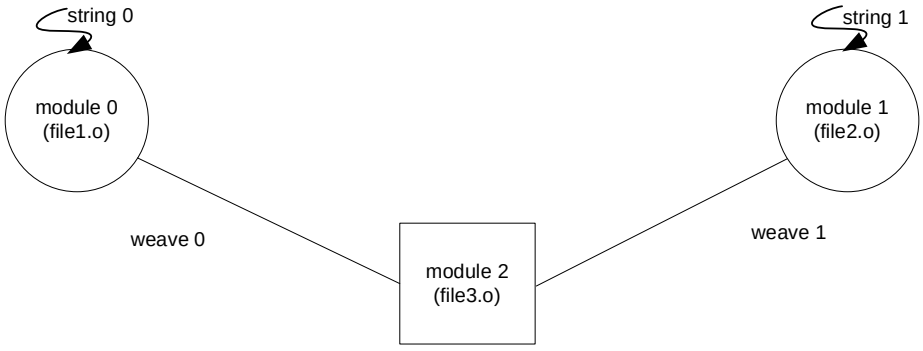
Figure 4.3 diagrammatically illustrates the complete process for developing a general Weaved application. An important aside is that the actual runtime structure of a Weaved application matches the high-level composition specification. This correspondence between execution and specification, and the minimal application-specific information in bootstrap modules, make automatic tapestry generation simple and generic across a diverse set of applications. In fact, we have used the same meta-language and script to generate tapestries for various network emulations as well as different parallel scientific applications.

The Weaves framework also provides several general purpose APIs:

- *Query APIs* provide information about a string, weave or module (The framework

Table 4.1: The API of the Weaves framework. Actions, inputs, and compositional issues associated with each API. Further detials are mentioned under Implementation.

API	**Action**	**Inputs**	**Issues**
`module_load`	Loads a runtime module from an object file on disk.	Location of the object file and an identifier (32-bit unsigned integer) for the module.	Splitting an application into appropriate object files. Providing a unique identifier for the module.
`weave_compose`	Composes together a set of modules into a weave at runtime.	The constituent module identifiers and an identifier (32-bit unsigned integer) for the weave.	Assuring the compatibility of the constituent modules. Assuring completeness and unambiguousness of the weave. Providing a unique weave identifier.
`resolve_ref`	Binds a reference to a definition.	The symbols for the target reference and the target definition. The identifiers of the modules that contain the reference and the definition.	Assuring the existence and the compatibility of the target reference and the target definition.
`string_init`	Starts a string through a weave.	The entry-function of the concerned weave and the required function parameters.	Deciding on the underlying thread package. Specifying the entry-function and associated parameters according to the rules of the thread package being used.
`string_cont*` *(This API provides support for 'string continuations'. String continuation is a specialized function. Its utility is explained in Chapter 5 (Use Cases).	Continues the execution of a certain string into *another* weave.	The identifier of the other weave.	Assuring the similarity or compatibility of the *current* and the *other* weaves. Details on string continuations are mentioned in the next section (Implementation).



(a)

```
void bootstrap ()
{
…
   /* load modules */
   mod[0] = load_module (“file1.o”);
   mod[1] = load_module (“file2.o”);
   mod[2] = load_module (“file3.o”);
…
   /* Compose weaves */
   weave[0] = compose_weave (mod[0], mod[2]);
   weave[1] = compose_weave (mod[1], mod[2]);
…
   /* 1 string through each weave */
   string_init (weave[0], foo);
   string_init (weave[1], bar);
…
}
```

(b)

```
#modules      /* object file: module ID */
file1.o: 0    /* object file1 is loaded as module 0 */
file2.o: 1    /* object file 2 is loaded as module 1 */
file3.o: 2    /* object file 3 is loaded as module 2*/

#weaves    /* weave ID: module ID, module ID, … */
0: 0, 2    /* weave 0 composes modules 0 and 2 */
1: 1, 2    /* weave 1 composes modules 1 and 2 */

#strings              /* string ID: weave ID, start function */
0: 0, foo (…)      // start string 0 in weave 0 at fn. foo ()
1: 1, bar (…)      // start string 1 in weave 1 at fn. bar ()
```

(c)

Figure 4.2: (a) A generic Weaved application. (b) Bootstrap pseudo-code for setting up the tapestry. (c) The corresponding configuration file.

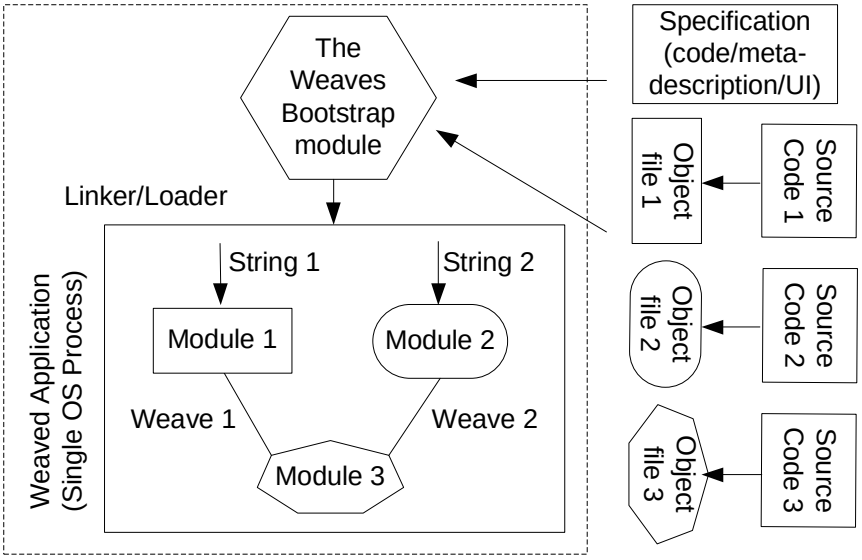


Figure 4.3: Development of weaved applications.

assigns a globally unique identifier to every module, bead, weave and string).

- *Destruction APIs* destroy and free memory allocated for modules and weaves.

- *String APIs* are based on those of the underlying thread. They control string functionality at runtime, including initialization, termination, runtime management and scheduling. The framework provides a special API for string continuation because it consists of framework-specific action. For initialization, termination, runtime management and scheduling of strings, the framework allows applications to directly call underlying thread APIs.

- *Global APIs* start, stop, pause and/or restart the entire tapestry. These APIs, though not directly related to this thesis, are useful for the dynamic configuration and com-

position of Weaved applications[3].

Currently, composing Weaved applications entails writing bootstrap modules or configuration files. Essentially, bootstrap modules or configuration files encode the *composition* of modules in a tapestry, where modules are runtime images of relocatable object files. This property makes them very similar to Makefiles, which encode the *composition* of object files in an executable or a shared library. Consequently, from the usability perspective, composing Weaved applications is comparable to writing Makefiles.

For simple tapestries, bootstrap modules and configuration files are fairly simple. However, larger tapestries, especially those involving individual reference resolutions, can complicate matters. These complications are comparable to those encountered while writing extensive makefiles. A GUI for tapestry specification, which is one of our current research directions, will make the framework more usable.

4.3 Implementation and Preliminary Evaluation

The Weaves framework is currently implemented on GNU/Linux. The implementation is heavily dependent on the Executable and Linkable File Format (ELF) [TIS95], the format of native relocatable objects used by most GNU/Linux systems. It is possible, therefore, that to follow the discussion in this section requires the reader's familiarity with the ELF

[3]Mukherjee and Varadarajan have discussed the details of the dynamic aspects of Weaved applications in [MV05b].

standard.

The basic implementations of a module, a weave, a string, co-strings, the monitor, and a tapestry are fairly obvious from their definitions. A module is the runtime image of a relocatable object file, i.e. a '.o' file. A weave is a collection of runtime images of relocatable objects dynamically composed into a subprogram. Individual reference-definition bindings consist of virtual address resolutions according to typical binding semantics [TIS95]. Currently, strings can be implemented as either kernel-level POSIX threads (pthreads) [BB06] or as user-level GNU Pth [Eng06]. Co-strings are simply multiple strings running through a single weave. The monitor is implemented as the main process thread. Finally, a tapestry is a physical OS process.

This thesis provides a description of the implementation of the core aspects of the framework which are loading modules, composing weaves, direct control over reference-definition bindings and string continuations. The implementations of other aspects such as remaining string APIs, the monitor and the general purpose APIs are either trivial or not of great significance to this thesis [4].

The framework's runtime loader and linker, called Load and Let Link (LLL) implements three of the core aspects, loading modules, composing weaves, and direct control over reference-definition bindings. Before discussing the details of LLL, it is important to mention a few things about the data structures used for modules, weaves and strings. The implementation of string continuations is discussed later in this chapter.

[4] See Mukherjee [Muk02] for further details.

The data structure for a module contains an identifier (a 32-bit unsigned integer), a list for storing global references in the module's code (`reference-list`), a list for storing external references in the module's code (`external-references`) and a list for storing dependency libraries of the module (`dependency-list`). The data structure for a weave contains an identifier (a 32-bit unsigned integer) and a list for storing constituent module identifiers. Among other items, a string's data structure maintains an identifier (a 32-bit unsigned integer), a stack for storing weave identifiers (`weave-stack`), and a thread identifier (which depends on the underlying thread package being used). The implementation also maintains three global hash tables, one for storing pointers to the module structures, one for storing pointers to the weave structures, and one for storing pointers to the string structures. Applications can lookup the desired data structures these tables using module identifiers, weave identifiers and string identifiers, respectively.

4.3.1 Load and Let Link (LLL) Weaves' Runtime Loader and Linker

The Weaves framework requires extensive runtime loading and linking capabilities to flexibly load and compose modules and randomly manipulate reference-definition bindings. Traditional native object loaders cannot support the flexibility demands of Weaves. For instance, existing loaders cannot load multiple copies of a native code object. Again, typical loaders do not provide an explicit interface to control the resolution of individual global references.

Therefore, Weaves provides its own tool—Load and Let Link (LLL)—for the dynamic loading and linking of modules.

4.3.2 The LLL Loader

The LLL loader maps given relocatable object files from the disk to corresponding modules in the memory. To load a module, Weaved applications must explicitly specify the location of the concerned relocatable object file. Applications must also provide a unique module identifier (a positive 32-bit unsigned integer).

If inputs, such as location and identifier, pass validity checks (existence of file and uniqueness of identifier, respectively), the loader converts the relocatable object (.o file) to a shared object (.so file). Most relocatable objects can be converted to respective shared objects without any concerns[5]. Conversion of a relocatable object to a shared object does not require referential completeness. Although it does result in static resolution of certain locally scoped references (such as those to static variables and read-only constants), global references are not resolved and all information on them is maintained within the resultant shared object.

If the conversion is successful, the loader verifies the content and format of the resultant shared object for compatibility. This verification requires detailed checks on ELF encoding and platform specific factors such as instruction set architectures [FSF05]. Once the object passes the compatibility tests, the loader maps the file from disk to memory subject

[5]To our knowledge, any relocatable can be converted to a corresponding shared object.

to rules/commands provided in the object encoding. Sometimes, an object is dependent on libraries for utility functions. For instance, an object that uses printf is dependent on glibc. When loading a module, LLL records every required dependency library in the module's `dependency-list`. Nevertheless, since dependency libraries are not a part of the core application, LLL relies on the underlying OS's support to load them.

After mapping the module and its dependencies (using the OS's loader), the LLL loader attempts to resolve every global reference to a definition *within* the module. Global references are looked up in the module's relocation sections. Definitions are looked up in the module's symbol table. During this phase, the loader records each encountered global reference and all associated information in the module's `reference-list`. Such information is instrumental in empowering Weaved applications with direct control over the resolution of individual global references. If a certain resolution fails, the loader adds the concerned reference and all associated information to the module's `external-references` list. An unresolved reference, therefore, does not hamper module loading. The loader follows regular resolution semantics stated in the ELF standard [TIS95]:

- For a successful resolution, the target reference is bound to the virtual memory address (VMA) of the concerned definition.
- For a failed resolution, the target reference is bound to a NULL value.

Modules are not runnable entities by themselves. Applications must compose them into weaves, the principal runnable component (subprogram) as defined by the Weaves frame-

work, before execution. Because a weave can comprise multiple modules, cross-linking among constituent modules has a higher precedence than linking to the dependency libraries. Hence, the loader does not attempt to resolve a module's references to its dependency libraries at this stage.

The loader maps every module to a unique location in the memory. Hence, functions and variables defined within a module remain independent of all other modules. Also, the loader allocates and initializes an independent data structure for each module. This disjunction among different modules enables complete encapsulation of each module. Furthermore, the loader identifies every module by a unique 32-bit number that the user assigns to it. As long as the user provides different identifiers, the loader has no issues in distinguishing multiple identical modules loaded from a single object file. Therefore, in tandem with disjunctive mapping, a number-based identification scheme enables multiple identical, but independent, instantiations of a module. Note that the LLL loader does not entail any modifications to relocatable object files.

4.3.3 The LLL Linker

The LLL linker binds together (composes) a set of modules to create a weave. To compose a weave, a Weaved application must explicitly specify identifiers of all the constituent modules and a unique identifier (a positive 32-bit unsigned integer) for the weave. Upon invocation, the LLL linker checks the validity of the inputs. If all the module identifiers are valid, and

the weave identifier is valid as well as unique, the LLL linker composes a weave according to the following steps:

1. It starts with the first module in the given set.

2. For each reference in the module's list of external references, it looks up a compatible definition in the other modules (specifically, the symbol tables of the modules) of the set. If it finds a compatible definition, it resolves the reference to the definition and continues searching through the rest of the modules. If it finds a second viable definition, it signals 'redefinition' and terminates weave composition. If it finds one, and only one, definition, it proceeds to the next external reference. If it does not find any matching definition, it tries resolving to the dependency libraries of the module using the OS's services. If it still does not find a matching definition, it signals 'unresolved reference' and proceeds to the next external reference. When all external references are handled, it proceeds to step 3.

3. If available, it takes the next module in the set and goes to step 2. Otherwise it signals 'successful weave composition' and exits.

Step 2 above illustrates several interesting properties of the LLL linker:

- The linker does not exit successfully as soon as it finds a compatible definition in order to stay within the traditional linking constructs of procedural programming. The Weaves framework defines a weave as a *subprogram*. Unambiguousness is implied in this definition. As a result, the linker must check for possible redefinitions.

- The linker might not detect multiple definitions in the absence of a corresponding external reference. COnceptually, however, the encapsulation of modules implies that each module privately scopes all its definitions within itself. An external reference expands the scope of all corresponding definitions beyond their container modules, thereby leading to potential ambiguity.

- Unresolved references do not lead to the termination of weave composition. Weaved applications can exercise direct control over resolution of individual references to fulfill referential completeness at a later time.

Like the LLL loader, the linker follows regular resolution semantics stated in the ELF standard [TIS95] in *normal* cases. However, shared modules present exceptional cases. As stated earlier, external references from a shared module (shared references, for short) might need to resolve to different definitions in the context of different weaves. The linker resolves these shared references to a special dynamic linker function—`rt_lll`—defined by LLL itself. A string accessing a shared reference automatically invokes the `rt_lll` function. When invoked, the function looks up a definition for the target reference *within the scope of the weave associated with the invoker string*. This looking up process steps similar to those followed during a normal weave composition (steps 1, 2, and 3 above), except that both multiple and unresolved definitions lead to string termination. Also, if the looking up process is successful, the target reference is *not* resolved to the obtained definition. Instead, `rt_lll` dynamically directs the invoker string to the obtained definition. Details of `rt_lll` and its dynamic mechanisms are similar to `rtld` (GNU `libc`'s runtime dynamic linker [FSF05]). However,

the implementation of `rt_lll` is quite complicated. Because the implementation of `rt_lll` is tertiary to the main theme of this thesis, for simplicity and focus, we refrain from further discussions of `rt_lll` in this document[6].

The LLL linker also implements direct control over the resolution of individual global references. To invoke this service, Weaved applications must provide an identification tuple for a target reference. The tuple contains at least two fields, the identifier of the module that contains the reference and the symbol associated with the reference. When a module contains multiple references to a symbol, further information such as the container function, sequence number and so forth is required. Once again, for conciseness, this thesis does not include excessive details on this topic. Upon invocation, the linker looks up the target reference's entry in the associated module's reference list. If no matching module or entry is found, the linker signals an error. If multiple matches are found, the linker requests more information from the application. If one and only one match is found, the linker proceeds to the next step.

The next step consists of looking up a corresponding definition to which the target reference is to be bound. For this purpose, Weaved applications must provide another identification tuple. This tuple must identify a unique definition, such as a global function or a global variable. The tuple has two fields, the identifier for a module and an associated symbol. If the module identifier is valid, the linker looks up a definition for the given symbol in the given module's symbol table. If it finds a valid definition, the linker resolves the target

[6]See Mukherjee and Varadarajan [MV05a] and Free Software Foundation [FSF05] for more details.

reference to that definition. Otherwise, it signals an error.

There are four interesting aspects of the resolution mechanism described above. Firstly, the resolution of individual references is completely under application control. Unlike typical linkers, and also unlike weave composition, the resolution is NOT based on symbol matching. Therefore, a reference to a symbol `foo` in one module can be arbitrarily resolved to a definition `bar` in another. Secondly, the mechanism is not limited to a subset of global references, such as undefined or external references. *Any* global reference, even one that already links to a valid definition, can be re-linked using this mechanism. These two aspects allow Weaved applications to arbitrarily control or modify the structure and functionality of modules, weaves and tapestries.

Thirdly, the individual reference resolution mechanism does not check for compatibility, such as type matching, between a reference and a definition. The main reason for this aspect is that native code objects do not maintain high-level information on typing and function signatures. Even traditional native object linkers cannot perform stringent checks for type and signature compatibility between references and definitions (such checks are usually performed at compilation). And fourthly, the direct resolution of an individual reference overrides the previous or current resolution of that reference, even if it links to `rt_lll`. The third and fourth aspects are causes for concern. Weaved applications, therefore, should carefully ensure that they provide compatible reference-definition pairs when invoking the individual reference resolution API.

The LLL linker composes weaves, the principal runnable subprogram as defined by the Weaves framework. Therefore, it is the most important part of the framework's implementation. Based on user-specified numeric identifiers, it can compose and distinguish identical weaves from similar sets of modules. Furthermore, using dynamic linking mechanisms, the LLL linker single-handedly enables the selective sharing of modules among multiple weaves. Lastly, with some help from the LLL loader, it enables direct control over the resolution of individual references to arbitrary definitions. Like the LLL loader, the LLL linker does not entail any modifications to object files.

4.3.4 Strings: Continuations and Evaluation

Strings are based on underlying threads. As mentioned earlier, strings can be initialized, terminated, managed and scheduled using corresponding thread APIs directly. However, string continuation comprises framework-specific action. Users must explicitly program a call to the string continuation API into a module. A string dynamically invokes the continuation API when it executes through a weave that contains the concerned module. As Table 4.1 describes, a call to continuation switches the caller string from its current weave to a different, but compatible, weave (the *target* weave).

To invoke the continuation API, a Weaved application must provide the identifier of the target weave. Upon invocation, the continuation API checks for the validity of the given weave identifier. If the check is successful, it traces the following steps:

1. It issues a query to obtain the weave within which the string is currently executing (the *current* weave).

2. It checks the compatibility of the current and the target weaves. If they are compatible[7], it proceeds to the next step. Otherwise, it signals an error and terminates the continuation.

3. It pushes the current weave's identifier onto the string's `weave-stack`.

4. It looks up the next instruction following the continuation call in the current weave, jumps to the corresponding instruction in the target weave and continues the execution of the string from there.

A string can progressively call continuations. Each subsequent call pushes the current weave identifier on the string's `weave-stack`. A string can return to the *immediately previous* weave by calling the end continuation API. The end continuation API works as follows:

1. It pops the immediately previous weave identifier from the caller string's `weave-stack`.

2. If the `weave-stack` is empty, it signals an error and terminates the call to end continuation. Otherwise, it proceeds to the next step.

3. It looks up the next instruction following the end continuation call in the current weave, jumps to the corresponding instruction in the previous weave and continues the execution of the string from there.

[7]Two weaves are compatible if and only if they comprise the same or identical modules.

String continuations are helpful in efficient information exchanges among strings running through compatible weaves (similar strings). Using continuations, a certain string can switch to the weave of another similar string and asynchronously modify its state. The other string notices the effects of such modifications without any explicit action.

A simple experiment has confirmed that strings are indeed lightweight and have low-context switching times. We created a baseline application that implemented a calibrated delay loop (busy wait of 107 seconds) and then implemented thread-based, process-based and weaved versions of the application. In each of these versions, there were n independent flows of execution over the same code, where each flow of control executed a loop that did 1/nth the baseline work. We measured the total time of execution in each case. Since each flow of execution did 1/nth of the work and there were n flows, the total time taken should have been the same as the baseline calibrated delay loop case, except for an additional context switching cost.

Figure 4.4 shows the results of the experiment on a single processor AMD Athlon workstation running Linux. The results show the run time for five versions of the experimental application: (a) baseline calibrated delay loop version, (b) pthread-based version, (c) Pth-based version, (d) process-based version, (e) Weaved version over pthreads, and (f) Weaved version over Pth. The results clearly show that the weaved implementations are significantly faster than the process-based one even in this simple case, where the copy-on-write semantics of the fork() call are very effective. Furthermore, the run time of the weaved version over pthreads is very close to the base run time of the pthread-based version. The marginal variation is

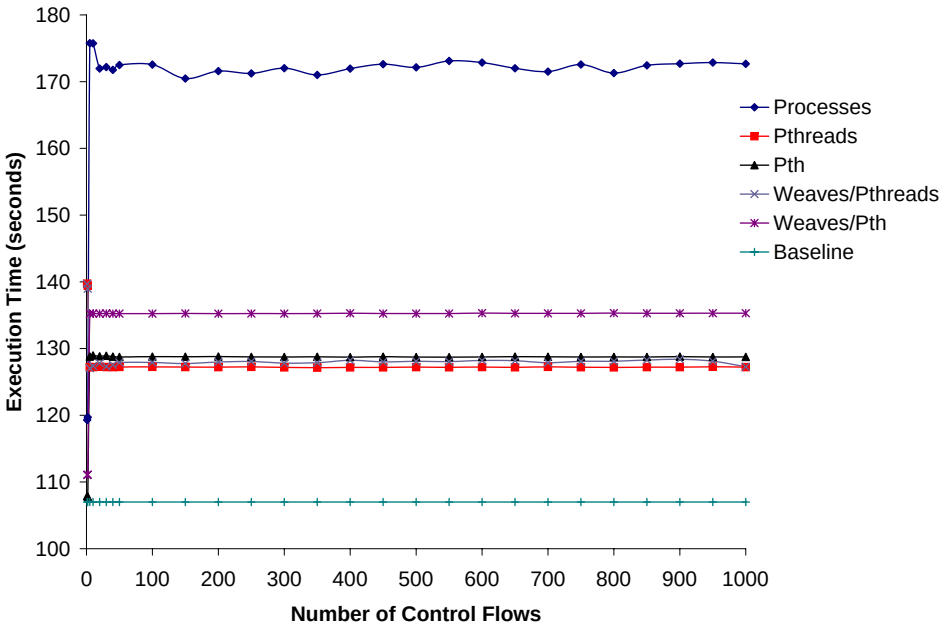


Figure 4.4: Comparison of context switch times of threads, processes, and strings. The baseline single process application implements a calibrated delay loop of 107 seconds.

due to the slightly higher weave creation cost, which is included in the run time. However, the run time of the weaved version over Pth is higher than the base Pth-based case. This increase in runtime occured because unlike pthreads, Pth is a user-level library and, hence, suffers from timer inaccuracies inherent in user-level libraries.

4.3.5 Portability

Currently, the Weaves framework is implemented over three architectures: x86, x86_64, and ia64. However, since most Linux-based systems subscribe to ELF and related semantics, the implementation is fairly architecture-neutral and can easily be extended to architectures such as Digital Alpha, Sun Solaris and Power PC. A port to Power PC is currently underway.

Porting to different operating systems such as Windows and OS-X poses bigger problems, because they do not comply with the ELF standard. However, most regular operating systems, including OS-X and Windows, allow the object file based decoupling of applications. Even though they have different formats, these object files are similar in structure and content to ELF relocatable objects. Theoretically, therefore, the Weaves framework is portable across a wide range of operating systems and architectures.

4.4 Properties of Weaved Applications

Encapsulation enforces decoupling between modules. Coupled with facilities for multiple instantiation, the encapsulation of modules empowers Weaved applications with the ability to have identical, but completely separate, copies of a relocatable object component within a single process. Furthermore, the ability to load referentially incomplete relocatable object files allows Weaved applications to exploit runtime modularization/componentization at arbitrary granularities. To avail themselves of these facilities, Weaved applications do not need to instrument any modifications on concerned relocatable objects.

Weaved applications realize each parallel program component as a weave, because a weave is an intra-process subprogram that can support a flow of execution. Weaves composed from disjunctive sets of modules are completely independent parallel subprograms within a single process. The Weaves framework allows a single module to be part of multiple weaves. Weaved applications can exploit this facility to realize arbitrary graph-like selective sharing of modules among various weaves (Figure 4.5:

- A weave can share, or not share, any number of modules with another weave.
- Transitively, any number of weaves can share, or not share, a single module.
- A weave can share, or not share, different modules with different weaves.

When references from different weaves resolve to a single definition of a function or a variable, the concerned definition is shared among the different weaves. Therefore, Weaved applica-

tions can exercise direct control over resolution of individual global references to extend selective sharing among weaves to the fine granularity of individual functions and variables:

- A weave can selectively share any number of functions and variables with another weave.
- Transitively, any number of weaves can share a single function or variable.
- A weave can selectively share different functions and variables with different weaves.

All the components of a Weaved application, including the fundamental module, are intra-process runtime units. Also, Weaved applications need not instrument any code modifications on modules either at the source level or at the level of relocatable objects. They can use the framework's facilities transparently.

Legacy procedural codes are easily available as relocatable objects. The Weaved versions of legacy parallel programs can transparently (without code modifications) load multiple encapsulated modules of legacy procedural codes into intra-process runtime environments. When they run strings through weaves composed from distinct sets of identical modules, the strings do not share any global variables. This capability directly addresses the first research challenge: the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

Strings running through weaves experience all the elements of sharing reflected in the compositions of those weaves. Therefore, Weaved versions of legacy parallel programs can trans-

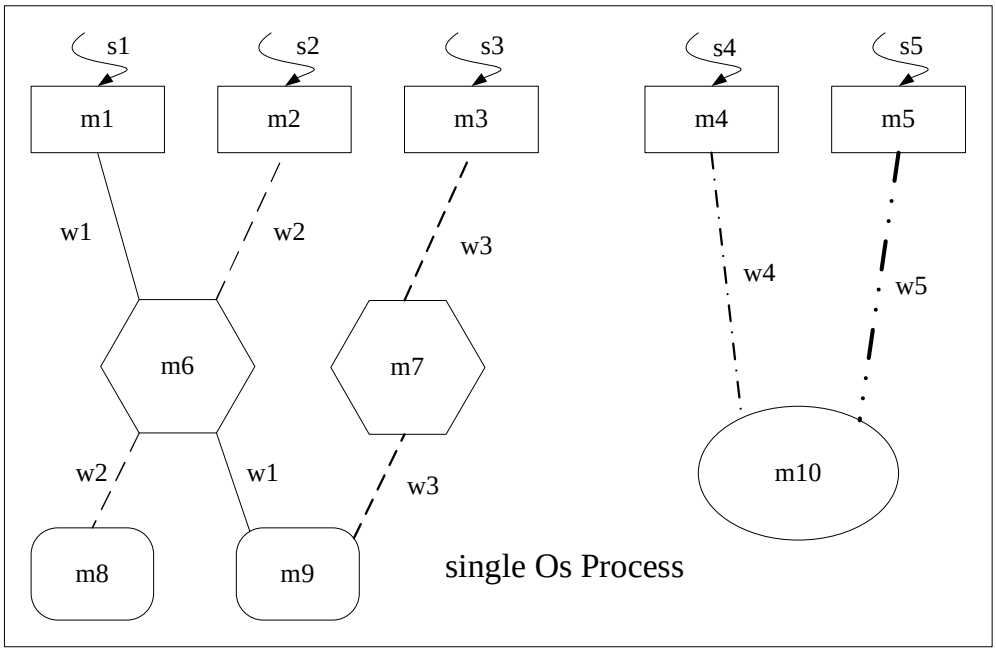


Figure 4.5: A sample tapestry essentially a complete parallel application executing as a single OS process. The figure shows the individual weaves (w), their constituent modules (m), strings (s), and their composition reflecting the structure of the application as whole. Identical shapes imply identical copies of a module. The lines connecting the modules imply external references being resolved between them.

parently realize arbitrary multi-granular selective sharing of functions and variables among strings. Because strings are essentially intra-process threads, this selective sharing among strings addresses the second research challenge: the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

Thus, Weaved versions of legacy parallel programs can run over lightweight intra-process threads without concerns of inadvertent namespace collisions and unintentional sharing. Furthermore, they can exploit low-overhead collaboration among constituent parallel threads. Together, these properties facilitate larger scales of parallelism for Weaved legacy parallel programs through better utilization of resources on a single multi-core machine or SMP workstation. Ultimately, larger scales of parallelism can facilitate more accurate modeling of large-scale parallel phenomena such as networks and multi-physics problems.

4.5 Summary

This chapter has provided a description of elements of the Weaves runtime framework for parallel programs. It has also illustrated the manner in which the framework addresses the research challenges and helps reach the overall goal of helping "unmodified" legacy parallel programs exploit the scalability provided by threads.

Weaved applications can load encapsulated runtime modules from relocatable object files. They can load multiple independent modules from a single object file without entailing any modifications to the concerned object file. By allowing direct runtime control over the resolu-

tion of individual references in a module's code, the Weaves framework empowers programs with the ability to manipulate their composition at fine granularities. Through modules, the Weaves framework supports the transparent encapsulation and multiple instantiation of legacy procedural codes in intra-process environments.

Just as the compile-time linking of object files creates an executable program, the runtime composition of a set of modules creates a weave. A weave is, therefore, an intra-process subprogram that can support a flow of execution. The framework allows a single module to be shared among multiple weaves, which can be leveraged to realize arbitrary graph-like sharing of modules among the different weaves. Direct control over the resolution of individual references can extend this selective sharing among weaves to finer granularities.

All components of Weaved applications, including the fundamental module, are intra-process runtime units. Also, Weaved applications need not instrument any code modifications on modules either at the source level or at the level of relocatable objects (native code patches). They can avail themselves of the framework's facilities transparently.

Legacy procedural codes are easily available as relocatable objects. Hence, the Weaved versions of legacy parallel programs can transparently (without code modifications) load multiple encapsulated modules of legacy procedural codes into intra-process runtime environments. When they run strings through weaves composed from distinct sets of identical modules, the strings do not share any global variables. This capability directly addresses the first research challenge: the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

Strings running through weaves experience all the elements of sharing reflected in the compositions of those weaves. Therefore, Weaved versions of legacy parallel programs can transparently realize arbitrary multi-granular selective sharing of functions and variables among strings. Since strings are essentially intra-process threads, this selective sharing among strings addresses the second research challenge: the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

Weaved versions of legacy parallel programs can run over lightweight intra-process threads without concerns of inadvertent namespace collisions and unintentional sharing. Furthermore, they can exploit low-overhead collaboration among constituent parallel threads. Together, these properties facilitate larger scales of parallelism for Weaved legacy parallel programs through better utilization of resources on a single multi-core machine or SMP workstation. Ultimately, larger scales of parallelism can facilitate more accurate modeling of large-scale parallel phenomena such as networks and multi-physics problems.

At the basic level, the Weaves framework offers its services as a library that supports simple APIs. Users can explicitly program the composition of Weaved applications using these APIs. The framework also provides a meta-language for specifying such compositions in a configuration file and a script that automatically creates and runs corresponding Weaved applications. Essentially, configuration files are very similar to Makefiles. Consequently, from the usability perspective, composing Weaved applications is comparable to writing Makefiles.

The Weaves framework is currently implemented on GNU/Linux over three architectures:

x86, x86_64, and ia64. Weaves' runtime loader and linker called Load and Let Link (LLL) implements the core aspects of the framework, which include loading modules, composing weaves and direct control over the resolution of individual references.

The implementation is heavily dependent on the Executable and Linkable File Format (ELF) [TIS95], the format of native relocatable objects used by most GNU/Linux systems. However, the implementation is fairly architecture-neutral. A port to the Power PC architecture is currently underway. Porting to different operating systems such as Windows and OS-X poses bigger problems. However, because most regular operating systems allow object file based decoupling of applications, theoretically, the Weaves framework is portable across a wide range of operating systems and architectures. Lastly, a preliminary experiment has confirmed that strings are indeed lightweight and more scalable than processes.

Chapter 5

Case Studies

This chapter provides a demonstration of the utility of the Weaves framework in software areas that comprise legacy parallel programs by presenting various case studies from the areas of network emulation and parallel scientific computing. Weaved instances of network emulation and parallel scientific programs are used to further illustrate the design, implementation and development of Weaved applications. For instance, one of the case studies explains the use of string continuations. Because these studies are for the purposes of illustration, we have chosen them from two radically different domains of parallel systems to emphasize the broad impact of the Weaves framework. These case studies were designed to exemplify the manner in which Weaved instances of network emulation and parallel scientific computing institute advances toward area-specific versions of our overall goal. Finally, this chapter also presents the results of experiments with large real-world applications to substantiate the effectiveness of this research.

5.1 Using Weaves for Network Emulation

This section gives a description of the Weaved instances of network emulation (see Chapter 2). First, a simple hypothetical case study is used to describe a Weaves-based approach to network emulation. This case study exemplifies the design, implementation and development of Weaved network emulations. Next, an experiment designed to corroborate that Weaved instances of network emulation can indeed run multiple independent threads of unmodified legacy network applications is detailed. The experiment also shows that the Weaves framework can facilitate user-level selective sharing of multiple real-world IP stacks among the variuos application threads. Then there follows an explanation of the results of experiments with large real-world network emulations performed using the Open Network Emulator (ONE) [Var02].

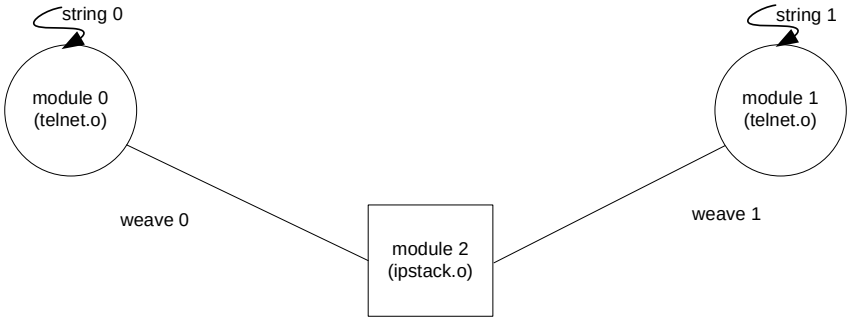
These results herald the ONE's ability to correctly model larger scales of parallel real-world network applications. In these experiments, the ONE used the Weaves framework to run unmodified real-world network applications and protocol stacks (legacy procedural codes) over intra-process threads. Furthermore, the ONE used the framework for low-overhead user-level selective sharing of multiple TCP/IP stacks among various application threads. Such uses of the Weaves framework were instrumental in helping the ONE transparently exploit overall resources for larger scales of parallelism. Ultimately, larger scales of parallelism enable more accurate testing of network protocols. The results, therefore, substantiate the efficacy of the current research.

5.1.1 A Simple Instance

Figure 5.1(a) depicts the simple hypothetical network scenario seen in Figure 2.2 To emulate this scenario using the Weaves framework, a user must compile the telnet code into one relocatable object (telnet.o) and the IP stack code into another relocatable object (ipstack.o), load two distinct modules of telnet.o (module 0 and module 1) and one module of ipstack.o (module 2), and compose two weaves as shown in the figure. The first weave (weave 0) comprises modules 0 and 2; the second weave (weave 1) comprises modules 1 and 2. Then, the user must start two strings, one each at the main entry functions of the telnet modules (module 0 and module 1).

These two strings run through separate telnet modules and do not interfere with each other. This Weaved realization helps separate the global variables of the two encapsulated telnet modules. It does not require any modifications to the telnet objects, which can be compiled from legacy procedural sources of telnet. The Weaved realization effectively addresses the network emulation version of the first research challenge: the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

Since both strings run through a single IP stack, they collaborate within the IP stack even though they comprise separate telnets. The two telnets interfere with each other within the stack, thereby emulating correct real-world operation. Advanced Weaved emulations, such as the one depicted in Figure 5.2, *selectively* share different independent IP stacks among different sets of application threads.



(a)

```
void bootstrap ()
{
…
  /* load modules */
  for ID = 0 TO 1
    mod[ID] = load_module (“telnet.o”);
  mod[2] = load_module (“ipstack.o”);
…
  /* Compose weaves */
  weave[0] = compose_weave (mod[0], mod[2]);
  weave[1] = compose_weave (mod[1], mod[2]);
…
  /* 1 string through each weave */
  for ID = 0 TO 1
    string_init (weave[ID], telnet_main);
…
}
```

(b)

```
#modules      /* object file: module ID */
telnet.o: 0, 1      /* object telnet is loaded as modules 0, 1 */
ipstack.o: 2        /* object file 3 is loaded as module 2*/

#weaves     /* weave ID: module ID, module ID, … */
0: 0, 2         /* weave 0 composes modules 0 and 2 */
1: 1, 2         /* weave 1 composes modules 1 and 2 */

#strings                    /* string ID: weave ID, start function */
0: 0, telnet_main (…)   // start string 0 in weave 0 at telnet’s entry fn.
1: 1, telnet_main (…)   // start string 1 in weave 1 at telnet’s entry fn.
```

(c)

Figure 5.1: Modeling the simple network scenario of Figure 2.2 using the Weaves framework. (a) Weaved setup of the tapestry. (b) Bootstrap pseudo-code (c) Configuration file.

These simple hypothetical Weaved realizations do not entail any modifications to either the telnet object or the IP stack object, which can be compiled from corresponding legacy procedural sources. In effect, these Weaved realizations show that the framework addresses the network emulation version of the second research challenge: the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

5.1.2 Experimental Corroboration

Figure 5.2 depicts the experiment that was designed to check whether Weaved instances of network emulation can indeed run multiple independent threads of unmodified legacy network applications. The experiment also checks whther the Weaves framework can facilitate user-level selective sharing of multiple real-world IP stacks among the variuos application threads.

The figure shows a simple network with two hosts each running a server and a client. We compiled real-world codes for the client, server, and the IP stack [Kne04] into relocatable objects (client.o, server.o, and ipstack.o respectively). The codes were written in C and contained global variables. We loaded two modules from each object (c1, c2 from client.o; s1, s2 from server.o; and ip1, ip2 from ipstack.o) and composed four weaves as follows:

- weave wv1: c1 and ip1.
- weave wv2: s1 and ip1.

- weave wv3: c2 and ip2.
- weave wv4: s2 and ip2.

Finally, we initialized four strings, one each at the start functions of c1, c2, s1, and s2. The entire emulation, comprising two independent virtual hosts, operated as a single OS process on a single processor AMD AthlonTM workstation (32-bit Intel architecture) running the Linux OS.

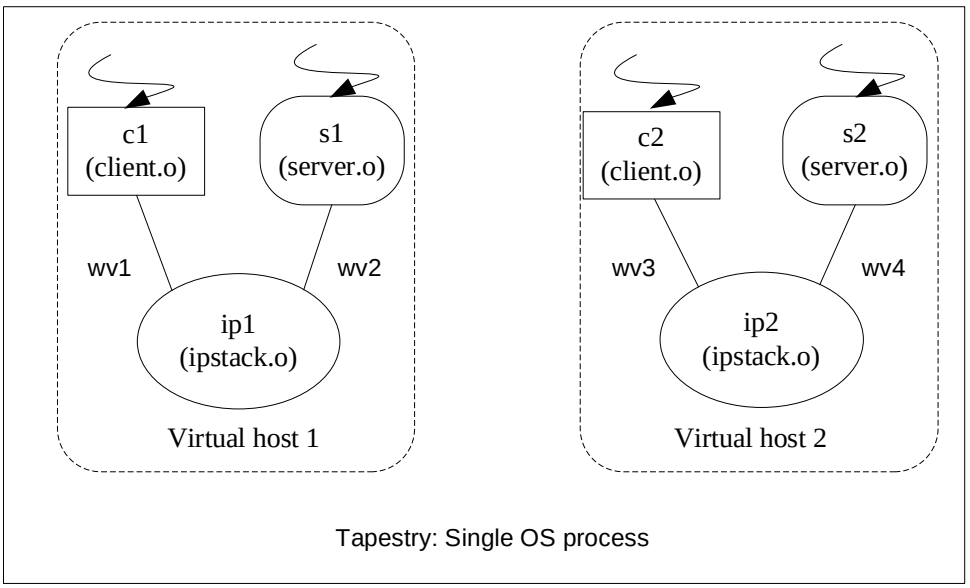


Figure 5.2: Weaved set up of the experimental network scenario. Both clients used identical real-world codes as did the servers. The IP stacks used identical real-world codes. The two hosts were completely independent, but ran within a single OS process.

To test whether the setup worked correctly, we transferred two different files (f1 and f2) between each client-server pair. Client c1 transferred file f1 to server s1 through IP stack ip1, and client c2 transferred file f2 to server s2 through IP stack ip2. Server s1 wrote the

received data to output file o1, and server s2 wrote the received data to output file o2. This setup did not entail any modifications to the client, server and IP stack codes.

After the transfer, we used Linux's diff command to compare files f1 and o1. The files were identical, as were files f2 and o2. Furthermore, the differences between f1 and f2 were identical to those between o1 and o2. These results show that the transfers took place properly. Unless the clients, servers and IP stacks were completely separate, the file transfers would have interfered with each other resulting in corrupted outputs. Furthermore, unless the two IP stacks were selectively shared between the different client-server pairs, the file transfers between the pairs would have run into errors.

The results, therefore, reasonably corroborate the correctness of the Weaved setup, because they show that the Weaves framework transparently facilitates (1) execution of legacy parallel programs (clients and servers) over lightweight intra-process threads without inadvertent sharing of global variables, and (2) user-level selective sharing of global variables (within the IP stacks) among threads of such legacy parallel programs. In effect, these results are indicative of the progress towards the overall goal of this research: helping "unmodified" legacy parallel programs exploit the scalability provided by threads.

5.1.3 Contextual Advances

The lateral Open Network Emulator (ONE) project [Var02] has reached prototype capability. Over the last few months, the distributed version of the ONE (dONE [BVB06]) was used to

experiment with large real-world network emulations. In these experiments, the ONE used the Weaves framework to run unmodified real-world network applications and protocol stacks (legacy procedural codes) over threads. Furthermore, the dONE used the framework for low-overhead user-level selective sharing of multiple TCP/IP stacks among various application threads.

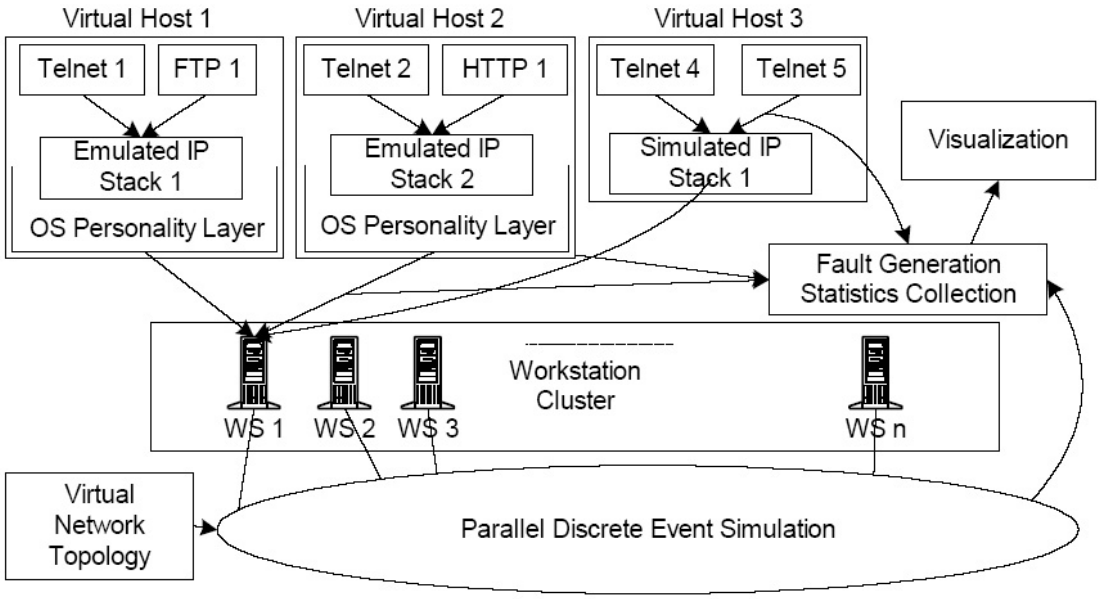


Figure 5.3: The Open Network Emulator (ONE) models thousands of simultaneously (parallely) running real-world network nodes and applications on a single machine.

These experiments were performed on the Linux 2.6.11 kernel over a cluster of eight dual-CPU Opteron machines for a total of sixteen processors with 2 GB of physical memory per machine. The machines were connected via a 10Gbps Infinicon Infiniband interface. Each machine was configured based on Redhat Fedora Core 2 in a diskless configuration.

Figure 5.3, seen earlier in Figure 2.1, depicts the overall setup of the experiments. The

dONE used the Weaves framework to run multiple parallel sender and receiver applications over multiple IP stacks to emulate multiple virtual nodes within a single process on a single physical machine. Collaboration across physical nodes was facilitated using the Message Passing Interface (MPI/LAM [PTL06]).

The codes used in the experiments consisted of senders and recievers written in the C language and a real-world TCP/IP stack extracted from the Linux 2.4 kernel [Kne04]. The TCP/IP stack contained a large number of global variables.

The results from the experiments show that all data transfers across various senders and receivers were without errors. Furthermore, the results confirm that the dONE can correctly model the real-world behavior of protocols such as TCP/IP. For a certain scenario of 50 sender-reciever pairs, the results are consistent and the variance negligible. Most pertinently, the results herald that the dONE can emulate a thousand virtual nodes over a single physical machine. On increasing the number of machines, the emulation speeds up super-linearly i.e. on increasing the number of machines by a factor of N, the time taken to run a certain emulation drops by a factor greater than N (Figure 5.4: This figure is reproduced from [BVB06].). This super-linear speedup attests the scalability of the dONE.

The exact details of the experiment and further analyses of results are beyond the scope of the current thesis. Interested readers can refer to [BVB06] for additional exposition.

These results show that the dONE can emulate large-scale networks consisting of thousands of network applications and virtual nodes on a limited number of machines. The Weaves

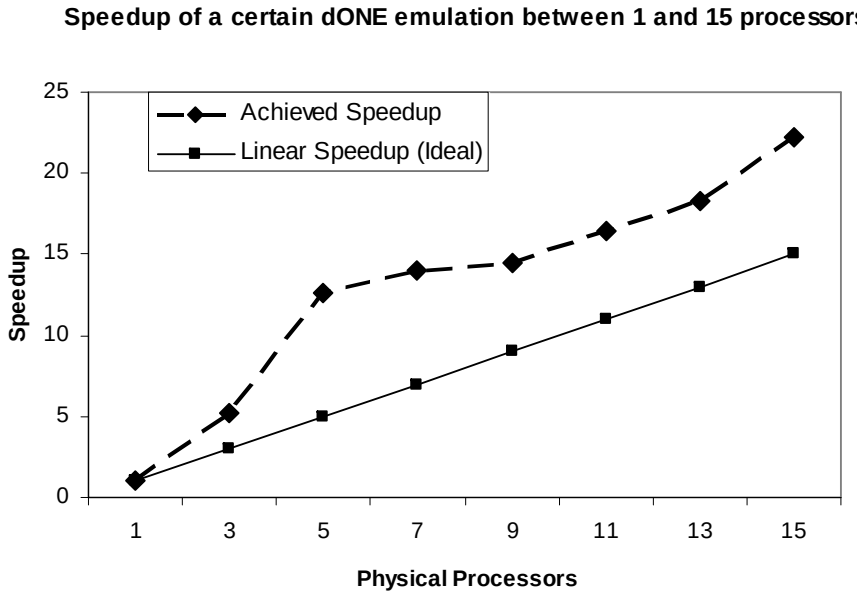


Figure 5.4: The dONE exhibits super-linear speedup when emulating real-world network nodes and applications. This figure is reproduced from [BVB06].

framework is instrumental in helping network emulations transparently exploit the scalability of threads for larger scales of parallelism, the overall goal of this research. Ultimately, larger scales of parallelism enable more accurate testing of network protocols. The results, therefore, substantiate the usefulness of the Weaves framework.

5.2 Using Weaves for Scientific Computing

This section provides a description of Weaved instances of parallel scientific applications that comprise collaborating partial differential equation (PDE) solvers (see Chapter 2). First, some simple hypothetical case studies are used to describe Weaves-based approaches to

parallel scientific computing. These case studies exemplify the design, implementation and development of Weaved scientific applications. One of the case studies explains the use of string continuations. Next, experiments designed to corroborate that Weaved instances of scientific applications can indeed run multiple independent threads of unmodified legacy PDE solvers is detailed. One of the experiments also shows that the Weaves framework can facilitate low-overhead collaboration between legacy PDE solver threads through user-level selective sharing of global variables. Then there follows an explanation of the results of experiments with large parallel scientific applications.

These experiments used the Weaves framework to run unmodified real-world scientific solvers (legacy procedural codes containing global variables) over intra-process threads. The results of one experiment show that, under identical resource limitations, a Weaved application can support nearly twice the number of unmodified parallel solvers as the corresponding process-based application can. The results of another experiment show that a Weaved application can transparently facilitate low-overhead collaboration among parallel solver threads through user-level sharing of global variables, which allows for greater scales of achievable parallelism than traditional MPI-based communication allows [KBA92, ANLUC06].

These results together indicate that the Weaves framework is instrumental in helping parallel scientific applications transparently exploit overall resources for larger scales of parallelism. Ultimately, larger scales of parallelism enable more accurate modeling of multi-physics phenomena. The results, therefore, substantiate the efficacy of the current research.

5.2.1 A Simple Instance

The Weaves framework opens up three possibilities for realizing the simple hypothetical instance of collaborating PDE solvers depicted in Figure 5.5 (seen also in Figure 5.5). The first two approaches assume that solvers and mediators are designed with an awareness of Weaves. The second approach uses string continuations for solver-mediator collaboration. The third and most radical approach reuses unmodified solver and mediator codes from traditional agent-based implementations.

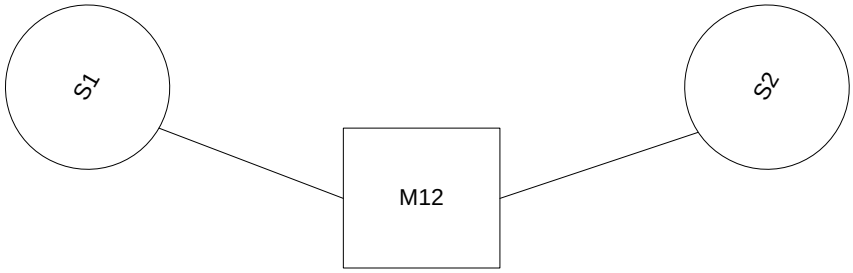


Figure 5.5: The simple PDE solver scenario of Figure 2.8.

Designing for Weaves

The design of Weaves-aware collaborating PDE solvers can follow different methods depending on the extent of affordable parallelism. If the number of available processors is large, both solvers and mediators can afford independent parallel strings of execution. Otherwise, only solvers can own a parallel string, while mediator modules are shared among adjacent solver strings.

Parallel Solvers and Mediators: In this method users must compile the solver and mediator instances into separate relocatable objects, solver.o and mediator.o, respectively. The solver object must define and store the global variables for boundary conditions and solution structures. The mediator object must make external references to boundary conditions and solution structures.

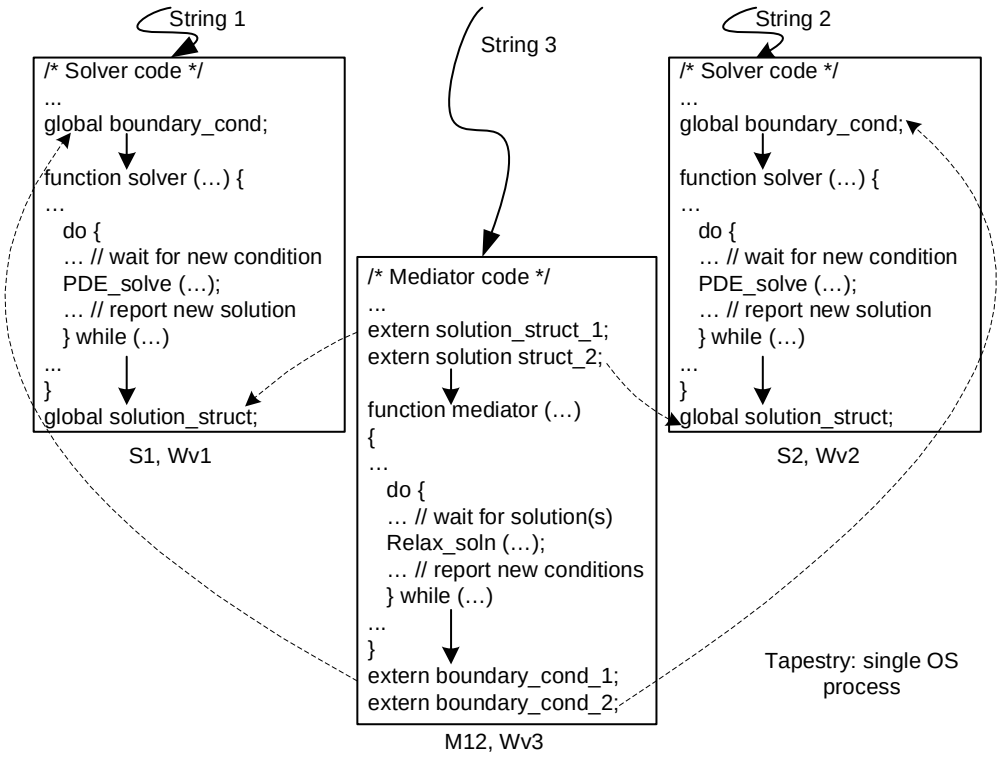


Figure 5.6: A possible Weaved realization of the Figure 5.5 scenario. S1, S2, and M12 are composed into separate weaves Wv1, Wv2 and Wv3. External references from M12 are explicitly bound to definitions within S1 and S2.

Figure 5.6 depicts the Weaved setup. To compose and run a complete tapestry of collaborating PDE solvers, users must load 2 modules of solver.o (S1 and S2) and one module of

mediator.o (M12). They must then compose each solver module into a solver weave (Wv1, Wv2), the mediator module into a mediator weave (Wv3) and exercise direct control over the resolution of M12's references to bind them to the different definitions of the solution structures and boundary conditions within the two solvers. Lastly, users must initiate parallel strings at the entry functions of the solvers and mediator.

The semantics of the overall application remain the same as in the traditional agent-based implementation. Initially, the solvers read in the initial PDE structure and start the first computations. At the end of a run, they write their solutions to their solution structures and wait for fresh boundary conditions. The mediator performs relaxations on solver solutions and writes new boundary conditions to the solvers. As soon as the new boundary conditions become available, the solvers start off again and the loop is repeated till a satisfactory state is reached.

Parallel Solvers and Shared Mediators: This method is based on the critical observation that neither the physics problem nor the computational basis behind an instance of collaborating PDE solvers requires that mediators be parallel independent flows of execution. The fact that they are is a direct result of the distributed agent model traditionally used to solve such problem instances.

Here also, users must compile the solver and mediator instances into separate relocatable objects, solver.o and mediator.o, respectively. The solver object must define the global boundary condition variables and solution structures. Additionally, each solver component

must explicitly call external mediator functions. The mediator object must make external references to the boundary variables and solution structures.

This time, users must load 2 modules of solver.o (S1 and S2) and one module of mediator.o (M12). Then they must compose S1 and M12 into weave Wv1, S2 and M12 into weave Wv2, and exercise direct control over the resolution of M12's references to bind them to the different definitions of the solution structures and boundary conditions within the two solvers. Finally, they must initiate one string at the entry function of S1 and another string at the entry function of S2. As a result, while the solvers have their own parallel flows of execution, the mediator does not. The mediator module is a part of the solver weaves.

Figures 5.7(a), 5.7(b), 5.7(c), and 5.7(d) show the corresponding tapestry layout and pseudo-codes for the solvers and the mediator. Because M12's external references resolve to definitions in both S1 and S2, Wv1 and Wv2 together ensure referential completeness of the tapestry, even though they are individually incomplete. Moreover, the mediator function in M12 is invoked twice (once each from S1 and S2) during a single relaxation act. An effective relaxation can be performed by either (c) the last invocation when both the solvers' solutions are available, or (d) the first invocation, which uses the solvers' solutions as and when they are needed and become available. In each case, Weaves' linking semantics transparently make sure that the invocation has access to the contexts of both Wv1 and Wv2, or S1 and S2.

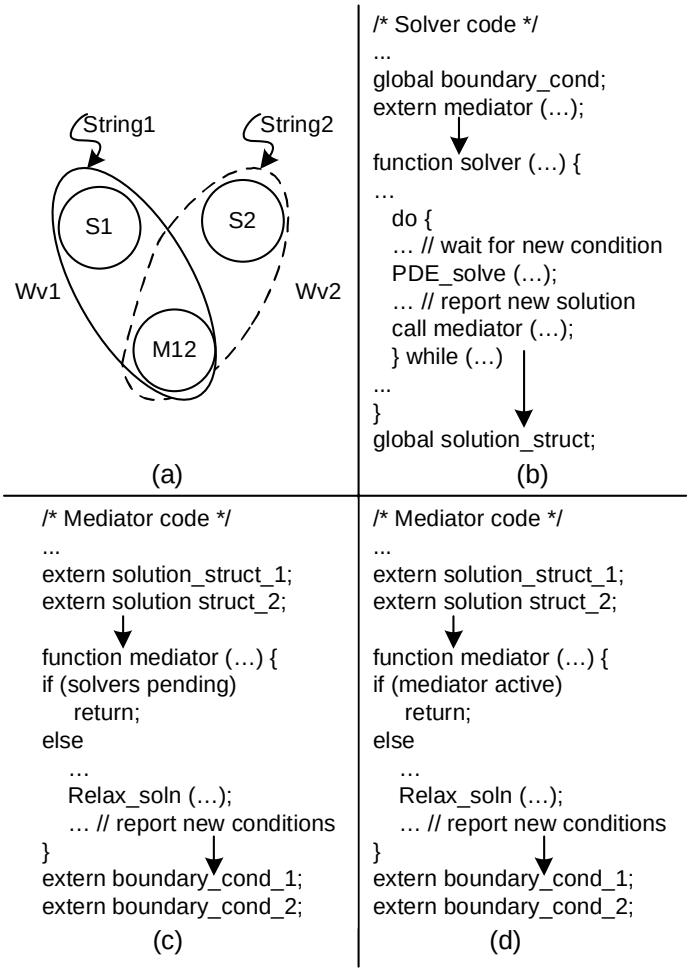


Figure 5.7: An alternate Weaved realization of the Figure 5.5 scenario. (a) Weaves Wv1 and Wv2 compose M12 with S1 and S2 respectively. (b) Typical code for S1 and S2. (c) Code for M12 if it needs all solutions. (d) Code for M12 if it uses solutions as and when needed and available.

Special Use of String Continuations

This section, presents a singular Weaves-based method to realize a special case of the Figure 5.5 scenario using the string continuation API. Assume that:

1. S1 and S2 are identical solvers.

2. M12 uses one solution at a time as needed and available.

3. M12 returns an identical boundary condition to S1 and S2.

To set up the tapestry users must follow the same steps as outlined in the description of the 'Parallel Solvers and Shared Mediators' method. However, in this case, M12 needs to define only two external references, one to a solution structure and one to a boundary condition. Figure 5.8(a) depicts the code for M12. If S1 calls the mediator function first, the Weaves framework transparently binds `solution_struct` in M12 to the definition in S1, based on the context of Wv1. When the mediator function needs solution information from S2, it invokes an explicit continuation into Wv2. The continuation causes a change in namespace such that `solution_struct` maps to the definition in S2. When the mediator function has computed a fresh boundary condition, it writes the new value to `boundary_cond` in S2, because the current context is Wv2. To write to S1, M12 issues an explicit call to end the continuation, which restores the context to that of Wv1. The overall application follows the same iterative semantics as in the other cases.

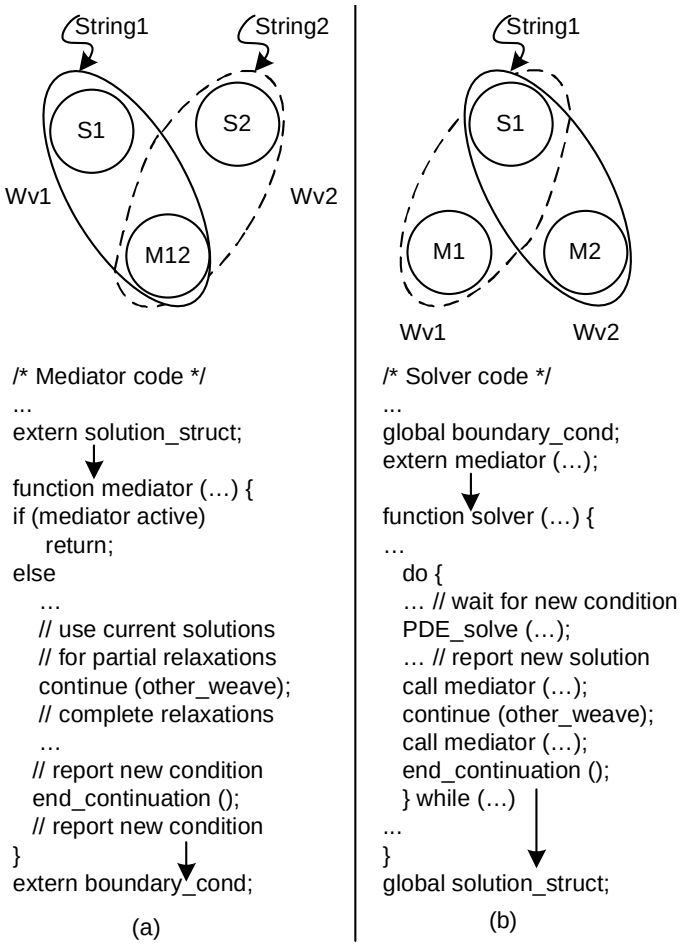


Figure 5.8: Continuations help map a single module to different weaves. (a) The tapestry setup. (b) An imaginary partial tapestry.

Figure 5.8(b) depicts a partial tapestry where a solver, S1, is a part of two weaves comprising identical mediators, M1 and M2. The solver (S1) consists of a single string, declares a single external reference to the mediator function and uses string continuations to invoke the identical, but independent, definitions of the mediator function in M1 and M2.

Reusing Agent-based Codes

The Weaves framework supports unmodified reuse of the agent-based solver and mediator codes. An agent-based approach assumes some form of an underlying message passing library, such as MPI [ANLUC06], that implements dependable communication primitives. The solver and mediator agents call the functions in the library to safely exchange boundary conditions and solution structures.

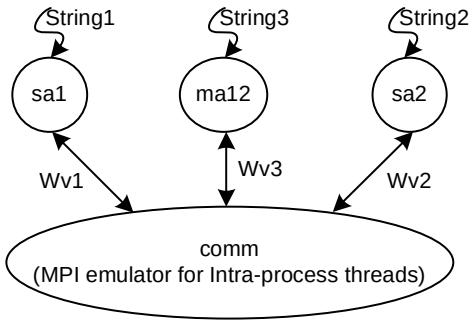


Figure 5.9: Weaving unmodified agent-based codes. Solver and mediator modules are composed into different weaves, but share a single thread-based MPI emulator.

Assuming MPI for solver-mediator communication, Figure 5.9 illustrates a Weaves-based method to set up a hypothetical tapestry that reuses the unmodified agent-based codes.

According to this method, users must compile the solver and mediator agent codes into the relocatable objects, s_agent.o and m_agent.o, respectively. Additionally, they must program a communication component that emulates dependable message transfers between intra-process threads using user-level global variables and must compile it into an object communicator.o. The functional interfaces of the communication component must be identical to those of the original communication library. Users must then load modules sa1 and sa2 from s_agent.o, module ma12 from m_agent.o, module comm from communicator.o and compose weaves as follows:

- weave wv1: sa1 and comm.
- weave wv2: sa2 and comm.
- weave wv3: ma12 and comm.

Finally, they must initialize three strings, one each at the start functions of sa1, sa2, and ma12. The solver and mediator agents run as independent virtual machine abstractions within a single process unaware of Weaves.

Summary

All the different Weaved realizations of the hypothetical collaborating PDE solver application run identical, but independent, threads of the solver programs. They transparently reuse legacy procedural codes for solvers and mediators even though they contain global variables.

The Weaves framework helps separate the global variables of encapsulated solver threads without any code modifications. In effect, the Weaved realizations address the scientific computing version of the first research challenge, the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

Furthermore, different Weaved realizations embody different granularities of selective sharing of global variables among parallel solver and mediator threads. While the agent-based realization embodies selective sharing of an entire communication module, the others demonstrate selective sharing at the fine granularity of individual solution structures and boundary condition variables. All the Weaved realizations reuse unmodified legacy scientific routines (`PDE_solver` and `Relax_soln`) from standard PSE toolboxes. Therefore, the Weaved realizations address the scientific computing version of the second research challenge, the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

However, the different Weaves-based methods exhibit a trade-off between extent of overall transparency and the overhead of inter-thread collaboration. The most transparent method reuses the unmodified agent-based traditional procedural codes, but incurs some extra overhead due to the copying of data during exchanges through the communication module. On the other hand, the other methods directly share solutions structures and boundary variables for lower overhead, but require globally stored solution structures and boundary condition variables to allow direct external access.

As an aside, in all Weaved methods, the runtime image of a tapestry closely matches the

graphical representation of the hypothetical collaborating PDE solvers application. Furthermore, the bootstrap code in any method requires minimal information about the internal codes of solvers and mediators, which shows that the Weaves framework is fairly domain neutral.

5.2.2 Experimental Corroboration

Figure 5.10 shows a special Weaved instance of parallel PDE solvers designed to check whether Weaved scientific applications can transparently run multiple identical, but independent, threads of solvers that contain global variables.

The experimental setup followed the basic Weaves-based semantics depicted in Figure 5.9. We assumed that all solvers were identical, but independent. We obtained solver codes from The Numerical Algorithm Group (NAG) [NAG06b] website under a trial license. Specifically, we used the Fortran Library Mark 21 for Intel Linux, compiled using GNU's Fortran Compiler `g77`. We chose the example solver program for Finite Difference Equations (Elliptic PDE, Multigrid Technique) provided along with the library to represent generic solvers. To model realistic long running problems and to aide timing measurements, we wrapped the main entry function in a loop of 10000 iterations. To ensure non thread safety, we added FORTRAN's DATA [NAG06a] directive for dummy initialization of a few elements of the solver's input PDE structure[1].

[1]From the programming perspective, FORTRAN variables are, largely, locally scoped. However, when associated with certain directives, some of these variables are allocated globally as a part of a program's

Our target solver program had three relevant characteristics:

1. It comprised validated, commercial, procedural, process-based, non-threadsafe and non-reentrant FORTRAN code.

2. It contained global variables.

3. Sources for its core code were NOT available. Only compiled object files were available as a library (.a archive).

To test that the solver program was non-threadsafe, we ran multiple threads through a single, otherwise correct, program. The outputs of the resultant threaded program were wrong, thereby indicating that the threads interfered with each other and disrupted the common global variables. This anomaly was transparently evaded by using the Weaves framework.

We compiled and linked the example solver program (d03edfe.f) with the required code objects (d03edf*.o) from the archive to generate a relocatable object (d03edfe.o). We loaded multiple modules of d03edfe.o, composed each module into an independent weave and started strings through each weave at the main entry function of the solver. All the strings ran to completion, producing results that correctly matched with normal process-based execution.

The correctness of the results reasonably corroborates the correctness of the Weaved setup and shows that the Weaves framework transparently facilitates execution of legacy parallel

binary image. From a threading perspective, therefore, these variables are invariably shared across all threads, thereby potentially hampering thread safety.

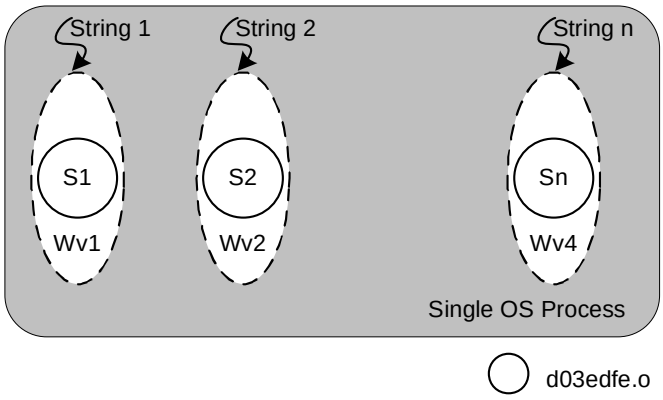


Figure 5.10: Weaved setup of the experimental PDE solver (d03edfe) scenario.

programs (non-threadsafe NAG solvers containing global variables) over lightweight intra-process threads, the overall goal of this research.

5.2.3 Contextual Advances

To confirm the utility of the Weaves framework, we ran two experiments with large parallel scientific applications. Both experiments used the Weaves framework to run unmodified real-world scientific solvers, legacy procedural codes containing global variables, over lightweight intra-process threads.

Scalability of Weaved Scientific Applications

To judge the scalability of Weaved realizations of legacy parallel scientific applications, we instantiated an NxN grid of solver (d03edfe) weaves and strings in the manner depicted in

Figure 5.10. Each element of the grid consisted of an identical solver module loaded from d03edfe.o and composed into an independent weave with a string started at the solver's entry function.

We then realized a process-based version of the same scenario. Here, a master process spawned NxN processes using Linuxs `fork()` command. Each child process called `execvp` to switch to an independent copy of the solver program. We used the same solver codes in both process-based and Weaves-based realizations.

We varied N from 1 to 31 and noted the total time of execution, from the start of the first solver to the end of the last, in every case. The outputs for both the Weaved and process-based realizations matched correctly upto N = 23. The Weaved realization correctly ran up to nearly 1000 solvers with a perfectly linear increase in the total execution time. In contrast, the process-based one scaled up to approximately 500 solvers only.

All runs were conducted on a 2GZ 32 bit Athlon processor with 1GB RAM running GNU/Linux. All measurements were averaged over 10 runs. Figure 5.11 sketches performance data obtained from the experiment. The data show that a Weaved realization can support nearly twice the number of unmodified non-threadsafe parallel solvers than a traditional process-based realization. These results indicate that the Weaves framework can help scientific applications exploit the scalability of threads without requiring modifications to the traditionally process-based programs that contain global variables.

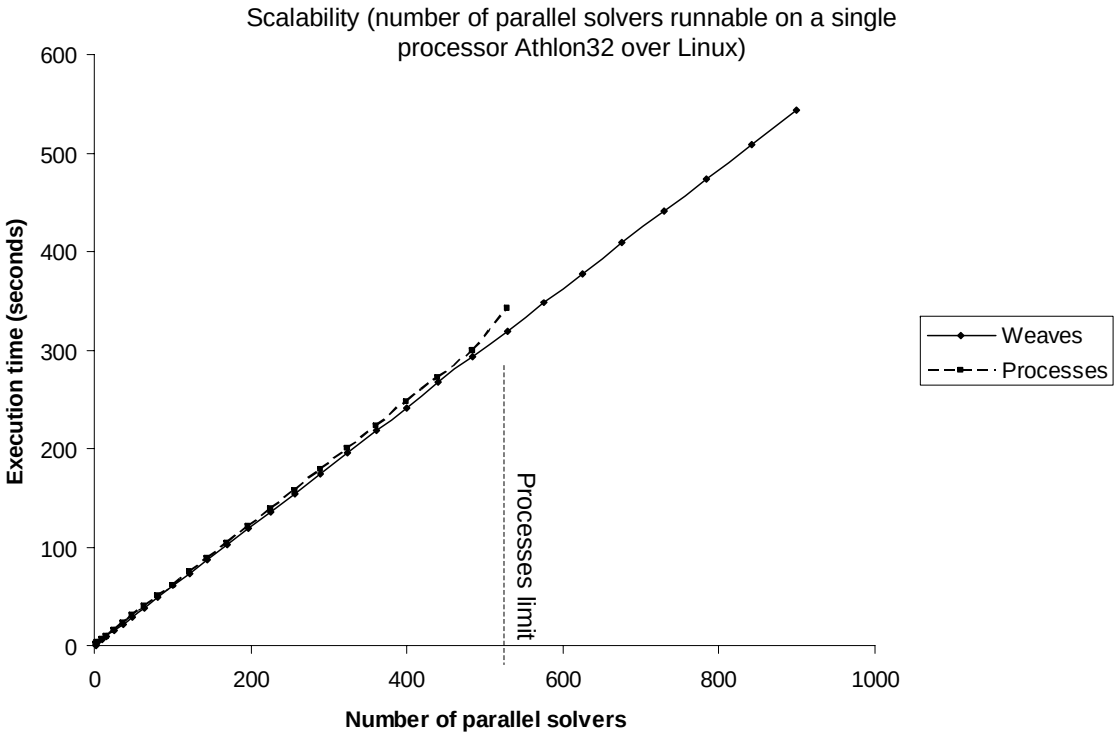


Figure 5.11: Scalability of Weaved scientific applications: Experimental results indicate that the Weaves framework can help applications exploit the scalability of threads without requiring modifications to traditional procedural process-based programs. The framework effects zero-overhead encapsulation of solvers.

Scalable Collaboration among Weaved Parallel Solvers

The previous experiment indicated the scalability of a Weaved scientific application in terms of the number of unmodified solver programs that can be accommodated on a single machine. Nevertheless, realistic scientific problems typically require collaboration amongst solvers through inter-solver communication.

To judge the Weaves framework's ability to transparently facilitate low-overhead collaboration among parallel solver threads through user-level sharing of global variables, we realized a Weaved version of Sweep3D [KBA92], an application that uses solvers for 3-Dimensional Discrete Ordinate Neutron Transport, on an 8-processor x86_64 SMP and compared the results with traditional MPI-based realizations. Figure 5.12 depicts the Weaves-based setup. We developed a simple MPI emulator for user-level inter-thread communication through shared global variables and compiled the emulator and the Sweep3D solver code into relocatable objects mpi.o and sweep3d.o, respectively. To setup a Weaved instance of Sweep3D comprising `nxm` solvers (or a `nxm` split), we loaded `nxm` modules of sweep3d.o and one module of mpi.o. We then composed `nxm` weaves, where each weave consisted of a unique sweep3d module and the mpi module, and initialized `nxm` strings, one string at the entry function of each Sweep3D module.

We reused unmodified legacy FORTRAN codes of Sweep3D for the Weaved realization. Sweep3D codes contain FORTRAN's COMMON directive for global allocation of certain variables [NAG06a], that is, they contain global variables and are non-threadsafe. Linux's

`objdump` tool was used to verify that Sweep3D's compiled object contained a number of globally stored variables.

All the Sweep3D strings ran to completion, producing results that correctly matched with normal process-based execution. This observation reasonably corroborates the correctness of the Weaved setup. It also shows that the Weaves framework facilitates user-level selective sharing of global variables within the MPI module among constituent threads of legacy Sweep3D solvers.

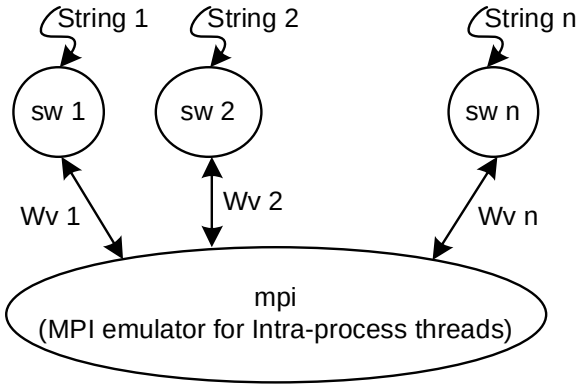


Figure 5.12: Weaved setup of the experiment using Sweep3D solvers.

We used a 150-cube input file with a 2x3 split (6 strings/processes) as a start point and increased the split to 2x4, 4x6, 6x9, and so on up to 10x15 (150 strings/processes). Figure 5.13 shows that the performance of the Weaved realization matched that of LAM-based [PTL06] and MPICH-based [ANLUC06] realizations as long as the number of strings/processes was less than the number of processors. When the number of strings/processes was increased beyond the number of processors (8), the Weaved realization performed much better. There-

fore, the Weaved realization clearly demonstrates scalable low-overhead collaboration. Both the MPI-based realizations, compiled and run with shared memory flags for lowest overhead, crashed beyond 24 processes (4x6 split).

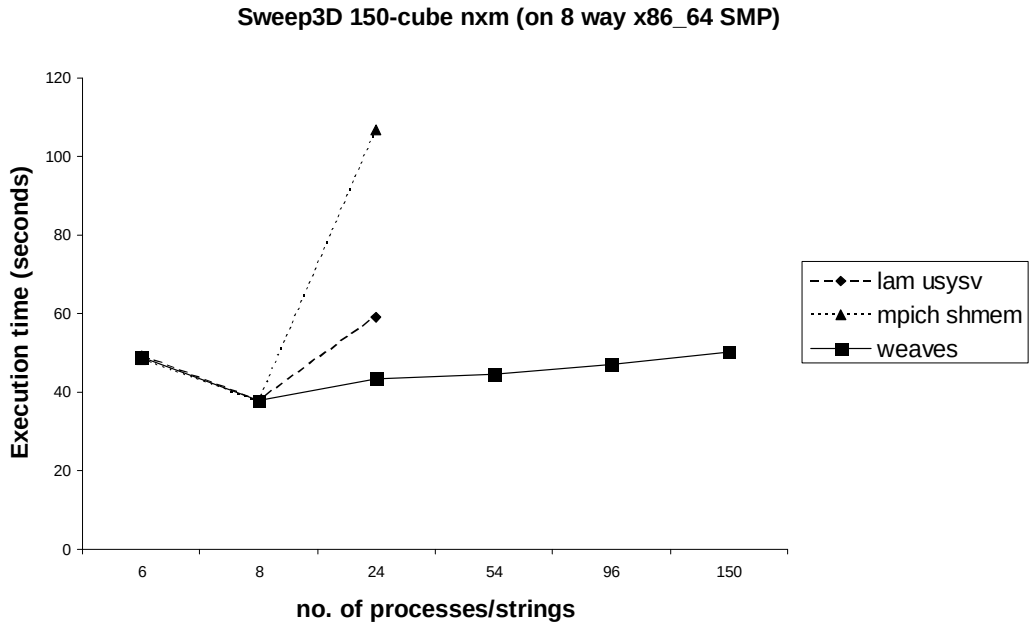


Figure 5.13: Comparison of performance results of Weaved Sweep3D against LAM-based and MPICH-based Sweep3D. The performance of the Weaved realization matched that of the LAM-based and MPICH-based realizations as long as the number of strings/processes was less than the number of processors. When the number of strings/processes was increased beyond the number of processors (8), the Weaved realization performed much better.

The LAM-based and MPICH-based realizations perform poorly beyond the 2x4 split (8 processes) because they rely on OS-level shared memory schemes for inter process communication (IPC), which do not scale beyond the number of processors. Their reliance on OS-level IPC is a direct consequence of following the process paradigm. The Weaves

framework works around this problem by facilitating low-overhead collaboration among the Sweep3D solver threads through a user-level sharing of global variables. In effect, these results show that the low-overhead collaboration facilitated by the Weaves framework allows for greater scales of achievable parallelism compared to traditional MPI-based collaboration [KBA92, ANLUC06].

Summary

Results from the first experiment show that Weaved scientific applications can support nearly twice the number of unmodified non-threadsafe parallel solvers as corresponding process-based applications. Results from the second experiment show that Weaved applications can transparently exploit low-overhead collaboration among parallel solver threads through a user-level sharing of global varaibles, which allows for greater scales of achievable parallelism than MPI-based collaboration allows.

Together, these results indicate that the Weaves framework is instrumental in helping legacy parallel scientific applications transparently exploit the scalability of threads for larger scales of parallelism, the overall goal of this research. Ultimately, larger scales of parallelism enable more accurate modeling of multi-physics phenomena. The results, therefore, substantiate the efficacy of the Weaves framework[2].

[2]Data and codes for both experiments can be obtained from http://blandings.cs.vt.edu/joy.

5.2.4 Configuring Weaves for HPC

This section provides an examination of various configurations of the Weaves framework for high performance scientific computing (HPC) on shared memory multi-processor machines (SMPs). Figure 5.14(a) shows one configuration invoking Weaves as part of a larger problem solving environment (PSE). Here, the PSE provides the tapestry specifications and the module codes and uses the framework to compose and execute them. Figure 5.14(b) shows another similar scenario with the Weaves framework operating within a larger performance modeling framework [AS00], such as POEMS [ABB+00, BBD00]. Here, POEMS supplies the modeling capability for performance characterization, while the Weaves framework takes care of the low-level composition and execution of unmodified scientific codes. Thus, systems such as POEMS can utilize the Weaves framework as a scalable and efficient substrate for scientific modeling.

Figure 5.14(c) shows another configuration that POEMS exemplifies, the simulation of MPI-based real-world codes using the MPI-SIM [PB98]. MPI-SIM uses a multi-threaded architecture to simulate MPI. However, it assumes that the linked code base is thread-safe. This assumption might not hold for distributed programs, especially those that use global variables. However, using the Weaves framework, the real-world codes can be run as independent virtual threads, thus presenting a thread-safe view of the application to MPI-SIM. This configuration of the framework enables a wider variety of real-world parallel codes to be used in simulations with MPI-SIM. We used a similar configuration for experiments with PDE solvers and Sweep3D.

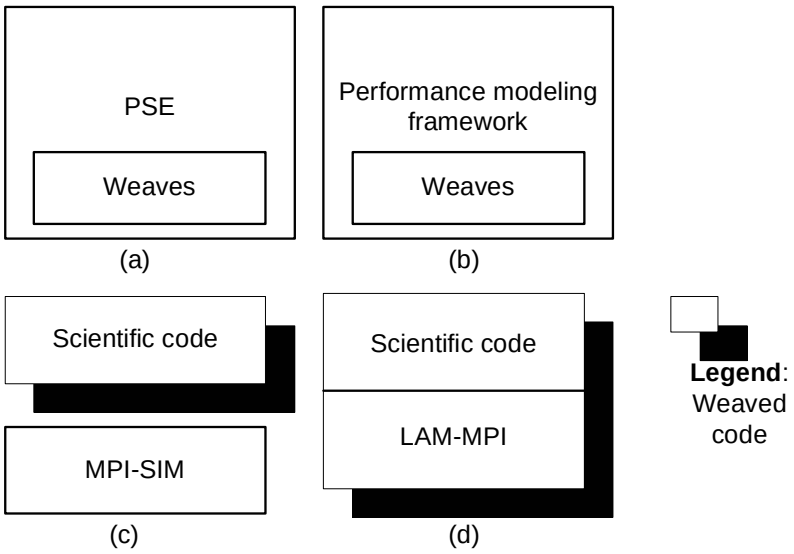


Figure 5.14: Relationship between the Weaves framework and (a) a problem solving environment and (b) a performance modeling framework. Advanced configurations of the Weaves framework for scientific computing: (c) Weaved scientific codes running over MPI-SIM and (d) Weaved scientific codes over Weaved MPI implementations.

Figure 5.14(d) depicts the most radical configuration of the Weaves framework for HPC. It loads modules of a real-world MPI library (e.g., the LAM MPI [PTL06] implementations) and links it to application modules. This configuration has the advantage of emulating real-world execution of not just the application, but its original communication interface, a truly native virtual machine abstraction for MPI codes. It facilitates the study of the effects of different MPI implementations on the performance of scientific codes. In this configuration, the only simulated entity is the communication channel, not the operation of MPI. The Open Network Emulator (ONE) has used a similar configuration of the framework for large scale network emulations.

5.3 Summary

This chapter has provided a demonstration of the utility of the Weaves framework in software areas that consist of legacy parallel programs. It has presented various case studies from the areas of network emulation and parallel scientific computing. Certain Weaved instances of network emulation and parallel scientific programs have further illustrated the design, implementation, and development of Weaved applications. One of the case studies has explained the use of string continuations. These case studies, purposely chosen from two radically different domains of parallel software systems, emphasize the broad impact of our work.

The experiments described in this chapter have corroborated that Weaved instances of net-

work emulation and parallel scientific applications can transparently run multiple identical, but independent, threads of legacy procedural programs that contain global variables. The experiments have also indicated that:

1. Weaved instances of network emulation can selectively share unmodified real-world IP stacks among independent telnet threads.

2. Weaved instances of scientific applications, such as Sweep3D, can selectively share global varaibles among independent solver threads.

Together, these experiments show that the Weaves framework facilitates user-level selective sharing of global variables among constituent parallel threads of legacy parallel programs.

Finally, we have explained the results of experiments with large real-world network emulations and parallel scientific applications. The results of experiments with network emulation show that Weaved realizations of real-world network scenarios transparently exploit the scalability of threads for larger scales of parallelism. A Weaved emulation can emulate a thousand virtual nodes over a single physical machine. On increasing the number of machines, the emulation speeds up super-linearly i.e. on increasing the number of machines by a factor of N, the time taken to run a certain emulation drops by a factor greater than N. In effect, such super-linear speedup exemplifies the scalability of Weaved network emulations.

The results of the experiments with parallel scientific applications show that Weaved realizations can support nearly twice the number of unmodified and non-threadsafe parallel solvers as the corresponding process-based realizations. The results also show that Weaved appli-

cations can transparently exploit low-overhead collaboration among parallel solver threads through user-level sharing of global variables, which allows for *significantly* greater scales of achievable parallelism than MPI-based collaboration does.

These results together indicate that the Weaves framework is instrumental in helping "unmodified" legacy parallel programs exploit the scalability of intra-process threads for larger scales of parallelism, the overall goal of this research. Ultimately, larger scales of parallelism enable more accurate testing of network protocols and more accurate modeling of multiphysics phenomena. The results, therefore, substantiate the effectiveness of the Weaves framework.

Chapter 6

Concluding Remarks

This thesis has proposed the Weaves runtime framework for parallel programs. Weaved applications can load encapsulated runtime modules from relocatable object files, as well as multiple independent modules from a single object file without entailing any modifications to the concerned object file. By allowing direct runtime control over the resolution of individual references in a module's code, the Weaves framework empowers programs with the ability to manipulate their composition at fine granularities. Through modules, the Weaves framework supports the transparent encapsulation and multiple instantiation of legacy procedural codes in intra-process environments.

Just as the compile-time linking of object files creates an executable program, the runtime composition of a set of modules creates a weave. A weave is, therefore, an intra-process subprogram that can support a flow of execution. The framework allows a single module to

be shared among multiple weaves that can be leveraged to realize arbitrary graph-like sharing of modules among the different weaves. Direct control over the resolution of individual references can extend this selective sharing among weaves to finer granularities.

All components of Weaved applications, including the fundamental module, are intra-process runtime units. Also, Weaved applications need not instrument any code modifications on modules either at the source level or at the level of relocatable objects (native code patches). They can use the framework's facilities transparently.

Legacy procedural codes are easily available as relocatable objects. Hence, the Weaved versions of legacy parallel programs can transparently (without code modifications) load multiple encapsulated modules of legacy procedural codes into intra-process runtime environments. When they run strings through weaves composed from distinct sets of identical modules, the strings do not share any global variables. This capability directly addresses the first research challenge: the need to transparently separate global variables used by identical, but independent, threads of a legacy parallel program.

Strings running through weaves experience all the elements of sharing reflected in the compositions of those weaves. Therefore, Weaved versions of legacy parallel programs can transparently realize arbitrary multi-granular selective sharing of functions and variables among strings. Since strings are essentially intra-process threads, this selective sharing among strings addresses the second research challenge: the need to transparently realize multi-granular selective sharing of global variables among the threads of a legacy parallel program.

6.1 Salient Contributions

The *main* contribution of the Weaves framework lies in its ability to facilitate execution of "unmodified" legacy parallel programs over lightweight intra-process threads without inadvertent sharing of global variables among constituent threads. This contribution allows legacy parallel programs to exploit the scalability of threads without any modifications.

A *second* contribution of the Weaves framework lies in its ability to transparently facilitate low-overhead user-level sharing of global variables among parallel threads of legacy parallel programs. This contribution allows exploitation of low-overhead collaboration between constituent parallel threads of a legacy parallel program.

Together, these contributions facilitate larger scales of parallelism for Weaved legacy parallel programs through better utilization of resources on a single multi-core machine or SMP workstation. Ultimately, larger scales of parallelism can facilitate more accurate modeling of large-scale parallel phenomena such as networks and multi-physics problems.

The Weaves framework institutes lateral advances in software areas that frequently encounter legacy parallel programs. This research has demonstrated these advances through experiments with large real-world network emulations and parallel scientific applications. These experiments in two radically different domains of parallel systems emphasize the broad impact of the Weaves framework.

The results from the experiments with network emulation show that Weaved realizations transparently exploit overall resources for larger scales of parallelism. A Weaved emulation

can emulate a thousand virtual nodes over a single physical machine. On increasing the number of machines, the emulation speeds up super-linearly i.e. on increasing the number of machines by a factor of N, the time taken to run a certain emulation drops by a factor greater than N. In effect, such super-linear speedup exemplifies the scalability of Weaved network emulations.

The results of the experiments with parallel scientific applications show that Weaved realizations can support nearly twice the number of unmodified and non-threadsafe parallel solvers as the corresponding process-based realizations can. The results also show that Weaved applications can transparently exploit low-overhead collaboration among parallel solver threads through user-level sharing of global variables, which allows for *significantly* greater scales of achievable parallelism than MPI-based collaboration allows.

These results together demonstrate that the Weaves framework is instrumental in helping legacy parallel programs transparently exploit overall resources for larger scales of parallelism. Ultimately, larger scales of parallelism enable more accurate testing of network protocols and more accurate modeling of multi-physics phenomena. These lateral advances, therefore, substantiate the efficacy of the Weaves framework.

6.2 Other Aspects

At the basic level, the Weaves framework offers its services as a library that supports simple APIs. Users can explicitly program the composition of Weaved applications using these

APIs. The framework also provides a meta-language for specifying the composition of a Weaved application in a configuration file and a script that automatically creates and runs the application from the meta-description. Essentially, configuration files are very similar to Makefiles. Consequently, from the usability perspective, composing Weaved applications is comparable to writing Makefiles.

Another noteworthy aspect of the Weaves framework is its current implementation. The Weaves framework is currently implemented on GNU/Linux over three architectures: x86, x86_64, and ia64. Weaves' runtime loader and linker called Load and Let Link (LLL) implements the core aspects of the framework, which include loading modules, composing weaves and direct control over the resolution of individual references. The implementation is fairly architecture-neutral. A port to the Power PC architecture is currently underway. Porting to different operating systems such as Windows and OS-X poses some problems. However, since most regular operating systems allow object file based decoupling of applications, theoretically, the Weaves framework is portable across a wide range of operating systems and architectures.

6.3 Summary

Parallel computing systems are becoming pervasive. An important effect of the increased adoption of parallel systems is that many contemporary applications are being explicitly programmed for large-scale parallelism. These applications benefit from mechanisms that

facilitate better exploitation of an individual machine or a single shared-memory multiprocessor (SMP) for larger scales of parallelism.

This research has proposed the Weaves runtime framework for parallel programs. The Weaves framework exploits lightweight intra-process threads and user-level low-overhead collaboration among parallel threads to facilitate larger scales of parallelism on an individual computer node or a single SMP machine. The framework is particularly beneficial to traditionally process-based parallel applications that use *legacy procedural codes*. The reason for these benefits is that the framework does not entail any modifications to these codes that have been validated and verified over decades of research and usage even if the codes contain global variables and are non threadsafe. To the best of our knowledge, the execution of unmodified legacy parallel programs over lightweight intra-process threads has never been attempted.

Ultimately, the Weaves framework helps legacy parallel programs transparently exploit overall resources for larger scales of parallelism. Larger scales of parallelism enable more accurate software modeling of large-scale parallel phenomena, such as real-world networks and multiphysics natural occurrences. This research has experimentally demonstrated lateral advances instituted by the Weaves framework in such diverse software areas as network emulation and parallel scientific computing that comprise large-scale legacy parallel applications. These lateral advances substantiate the efficacy of the current research.

Chapter 7

Ongoing Work

Ongoing work on the Weaves framework mainly focuses on reconfigurability or adaptivity of unmodified applications. The framework's facilities for flexible runtime loading and fine-grain dynamic linking of native code objects allows for unforeseen and arbitrary expansion, contraction and substitution in unmodified application code-bases at runtime. Such dynamic code mutations are useful for (a) adaptive applications that need to rewire themselves in response to dynamic conditions, (b) code swapping for mission critical systems and (c) automatic code overlaying for programs in constrained environments. This chapter showcases the adaptivity of Weaved applications through some case studies. It also mentions some other aspects of ongoing work.

7.1 Adaptivity of Weaved Applications

Extra facilities for runtime flexibility offered by a binary loading and linking framework are available to many high-level languages, frameworks, and models. The Weaves framework's runtime support for loading modules at arbitrary granularities is useful to programs seeking to expand their code base in a flexible manner. The ability of Weaved applications to load referentially incomplete modules and control their composition at the fine granularity of individual references is useful to programs trying to reduce resource consumption through automatic code overlaying.

Current software development processes support a fair degree of modularity and interface standardization. For instance, procedural programmers, who use languages such as C and FORTRAN, often compartmentalize programs into separate source files and compile them into corresponding object files (.o). Once they have fixed the cross-linking interfaces of these object files, the codes contained therein can be changed. The problem, as Figure 7.1(a) depicts, is that this compartmentalization is restricted to the pre-runtime domain. To create an executable, programmers must integrate the objects into *one* executable file, which obfuscates the reified structure of the application at runtime.

In contrast, the Weaves framework allows applications load runtime modules from objects files at arbitrary granularities without affecting the compartmentalization. Moreover, as seen in Figure 7.1(b), the framework maintains the decoupling among modules even after they are composed into a runable application weave. Applications can use these facilities to

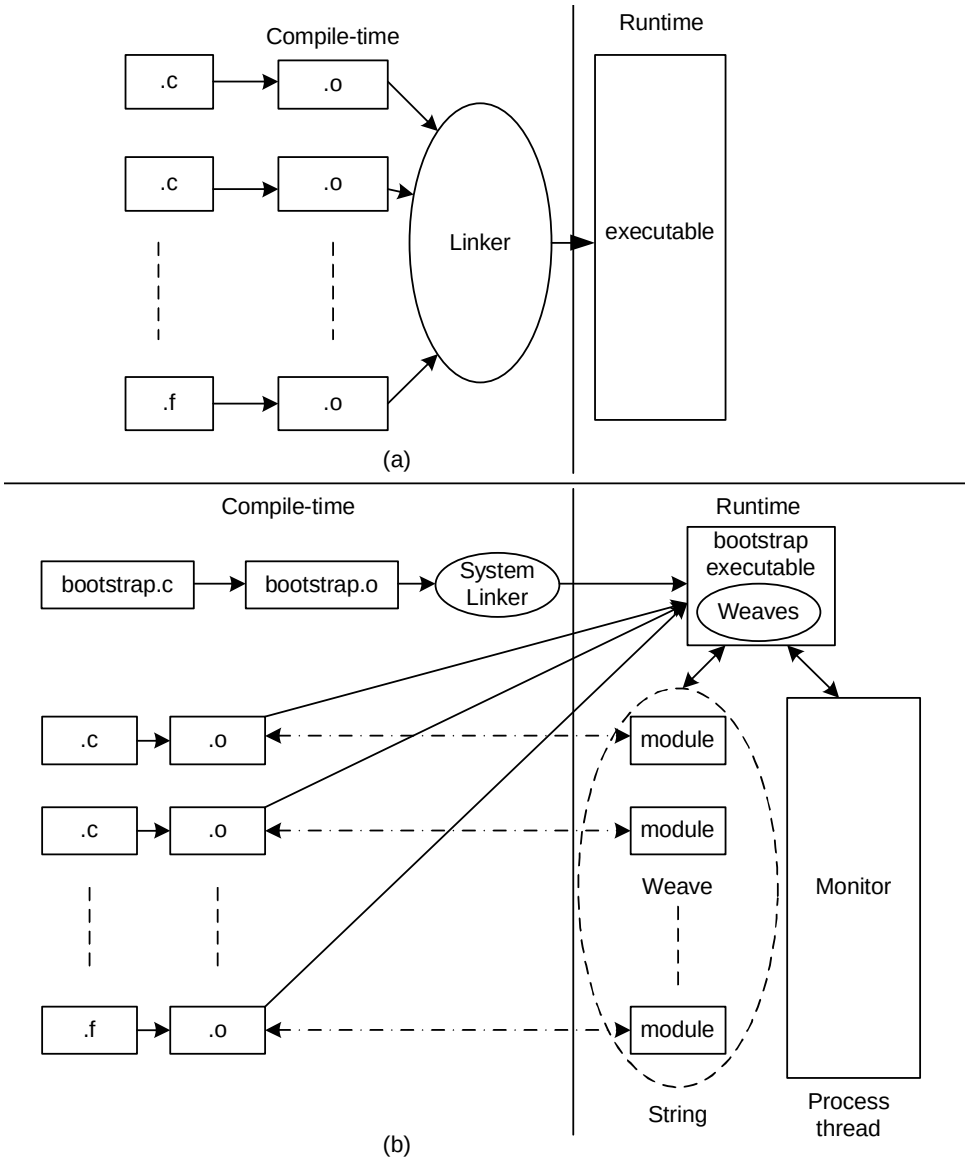


Figure 7.1: (a) Normal loading and linking. (b) Weaved application linking.

exploit the source file based decoupling at runtime. This runtime decoupling, in turn, aids interface-oriented code swapping or on-the-fly component replacement.

In effect, the Weaves framework can build a minimal functional program from a set of modules, while retaining the ability for future modifications. It facilitates runtime decoupling, runtime composition, and dynamic adaptation of applications. Using direct control over the resolution of individual references, Weaved applications can tune dynamic composition to their needs. They can explicitly specify different dynamic resolution handlers for different references. Using the 'monitor' component and the APIs for global control over a tapestry, Weaved applications can asynchronously modify their constitution at runtime.

Whereas adaptive programming techniques, models and frameworks help architect application reconfigurability at a higher level, an orthogonal low-level framework such as Weaves complements them by providing the necessary runtime support. Most importantly, the Weaves framework can facilitate unforeseen adaptivity for legacy procedural programs without *any* modifications to their codes. Some aspects of the framework can be more productive when coupled with strategies for dynamic state and stack manipulation [Hef04].

7.2 Dynamic Code Expansion

This section briefly exemplifies the Weaves framework's ability to support arbitrary expansion in application code bases through a hypothetical case of network emulation. Weaved network emulations can model dynamic characteristics of real-world networks. For instance,

to emulate a new machine joining a network, a Weaved emulation simply needs to load a new IP stack module. Again, as Figure 7.2 shows, to emulate a new FTP[1] application joining an existing machine, the emulation needs to load a FTP module, compose it with the concerned IP stack and start a string at the new FTP's start function. In Figure 7.2, part A depicts an emulation of two network hosts. Each host initially consists of a telnet (T) running over an IP stack (IP). The figure shows the weaves (W) and the strings (S). Part B shows the configuration after an FTP (F) joins each host.

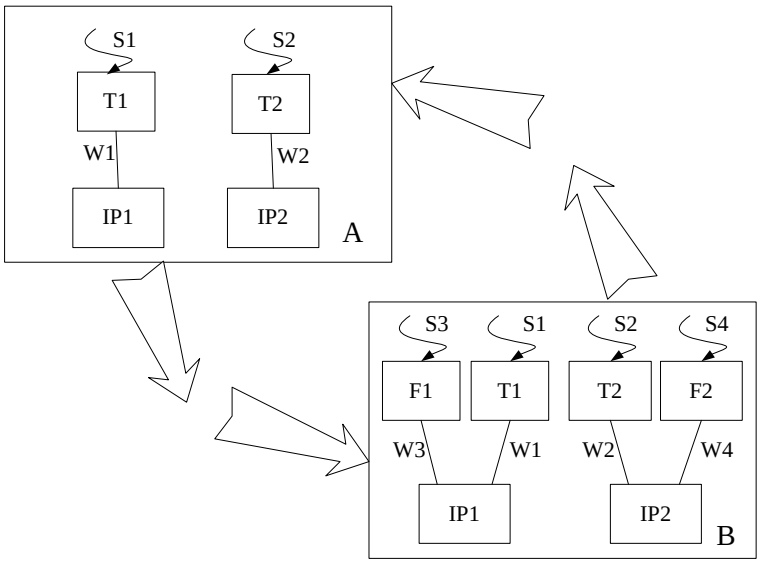


Figure 7.2: Modeling network dynamics using the Weaves framework.

We use the term 'spatial adaptivity' for dynamic inclusion of program elements, because it institutes an *unforeseen* expansion in an application's code *space*. Weaved applications can dynamically compose or recompose old and new modules into different or additional weaves. The Weaves framework does not entail any modifications to network application codes to

[1]File Transfer Protocol

facilitate spatial adaptivity.

7.3 Dynamic Code Swapping

Mission critical systems often consist of performance driven software programs that need to run continuously over long periods of time. These long running programs cannot be easily upgraded without the ill effects of costly downtime. Moreover, many high-performance programs use optimistic algorithms. In simple terms, optimistic algorithms advocate calling a certain function among many alternatives based on a 'best guess' according to available information. However, an optimistic selection can be erroneous, thereby requiring runtime swapping out of the erroneous selection in favor of a better alternative function. The Weaves framework's support for application controlled dynamic linking of programs at multiple granularities is useful for dynamic code swapping.

To exemplify the Weaves framework's ability to support dynamic code swapping, we ran a simple experiment where a program component was asynchronously replaced by a better implementation. This experiment consisted of an application to sort integers. The application's `main` routine continually generated a set of random numbers and then called the `sort` routine to sort the set through a standardized interface:

```
void sort (int *array, int size);
```

As Figure 7.3 shows, we programmed the sources for the `main` and the `sort` functions in C in two files, main.c and sort.c, respectively, with an external reference (`sort`) from the

former to the latter, such that an executable `sorter` was generated as:

```
gcc o sorter main.c sort.c
```

For Weaves-based modules, we compiled the *unmodified* source files into objects main.o and sort.o. We loaded one module of each object, composed them into one weave and started a string at the entry function in main.o. The main process thread, the monitor, then waited for user commands to perform requested modifications. Our default sort routine was `bubblesort`.

Midway through execution, without stopping the application, we asked the monitor to load an implementation of `mergesort` as another module and dynamically redirected the reference to `sort` in `main` to the definition `mergesort` in the new module. The output showed a seamless handover from `bubblesort` to `mergesort`. We then retraced these steps with an implementation of `quicksort`. The output again showed a seamless handover from `mergesort` to `quicksort`.

This experiment shows that if a better code component becomes available, the Weaves framework can help an application asynchronously switch to it at runtime. This facility of the framework can help performance driven applications evolve over time and dynamically adjust to unforeseen performance requirements. For mission critical systems, the Weaves framework can facilitate runtime upgrading as and when better code components become available. The experiment also shows that the Weaves framework does not entail any modifications to the original program's sources to facilitate code swapping.

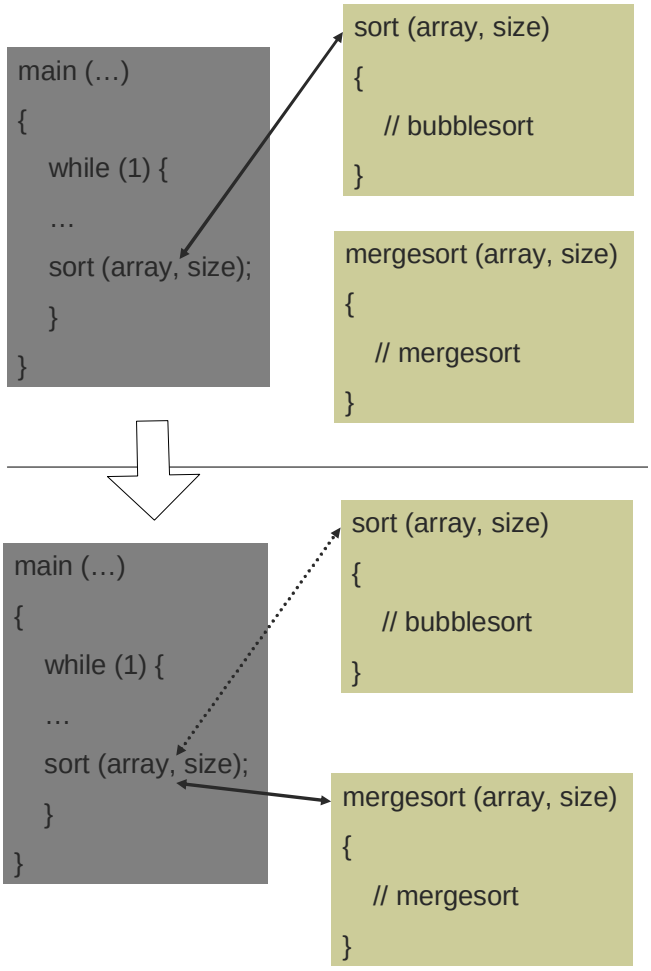


Figure 7.3: Dynamic code swapping using the Weaves framework.

7.4 Dynamic Code Overlaying

With continuing trends in processor miniaturization, computing power is becoming more commonplace. Embedded processors in handheld devices, cellular telephony and consumer appliances account for the majority of the processor market. These devices have relatively small memory footprints and experience dynamic resource constraints. In contrast, application software for these devices—user interface libraries, web browsers, email clients and so forth--continue to increase in complexity. Using the Weaves framework's support for dynamic loading and controlled composition of codes at various granularities, large programs, such as web browsers, can automatically prune their codes at runtime in response to dynamic resource constraints.

To showcase the ability of the Weaves framework to support dynamic overlaying of program codes, we describe experiences with a minimal web-browser called Dillo [Vik06] freely available under the GNU Public License. We chose Dillo for 3 reasons:

1. Its source-code is freely available.

2. Its memory efficient minimalism is apt for mobile and limited-resource devices.

3. Its source is in traditional procedural C.

Even though programmed in C, the code for Dillo is fairly modularized with different components programmed in different source files. For simplicity, we arbitrarily split the application into 4 modules (1, 2, 3 and 4) while maintaining, as far as possible, the linking sequence

decreed by the application's default building mechanism. Of these, module 2 consisted of implementations for `find_text` (searching for text strings) and `select_link` ('clicking' on a hyperlink) functionalities.

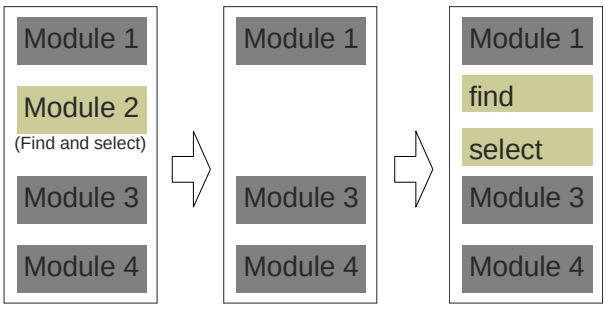


Figure 7.4: Dynamic code pruning in memory constrained Weaved applications.

As Figure 7.4 shows, we loaded and linked the modules and started a string at the main function of Dillo in module 3. The process thread was then used as the monitor. To exemplify that the Weaved version of Dillo could automatically prune its code based upon unforeseen memory constraints, we asked the monitor to asynchronously unload module 2 at runtime, thereby reducing memory consumption at the expense of some functionality, `find_text` and `select_link`. The rest of the application was left untouched and the unresolved references, resulting from the unloading of module 2, were redirected to a single minimal function that returned 0 (or NULL). As expected, the browser continued to run correctly and smoothly in every respect, except that the `find_text` and `select_link` functionalities were not available.

We then split the object file corresponding to module 2 into separate objects for `find_text` and `select_link` while the browser was still running, dynamically loaded the two new modules (find and select), and composed them with modules 1, 3 and 4. The browser could then

be contracted, expanded, or upgraded at a finer granularity, that is, one could unload and upgrade `find_text` or `select_link` individually.

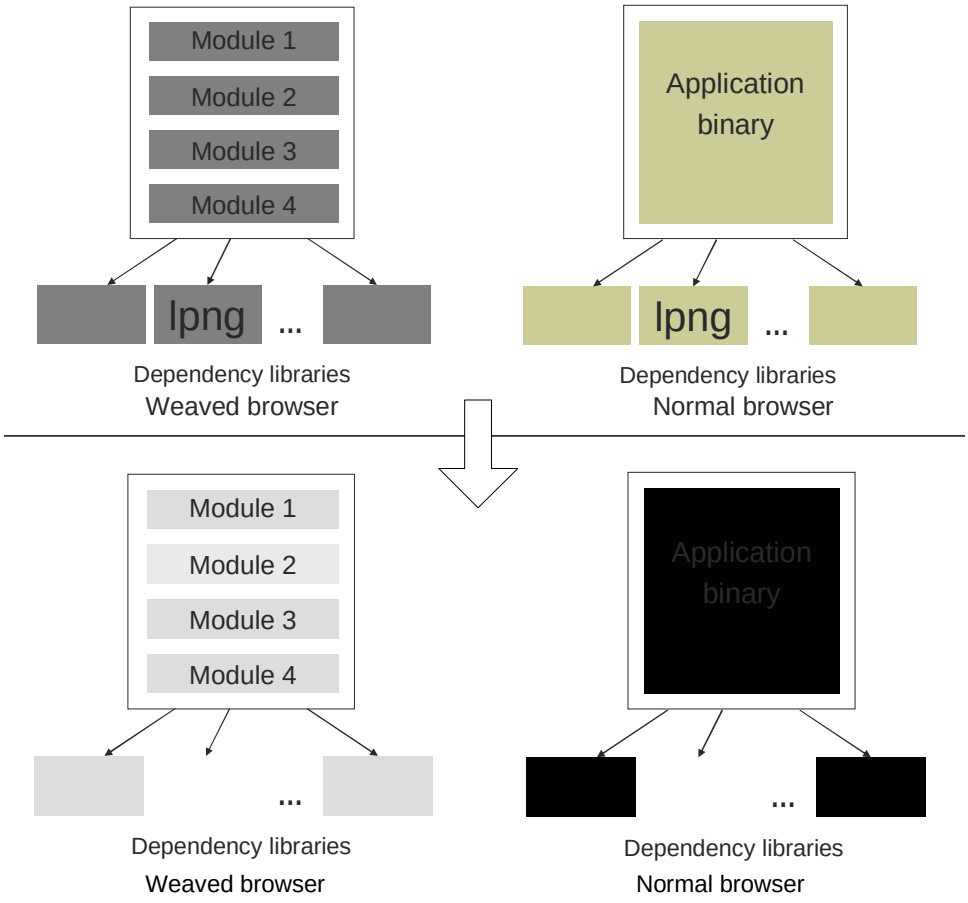


Figure 7.5: Automatic adjustment of Weaved applications to available software infrastructure.

Figure 7.5 depicts another experience that demonstrates the utility of the Weaves framework for automatic overlaying of programs. We built Dillo into its standard executable program as well as into a Weaves-based version. The original and the Weaved versions of the browser ran identically with identical software infrastructure requirements, such as dependency libraries.

We then ported both of them to a different, but architecture-compatible, machine that lacked a dependency library for portable network graphics, `libpng` [Roe06]. The original executable failed to run due to the lack of a dependency library. The Weaved version, however, promptly started up with all facilities except the portable network graphics. The framework automatically redirected all unresolved references to libpng to a minimal NULL returning function, thereby preventing adverse effects on the rest of the application.

Neither of the experiences described here demanded *any* modification to the Dillo browser's codes. These experiences show that the Weaves framework can be useful to applications that run under dynamic and unforeseen resource constraints on miniature devices. The framework can automatically contract an unmodified application according to available memory as well as available software infrastructure. It can also dynamically reinstate pruned codes when resource conditions are favorable. Finally, it allows runtime re-componentization of unmodified applications at multiple granularities.

7.5 Other Aspects of Ongoing Work

This section presents various other aspects of the Weaves framework that are currently being researched and experimented with. Brief discussions of these aspects follow:

- A comprehensive analyses of Weave's loading and linking toolkit (Load and Let Link or LLL) is being pursued. Focus is on performance overhead of LLL and testing LLL on standard benchmarks. Specific test applications consist of network filtering programs

and runtime program tracing utilities.

- Many programs comprise performance intensive core routines. These routines are typically available as commodity libraries from various vendors. Currently, applications link against one such library only. However, these routines are often tailored and optimized differently by different vendors. The result is that each vendor's product is selectively better than others. This aspect of ongoing work seeks to use the Weaves framework to selectively link to more than one of these similar libraries and use the best routines from each, thereby enhancing the overall performance of an application.

- Interposition, or 'call hijacking' for code injection, has several advantages. For instance, interposition facilitates low-overhead tracing of operational applications. Currently, interposition is mainly restricted to calls from executables to libraries. By extending compositionality to the finer granularity of relocatable objects, the Weaves framework offers a way for utilizing interposition for fine-grain tracing. This aspect of ongoing work intends to demonstrate the usefulness of the framework for fine-grain interposition on some standard benchmarks and real-world programs.

- Temperature and power consumption control are gaining importance in the backdrop of increased use of powerful supercomputers and clusters. Thermal control is currently exercised at the hardware level through various throttling mechanisms. This aspect of ongoing work attempts to utilize the Weaves framework to asynchronously slow down or speed up applications depending on the temperature level detected by on-board sensors.

Essentially, this work intends to use controlled dynamic composition of unmodified programs to automatically insert halting code at function boundaries. Once the slowing down of applications has allowed the cooling system enough time to dissipate the excess heat, the halting code can be automatically removed and application execution can be resumed at full speed.

- Finally, ongoing work also looks at advanced Weaved network emulations and Weaves-based modeling of complex real-world scientific problems.

7.6 Summary

This chapter has presented aspects of ongoing work on the Weaves framework, which mainly focuses on reconfigurability or adaptivity of unmodified applications. The framework's facilities for flexible runtime loading and fine-grain dynamic linking of native code components allows for unforeseen and arbitrary expansion, contraction and substitution in unmodified applications at runtime. Ongoing work endeavors to use these dynamic code mutations for (a) adaptive applications that need to rewire themselves in response to dynamic conditions, (b) code swapping for mission critical systems and (c) automatic code overlaying for programs in constrained environments. This chapter has showcased the adaptivity of Weaved applications through some case studies.

Other aspects of ongoing work include (1) testing and analyses of the Weaves framework's runtime loader and linker (LLL), (2) using the framework to selectively link a program to

multiple commodity libraries for performance gains, (3) using control over linking of Weaved programs for low-overhead tracing through fine-grain interposition, and (4) exploiting the framework's support for controlled program composition to throttle applications according to temperature and power constraints. Finally, ongoing work also looks at advanced Weaved network emulations and Weaves-based modeling of complex real-world scientific problems.

Bibliography

[ABB+00] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. POEMS: End-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, 26(11):1027–1048, 2000.

[ANLUC06] Argonne National Laboratory and University of Chicago. MPICH2, 2006. http://www-unix.mcs.anl.gov/mpi/mpich/, Last accessed on July 17 2006.

[AP99] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *Proceedings of the ACM SIGCOMM '99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 263–274. ACM Press, 1999.

[AS94] C. Alaettingoglu and A. U. Shankar. Design and implementation of MARS: A routing testbed. *Journal of Internetworking: Research and Experience*, 5(1):17–41, 1994.

[AS00] V. S. Adve and R. Sakellariou. Application representations for multiparadigm performance modeling of large-scale parallel scientific codes. *International Journal of High Performance Computing Applications*, 14(4):304–316, 2000.

[BB06] B. Barney. POSIX threads programming, 2006. http://www.llnl.gov/, Last accessed on July 17 2006.

[BBD00] J. C. Browne, E. Berger, and A. Dube. Compositional development of performance models in POEMS. *International Journal of High Performance Computing Applications*, 14(4):283–291, 2000.

[BBH+98] H. E. Bal, R. Bhoedjang, R. F. H. Hofman, C. J. H. Jacobs, K. Langendoen, and T. Rühl. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998.

[BKdSH01] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger. Object-based adaptive load balancing for MPI programs. In *Proceedings of the International Conference on Computational Science, LNCS 2074*, pages 108–117. Springer Verlag, 2001.

[BP96] L. S. Brakmo and L. L. Peterson. Experiences with network simulation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 80–90. ACM Press, 1996.

[BVB06] C. Bergstrom, S. Varadarajan, and G. Back. The distributed Open Network Emulator: Using relativistic time for distributed scalable simulation. In *Pro-*

ceedings of the 20th IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2006), pages 19–28. IEEE Computer Society, 2006.

[CCA04] The Common Component Architecture Forum. CCA: Common component architecture, 2004. http://www.cca-forum.org/, Last accessed on July 17 2006.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CK01] K. M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. Technical report, California Institute of Technology, Pasadena, CA, USA, 2001.

[CS03] M. Carson and D. Santay. NIST Net: A linux-based network emulation tool. *Computer Communication Review*, 33(3):111–126, 2003.

[DGTY95] J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 95)*, pages 19–28. ACM press, 1995.

[DHRR99] T. T. Drashansky, E. N. Houstis, N. Ramakrishnan, and J. R. Rice. Networked agents for scientific computing. *Communications of the ACM*, 42(3):48–54, 1999.

[EII00] Ericsson IP Infrastructure (formerly Torrent Networks). Virtual Net: Virtual host environment, 2000.

[Eng06] R. S. Engelschall. GNU Pth: A user-level thread library, 2006. http://www.gnu.org/software/pth/, Last accessed on July 17 2006.

[Fos96] I. T. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1996.

[FOT92] I. T. Foster, R. Olsen, and S. Tuecke. Productive parallel programming: The PCN approach. *Journal of Scientific Computing*, 1(1):51–66, 1992.

[FSF05] Free Software Foundation. GNU C library, 2005. http://www.gnu.org/, Last accessed on July 17 2006.

[FT89] I. T. Foster and S. Taylor. Strand: A practical parallel programming tool. In *Proceedings of the North American Conference*, pages 497–512. MIT Press, 1989.

[FV06] K. Fall and K. Varadhan. Network emulation with the NS simulator, 2006. http://www.isi.edu/nsnam/ns/doc/index.html, Last accessed on July 17 2006.

[GB82] D. Gelernter and A. J. Bernstein. Distributed communication via global buffer. In *Proceedings of the Symposium on Princiles of Distributed Computing*, pages 10–18. ACM Press, 1982.

[Hef04] M. Heffner. A runtime framework for adaptive compositional modeling. Master's thesis, Virginia Tech, Blacksburg, VA, USA,2004.

[HK97] A. Helmy and S. Kumar. Virtual InterNetwork Testbed, 1997. http://www.isi.edu/, Last accessed on July 17 2006.

[HP88] N. C. Hutchinson and L. L. Peterson. Design of the x-kernel. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 65–75. ACM Press, 1988.

[HSK99] X. W. Huang, R. Sharma, and S. Keshav. The ENTRAPID protocol development environment. In *Proceedings of the IEEE INFOCOM '99 Conference on Computer Communications*, pages 1107–1115. IEEE press, 1999.

[JL05] F. H. Carvalho Jr. and R. D. Lins. The # model: Separation of concerns for reconciling modularity, abstraction and efficiency in distributed parallel programming. In *Proceedings of the 2005 ACM Symposium on Applied Computation (SAC 05)*, pages 1357–1364. ACM press, 2005.

[KBA92] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of the 3D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65(198), 1992.

[Kes97] S. Keshav. Real 5.0 overview, 1997. http://www.cs.cornell.edu/skeshav/real/, Last accessed on July 17 2006.

[Kne04] C. Knestrick. LUNAR: A user-level stack library for network emulation. Master's thesis, Virginia Tech, Blacksburg, VA, USA, 2004.

[LM01] H. P. Langtangen and O. Munthe. Solving systems of parallel differential equations using OOP techniques with coupled heat and fluid flow as example. *ACM Transactions On Mathematical Software*, 27(1):1–26, 2001.

[MDB03] N. Mahmood, G. Deng, and J. C. Browne. Compositional development of parallel programs. In *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing (LCPC03)*, pages 109–126. Springer Verlag, 2003.

[MR92] H. S. McFaddin and J. R. Rice. Collaborating PDE solvers. *Applied Numerical Mathematics*, 10:279–295, 1992.

[MS06] Microsoft Corporation. COM: Component object model technologies, 2006. http://www.microsoft.com/com/default.mspx, Last accessed on July 17 2006.

[Mu99] M. Mu. Solving composite problems with interface relaxation. *SIAM Journal on Scientific Computing*, 20(4):1394–1416, 1999.

[Muk02] J. Mukherjee. A compiler directed framework for parallel compositional systems. Master's thesis, Virginia Tech, Blacksburg, VA, USA, 2002.

[MV05a] J. Mukherjee and S. Varadarajan. Develop once deploy anywhere: Achieving adaptivity with a runtime linker/loader framework. In *Proceedings of the 4th workshop on Reflective and adaptive middleware systems (ARM '05)*. ACM Press, 2005.

[MV05b] J. Mukherjee and S. Varadarajan. Weaves: A framework for reconfigurable programming. *International Journal for Parallel Programming (Special Issue)*, 33(2):279–305, 2005.

[NAG06a] Numerical Algorithms Group. NAG library manual: Thread safety, 2006. http://www.nag.co.uk/downloads/fldownloads.asp, Last accessed on July 17 2006.

[NAG06b] Numerical Algorithms Group. NAG software downloads: Fortran library, 2006. http://www.nag.co.uk/downloads/fldownloads.asp, Last accessed on July 17 2006.

[NBF96] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996.

[NsN06] NsNam. The network simulator – ns, 2006. http://nsnam.isi.edu/, Last accessed on July 17 2006.

[OAR04] OpenMP Architecture Review Board. Official OpenMP specifications, 2004. http://www.openmp.org/drupal/node/view/8, Last accessed on July 17 2006.

[OMG06] Object Management Group. CORBA basics, 2006. http://www.omg.org/, Last accessed on July 17 2006.

[OPN05] OPNET Technologies. OPNET modeler: Network modeling and simulation environment, 2005. http://www.opnet.com/products/modeler/home.html, Last accessed on July 17 2006.

[OPN06] OPNET Technologies. OPNET, 2006. http://www.opnet.com, Last accessed on July 17 2006.

[Pax99] V. Paxson. End-to-end internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[PB98] S. Prakash and R. Bagrodia. MPI-SIM: Using parallel simulation to evaluate MPI programs. In *Proceedings of the 1998 Winter Simulation Conference (WSC 98)*, pages 467–474. ACM press, 1998.

[PCB94] S. Parkes, J. A. Chandy, and P. Banerjee. A library-based approach to portable, parallel, object-oriented programming: Interface, implementation, and application. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (SC 94)*, pages 69–78. IEEE/ACM, 1994.

[PTL06] Pervasive Technology Labs at Indiana University. LAM/MPI parallel computing, 2006. http://www.lam-mpi.org/, Last accessed on July 17 2006.

[Ram04] H. Ramankutty. Inter-process communication, 2004. http://linuxgazette.net/, Last accessed on July 27 2006.

[Ric98] J. R. Rice. An agent-based architecture for solving partial differential equations. *SIAM News*, 31(6), 1998.

[Riz97] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communications Review*, 27(1):31–41, 1997.

[RL98] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, 1998.

[Roe06] G. Roelofs. `libpng`, 2006. http://www.libpng.org/, Last accessed on July 17 2006.

[RTV99] J. R. Rice, P. Tsompanopoulou, and E. A. Vavalis. Interface relaxation methods for elliptic differential equations. *Applied Numerical Mathematics*, 32:291–245, 1999.

[Sat02] M. Sato. OpenMP: Parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS 02)*, pages 109–111. IEEE Computer Society, 2002.

[SteJr90] G. L. Steele Jr. Making asynchronous parallelism safe for the world. In *Conference Records of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 218–231. ACM Press, 1990.

[Ste97] W. R. Stevens. *UNIX network programming*. Prentice-Hall of India, 1997.

[Ter01] M. Terwilliger. PARSEC: Parallel simulation environment for complex systems, 2001. http://may.cs.ucla.edu/projects/parsec, Last accessed on July 17 2006.

[TIS95] Tools Interface Standards Committee. *Executable and Linkable Format (ELF) Specification*, 1995.

[Var02] S. Varadarajan. Weaving a code tapestry: A framework for reconfigurable programming, *DOE Early Career Proposal*. 2002.

[Vik06] J. Viksell. Dillo, 2006. http://www.dillo.org/, Last accessed on July 17 2006.

[VR05] S. Varadarajan and N. Ramakrishnan. Novel runtime systems support for adaptive compositional modeling in PSEs. *Future Generation Computing Systems (Special Issue)*, 21(6):878–895, 2005.

[VYW+02] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI '02)*. USENIX Association, 2002.

[Yuj01] L. Yujih. Distributed simulation of a large-scale radio network. *OPNET Technologies' Contributed Papers*, 2001.

Vita

Joy Mukherjee was born on the 16th of May 1978 at Ranchi, India. He graduated with a Bachelor of Technology (Honors) Degree in Computer Science and Engineering from the Indian Institute of Technology, Kharagpur, India in July 2000. He received the degree of Master of Science in Computer Science from Virginia Tech, Blacksburg, Virginia, USA in December 2002. His academic interests include Computer Systems, Systems Support for Programming Languages, Adaptive Software Systems, Parallel Computing, Binary Technologies, and Compilers. As asides he indulges in novels, progresses in the natural sciences, traveling, sketching, music, and various outdoor sports. He is to join Oracle, India for research and development of systems software support for reliable computer clusters.