

STAPL: The Standard Template Adaptive Parallel Library ^{*}

Nancy M. Amato Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science, Texas A&M University
{amato,rwenger}@cs.tamu.edu

1 Introduction

In sequential computing, standardized libraries are essential for simplifying program development. They allow routine problems to be handled economically and increase the scale of problems that can be addressed. They do so by providing routines, types, and frameworks for common operations (in general programming and in specific fields) so programmers can concentrate on higher level problems. Due to the added complexity of parallel programming, the potential impact of libraries could be even more profound than for sequential computing. Good parallel libraries can insulate less experienced users from much of the complexity of parallelism by providing routines that are easily interchangeable with their sequential counterparts, while allowing more sophisticated users sufficient control to achieve higher performance gains. There have been some successful specialized libraries of elementary parallel algorithms [3, 4, 15]. However, designing general, efficient, and portable parallel libraries is a challenge that has not yet been met, mainly due to the difficulty of managing concurrency and the diversity of parallel and distributed architectures.

STAPL (the Standard Template Adaptive Parallel Library) [2, 12] is a framework for parallel processing consisting of C++ library code, design rules for extending that code, and optimization tools. Its goal is to allow the user to work at a high level of abstraction and hide many details specific to parallel programming, while allowing a high degree of portability and performance adaptability across computer systems. STAPL's core is a library of ISO Standard C++ components with interfaces similar to the (sequential) ISO C++ standard library. By providing its facilities within an ISO standard mainstream language, STAPL gives developers of parallel software the benefits of the huge investment in mainstream tools, compilers, and libraries. Conversely, STAPL makes parallel techniques accessible to mainstream programmers by providing a degree of automatic parallelization and by providing its facilities within mainstream development environments and on mainstream hardware. As the price of parallel systems drops within the reach of even modest (in terms of size and programming skills) computing establishments, this could have an important long-term impact.

STAPL offers the parallel system programmer a shared object view of the data space. The objects are distributed across the memory hierarchy which can be shared and/or distributed address spaces. Internal STAPL mechanisms assure an automatic translation from one space to another, presenting to the less experienced user a flat, UMA like, unified data space. For more experienced users the local/remote distinction of accesses can be exposed and performance enhanced. STAPL supports the SPMD model of parallelism with essentially the same consistency model as that of OpenMP. To exploit large systems like the DOE machines and IBM's BlueGene/L, STAPL allows for (recursive) nested parallelism (as in NESL [4]). STAPL's ARMI communication library [14] and run-time system are compatible with both message passing (e.g., MPI) and shared memory systems (e.g., OpenMP). Because performance of parallel algorithms is sensitive to system architecture (latency, topology, etc.) and to application data (data type, distribution, density, etc.), STAPL is designed to continually adapt to the system and the data at all levels – from selecting the most appropriate algorithmic implementation [17] to balancing communication granularity with latency by self-tuning the message aggregation factor, etc.

An important goal of STAPL is to provide an easily extensible base and building blocks for the development of domain specific libraries by specialists in the domain who are not necessarily expert parallel programmers. To ensure its applicability and usability, STAPL is being refined and extended through the concurrent development of domain specific libraries for particle transport computations (computational physics) and protein folding simulation (computational biology). Both applications, like many problems in the physical and life sciences, make heavy use of dynamic linked data structures (e.g., graphs). Hence, in addition to the usual vector-based operations, STAPL provides built-in support for irregular data structures such as lists, sparse matrices, and graphs.

^{*}This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, ACI-0326350, and by the DOE.

	STAPL	AVTL	CHARM++	CHAOS++	CILK	NESL	POOMA	PSTL	SPLIT-C
Paradigm	S/M	S	M	S	S/M	S/M	S	S	S
Architecture	S/D	D	S/D	D	S/D	S/D	S/D	S/D	S/D
Nested Par.	yes	no	no	no	yes	yes	no	no	yes
Adaptive	yes	no	no	no	no	no	no	no	no
Generic	yes	yes	yes	yes	no	yes	yes	yes	no
Irregular	yes	no	yes(limited)	yes	yes	yes	no	yes	yes
Data decomp	auto/user	auto	user	auto/user	user	user	user	auto/user	user
Data map	auto/user	auto	auto	auto/user	auto	auto	user	auto/user	auto
Scheduling	block, dyn, partial self-sched	user MPI-based	prioritized execution	based on data decomposition	work stealing	work and depth model	pthread scheduling	Tulip RTS	user
Overlap comm/comp	yes	no	yes	no	no	no	no	no	yes

Table 1: Related Work. For the paradigm, S and M denote SPMD and MIMD, resp. For the architecture, S and D denote shared memory and distributed, respectively.

2 Related Work

There is a relatively large body of work that has similar goals to STAPL. Table 1 gives an overview of selected projects. We briefly comment on some and attempt to compare them with STAPL. For further work in this area see [18]. The Parallel Standard Template Library (PSTL) [11, 10] has similar goals to STAPL; it uses parallel iterators in place of STL iterators and provides some parallel algorithms and containers. NESL [4], CILK [7] and SPLIT-C [6] provide the ability to exploit nested parallelism through their language support (all three are extended programming languages with NESL providing a library of algorithms). However, only STAPL is intended to automatically generate recursive parallelization without user intervention. Most listed packages (STAPL, Amelia [16], CHAOS++ [5] and to a certain extent CHARM++ [1]) use a C++ template mechanism and assure good code reusability. STAPL emphasizes irregular data structures like trees, lists, and graphs, providing parallel operations on such structures. Charm++ and CHAOS++ also provide support for irregular applications through their chore objects and inspector/executor, respectively. POOMA [13], PSTL [11, 10], and STAPL borrow from the STL philosophy, i.e., containers, iterators, and algorithms. STAPL’s RMI-based communication infrastructure is extremely flexible, supporting overlapping of communication/computation, and the simultaneous use of both message passing and shared memory (MPI and OpenMP). Charm++ provides similar support through message driven execution and a dynamic object creation mechanism. The split phase assignment ($:=$) in Split-C also allows for overlapping communication with computation.

STAPL is distinguished by its emphasis on dynamic and aggressive adaptive capabilities which are critical for STAPL’s support of irregular applications that rely on dynamic linked data structures. For example, STAPL provides support for both automatic and user specified policies for scheduling, data decomposition and data dependence enforcement and it is unique in its goal to automatically select the best algorithm from its library options by analyzing the data, architecture and current run-time conditions.

3 STAPL Architecture

The STAPL software infrastructure consists of platform independent and platform dependent components. These are revealed to the programmer at an appropriate level of detail through a hierarchy of abstract interfaces.

STAPL components. The platform independent STAPL components include the core parallel library and an abstract interface to the communication library and run-time system. The core library consists of parallel algorithms (pAlgorithms) and distributed data structures (pContainers). A pContainer is the parallel equivalent of an STL container. Its data is distributed, but it offers the user a shared object view. The pContainer distribution can be specified by the user or computed automatically using methods in STAPL’s optimization library. The shared object view is achieved through an internal object translation method that can map, transparently, any part of an object to the thread to which it belongs. A pAlgorithm is the parallel equivalent of an STL algorithm. To bind pAlgorithms to pContainers we introduce the pRange class which supports random access to distributed pContainer data. Random access is essential to parallel processing – every thread must have independent access to its own data to process it in parallel. Adaptive behavior is supported by providing algorithms with the same interface and functionality, but which perform differently depending on the architecture and data involved. Data dependences in pAlgorithms are encoded by dependence graphs (DDGs) stored in the pRange and are enforced at run-time. The traversal of the DDG, i.e., the execution schedule, can be selected from a set of STAPL-provided policies (e.g., wavefront, GSS, etc), or can be provided by the user. Communication and synchronization use the remote method invocation (RMI) communication abstraction that assures mutual exclusion at the destination but hides the lower level implementation (e.g., MPI, OpenMP).

The STAPL run-time system (RTS) is a collection of platform specific components that needs to be adapted whenever STAPL is ported to a new system. The executor object is responsible for the parallel execution of pAlgorithms

while respecting the constraints imposed by the pRange’s DDG. Here we detect and translate remote references to a shared object, and translate the generic RMI to MPI, OpenMP, pthreads, or native communication primitives. We tailor the memory management to the specifics of the hardware, e.g., processor aware allocation, mapping of virtual processor to physical processor, etc. Many low-level optimizations are performed here.

The STAPL optimization toolbox provides support for general high-level optimizations, automatic translation of sequential C++ STL into parallel STAPL, and for optimizing domain-specific library extensions. It collects information about types, algorithm and container usage, and memory layout that can be used for the run-time tuning of programs. It interfaces with the core library and the STAPL RTS.

STAPL interfaces. We distinguish two main views of STAPL. From the user’s perspective, STAPL has a three layer, hierarchical architecture whose interfaces incrementally expose information about and control of parallelism, data distribution, and platform dependent features. In contrast, the STAPL optimization tools typically access all STAPL components, from platform independent library components to platform dependent components such as the native communication library.

For users, STAPL provides three levels of abstraction appropriate to an *application developer* (level 1), a *library developer* (level), and a *run-time system developer* (level 3).

At the highest level, STAPL offers the application developer an STL compatible interface to a generic parallel machine. Parallel programs can be composed by non-expert parallel programmers using building blocks from the core STAPL library. Users don’t have to (but can) be aware of the distributed nature of the machine.

At the intermediate level, STAPL exposes sufficient information to allow a library developer to implement new STAPL-like algorithms and containers, e.g., to expand the STAPL base or build a domain specific library. This is the lowest level at which the “usual” user of STAPL operates. The shared object view and abstract interface to the machine and RTS result in platform independent portable code. Nevertheless, remote and local accesses may be distinguished, and a new container class must specify the data types it stores to enable RMI to pack/unpack container objects. For new pAlgorithms, the developer must (provide a method to) define a DDG describing the algorithm’s dependences.

At the lowest level, the RTS developer has access to the implementation of the communication and synchronization library, the interaction between OS, STAPL thread scheduling, memory management and machine specific features such as topology and memory subsystem organization.

4 Program Development with STAPL

This section illustrates STAPL’s flexibility and ease of use, and provides performance results for two case studies using STAPL.v0. All experiments were run on a 16 processor HP V2200 with 4GB of memory running in dedicated mode. Speedups reported represent the ratio between the sequential algorithm’s running time and its parallel counterpart.

4.1 Molecular Dynamics (automatic vs. manual STL to STAPL translation)

We used STAPL to parallelize a molecular dynamics code written by Danny Rintoul at Sandia National Lab that makes extensive use of STL algorithms and containers. Both (i) automatic translation and (ii) manual recoding were used to translate the STL program into an equivalent STAPL program. The section of code parallelized with STAPL accounts for 40% to 49% of the sequential execution time. It uses STL algorithms (e.g., `for_each`, `transform`, `accumulate`) and parallel `push_back*` operations on vectors.

STAPL provides two ways to automatically translate STL code to STAPL code: *full translation* (the entire code is translated), and *partial translation* (only user defined sections of the code are translated). In both full and partial translation the user must decide which sections of the STL code are safely parallelizable. Without compiler support, it is not possible to identify data dependencies. For full translation, the only change necessary is to add STAPL header files. For partial translation, the sections to be parallelized are enclosed by the user inserted STAPL preprocessing directives `#include < start_translation >`, and `#include < stop_translation >` (see Figures 1 and 2). For the molecular dynamics code we have used partial translation.

Our experimental results, shown in Figure 3, indicate that although automatic translation incurs some run-time overhead, it is simpler to use and the performance is very close to hand parallel (less than 5% performance deterioration). In both cases, the STAPL code achieved scalable speedups.

4.2 Generic Particle Transport Solver (Programming in STAPL)

Within the framework of our DOE ASCI project we have developed, from scratch, a particle transport library. It is a numerical intensive parallel application written entirely in C++ (so far, about 25,000 lines of code). Its purpose is the development of a general testbed for solving transport problems on regular and arbitrary (irregular) grids. Its central

*A `push_back` operation appends an element to the end of a container.

```

std::vector<int> v(400,000);
int sum=0;
...// Execute computation on v
std::accumulate(v.begin(),v.end(), sum);
...// Rest of the computation

```

Figure 1: Original STL code

```

std::vector<int> v(400,000);
int sum=0;
...// Execute computation on v
#include <start_translation>
std::accumulate(v.begin(),v.end(), sum);
#include <stop_translation>
... Rest of the computation

```

Figure 2: STL to STAPL code

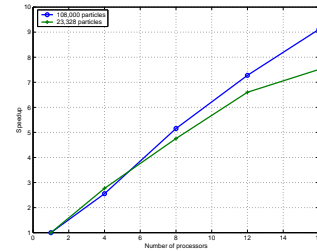


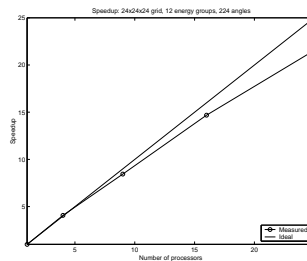
Figure 3: Hand parallel MD.

algorithmic component is a discrete ordinate parallel sweep across a spatial grid in each direction of particle travel [8, 9] which accounts for 50% to 80% of total execution time. Due to the very large data sizes and the enormous amount of computation the code has to scale up to 10,000 processors. The primary data structure is the spatial discretization grid, which is provided as input. The primary algorithm is called a solver, which usually consists mainly of grid sweeps. The solver method may also be given.

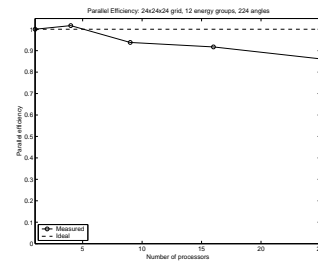
We started with a sequential version of the code written in STL and then transformed it using the `pfor_each` template, which applies a work function to all the elements in a domain. There are three important routines: particle scattering (fully parallel), sweep (partially parallel), and a convergence test (reduction). The `pContainer` that receives the input data is called the `pGrid`; it represents a distributed spatial discretization grid and is a `pGraph`. There is one `pRange` for each direction of particle travel, all built on the same `pGrid`, but with different DDGs. The scattering routines are fully parallel. For the partially parallel sweep loop (which performs graph traversals), the DDGs are computed in the application and passed to STAPL. For the scattering routines, the work function is a routine that computes scattering for a single spatial discretization unit (Cell Set). For the sweep, the work function sweeps a Cell Set for a specific direction of particle travel. For the convergence routines, the work function tests convergence for a Cell Set. *The work function (sweep) was written by the physics/numerical analysis team unaware of parallelism issues. We just needed to put a wrapper around it (the interface required by STAPL).*

Experiments were run in the parallel queue (not dedicated) on a SGI Origin 2000 server with 32 R10000 processors and 8GB of physical memory. The input data was a 3D mesh with 24x24x24 nodes. The energy discretization consisted of 12 energy groups. The angular discretization had 228 angles. The six parts of the code parallelized with STAPL, their execution profile and speedup are shown in Table 4. Speedups are for 16 processors in the experimental setup described above. Our speedups and efficiencies for up to 25 processors are shown in Figure 4.

Code Region	% Seq.	Speedup
Create computational grid	10.00	14.50
Scattering across group-sets	0.05	N/A
Scattering within a group-set	0.40	15.94
Sweep	86.86	14.72
Convergence across group sets	0.05	N/A
Convergence within group sets	0.05	N/A
Total	97.46	14.70



(b)



(c)

Figure 4: (a) Profile and speedups on 16 processor SGI Origin 2000. (b) Speedup, and (c) parallel efficiency (average)

References

- [1] *The CHARM++ Programming Language Manual*. <http://charm.cs.uiuc.edu>, 2000.
- [2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, Aug 2001.
- [3] Guy Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [4] Guy Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

- [5] C. Chang, A. Sussman, and J.H. Saltz. Object-oriented run-time support for complex distributed data structures. Technical Report UMIACS-TR-95-35, UMIACS, University of Maryland, March 1995.
- [6] David Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *International Conference on Supercomputing*, November 1993.
- [7] Matteo Frigo, Charles Leiserson, and Keith Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.
- [8] Adolfo Hoisie, Olaf Lubeck, and Harvey Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. Technical Report LAUR-98-3316, Los Alamos National Laboratory, August 1998.
- [9] Adolfo Hoisie, Olaf Lubeck, and Harvey Wasserman. Scalability analysis of multidimensional wavefront algorithms on large-scale SMP clusters. In *Proceedings of Frontiers '99: The 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 4–15, Annapolis, MD, February 1999. IEEE Computer Society.
- [10] Elizabeth Johnson. *Support for Parallel Generic Programming*. PhD thesis, Indiana University, 1998.
- [11] Elizabeth Johnson and Dennis Gannon. HPC++: Experiments with the parallel standard library. In *International Conference on Supercomputing*, 1997.
- [12] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library. In *Proc. of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburgh, PA, May 1998.
- [13] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn. POOMA: A Framework for Scientific Simulations of Parallel Architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, chapter 14, pages 547–588. MIT Press, 1996.
- [14] Steven Saunders and Lawrence Rauchwerger. ARMI: an adaptive, platform independent communication library. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 230–241. ACM Press, 2003.
- [15] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.
- [16] Thomas Sheffler. A portable MPI-based parallel vector template library. Technical Report RIACS-TR-95.04, Research Institute for Advanced Computer Science, March 1995.
- [17] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, 2005. To appear.
- [18] Gregory Wilson and Paul Lu. *Parallel Programming using C++*. MIT Press, 1996.