TINKERTOY PARALLEL PROGRAMMING: A CASE STUDY WITH ZOLTAN

KAREN D. DEVINE AND BRUCE HENDRICKSON
DISCRETE ALGORITHMS AND MATHEMATICS DEPARTMENT
SANDIA NATIONAL LABORATORIES
ALBUQUERQUE, NM 87185-1111
{KDDEVIN,BAHENDR}@SANDIA.GOV *

Abstract. As the need for complex parallel simulation software grows, better strategies for efficient and effective software development become important. We advocate a toolkit — or "tinkertoy" — approach to parallel application development. By providing efficient implementations of basic services commonly needed by applications, toolkits allow application developers to benefit from others' research, compare algorithms, and save time for their own development. Unlike large frameworks, toolkits provide these services with light-weight interfaces and little or no restriction on application data structures, making them easy to use in both new and existing applications. In this paper, we describe features of effective toolkit design, using the Zoltan parallel, dynamic data management toolkit as an example.

1. Introduction. Developing software for parallel scientific simulations is always a challenge. Parallel simulations require a wide range of capabilities, from meshing tools and data managers to solvers and visualization tools. Dynamic and/or adaptive simulations present an even greater challenge, as load redistribution, synchronization, and more complicated data structures must be managed. High parallel performance is always desired, requiring expertise by the software developer in efficient algorithm design and implementation. Development schedules are often tight. And parallel architectures change with each new generation of computers, requiring portability of codes and providing a "moving target" for performance optimization. In such an environment, what is the best approach to the development of complex adaptive software?

Several approaches to parallel software development exist, each with its own advantages and disadvantages. In the first approach, application developers could do all the software development themselves. This approach is attractive because it gives developers total control of the software. This control, however, comes at a severe price. The do-it-yourself approach is time consuming, as much effort is spent writing code that is often available elsewhere — reinventing the wheel, so to speak. Moreover, developers must spend much of that time writing code in areas outside their areas of expertise and interest, resulting in non-expert and, quite possibly, less efficient implementations of many parts of the software.

Software frameworks provide a second approach to software development. Frameworks have been successful for some parallel applications because they provide a wide range of services and data structures to specific classes of applications. SIERRA [15] and Overture [9] are examples of successful adaptive simulation frameworks. Such frameworks provide many capabilities in one package, which is a significant advantage to applications that need all the capabilities. However, frameworks typically are large and can have substantial overhead, which is a disadvantage to applications needing only a small subset of a framework's capabilities. Frameworks are also difficult to add to existing applications; instead, existing applications must be incorporated like new applications into the framework. To use a framework, application developers

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

must learn both its interfaces and data structures, which is often a time-consuming task. In addition, framework use makes application developers highly dependent on the framework's developers, perhaps causing an undesirable loss of control in terms of enhancements and schedules for the application developers.

As an alternative, we advocate a toolkit — or "tinkertoy" — approach to software development. The original tinkertoy, made for children by Hasbro, is a set of simple wooden pieces that can be interconnected in different ways to make surprisingly complex structures and machines. Similarly, in a tinkertoy approach to software, applications are constructed of small, simple software parts with flexible, easy-to-use interfaces. Toolkits contain basic services commonly needed by applications. Application developers can then put together these services to create a larger application. For example, an application could be constructed from an adaptive meshing toolkit (e.g., Pyramid [32], AOMD [36, 37]), a dynamic load-balancing toolkit (e.g., Zoltan [14, 13, 12], DRAMA [27], ParMETIS [23]), a linear and non-linear solver library (e.g., Trilinos [19, 20], Aztec [22], PETSc [3, 4]), and some visualization tools (e.g., VTK [39]). Application developers, then, can concentrate on the simulation details in which they are most interested (e.g., physics or engineering).

This toolkit approach has a number of advantages. Because they typically provide a smaller number of basic services, toolkits are less cumbersome than frameworks; application developers can select and use only the functionality they need. Because toolkits generally use library interfaces and private data structures, developers can incorporate them easily into both new and existing applications. Application developers need to learn only the toolkit interfaces, rather than all its internal data structures, so start-up time is shorter than with framework use. Well-designed toolkits can, like tinkertoys, be easily hooked together to build larger and more complex functionality, beyond the scope of any single library. And since most toolkits are developed by experts in the toolkits' capabilities, application developers benefit in terms of both development time and algorithmic efficiency by using toolkits.

Toolkits do, however, have some of the same trust and dependence issues as frameworks, although to a lesser degree. There are often competing toolkits providing similar functionality, so an application developer can switch if the need arises. Also, the simple interfaces associated with well-designed toolkits facilitate replacement if necessary. Toolkit users are dependent upon toolkit developers to provide correct algorithms and customer support. Open-source distribution used by many toolkits can increase reliability and trust by allowing users to inspect the implementations and by providing a broad testing community for the software. Toolkits also have some memory and performance overhead due to separation of toolkit data structures from the applications, but with careful design, these costs can be kept acceptably low.

There are, of course, hybrids of all these approaches. The Common Component Architecture (CCA), for example, provides interfaces that allow plug-and-play interoperability of components, in line with the toolkit philosophy [2]. The components, however, are launched within a framework (e.g., Ccaffeine [1]) that manages the components' operation. For this paper, however, we will focus on the straightforward toolkit approach to parallel computing.

Is it really possible to build complex applications out of tinkertoys? It is widely accepted that linear solvers can be encapsulated as libraries, but what about the needs of complex, adaptive parallel applications like adaptive mesh calculations or particle simulations? It is often presumed that these kinds of dynamic applications require such intricate control over data structures that toolkits cannot easily be applied.

One goal of this paper is to give affirmative answers to these questions. We believe that even complicated, adaptive computations can be efficiently and effectively constructed from simple tools. As a second goal, the paper describes our attempt to instantiate this vision through Zoltan — a toolkit for adaptive parallel computation built with the tinkertoy philosophy. Finally, through Zoltan we have been exploring the possibilities and limits of tinkertoys. Specifically, what types of functionality can be delivered through application-independent toolkits, and what can be provided only by applications or frameworks? This paper reports on our current understanding of this important issue.

2. Zoltan Overview. The Zoltan toolkit is a collection of data management services for parallel unstructured, adaptive and dynamic applications, available as open-source software from http://www.cs.sandia.gov/Zoltan. It is designed to simplify the load balancing, data movement, unstructured communication, and memory usage difficulties that arise in dynamic applications such as adaptive finite element methods, particle methods, and multiphysics simulations. Zoltan's data-structure neutral design allows it to be used by a wide range of applications without imposing restrictions on application data structures. Its object-based interface provides a simple and inexpensive way for application developers to use the library and researchers to make new capabilities available under a common interface.

As we detail in the subsequent sections of this paper, Zoltan provides tools that help application developers without imposing strict frameworks on them. For example, it includes parallel partitioning algorithms and data migration tools that help redistribute data to reflect, for example, changing processor workloads resulting from creation of elements in adaptive finite element methods. Zoltan also includes distributed data directories, dynamic memory debugging tools, and unstructured communication services that enable applications to perform complicated communication using only a few simple primitives. Zoltan is used in a variety of applications, including contact detection and crash simulations [10, 24], adaptive finite element methods [15, 8, 25], parallel circuit simulations [21], multiphysics simulations [38], and linear solvers and preconditioners [19, 20].

3. The Promise and Limitations of Toolkits. The success of tinkertoy computing depends upon a number of software design features:

- Functionality: toolkits must solve problems that appear in multiple applications.
- Portability: toolkits must be portable across multiple parallel platforms.
- Interfaces: toolkits' software interfaces should be easy to use.
- Added value: toolkits should give application developers greater performance and flexibility.
- Low overhead: the overhead due to toolkit use must be small, both in memory and in runtime.
- Support: toolkit developers should help application developers use their toolkits effectively.

Each of these issues provides challenges for advocates of tinkertoy parallel computing. We discuss these challenges below in the context of support for adaptive parallel computations, and explain how we have tried to address them within Zoltan.

3.1. Functionality. Selection of services provided by a toolkit is a critical design step for toolkit designers. Toolkits should include services commonly needed by applications. Services should remain independent of each other as much as possible

so that application developers can select and use only the tools that they want. In addition, services within a toolkit should be related to and complement each other. Toolkit developers should fight the urge to incorporate every possible service, so that toolkits do not become too large and difficult to use (and, indeed, start resembling frameworks).

As an example, efficient parallel implementation of adaptive applications requires dynamic load balancing to redistribute work to processors after adaptive refinement occurs. Dynamic load balancing involves both the computation of a new partitioning of data and workload, and movement of data to new processors. Moreover, dynamic data redistribution creates new needs for applications as they dynamically delete and insert data in their data structures, re-locate needed off-processor data, and build new communication patterns. The Zoltan toolkit includes functionality to address many of these related needs.

Zoltan's main utility is a suite of dynamic load-balancing algorithms that compute new distributions of data to processors. Since dynamic load balancers must run side-by-side with applications, Zoltan is implemented in parallel and is scalable in both execution and memory usage. For load balancing, it takes an existing distributed partition as input and computes a description of the new partition in terms of objects to be transferred between processors. Many of the partitioning algorithms are incremental; that is, small changes in processor work loads result in only small changes in the resulting decompositions. Zoltan's partitioning algorithms support non-uniform partition sizes and unequal numbers of partitions and processors. Additional utilities that compute which processors' partitions intersect a given point or region in space are provided for geometric partitioning methods; these utilities are key kernels of parallel contact detection simulations [10, 24].

After obtaining a map of a new decomposition, applications must move data from their old processors to their new processors. This data migration requires deletions and insertions from the application data structures, along with communication between the processors. A general-purpose toolkit like Zoltan can do little to help with the manipulation of application-specific data structures. However, because Zoltan has knowledge of both the old and new partitions, it can easily communicate object data among processors. In fact, using user-supplied functions to pack and unpack data into communication buffers, Zoltan's data migration tools can perform all communication necessary to send data to their new location.

Zoltan's distributed data directories (based on the rendezvous algorithm of Pinar and Hendrickson [34]) provide additional functionality related to dynamic data redistribution. After repartitioning, for example, a processor may need to rebuild ghost cells and lists of objects to be communicated; it may know which objects it needs, but may not know where they are located. Using Zoltan to locate this off-processor data, processors register data along with their processor numbers in a directory that is distributed evenly across processors in a predictable way (e.g., a linear decomposition of the data or a hashing of data to processors). Then, other processors obtain the processor number of a given object by sending a request for the information to the processor holding the directory entry. Thus, the total memory usage is linear in the amount of data and communication cost for look-ups is constant. Moreover, since the directory is distributed, no communication bottlenecks develop (as they would for a directory located completely on one processor).

The Zoltan toolkit provides further capability to dynamic applications with complicated and/or changing communication patterns. For example, multi-physics sim-

ulations and crash simulations may require complicated communication patterns to transfer data between decompositions for different simulation phases. To simplify this communication, Zoltan provides an unstructured communication package that generates a communication "plan" with information about sends and receives for a given processor. The plan may be used and reused throughout the application, or it may be destroyed and rebuilt when communication patterns change. Simple communication primitives in the toolkit insulate users from details of sends and receives.

Similarly, memory usage in dynamic applications can change throughout the simulation. After repartitioning, for example, new memory is needed for imported data and exported data's memory is freed. Memory leaks are common in developing software. While there are many software development tools that enable users to track memory bugs [35, 41], these tools are often not available on state-of-the-art parallel computing platforms. Thus, Zoltan provides basic in-application memory-debugging tools that are simple wrappers around memory allocation routines. The wrappers record information (e.g., line number, file name) about memory operations, allowing developers to track memory leaks and print memory-usage statistics.

While these related tools operate well together, an important feature of Zoltan's toolkit design is separation between tools. Application developers can use only the tools they want; for example, they can use Zoltan to compute decompositions but perform all data migration themselves. They can build Zoltan distributed data directories that are completely independent of load balancing. They can use Zoltan's unstructured communication tools within statically balanced applications. Or they can use Zoltan to perform all the data management tasks associated with load balancing. In this way, Zoltan provides full service for dynamic partitioning, while allowing developers the flexibility to use Zoltan's tools in a variety of ways — even ways not orignally envisioned by Zoltan's designers.

3.2. Portability. A toolkit is useful to a broad community only if it is portable across many platforms. In addition to allowing toolkit use on many current architectures, portability allows the toolkit to be used across generations of machines. Developers are more apt to be willing to use toolkits if they know that the software will continue to work as machines are upgraded or replaced.

To ensure portability, toolkits must rely on standards as much as possible. They should use only standard language features, to prevent compilation difficulties. Since many cutting-edge language features are not supported by older compilers, toolkits should include code that is as simple as functionally possible. Toolkit dependence on other libraries should be kept to a minimum; few things are more frustrating than trying to build a toolkit only to discover that many other libraries must be located, purchased, downloaded, and/or installed first. Necessary dependencies should take advantage of "standard" libraries (MPI [28, 29], OpenGL [40], BLAS [5], etc.) as much as possible.

The Zoltan toolkit is implemented in ANSI C, with an optional Fortran90 interface available. It uses MPI for all communication; it can use any version of MPI and has been tested using MPICH, LAM, and system-specific MPI libraries for IBM, DEC, and Intel architectures. To use Zoltan's graph partitioners, applications must link with Zoltan and either ParMETIS [23] or Jostle [44, 43]; a compatible version of ParMETIS is distributed with Zoltan to simplify building and configuring both libraries. If graph partitioning is not needed, however, dependence on ParMETIS and/or Jostle can be excluded from Zoltan.

3.3. Easy-to-Use Interfaces. Toolkits' capabilities should be easily accessible by many different applications. To accomplish this goal, several features are needed. There should be separation between the application and toolkit data structures, so that toolkit use is not restricted to a particular application. Toolkits should have simple interfaces that do not require extensive programming by the application developer. And toolkits should fit easily into both existing and new applications, allowing application developers to retrofit and update their existing codes.

Separation between application and toolkit data structures is achieved through data-structure neutral toolkit design. In a data-structure neutral design, details of the toolkit data structures are hidden from the application and vice versa. Thus, the toolkit does not impose data structures upon an application as frameworks do. This separation can be achieved in several ways. Some toolkits (e.g., ParMETIS [23], Jostle [43]) require the application to build specific data structures (e.g., graphs) for the toolkit to use. While this approach is acceptable, it has the drawback that data structure changes in the toolkit require changes to both the toolkit interface and application. It also burdens the application programmer with the task of creating a complex data structure, and it may incur a significant memory overhead if the library creates yet another copy of the data structure. Other toolkits (e.g., Trilinos [19, 20]) provide an object interface (e.g., a matrix) and methods for performing operations on the objects (e.g., transposition). This interface allows greater code hiding than the previous approach.

The Zoltan toolkit uses a callback function approach, in which Zoltan calls user-supplied functions to obtain needed application data. The functions answer questions like, "How many data items are owned by this processor?" and "What are the geometric coordinates of the data items on this processor?" The application developer must provide simple functions that answer these queries. Then Zoltan calls the functions to build appropriate data structures for the particular tool requested. This approach has several advantages for both ease of use and ease of maintenance. First, once application developers implement the callback functions, they can access all technology within Zoltan without additional construction of data structures; as new capabilities are added to the toolkit, users can access them with little effort. Second, by not requiring users to build data structures for them, Zoltan developers can use the most efficient data structures for their algorithms and can improve them without impacting the applications. Third, the user interface remains unchanged regardless of any internal changes in Zoltan, allowing users to upgrade versions of Zoltan with no change to their applications. Finally, at no time does the application developer have to build (or debug!) complicated data structures for use within Zoltan.

Toolkit interfaces should be simple to understand and utilize for users with various levels of interest and expertise. Only a small set of functions should be needed to invoke the toolkit's basic capabilities; additional functions can be provided to support more advanced features. Parameters can be used to control toolkit functionality, but reasonable default values should be provided. With this layered approach, beginning users can benefit from the basic toolkit functionality, while more advanced and interested users can experiment with a broader range of options. Zoltan uses only a small set of callback functions and makes them easy to write by requesting only information that is, in general, easily accessible to applications. For the most basic partitioning algorithms, Zoltan requires only four callback functions; these functions return the number of objects owned by a processor, a list of weights and names for owned data, the dimensionality of the problem, and coordinates of a given owned object. More so-

phisticated graph-based partitioning and matrix-ordering algorithms require only two additional callback functions, returning the number of edges per data object and edge lists for data objects. All algorithms have parameters that can alter the algorithms' performance and results; default values are set to reflect the most common scenarios for algorithm use.

Toolkits should be easy to use in both new applications and existing ones. When toolkits allow individual tools to be used independently, application developers can incorporate the toolkits incrementally into their applications. For example, an application developer may replace a load-balancing scheme in an existing dynamic application with a partitioning algorithm from Zoltan, but continue using the data migration code previously written in the application.

3.4. Added value. Since toolkits are most often implemented by researchers in the areas addressed by the toolkit, they can provide high performance implementations of state-of-the-art algorithms. Thus, by using toolkits, application developers can focus on their particular areas of interest, rather than concern themselves with every detail of the parallel simulation. Instead of trying to understand the state of the art in every field, they can concentrate on research in their own field. Likewise, they can provide valuable user feedback to toolkit developers, creating a synergy that benefits both application developers and toolkit researchers.

Toolkits can also add value to applications by providing a number of different algorithms whose effectiveness can be compared within an application. For example, there is no single partitioning strategy that is effective for all parallel computations. Some applications require partitions based upon only the workloads and geometry of the problem; others benefit from explicit consideration of dependencies between objects. Some applications require the highest quality partitions possible, regardless of the cost to generate them; others can sacrifice some quality as long as new partitions can be generated quickly. Most importantly, an application developer may not know in advance which strategy works best in his application. By providing a collection of algorithms and a convenient way to compare them, toolkits can significantly improve application performance with little additional effort required by application developers. By facilitating easy algorithmic comparisons, toolkits also help advance algorithmic research.

In the Zoltan library, we have included a suite of parallel partitioning algorithms. Three classes of algorithms are provided: geometric bisection, space-filling curves, and graph partitioning. Within each class, several different algorithms are implemented. Geometric algorithms include Recursive Coordinate Bisection [7] and Recursive Inertial Bisection [42]. Space-filling curve partitions are generated via a binned Hilbert Space-Filling Curve algorithm [18, 6], Octree partitioning [26, 16, 11], or a Refinement Tree Partitioning algorithm design especially for adaptive mesh refinement applications [30, 31]. Graph partitioning is provided through easy-to-use interfaces to ParMETIS [23] and Jostle [43]. Once users write the callback functions for each class, switching between classes and methods requires only a single parameter change with the new algorithm name. In this way, developers can easily compare algorithms within their applications to find the strategy that works best for them.

3.5. Low Overhead. The performance obtained using a toolkit will almost never be as high as the performance of an equivalent algorithm embedded directly within an application. Data separation and general application interfaces require additional memory use and computation time for creating toolkit data structures.

However, with careful design, this overhead can be kept acceptably low; the additional cost can be tolerated when the application benefits from toolkit functionality.

The amount of overhead that can be tolerated depends, of course, on the application's use of the toolkit. Greater overhead can be tolerated if the tools are invoked infrequently. Since we expect dynamic load balancing and its related functionality to be executed frequently during a simulation, however, low overhead is important in Zoltan. Multiple versions of many callback functions (e.g., list-based functions that return arrays of data versus iterator functions that return one data item at a time) are provided to applications, allowing application developers to pick the interface most suitable for their data structures. Callbacks allow Zoltan's algorithms to directly build the data structures they need; no intermediate data structure is used. The callbacks also allow the algorithms to obtain only the data they need; for example, geometric algorithms do not require graph information and, thus, do not obtain that information from the application.

Table 3.1 provides evidence that Zoltan's callback function interface adds only a small overhead to a simulation. Experiments using several Zoltan partitioning algorithms were run using the mesh-based driver application distributed with Zoltan. Two types of overhead are measured: the overhead associated with calling the user-specified callback functions (e.g., to obtain coordinate or graph information), and the overhead associated with building the data structures needed for load balancing. While the callback function overhead would not exist if the load balancing algorithms were embedded directly in the applications, most of the overhead needed to build data structures for load balancing would still be incurred. Cases where the application could use exactly the same data structures for computation and partitioning are rare; for example, few applications' computations use as their native data structure the compressed, distributed graph structure commonly used in graph partitioning. Thus, the overhead of building the data structures cannot be completely disregarded for embedded partitioning algorithms.

In the experiments, a three-dimensional, unstructured finite element mesh with about one million elements was randomly distributed on 16 processors. New decompositions were computed using three of Zoltan's partitioning algorithms: Recursive Coordinate Bisection (RCB), Hilbert Space-Filling Curves (HSFC), and a graph-based method (ParMETIS PartKWay). The time reported in the table is just that associated with the partitioning itself. Once the partitioning has been computed, additional time will be required to migrate data and to update the data structures on each processor. In our experience, these latter operations are several times as costly as the partitioning itself, yet they are independent of the use of a toolkit. This further reduces the significance of the overhead times in the table.

The time spent in callback functions, time spent building data structures (including the callback-function time), and total partitioning time (including both callbacks and data structure construction) were measured. The geometric algorithms RCB and HSFC have similar amounts of overhead, since they both use only geometric information about data to be partitioned. Because the graph-based partitioner requires more application data (e.g., the edge lists for each graph vertex) and more complicated data structures (i.e., a distributed graph), its overheads are somewhat higher. Still, for all algorithms, the callback overhead is less than 9% of the total partitioning time. Similarly, the time required to build data structures is less than 15% of the partitioning time for all three algorithms. And as discussed above, this time is likely to be required even by embedded partitioning implementations.

	RCB	HSFC	ParMETIS PartKWay
Time for callback functions (secs)	0.038	0.037	0.096
Percentage of total partitioning time in callbacks	4.4%	8.7%	3.6%
Time to build data structures (secs)	0.066	0.059	0.288
Percentage of total partitioning time in build	7.7%	14.0%	10.7%
Total partitioning time (secs)	0.865	0.423	2.674

TABLE 3.1

Overheads incurred when using Zoltan in a 3D mesh-based application. Callback overhead measure only time spent in callback functions; this time is unique to Zoltan. Build overhead includes time spent contructing load-balancing data structures; this time would most likely be incurred by applications using embedded load balancing.

As the table indicates, geometric partitioners are often faster than graph based partitioners. However, there may be compensating differences in partition quality. The goal of this table is merely to quantify the overhead associated with the use of Zoltan, and not to compare the merits of different partitioning strategies.

We must emphasize that the overhead incurred in using Zoltan can vary depending upon the application and its implementation of the callback functions. If the application gives Zoltan expensive implementations of callback functions, the callback-function overhead will, of course, increase.

3.6. Support. Convincing application developers to use someone else's code is often a difficult proposition. Their reluctance is understandable, as they cannot always ascertain the quality of outside code, and they have justifiable worries about long-term maintenance and development. In this regard, toolkits are more attractive than frameworks, because the commitment by an application developer is less with the former than the latter. But although reduced in significance, support issues are still important. Toolkit designers can help ease these concerns in several ways. First, open-source distribution of toolkit software allows users to study and experiment with the software. A number of open-source licenses are available with varying levels of protection for toolkit developers [33]. Zoltan is distributed using the GNU Lesser General Public License [17], which allows free use of Zoltan software and guarantees that redistributions of Zoltan are also free. Second, toolkit developers must provide documentation for their software, with User's Guides describing functionality and options. User's guides should provide detailed descriptions of capabilities, options, interfaces, and configurations, and should include usage examples when appropriate. Zoltan's hyperlinked, web-based User's Guide [14] has proved to be useful both to users and Zoltan developers. Third, simple codes using the toolkit can be distributed with the toolkit. While allowing users to verify their toolkit installation, these codes serve as examples that application developers can study in learning to use the toolkit. Most toolkit developers have such programs available for their own testing; including the examples with the toolkit distribution takes little additional effort. For example, code, instructions, and sample inputs and outputs for the Zoltan regression test programs are included in the Zoltan distribution. And finally, the promise of customer support from toolkit developers encourages application developers to at least give toolkits a try.

4. Future Work. The difficulty of software development continues to be a principle impediment to the adoption of high-performance computing. It is our belief that

well executed toolkits will play a growing role in addressing this problem. Although they provide less support than full-fledged frameworks, toolkits have the advantage of greater flexibility and incremental adoption. But toolkits can never address the detailed manipulation of application-specific data structures. Thus, we anticipate a continuing need for frameworks, and hope for a decline in the heroic but inefficient practice of all-in-one application development. We look forward to a day when the community has built a diverse set of toolkits with clean interfaces, and application developers can mix and match them like tinkertoys to quickly build complex yet high-performing applications.

We feel that Zoltan offers an attractive model for toolkit development. It provides a diverse set of related services via a simple interface and low overhead in both memory and runtime. We continue to add functionality to Zoltan including support for heterogeneous parallel architectures and alternative models of load balancing. However, in doing so we are constantly facing choices about the appropriate breadth of functionality of the toolkit and about the complexity of the interface. In our experience, good toolkit development requires both discipline and humility. We want to avoid adding functionality capriciously, and to focus on the areas in which we are best able to provide novel support to application developers.

We hope that the proven success of Zoltan will attract attention to the promise of tinkertoy parallel programing. More specifically, we hope to persuade others to build additional toolkits with complimentary functionality, and to convince application developers of the promise this approach has in the development of more, and more complex, simulation codes.

Acknowledgments. The ideas and opinions in this paper have grown out of our many years of experience developing supporting infrastructure for parallel computations. Along the way we have learned from many colleagues including Steve Plimpton, Mike Heroux, Erik Boman, John Shadid, Bob Heaphy, Courtenay Vaughan, and Bill Mitchell.

REFERENCES

[1] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, *The CCA core specification in a distributed memory spmd framework*, Concurrency Computat., 14 (2002), pp. 1–23.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, *Toward a common component architecture for high-performance scientific computing*, in Proceedings of the 1999 Conference on High Performance Distributed Computing, Redondo Beach, CA, August 1999.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc users manual*, Tech. Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2002.

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhauser Press, 1997, pp. 163–202.

[5] Basic Linear Algebra Subprograms Technical (BLAST) Forum, University of Tennessee, *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*, Knoxville, Tennessee, 2001. `http://www.netlib.org/blas/blast-forum/`.

[6] A. C. Bauer, *Efficient Solution Procedures for Adaptive Finite Element Methods — Applications to Elliptic Problems*, PhD thesis, State University of New York at Buffalo, 2002.

[7] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. Computers, 36 (1987), pp. 570–580.

[8] E. A. Boucheron, K. H. Brown, K. G. Budge, S. P. Burns, D. E. Carroll, S. K. Carroll, M. A. Christon, R. R. Drake, C. G. Garasi, T. A. Haill, J. S. Peery, S. V.

Petney, J. Robbins, A. C. Robinson, R. Summers, T. E. Voth, and M. K. Wong, *ALEGRA: User Input and Physics Descriptions Version 4.2*, Sandia National Laboratories, Albuquerque, NM, 2002. Tech. Report SAND2002-2775.

[9] D. L. Brown, G. S. Chesshire, W. D. Henshaw, and D. J. Quinlan, *OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments*, in Eighth SIAM Conf. on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997, SIAM.

[10] K. H. Brown, M. W. Glass, A. S. Gullerud, M. W. Heinstein, R. E. Jones, and T. E. Voth, *ACME Algorithms for Contact in a Multiphysics Environment, API Version 1.3*, Sandia National Laboratories, Albuquerque, NM, 2003. Tech. Report SAND2003-1470.

[11] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, *Dynamic octree load balancing using space-filling curves*, Tech. Report CS-03-01, Williams College Department of Computer Science, 2003.

[12] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4 (2002), pp. 90–97.

[13] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan, *Zoltan: A Dynamic Load Balancing Library for Parallel Applications; Developer's Guide*, Sandia National Laboratories, Albuquerque, NM, 1999. Tech. Report SAND99-1376 `http://www.cs.sandia.gov/Zoltan/dev\_html/dev.html`.

[14] ———, *Zoltan: A Dynamic Load Balancing Library for Parallel Applications; User's Guide*, Sandia National Laboratories, Albuquerque, NM, 1999. Tech. Report SAND99-1377 `http://www.cs.sandia.gov/Zoltan/ug\_html/ug.html`.

[15] H. C. Edwards, *SIERRA framework version 3: Core services theory and design*, Tech. Report SAND2002-3616, Sandia National Laboratories, Albuquerque, NM, 2002.

[16] L. G. Gervasio, *Octree load balancing techniques for the dynamic load balancing library*, master's thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 1998.

[17] *GNU lesser general public license*. `http://www.gnu.org/copyleft/lesser.html`.

[18] R. Heaphy, *Load balancing contact deformation problems using the Hilbert space filling curve*. In preparation, 2003.

[19] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, *An overview of Trilinos*, Tech. Report SAND2003-2927, Sandia National Laboratories, Albuquerque, NM, 2003.

[20] M. A. Heroux and J. M. Willenbring, *Trilinos Users Guide*, Sandia National Laboratories, Albuquerque, NM, 2003. Tech. Report SAND2003-2952.

[21] S. A. Hutchinson, E. R. Keiter, R. J. Hoekstra, L. J. Waters, T. V. Russo, E. L. Rankin, and S. D. Wix, *Xyce Parallel Electronic Simulator User's Guide, Version 1.0*, Sandia National Laboratories, Albuquerque, NM, 2002. Tech. Report SAND2002-3790.

[22] S. A. Hutchinson, J. N. Shadid, and R. S. Tuminaro, *Aztec user's guide: Version 1.0*, Tech. Report SAND95-1559, Sandia National Laboratories, Albuquerque, NM, 1995.

[23] G. Karypis, K. Schloegel, and V. Kumar, *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 3.1*, University of Minnesota, Minneapolis, 2003.

[24] J. R. Koteras and A. S. Gullerud, *Presto User's Guide: Version 1.05*, Sandia National Laboratories, Albuquerque, NM, 2003. Tech. Report SAND2003-1089.

[25] S. J. P. Lawrence C. Musson and R. C. Schmidt, *MEMS fabrication modeling with ChISELS: A massively parallel 3D level-set based feature scaled modeler*, in Proc. 2003 Nanotechnology Conference and Trade Show, vol. 3, San Francisco, CA, February 2003. Also Sandia National Laboratories Tech. Rep. SAND2002-3994C.

[26] R. M. Loy, *Adaptive Local Refinement with Octree Load-Balancing for the Parallel Solution of Three-Dimensional Conservation Laws*, PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 1998.

[27] B. Maerten, D. Roose, A. Basermann, J. Fingberg, and G. Lonsdale, *DRAMA: A library for parallel dynamic load balancing of finite element applications*, in Proc. Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, TX, 1999.

[28] Message Passing Interface Forum, University of Tennessee, *MPI: A Message-Passing Interface Standard*, Knoxville, Tennessee, 1995. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

[29] ———, *MPI-2: Extensions to the Message-Passing Interface*, Knoxville, Tennessee, first ed., 1997. `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`.

[30] W. F. Mitchell, *Refinement tree based partitioning for adaptive grids*, in Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, 1995, pp. 587–592.

[31] ———, *The refinement-tree partition for parallel solution of partial differential equations*, NIST Journal of Research, 103 (1998), pp. 405–414.

[32] C. D. Norton, J. Z. Lou, and T. Cwik, *Status and directions for the PYRAMID parallel unstructured AMR library*, in Eighth Intl. Workshop on Solving Irregularly Structured Problems in Parallel (15th IPDPS), San Francisco, CA, 2001, IPDPS.

[33] *The approved licenses*. Links to numerous open source licenses at `http://www.opensource.org/licenses/index.php`.

[34] A. Pinar and B. Hendrickson, *Communication support for adaptive computation*, in Proc. 10th SIAM Conf. Parallel Processing for Scientific Computing, Portsmouth, VA, March 2001.

[35] *IBM Rational PurifyPlus*. `http://www.rational.com`.

[36] J.-F. Remacle, O. Klaas, J. E. Flahery, and M. S. Shephard, *Parallel algorithm oriented mesh database*, Eng. Comput., 18 (2002), pp. 274–284.

[37] J.-F. Remacle and M. S. Shephard, *An algorithm oriented mesh database*, Int. J. Numer. Meth. Engng., 58 (2003), pp. 349–374.

[38] A. Salinger, K. Devine, G. Hennigan, H. Moffat, S. Hutchinson, and J. Shadid, *MPSalsa A Finite Element Computer Program for Reacting Flow Problems, Part 2 – User's Guide*, Sandia National Laboratories, Albuquerque, NM, 1996. Tech. Report SAND96-2331.

[39] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, 3rd Edition*, Kitware, Inc., 2003.

[40] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification (Version 1.5)*, Silicon Graphics, Inc., 2003. `http://www.opengl.org/developers/documentation/specs.html`.

[41] J. Seward and N. Nethercote, *Valgrind, stable release 20031012*, 2003. `http://devel-home.kde.org/~sewardj/`.

[42] V. E. Taylor and B. Nour-Omid, *A study of the factorization fill-in for a parallel implementation of the finite element method*, Int. J. Numer. Meth. Engng., 37 (1994), pp. 3809–3823.

[43] C. Walshaw, *The Parallel JOSTLE Library User's Guide, Version 3.0*, University of Greenwich, London, UK, 2002.

[44] C. Walshaw, M. Cross, and M. G. Everett, *Parallel dynamic graph partitioning for adaptive unstructured meshes*, J. Parallel Distrib. Comput., 47 (1997), pp. 102–108.