Standard Templates Adaptive Parallel Library (STAPL)

Lawrence Rauchwerger, Francisco Arzu, and Koji Ouchi

Dept. of Computer Science
Texas A&M University
College Station, TX 77843-3112
http://www.cs.tamu.edu/faculty/rwerger
{rwerger,farzu,kouchi}@cs.tamu.edu

Abstract. STAPL (Standard Adaptive Parallel Library) is a parallel C++ library designed as a superset of the STL, sequentially consistent for functions with the same name, and executes on uni- or multi-processors. STAPL is implemented using simple parallel extensions of C++ which provide a SPMD model of parallelism supporting recursive parallelism. The library is intended to be of generic use but emphasizes irregular, non-numeric programs to allow the exploitation of parallelism in areas such as geometric modeling or graph algorithms which use dynamic linked data structures. Each library routine has several different algorithmic options, and the choice among them will be made adaptively based on a performance model, statistical feedback, and current run-time conditions. Built–in performance monitors can measure actual performance and, using an extension of the BSP model predict the relative performance of the algorithmic choices for each library routine. STAPL is intended to possibly replace STL in a user transparent manner and run on small to medium scale shared memory multiprocessors which support OpenMP.

1 Motivation

Although multi-processors have become commercially viable, exploiting their potential to obtain scalable speedups has remained an elusive goal, and today their use is still mostly confined to research environments. This lack of popularity among users is due to the difficulty of developing parallel applications that can efficiently exploit the hardware. We believe that parallel processing can be successful only when the effort to achieve scalable performance across a variety of applications and architectures is comparable to that of developing sequential codes.

In sequential computing, standardized libraries have proven to be valuable tools for simplifying the program development process by providing routines for common operations that allow programmers to concentrate on higher level problems. Similarly, libraries of elementary, generic, parallel algorithms would provide important building blocks for parallel applications or specialized libraries

[2,3,6]. Due to the added complexity of programming parallel machines, we believe that the potential impact of libraries on the future of parallel computing will be more profound than for sequential computing. Properly designed libraries could insulate naive users from managing parallelism by providing routines that are easily interchangeable with their sequential counterparts, while allowing more sophisticated users to use their expertise to extract higher performance gains.

Unfortunately, however, designing parallel libraries that are both portable and efficient is a challenge that has so far not been met. This is due mainly to the difficulty of managing concurrency and the wide variety of parallel and distributed architectures. For example, due to the differing costs of an algorithm's communication patterns on different memory systems, the best algorithm on one machine is not necessarily the best on another. On a given machine, the algorithm of choice may vary depending upon the data and run-time conditions (e.g., network traffic and system load). We believe programmers should be liberated from such concerns by parallel libraries that automatically determine which algorithm to use and how to schedule it based on a performance model, statistical feedback, and run-time conditions.

An important constraint on the development of any software is its interoperability with existing codes and standards. The dissemination and eventual adoption by the public of any new library depends on how well programmers can interface the old programs with the new software packages. At the same time, extending or building on top of existing work can greatly reduce both the developing efforts as well as the users' learning experience. It is for this reason that we have chosen to develop a parallel template library (STAPL) that offers full compatibility with the recently ANSI adopted Standard Template Library (STL) [5].

In a departure from previous approaches to libraries which have almost exclusively targeted scientific, numerical applications, STAPL will emphasize irregular, non-numeric programs. It will allow users to exploit parallelism when dynamic linked data structures replace vectors as the fundamental data structure in application areas such as geometric modeling or when algorithms operate on graphs. While we understand the difficulty of this task we believe that modern applications in all fields are rapidly evolving in this direction.

For STAPL to gain widespread acceptance and use, it is essential that the library routines achieve reasonable performance across a wide spectrum of applications and architectures and free its users from problems related to portability and algorithm choice. In STAPL, each library routine will have several different algorithmic options, and the choice among them will be made adaptively based on a performance model, statistical feedback, and current run-time conditions. Built–in performance monitors will measure actual performance and, using a an extension of the BSP model [1] that incorporates system specific information, STAPL will predict the relative performance of the algorithmic choices for each library routine and thus become an adaptive library.

2 STAPL General Specifications

STAPL will be a parallel C++ library with functionality similar to STL. To ease the transition to parallel programming and to insure portability and continuity for the current use of STL, STAPL will be a superset of the STL, it will be sequentially consistent for functions with the same name, and will execute on uni- or multi-processors. These characteristics will have the added benefit of introducing programmers to parallelism in a rather smooth and painless manner.

STAPL will be implemented using simple parallel extensions of C++ which provide a SPMD model of parallelism and will support recursive (nested) parallelism (as in NESL [2]). Although nested parallelism is not widely supported in the currently targeted commercial DSM machines we believe that it is an important feature that needs to be provided from the very beginning for several reasons: (i) large parallel processors have a hierarchical topology and will support a hierarchical run-time system in the near future, (ii) current compilers do not exploit well parallelism at the multi- and microprocessor level (at the same time) – but we believe improvement will come soon, and (iii) library functions are used as basic, elementary blocks which can be themselves nested or incorporated in a larger parallel application, thus requiring appropriate support. We intend to use the machine native run-time system and, as soon as it becomes available, generate OpenMP directives, thus insuring portability across platforms.

We have defined and have initial implementations of the three key extensions: (i) `pforall`, (ii) `prange`, and (iii) `pcontainer`. The `pforall` function applies a function to every object in a container in parallel and its implementation is so far the only architecture specific component of STAPL. The `prange` class is a parallel equivalent of the iterator class in STL that allows random access to all objects in a (certain range) in a container. The `pcontainer` is the parallel equivalent of the STL container and offers (semi–) random access to its elements, a pre-condition for parallel processing.

We will embrace the STL design philosophy of separating container from generic algorithm, which will be connected by `pranges`. There will be three types of parallel algorithms in STAPL. First, parallel algorithms with semantics identical to their sequential counterparts (e.g., sort, merge, reverse). Second, parallel algorithms with enhanced semantics. For example, a sequential `find` operation might return the first element found while a parallel `find` operation might return any (or all) elements found. Third, parallel algorithms with no sequential equivalent in STL, e.g., parallel prefix, a basic operation in many parallel algorithms. These algorithms will be implemented in the extended C++ language mentioned above and the containers will be manipulated through `pranges`. STAPL will use its own specially designed `pcontainers` that will allow pseudo-random access to its elements and thus be usable in parallel computation. For compatibility STL containers and `pcontainers` can co-exist in the same program. Furthermore, STAPL functions can use STL containers which, when necessary, can be translated internally into their parallel counterparts (`pcontainers`). For example, a linked list which does not support random access will be translated internally (without changing the interface) into a parallel linked list (e.g., one sublist for each processor). Initial speedups are not expected to be very high because of the translation overhead. Thus, we will also provide the user with the means to directly use `pcontainers` which will avoid this translation cost. For example, directly invoking a parallel sorting algorithm

```
void psort(container.prange);
```

would avoid any internal translation required to run the original STL `sort` in parallel.

Adding Adaptive Capabilities to STAPL. The goal of this phase is to add features to the STAPL routines that optimize performance by selecting the best algorithm, the number of processors, scheduling, and data layout, thus guaranteeing a certain level of performance across platforms and applications. The two major components that will confer adaptive capabilities to STAPL are execution-time instrumentation for performance monitoring and a relatively simple yet accurate performance model for parallel computation.

For example, load imbalance, network congestion, and cache miss ratios are factors that should be taken into account when determining which and how many processors should be used; hardware monitors now available on modern machines (e.g., SGI Origin2000) provide low overhead performance monitoring capabilities. Also, the algorithm's impact on system resources (e.g., memory traffic and workload) should be considered.

The performance model, a modified BSP model [1] is currently being developed and experimentally validated by our collaborators. Finally, the library will provide performance feedback to the programmer, either during or after program execution which can be used to design more efficient methods.

3 STAPL Components Overview

In this section we briefly present the basic STAPL components and their overall organization.

STL consists of three major components; `containers`, `iterators` and `generic algorithms`. Containers are data structures such as vectors, lists, sets and maps, and generic algorithms are operations such as searching, sorting and merging. Algorithms are generic in the sense that every algorithm can work on a variety of containers. This is accomplished by defining algorithms in terms of `iterators` which are generalized C++ pointers. Iterator types are specific to the different type of container they traverse, e.g., forward, backward, random.

STAPL consists of five major components: `p_containers`, `iterators`, `p_ranges`, the function `p_for_all` and `p_algorithms`. Although `p_for_all()` is the main component which manages parallelism, it is usually hidden from users. Users can write STAPL parallel programs just by using p_containers, p_ranges and p_algorithms. Also, `p_ranges` can be constructed via STL (sequential) containers and iterators.

Parallelism is embedded as follows. A p_range is a collection of iterators to which parallel processing is applied. A p_range is divided into subranges of type s_range each of which is a minimum quantum processed by a single processor

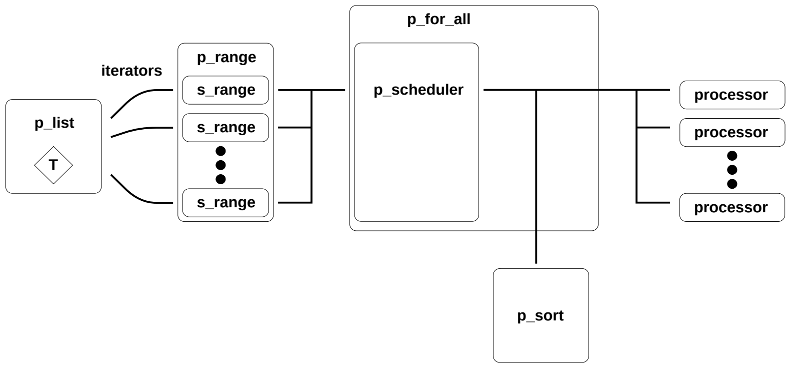

Fig. 1. STAPL Components

at a time. The function `p_for_all()` has a subcomponent `p_scheduler` which gets another s_range, associates it with a copy of p_algorithm and executes the algorithm over the s_range on some processor. Note that we provide random access to the set of s_ranges. Figure 1 shows the overal organization of STAPL's major components.

3.1 P_ranges and P_containers

The p_range class in STAPL is a generalization of the STL iterator. It allows, in essence, semi-parallel access to all objects in a container or within a certain range. The p_range type provides the same begin() and end() functions that the container provides, since a p_range can be seen as a parallel adapter to the container. p_ranges allow the programmer to work with different containers in a uniform manner.

To support nested parallelism the p_range is partioned in a set of sub–p_ranges. At the lowest level of parallelism every p_range will hold a group of serial ranges, called s_ranges, which are a range of contiguous elements in the container. The s_range is the minimum quantum that the scheduler can send to a processor for parallel processing. The union of all s_ranges constitutes the p_range of a container (or p_container). At the limit this means that if we have a list of N elements and every s_range has only one element, we obtain full random access to the list. The ratio between the number of s_ranges of a container to the number of processors (currently kept at 1:1) is a user modifiable parameter and will be adaptively tunable. The scheduler maps each s_range to a specific processor at run time. The p_range class provides methods for partitioning the container or p_container in an efficient way. Any modification of the STAPL P_container (e.g., dynamic insertion or deletion of elements) will be reflected in the information kept by its associated p_range and will be memorized for future instantiations. On the other hand, a STL container is unable to inform its associated p_range of any structural change and the responsibility of keeping its associated p_range up-to-date rests with the programmer.

Every STL container type, has a STAPL counterpart (e.g., `p_vector`, `p_list`). In addition STAPL includes new, complex data structures which are not trivially parallelized (e.g., `p_graph`, `p_hash`) but are necessary in modern, non-numeric applications.

While similar to and backwards compatible to sequential STL containers, p_containers support p_ranges via new member functions and record their distribution information. For example, the p_list (Parallel List) can reuse its per processor distribution information from previous instances, thus avoiding an expensive re-distribution operation. The beginning and ending of each sub-list or s_range (sequential range) can be randomly accessed via its s_range index, but the components inside this s_range will be traversed with the same limitations as that of the sequential iterators for that container (e.g., forward iterators for lists, random iterators for vectors) . This organization offers "semi-random" access to the p_container (and containers). All generic parallel algorithms in STAPL have to follow this rule to work with the different types of containers.

3.2 P_Forall Function Template

P_forall is, so far, the only STAPL parallel programming primitive and is the only machine dependent function template. The function p_forall() applies in parallel the function passed in as an argument to the container p_range. It can work with one or many p_ranges at the same time, depending on the needs of the parallel operation.

On every processor the argument function used in p_forall() is applied to its corresponding s_range (group of elements). The p_forall() construct does not guarantee by itself that its application is correct, i.e. that it will satisfy all the data dependence conditions associated with the concurrent application of the chosen function. This responsibility will rest with the programmer, as in most other languages that support parallel extensions.

The current version of STAPL generates calls to the native run time systems, e.g., the SGI's m_fork library, to implement its parallel processing environment. p_forall() will generate Open-MP standard directives in the near future. When the run-time system allows it, p_forall() supports nested parallelism.

An instantiation of a p_forall causes each processor to create its own copy of the argument function class via its copy constructor. Any instance variables in the function are automatically privatized to each processor's stack. It is expected that most private storage will be allocated and initialized in the prologue() method and freed up in the epilogue() method of this function.

The p_forall() also accepts an optional parameter through which the programmer can provide their own scheduler. If not specified, a default scheduler provided by STAPL will be used.

3.3 Generic P_algorithms

P_algorithms are generic parallel algorithms. P_algorithms are written in terms of p_ranges and iterators. There are three types of parallel algorithms in STAPL.

1. Parallel algorithms with semantics identical to their sequential counterparts (e.g., sort, merge, reverse)
2. Parallel algorithms with enhanced semantics. For example, a sequential find operation might return the first element found while a parallel find operation might return any (or all) elements found.
3. Useful parallel algorithms which are often used in parallel programs (e.g., p-way merge, parallel prefix.).

Figures 3 and 2 represent examples of the speedup obtained for two sorting algorithms, radix sort and merge sort respectively, that have been implemented in STAPL and executed on an SGI Power Challenge. Note that the represent the STL implementation as a baseline (number of processors = 0) followed by the speedup obtained with STAPL on 1, 2, 4 and 8 processors.

In the following section we present some criteria for adaptively choosing and tuning the parallel algorithm.

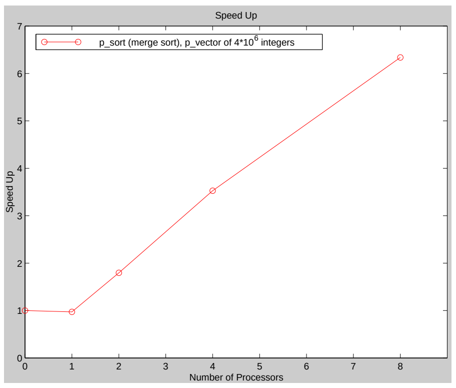

Fig. 2. STAPL Merge Sort Speedup

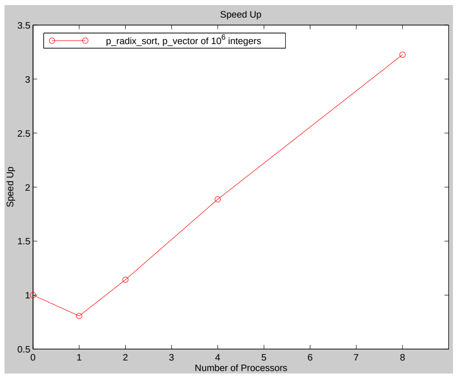

Fig. 3. STAPL Radix Sort Speedup

3.4 Adaptive Features of STAPL

STAPL will provide mechanisms for both programmer directed and automatic algorithm selection. Optional arguments will allow programmers to use *a priori* knowledge of the input to specify which method to employ. If the choice of algorithm is left to STAPL, then a newly developed performance model [1] will be used to determine the best algorithm (possibly sequential) for the given system and data size. Also, the library will be able to sample the input data and to help choose the most appropriate method. Then, the algorithm selection feature will compute the best algorithm based on the performance model and other information such as sensitivity to input data. We will provide methods that automatically analyze this information at execution-time and determine the algorithm of choice.

The Modified BSP model. The BSP is an attractive performance model due to its relative simplicity. Briefly, it measures the time complexity of an algorithm by breaking it into so called *supersteps* (computation and communication taking place between barrier synchronizations) and analyzing each superstep separately. The machine independent complexity is expressed in terms the data size n, number of processors p, and parameters g and l accounting for communication costs such as bandwidth and synchronization. Each machine has different values of g and l which are measured experimentally. The algorithm selection is made by evaluating the complexities using the machine's values for g and l, the available processors p, and the data size n. Amato *et al.*[1] have recently enhanced this model by taking into account more general architectural features and predicting upper and lower bounds on performance, thus making it a more practical model of parallel performance.

Performance Monitoring. Adaptive tuning of performance requires real time data; it is therefore imperative for such an adaptive library to incorporate from the very beginning execution-time instrumentation for performance monitoring. We have started early on with the implementation of execution-time instrumentation for performance monitoring. For example, load imbalance, network congestion, and cache miss ratios, factors that may determine which and how many processors should be used can be easily measured by using the hardware monitors now available on modern machines (e.g., SGI Origin2000). Finally, the library will provide performance feedback to the programmer either during or after program execution, to be used to design more efficient methods.

3.5 Relation to Other Work.

The goal of this research is not to substitute but to complement other current efforts to parallelize C++ with static, compile-time methods. The other parallel STL implementations known to us (e.g., [4]) target scientific applications which employ array data structures (which support random access). STAPL tries to explore linked dynamic structures (for which random access iterators do not exist) and their application in non-numeric applications.

References

1. N. M. Amato, A. Pietracaprina, G. Pucci, L. K. Dale and J. Perdue, "A Cost Model for Communication on a Symmetric Multiprocessor", TR 98004, Dept. of Computer Science, Texas A&M University, January, 1998.
2. G.E. Blelloch, "NESL: A nested data-parallel language," Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, April 1993.
3. G.E. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, 1990.
4. E. Johnson and D. Gannon, *HPC++: Experiments with the Parallel Standard Library*, In *Proc. of the 1997 Int. Conf. on Supercomputing*, 1997, pp. 124–131.
5. D. Musser and A. Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.
6. R. Sedgewick, *Algorithms in C++*, Addison-Wesley, 1992.