Scalable Execution of Legacy Scientific Codes

Joy Mukherjee, Srinidhi Varadarajan, and Naren Ramakrishnan

660 McBryde Hall, Dept of Computer Science, Virginia Tech, Blacksburg, VA 24061
{jmukherj, srinidhi, naren}@cs.vt.edu

Abstract. This paper presents Weaves, a language neutral framework for scalable execution of legacy parallel scientific codes. Weaves supports scalable threads of control and multiple namespaces with selective sharing of state within a single address space. We resort to two examples for illustration of different aspects of the framework and to stress the diversity of its application domains. The more expressive collaborating partial differential equation (PDE) solvers are used to exemplify developmental aspects, while freely available Sweep3D is used for performance results. We outline the framework in the context of shared memory systems, where its benefits are apparent. We also contrast Weaves against existing programming paradigms, present use cases, and outline its implementation. Preliminary performance tests show significant scalability over process-based implementations of Sweep3D.

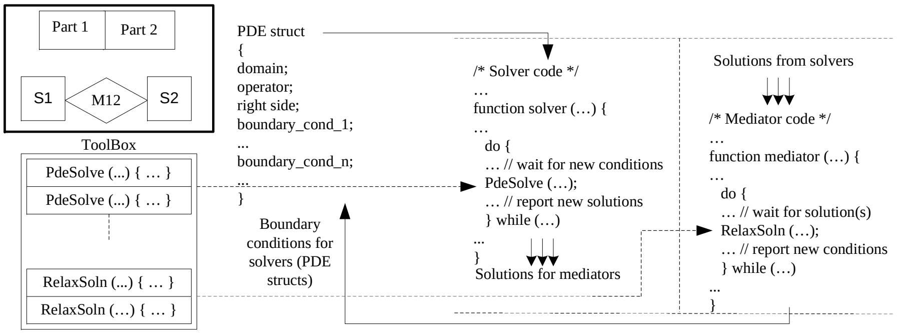
1 Introduction

The past decade has witnessed increasing commoditization of scientific computing codes, leading to the prevailing practice of compositional software development. The ability to combine representations for different aspects of a scientific computation to create a representation for the computation as a whole is now considered central to high-level problem solving environments (PSEs)[1]. Common solutions for compositional scientific software [2],[3],[4] are primarily targeted at the higher end of distribution and decoupling among processors – clusters, distributed memory supercomputers, and grids. As a result, they do not require expressive mechanisms for realizing certain facets of parallelism. For instance, most of them follow the process model along with message passing for data exchanges. Nevertheless, processes can lead to problems of scalability when resources are limited.

This paper concentrates on shared memory multiprocessor (SMP) machines. It introduces Weaves—a language neutral framework for scalable execution of scientific codes—for such platforms. Weaves exploits typical shared memory properties to enrich scalability of unmodified scientific applications. It facilitates creation of multiple namespaces within a single process (address) space with arbitrary sharing of components (and their state). The framework supports lightweight threads of control through each namespace thus enabling scalable exchange of state, code, and control information. We resort to two examples for illustration of different aspects of the framework and to stress the diversity of its application domains. The more expressive collaborating partial differential equation (PDE) solvers are used to exemplify developmental aspects, while freely available Sweep3D is used for performance results. We discuss related work, outline Weaves' design and implementation, and present use-cases.

2 Collaborating PDE Solvers

Our first example application involves collaborating partial differential equation (PDE) solvers [4], an approach for solving heterogeneous multi-physics problems using interface relaxation [5], [6] (see Fig. 1). Mathematical modeling of the multi-physics problem distinguishes between solvers and mediators. A parallel PDE solver is instantiated for each of the simpler problems and a parallel mediator is instantiated for every interface, to facilitate collaboration between the solvers. Fig. 1 illustrates typical solver and mediator codes. Among other parameters, a solver takes boundary conditions as inputs to compute solutions. The *PdeSolve* routine is chosen from a problem solving environment (PSE) toolbox depending on the PDE problem characteristics. Different *PdeSolve* routines may implement different algorithms, but could use identical names and signatures. Further, a composite problem might use the same solver on all subdomains or adopt different solvers. After computing solutions, a solver passes the results to mediators and waits till the mediators report back fresh boundary conditions. Upon the receipt of new conditions, it may recompute the solutions and repeat the whole process till a satisfactory state is reached. A mediator relaxes the solutions from solvers and returns improved values [6]. The *RelaxSoln* routines are also chosen from the PSE toolbox depending on the problem instance. Once



```
PDE struct
{
domain;
operator;
right side;
boundary_cond_1;
...
boundary_cond_n;
...
}
```

```
/* Solver code */
…
function solver (…) {
…
  do {
   … // wait for new conditions
   PdeSolve (…);
   … // report new solutions
  } while (…)
...
}
```

```
/* Mediator code */
…
function mediator (…) {
…
  do {
   … // wait for solution(s)
   RelaxSoln (…);
   … // report new conditions
  } while (…)
...
}
```

Fig. 1. *Inset (top-left)*: Simple composite multy-physics problem with two subdomains or parts. Each part is modeled by a PDE solver (S). The mediator (M) is responsible for agreement along the interface. Realistic scenarios can involve several solvers and mediators with complex graph-like connections. *Center*: Typical solver and mediator codes are shown. *PdeSolve* and *RelaxSoln* routines are chosen from a PSE toolbox.

again, multiple *RelaxSoln* routines may expose identical names and signatures and there is a choice of using the same or different mediator algorithms.

The PDE solver problem exemplifies three requirements from a parallel programming perspective: (i) Arbitrary state sharing: A part of solver state (corresponding to a boundary) should be accessible to a mediator. Additionally, different segments of solver state should be accessible to different mediators. (ii) Transparency: PDE solution and relaxation routines are mostly legacy procedural codes validated over decades of research. Modification to their sources should be minimized. (3) Scalability: Complex problem instances such as modeling of turbines and heat engines may involve thousands of solvers and mediators. Solution approaches should, therefore, be scalable.

Traditionally, the collaborative PDE solvers problem has been approached using agent technology [4] in highly distributed environments such as clusters and distributed memory machines. Agent-based solutions use message passing as an indirect representation of procedural invocations for arbitrary state sharing. Nevertheless, continuing rise of low-cost SMPs and increases in 'arity' of nodes open up new possibilities. On SMP machines, multiple flows of execution may run simultaneously on different processors, but over the same operating system (OS). Weaves exploits this proximity to manifest efficient state sharing through direct in-memory data accesses within a single address space. Transparency is harnessed through component-based composition of solvers and mediators. Scalability is gained through use of lightweight threads for flows of control.

3 Related Work

To our knowledge, not much research has been directed at providing scalability without code modification. Nevertheless, we contrast against some general approaches to parallel programming since they are powerful enough to be used for various purposes. For instance, the traditional agent-based message passing scheme may be implemented on SMPs by modeling each solver and mediator as an independent process. This approach aids code-reuse. However, inter-process communication and process switching overheads hamper scalability [1]. Scalability issues with multiple independent processes emphasize the use of lightweight intra-process threads for parallel flows of control. Techniques for concurrent [3], [7],[8], compositional [9], [10], and object-oriented [2] programming can enable scalable state sharing over lightweight threads through the use of in-memory data structures. However, all of these mechanisms resort to varying degrees of source-level constructs to enforce encapsulation (separation of namespace) and therefore do not meet our transparency requirement. (Recall that legacy PDE solver and relaxation codes may use identical names or symbols for functions/data).

4 Weaves

From the previous section we deduce that on SMP machines, the transparency and scalability requirements of the collaborating PDE solver example reduce to

(1) use of traditional procedural codes and programming techniques, and (2) use of lightweight intra-process threads for modeling parallel flows of control. The need is selective separation of state and namespace of intra-process threads without resorting to source-level programming. Scalability and transparency debar OS level solutions. These observations lead to the first step towards Weaves.

4.1 Design and Definitions

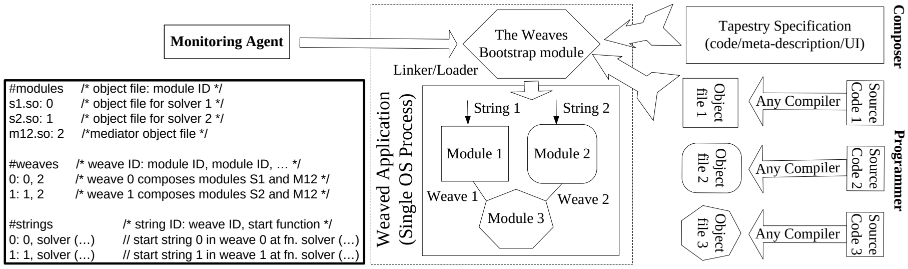
The Weaves framework creates encapsulated intra-process components called modules from source written in any language. A module is an encapsulated runtime image of a compiled binary object file. Each module defines its own namespace. Multiple identical modules have independent namespaces and global data within the address space of a single process. A module may make references to external definitions. Weaves offers application control over reference-definition bindings. While encapsulation of modules enforces separation, individual references may be explicitly redirected to achieve fine-grain selective sharing. References from different modules pointing to a particular definition result in sharing of the definition among the modules. The minimal definition of the module allows Weaves to flexibly work with high-level frameworks, models, and languages.

At the core of the Weaves framework is the definition of a weave. A weave[1] is a collection of one or more modules composed into an intra-process subprogram. From the viewpoint of procedural programs, weave composition from modules is similar to linking object files for executable generation. However, there are important differences: (1) Weave composition is an intra-process runtime activity (recall that modules are intra-process runtime entities). (2) Weave composition does not necessitate resolution of all references within constituent modules. (3) Reference redirections may be used later to fulfill completeness or to transcend weave boundaries. A weave unifies namespaces of constituent modules. Hence, identical modules cannot be included within a single weave. However, different weaves may comprise similar, but independent, modules. This facility of weaves helps create multiple independent copies of identical programs within a process space. Going a step further, the Weaves framework allows a single module to be part of multiple weaves. These weaves, therefore, share the contents of the common module. This lays the foundation for selective sharing and separation of state within the Weaves framework. Individual reference redirections extend such sharing among weaves in arbitrary ways.

A string is the fundamental unit of execution under Weaves. It is a lightweight intra-process thread. A string executes within a weave. Multiple strings may simultaneously run through the same or different weaves. The Weaves framework emulates the process model when one string is initiated through each one of multiple independent weaves. It emulates the traditional threads model when multiple strings are initiated through a single weave. Apart from these extremes, the framework also provides for arbitrary compositions of strings (realized through

[1] We use 'weave' to indicate the unit of composition and 'Weaves' to refer to the overall framework

the composition of associated weaves). A runtime issue can arise when a module is part of more than one weave – external references from a shared module may have to resolve to different definitions depending on the weave. We use late binding mechanisms to resolve this on-the-fly. Whenever a string accesses such an external reference, the framework's runtime environment queries the string for its weave and resolves to a definition accordingly. Weaves requires a minimal bootstrapping module (which runs on the main process thread of every application) to set up the target application – load modules, compose weaves, and start strings. The main process thread may later be used to monitor and externally/asynchronously modify the application constitution at runtime [1]. The entire application—including all modules, weaves, strings, and the monitor—runs within a single OS process. The framework provides a meta-language for specification of application configuration in a file and a script that automatically generates a bootstrap module, builds it, and initiates a live application from such a meta-description. A simple configuration file for a Weaves-based approach to realize the Fig. 1 (inset) scenario is shown in Fig. 2 (inset). One direction of current research on Weaves aims at an integrated GUI for tapestry specification and automatic execution. Fig. 2 diagrammatically illustrates the complete development process of a general application using Weaves.



```
#modules     /* object file: module ID */
s1.so: 0       /* object file for solver 1 */
s2.so: 1       /* object file for solver 2 */
m12.so: 2    /*mediator object file */

#weaves     /* weave ID: module ID, module ID, … */
0: 0, 2        /* weave 0 composes modules S1 and M12 */
1: 1, 2        /* weave 1 composes modules S2 and M12 */

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

Fig. 2. *Inset (bottom-left)*: A simple configuration file for a Weave-based approach to realize the Fig 1. (inset) scenario. Here, codes for S1, S2, and M12 are compiled into objects S1.so, S2.so, and M12.so respectively. Each solver module is composed into a distinct weave with the single mediator module. *Center*: Diagrammatic illustration of the development process of a general application using Weaves.

4.2 Implementation

Weaves' current prototype implementation works on x86 (32 and 64 bit) and ia64 architectures running GNU/Linux. Being a binary-based framework, it relies on the Executable and Linkable File Format (ELF) [11] used by most UNIX systems for native objects. It recognizes shared object (.so) files as loadable modules. Shared objects define encapsulated/independent namespaces and are easily created from most relocatable objects (.o) compiled with position independent options (-fPIC for gcc). The implementation of a weave follows from the basic design. A string can be customized to use either POSIX threads (pthreads) or GNU's user-level threads, Pth.

Weaves's runtime environment requires extensive binary loading and linking capabilities to load and compose modules, and randomly manipulate reference-definition bindings. However, traditional binary loader services are not sufficient to support the demands of Weaves. For instance, typical loaders do not provide an explicit interface to connect a reference to an arbitrary definition. Hence, Weaves provides its own tool—Load and Let Link (LLL)—for dynamic loading and linking of modules [13]. The LLL loader maps given object files on disk to corresponding modules in memory. It can load multiple identical, but independent, modules from the same shared object file. LLL does not try to resolve external references at load-time, since any attempt to resolve them at this time would result in avoidable overhead. All cross-binding actions involving multiple modules are delegated to the linker, which dynamically composes a weave given an ordered set of modules. Additionally, the linker is invoked for runtime relocations of external references from a shared module. Finally, it provides an interface for explicitly binding a reference in a module to a definition in another.

5 Weaving PDE Solvers

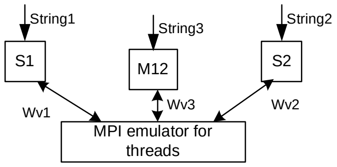


Fig. 3. Weaving unmodified agent-based codes (Wv implies a weave)

Weaves opens up various possibilities for implementing collaborating PDE solvers. However, due to space limitations, we discuss only the most radical approach that reuses unmodified solver and mediator codes from traditional agent-based implementations. Here, unmodified solver and mediator agent codes are compiled into object components. Additionally, a communication component is programmed to emulate dependable and efficient messages transfers between intra-process threads through in-memory data structures. The component's interfaces are identical to those of the distributed communication library used in the agent-based solution. At runtime, all solvers and mediators are loaded as distinct modules. The communication component is loaded as a single module. Every solver module is composed with the communication module into a solver weave. Every mediator module is composed with the communication module into a mediator weave. Parallel strings are then fired off at the main functions of solver and mediator modules. The solvers and mediators run as independent virtual machine abstractions unaware of Weaves. Fig. 3 diagrammatically illustrates a corresponding tapestry for a simple case assuming MPI for communication.

6 Performance Results

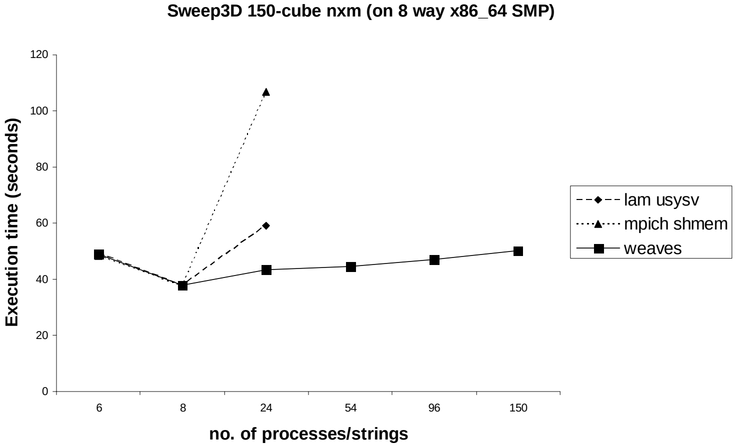


Fig. 4. Comparison of performance results of Weaved Sweep3D against LAM and MPICH based ones.

For preliminary performance results, we ran a Weaved version of Sweep3D [14] (an application for 3 dimensional discrete ordinates neutron transport) on an 8-way x86_64 SMP and compared results with traditional MPI-based implementations. The Weaved-setup was similar to Fig. 3. We developed a simple MPI emulator for in-memory data-exchange among threads. Multiple Sweep3D modules were composed with a single MPI module for communication. We used a 150-cube input file with a 2x3 split (6 processes/strings) as a start point and increased the split to 2x4, 4x6, 6x9, and so on upto 10x15 (150 processes/strings). The performance of the Weaved implementation matched that of LAM [12] and MPICH [15] as long as the number of processes/strings was lesser than the number of processors. Beyond that, the Weaved implementation performed much better thereby clearly demonstrating scalability (Fig.4). Both the MPI implementations (compiled and run with shared memory flags) crashed beyond 24 processes (4x6 split). The sharp performance degradation of LAM and MPICH are primarily due to systems-level shared memory schemes, which do not scale beyond the number of processors. The use of systems-level shared-memory is a direct consequence of reliance on the process-model. Weaves works around this problem by emulating processes within a single address-space.

7 Discussions

Weaved implementations of collaborating PDE solvers exploit lightweight intra-process threads and direct in-memory state sharing for scalability. Furthermore, they reuse legacy procedural codes for *PdeSolve* and *RelaxSoln* routines for transparency. Lastly, the bootstrap module requires minimal information about the internals of solver and mediator codes. Thus, Weaves can be used to flexibly compose a wide range of solver-mediator networks as well as applications from other domains. Preliminary performance tests show significant scalability over process-based implementations of Sweep3D. A prototype of the Weaves framework is available for download from http://blandings.cs.vt.edu/joy.

References

1. Varadarajan, S, Ramakrishnan, N.: Novel Runtime Systems Support for Adaptive Compositional Modeling in PSEs. Future Generation Computing Systems (Special Issue), 21(6) (June 2005), 878-895.
2. Chandy, K. M., Kesselman, C.: Compositional C++: Compositional Parallel Programming. Technical Report CaltechCSTR:1992.cs-tr-92-13, California Institute of Technology CA USA (2001).
3. Foster, I.: Compositional Parallel Programming Languages. ACM Transactions on Prog. Lang. and Sys., 18(4) (July 1996), 454-476.
4. Drashansky, T. T., Houstis, E. N., Ramakrishnan, N., Rice J. R.: Networked Agents for Scientific Computing. Communications of the ACM, 42(3) (March 1999), 48-54.
5. McFaddin, H. S., Rice, J. R.: Collaborating PDE Solvers. Applied Numerical Mathematics, 10 (1992), 279-295.
6. Rice, J. R.: An Agent-based Architecture for Solving Partial Differential Equations. SIAM News, 31(6) (August 1998).
7. Carriero, N., Gelernter, D.: Linda in Context. Communications of the ACM, 32(4). (April 1989) 444-458.
8. Sato, M.: OpenMP: Parallel Programming API for Shared Memory Multiprocessors and On-Chip Multiprocessors. In Proceedings of the 15th International Symposium on System Synthesis (ISSS ?02), Kyoto Japan (October 2-4 2002).
9. Common Component Architecture: http://www.cca-forum.org/
10. Mahmood, N., Deng, G., Browne, J. C.: Compositional Development of Parallel Programs. In Proceedings of the 16th Workshop on Langs. and Compilers for Parallel Computing (LCPC?03), College Station TX (2003).
11. Tools Interface Standards Committee: Executable and Linkable Format (ELF) Specification, (May 1995).
12. LAM MPI: http://www.lam-mpi.org/
13. Mukherjee, J., Varadarajan, S.: Weaves: a framework for reconfigurable programming. International Journal forParallel Programming, 33(2) (June 2005) 279-305.
14. Koch, K. R., Baker, R. S., Alcouffe, R. E.: 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).
15. Mpich: http://www-unix.mcs.anl.gov/mpi/mpich/