STAPL: An Adaptive, Generic Parallel C++ Library*

Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase,
Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger

Dept. of Computer Science, Texas A&M University,
College Station, TX 77843-3112
{pinga, alinj, silviusr, sms5644, tgs7381, gabrielt, nthomas,
amato, rwerger}@cs.tamu.edu

Abstract. The Standard Template Adaptive Parallel Library (STAPL) is a parallel library designed as a superset of the ANSI C++ Standard Template Library (STL). It is sequentially consistent for functions with the same name, and executes on uni- or multi-processor systems that utilize shared or distributed memory. STAPL is implemented using simple parallel extensions of C++ that currently provide a SPMD model of parallelism, and supports nested parallelism. The library is intended to be general purpose, but emphasizes irregular programs to allow the exploitation of parallelism for applications which use dynamically linked data structures such as particle transport calculations, molecular dynamics, geometric modeling, and graph algorithms. STAPL provides several different algorithms for some library routines, and selects among them adaptively at runtime. STAPL can replace STL automatically by invoking a preprocessing translation phase. In the applications studied, the performance of translated code was within 5% of the results obtained using STAPL directly. STAPL also provides functionality to allow the user to further optimize the code and achieve additional performance gains. We present results obtained using STAPL for a molecular dynamics code and a particle transport code.

1 Motivation

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 provide important building blocks for parallel applications and specialized libraries [7, 6, 20]. Due to the added complexity of parallel programming, the potential impact of libraries could be even more profound than for sequential computing. Indeed, we believe parallel libraries are crucial for moving parallel computing into the mainstream since they offer the only viable means for achieving scalable performance across a variety of applications and architectures with programming efforts comparable to those of developing sequential codes. In particular, properly designed parallel libraries could insulate less experienced users

* This research supported in part by NSF CAREER Awards CCR-9624315 and CCR-9734471, NSF Grants ACI-9872126, EIA-9975018, EIA-0103742, and by the DOE ASCI ASAP program grant B347886.

H. Dietz (Ed.): LCPC 2001, LNCS 2624, pp. 193–208, 2003.

from managing parallelism by providing routines that are easily interchangeable with their sequential counterparts, while allowing more sophisticated users sufficient control to achieve higher performance gains.

Designing parallel libraries that are both portable and efficient is a challenge that has not yet 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. Even on a given machine, the algorithm of choice may vary according to the data and run-time conditions (e.g., network traffic and system load).

Another important constraint on the development of any software package is its inter-operability with existing codes and standards. The public dissemination and eventual adoption of any new library depends on how well programmers can interface old programs with new software packages. Extending or building on top of existing work can greatly reduce both developing efforts and the users' learning curve.

To liberate programmers from the concerns and difficulties mentioned above we have designed STAPL (Standard Template Adaptive Parallel Library). STAPL is a parallel C++ library with functionality similar to STL, the ANSI adopted C++ Standard Template Library [17, 22, 13]. To ease the transition to parallel programming and ensure portability across machines and programs, STAPL is a superset of STL that is sequentially consistent for functions with the same name. STAPL executes on uni- or multi-processor architectures with shared or distributed memory and can co-exist in the same program with STL functions. STAPL is implemented using simple parallel extensions of C++ which provide a SPMD model of parallelism and supports nested (recursive) parallelism (as in NESL [7]). In a departure from previous parallel libraries which have almost exclusively targeted scientific or numerical applications, STAPL emphasizes irregular programs. In particular, it can exploit parallelism when dynamic linked data structures replace vectors as the fundamental data structure in application areas such as geometric modeling, particle transport, and molecular dynamics.

STAPL is designed in layers of abstraction: (i) the *interface layer* used by application programmers that is STL compatible, (ii) the *concurrency* and *communication layer* that expresses parallelism and communication/synchronization in a generic manner, (iii) the *software implementation layer* which instantiates the concurrency and communication abstractions to high level constructs, and (iv) the *machine layer* which is OS, run-time system (RTS) and architecture dependent. The *machine layer* also maintains a performance data base for STAPL on each machine and environment.

STAPL provides portability across multiple platforms by including its own (adapted from [18]) run-time system which supports high level parallel constructs (e.g., `forall`). Currently STAPL can interface directly to Pthreads and maintains its own scheduling. It can issue MPI and OpenMP directives and can use the native run-time system on several machines (e.g., HP V2200 and SGI Origin 2000). Thus, there is no need for user code modification when porting a program from one system to another. Only the *machine* layer needs to be modified when STAPL is ported to a new machine.

We have defined and implemented several key extensions of STL for STAPL: parallel containers and algorithms (`pContainer` and `pAlgorithms`), and an entirely new construct called `pRange` which allows random access to elements in a pContainer. Analogous to STL iterators, pRanges bind pContainers and pAlgorithms. Unlike STL iterators, pRanges also include a `distributor` for data distribution and a `scheduler` that can generically enforce data dependences in the parallel execution according to execution data dependence graphs (DDGs). STAPL allows for STL containers and algorithms to be used together with STAPL pContainers and pAlgorithms in the same program. STAPL provides a means of automatically transforming code that uses STL to code that uses STAPL. This parallelizes the application with very little user modification, but incurs some run-time overhead. To obtain even better performance, STAPL allows users to avoid the translation overhead by directly writing applications using pContainers, pAlgorithms, and pRanges. STAPL provides recursive data decomposition through its pRange which allows programs to be naturally mapped to hierarchical architectures.

To achieve wide adoption, STAPL must obtain reasonable performance across a wide spectrum of applications and architectures and free its users from problems related to portability and algorithm selection. This is achieved in STAPL by adaptive selection among various algorithmic options available for many STAPL library routines. Built–in performance monitors will measure actual performance, and using performance models [3, 2] that incorporate system specific information and current run-time conditions, STAPL will adaptively select an appropriate algorithm for the current situation, based on the performance monitors and performance models.

2 Related Work

There is a relatively large body of work that has similar goals to STAPL. Table 1 gives an overview of different projects. We will now briefly comment on some of them and attempt to compare them with STAPL. For further work in this area see [23].

	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, resp.

The Parallel Standard Template Library (PSTL) [15, 14] has similar goals to STAPL; it uses parallel iterators in place of STL iterators and provides some parallel algorithms and containers. NESL [7], CILK [10] and SPLIT-C [9] 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 of listed packages (STAPL, Amelia [21], CHAOS++ [8] 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 application through their chare objects and inspector/executor, respectively. Both POOMA [19] and STAPL borrow from the STL philosophy, i.e., containers, iterators, and algorithms. The communication/computation overlapping mechanism is present in the STAPL executor, which also supports simultaneous use of both message passing and shared memory (MPI and OpenMP) communication models. 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 further distinguished in that it emphasizes both automatic support and user specified policies for *scheduling, data decomposition and data dependence enforcement*. Furthermore, STAPL is unique in its goal to *automatically select the best performing algorithm* by analyzing data, architecture and current run-time conditions.

3 STAPL – Philosophy, Interface, and Implementation

STL consists of three major components: `containers`, `algorithms`, and `iterators`. Containers are data structures such as vectors, lists, sets, maps and their associated methods. Algorithms are operations such as searching, sorting, and merging. Algorithms can operate on a variety of different containers because they are defined only in terms of templated iterators. Iterators are generalized C++ pointers that abstract the type of container they traverse (e.g., linked list to bidirectional iterators, vector to random access iterators).

STAPL's interface layer consists of five major components: `pContainers`, `pAlgorithms`, `pRanges`, `schedulers/distributors` and `executors`. Figure 1 shows the overall organization of STAPL's major components. The `pContainers` and `pAlgorithms` are parallel counterparts of the STL containers and algorithms; pContainers are backwards compatible with STL containers and STAPL includes pAlgorithms for all STL algorithms and some additional algorithms supporting parallelism (e.g., parallel prefix). The `pRange` is a novel construct that presents an abstract view of a scoped data space which allows *random access* to a partition, or subrange, of the data space (e.g., to elements in a pContainer). A pRange can recursively partition the data domain to support nested parallelism. Analogous to STL iterators, pRanges bind pContainers and pAlgorithms. Unlike STL iterators, pRanges also include a `distributor` for data distribution and a `scheduler` that can generically enforce data dependences in the parallel execution according to data dependence graphs (DDGs). The STAPL `executor` is responsible for executing subranges of the pRange on processors based on the specified execution schedule. Users can write STAPL programs using pContain-

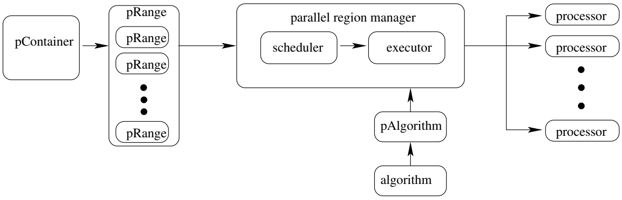


Fig. 1. STAPL Components

ers, pRanges and pAlgorithms, and, optionally, their own schedulers and executors if those provided by STAPL do not offer the desired functionality.

Application programmers use the interface layer and the concurrency/communication layer, which expresses parallelism and communication generically. The software and machine layers are used internally by STAPL, and only the machine layer requires modification when porting STAPL to a new system. In STAPL programmers can specify almost everything (e.g., scheduling, partitioning, algorithmic choice, containers, etc) or they can let the library decide automatically the appropriate option.

In the remainder of this section we present a more detailed discussion of the basic STAPL components and their current implementation.

3.1 pRanges

A `pRange` is an abstract view of a scoped data space providing *random access* to a partition of the data space that allows the programmer to work with different (portions of) containers in a uniform manner. Note that random access to (independent) work quanta is an essential condition for parallelism. Each subspace of the scoped data space is disjoint and can itself be described as a pRange, thus supporting nested parallelism. A pRange also has a relation determining the computation order of its subspaces and relative weights (priorities) for each subspace. If the partition, ordering relation, and relative weights (execution priorities) are not provided as input, then they can be self-computed by the pRange or imposed (by STAPL or the user) for performance gains.

pRange Implementation So far we have implemented the `pRange` for pvectors, plists and ptrees. The pRange and each of its subranges provide the same begin() and end() functions that the container provides, which allows the pRange to be used as a parallel adapter of the container. For example,

```
        stapl::pRange(pContainer.begin(), pContainer.end())
```

constructs a pRange on the data in pContainer. STL has no direct sequential equivalent of pRange, but a structure to maintain a range could be implemented as a simple pair of iterators. For example, a range on a sequential STL vector of integers can be constructed as follows.

```
std::pair<std::vector<int>::iterator,std::vector<int>::iterator>
       seqRange(seqContainer.begin(),seqContainer.end());
```

```
1. stapl::pRange<stapl::pVector<int>::iterator>
       dataRange(segBegin, segEnd);
2. dataRange.partition(4);
3. stapl::pRange<stapl::pVector<int>::iterator>&
        dataSubrange = dataRange.get_subrange(3);
4. dataSubrange.partition(4);
```

Fig. 2. Creating a pVector `dataRange` from iterators, partitioning it into 4 subranges, selecting the 3rd subrange `dataSubrange`, and sub-partitioning it into 4 (sub)subranges.

The pRange provides random access to each of its subranges (recursively), while the elements within each subrange at the lowest level (of the recursion) must be accessed using the underlying STL iterators. For example, a pRange built on a list would provide bidirectional iterators to the begin and end of each subrange, and elements within the subranges could only be accessed in a linear fashion from either point using the bidirectional iterators. In contrast, a pRange built on a vector would provide random access iterators to the begin and end of each subrange, and internal elements of each subrange could be accessed in a random manner using them.

STAPL provides support for nested parallelism by maintaining the partition of a pRange as a set of subranges, each of which can itself be a complete pRange with its own partition of the data it represents (see Figure 2). This allows for a parallel algorithm to be executed on a subrange as part of a parallel algorithm being executed on the entire pRange. The bottom (hierarchically lowest level) subrange is the the minimum quantum of work that the executor can send to a processor.

The pRange can partition the data using a built in distributor function, by a user specified map, or it might be computed earlier in the program. A simple example of distribution tuning is static block data distribution where the chunk sizes are either precomputed, given by the user, or automatically computed by the pRange and adjusted adaptively based on a performance model and monitoring code. In the extreme case, each data element is a separate subrange, which provides fully random access at the expense of high memory usage. Usually, larger chunks of data are assigned to each subrange.

3.2 pContainers

A `pContainer` is the parallel equivalent of the STL container and is backward compatible with STL containers through its ability to provide STL iterators. Each pContainer provides (semi–) random access to its elements, a prerequisite for efficient parallel processing. Random access to the subranges of a pContainer's data is provided by an internal pRange maintained by the pContainer. The internal pRange is updated when the structure of the pContainer is modified (e.g., insertion or deletion of elements) so that a balanced, or user-defined, partition can be maintained.

pContainer Implementation The pContainers currently implemented in STAPL are `pvector`, `plist` and `ptree`. Each adheres to a common interface and maintains

```
stapl::pVector<int> pV(i,j);
stapl::pSort(pV.get_pRange());
```

(a)

```
std::vector<int> sV(i,j);
std::sort(sV.begin(),sV.end());
```

(b)

Fig. 3. (a) STAPL and (b) STL code fragments creating (line 1) and sorting (line 2) pContainers and Containers, resp.

an internal pRange. Automatic translation from STL to STAPL and vice versa requires that each pContainer provides the same data members and member functions as the equivalent STL containers along with any class members specifically for parallel processing (e.g., the internal pRange). Thus, STAPL pContainer interfaces allow them to be constructed and used as if they were STL containers (see Fig. 3).

The pContainer's internal pRange maintains (as persistent data) the iterators that mark the boundary of the subranges. The continual adjusting of subranges within the internal pRange may eventually cause the distribution to become unbalanced. When the number of updates (insertions and deletions) made to a pContainer reach a certain threshold (tracked using an update counter in the pContainer) the overall distribution is examined and the subranges and pRange are adjusted to bring the distribution back to a near balanced state, or a user-defined distribution if one is provided. The maintenance of a distributed internal pRange is critical to the performance of STAPL so that a redistribution of a pContainer's data before beginning execution of each parallel region can be avoided.

When possible, each pContainer's methods have been parallelized (e.g., the pVector copy constructor). The methods may be parallelized in two ways: (i) internal parallelization – the method's operation is parallel, and (ii) external parallelization (concurrency) – the method may be called simultaneously by different processors in a parallel region. These two approaches to parallelization coexist and are orthogonal. A pContainer method can utilize both methods of parallelism simultaneously to allow for nested parallelism.

3.3 pAlgorithms

A `pAlgorithm` is the parallel counterpart of the STL algorithm. There are three types of pAlgorithms in STAPL. First, pAlgorithms with semantics identical to their sequential counterparts (e.g., sort, merge, reverse). Second, pAlgorithms with enhanced semantics (e.g., a parallel `find` could return any (or all) elements found, while a sequential find generally returns only the first element). Third, pAlgorithms with no sequential equivalent in STL.

STL algorithms take iterators marking the start and end of the input as parameters. STAPL pAlgorithms take pRanges as parameters instead. STAPL provides a smooth transition from STL by providing an additional interface for each of its pAlgorithms that is equivalent to its STL counterpart and automatically constructs a pRange from the iterator arguments. The pAlgorithms express parallelism through calls to a parallel region manager, which frees the STAPL user from low level issues such as construction of parallel structures, scheduling and execution. STAPL also allows users to implement custom pAlgorithms through the same interface.

pAlgorithm Implementation Currently, STAPL provides parallel equivalents for all STL algorithms that may be profitably parallelized. Some algorithms perform sequentially very well and we have chosen to focus our efforts on exploiting parallelism on other algorithms (this may change as STAPL matures and more systems are studied). STAPL pAlgorithms take the pRanges to process as arguments along with any other necessary data (e.g., a binary predicate). See Fig. 3 for examples of pSort and sort.

The pAlgorithms in STAPL are implemented by expressing parallelism through the *parallel region manager* of STAPL's concurrency and communication layer. The parallel region manager (e.g., `pforall`) issues the necessary calls to the STAPL runtime system to generate or awaken the needed execution threads for the function, and passes the work function and data to the execution threads. Each pAlgorithm in STAPL is composed of one or more calls to the parallel region manager. Between parallel calls, the necessary post processing of the output of the last parallel region is done, along with the preprocessing for the next call. The arguments to the parallel region manager are the pRange(s) to process, and a `pFunction` object, which is the work to be performed on each subrange of the pRange.

The pFunction is the base class for all work functions. The only operator that a pFunction instance must provide is the () operator. This operator contains the code that works on a subrange of the provided pRanges. In addition, a pFunction can optionally provide prologue and epilogue member functions that can be used to allocate and deallocate any private variables needed by the work function and perform other maintenance tasks that do not contribute to the parallel algorithm used in the work function. The pFunction and parallel construct interfaces can be accessed by STAPL users to implement user-defined parallel algorithms. Figure 4 is an example of a simple work function that searches a subrange for a given value and returns an iterator to the first element in the subrange that matched the value. The example assumes the == operator has been defined for the data type used.

```
template<class pRange, class T>
class pSearch : public stapl::pFunction {
private:
  const T value;
public:
  pSearch(const T& v) : value(v) {}
  typename pRange::iterator operator()(pRange& pr) {
    typename pRange::iterator i;
    for (i = pr.begin(); i != p.end(); i++)
      if (*i == value) return i;
    return pr.end();
  }
};
```

Fig. 4. STAPL work function to search a pRange for a given value

3.4 Scheduler/Distributor and Executor

The `scheduler/distributor` is responsible for determining the execution order of the subspaces in a pRange and the processors they will be assigned to. The schedule must enforce the natural data dependences of the problem while, at the same time, minimizing execution time. These data dependences are represented by a Data Dependence Graph (DDG).

The STAPL `executor` is responsible for executing a set of given tasks (subranges of a pRange and work function pairs) on a set of processors. It assigns subranges to processors once they are ready for processing (i.e., all the inputs are available) based on the schedule provided by the scheduler. There is an executor that deals with the subranges at each level in the architectural hierarchy. The executor is similar to the CHARM++ message driven execution mechanism [1].

Scheduler/Distributor and Executor Implementation STAPL provides several schedulers, each of which use a different policy to impose an ordering on the subranges. Each scheduler requires as input the pRange(s) that are to be scheduled and the processor hierarchy on which to execute. The *static scheduling* policy allows *block scheduling* and *interleaved block scheduling* of subranges to processors. The *dynamic scheduling* policy does not assign a subrange to any given processor before beginning parallel execution, but instead allows the executor to assign the next available subspace to the processor requesting work. The *partial self scheduling* policy does not assign subspaces to a specific processor, but instead creates an order in which subspaces will be processed according to their weight (e.g., workload or other priority) and the subspace dependence graph. The executor then assigns each processor requesting work the next available subspace according to the order. Finally, the *complete self scheduling* policy enables the host code to completely control the computation by indicating an assignment of subspaces to particular processors and providing a subspace dependence graph for the pRange. If no ordering is provided, then STAPL can schedule the subspaces according to their weights (priorities), beginning with the subspaces that have the largest weights, or, if no weights are given, according to a round robin policy.

The recursive pRange contains, at every level of its hierarchy, a DAG which represents an execution order (schedule) of its subranges. In the case of a `doall` no ordering is needed and the DAG is degenerate. The subranges of a recursive pRange (usually) correspond to a certain data decomposition across the processor hierarchy. The `distributor` will, at every level of the pRange, distribute its data and associated (sub) schedule (i.e., a portion of the global schedule) across the machine hierarchy. The scheduler/distributor is formulated as an optimization problem (a schedule with minimum execution time) with constraints (data dependences to enforce, which require communication and/or synchronization). If the scheduler does not produce an actual schedule (e.g., a fully parallel loop) then the distributor will either compute an optimal distribution or use a specified one (by the user or by a previous step in the program).

Each pRange has an `executor` object which assigns subspaces (a set of nodes in a DDG) and work functions to processors based on the scheduling policy. The executor maintains a ready queue of tasks (subspaces and work function pairs). After the current task is completed, the executor uses point-to-point communication primitives to transmit, if necessary, the results to any dependent tasks. On shared memory systems, synchronizations (e.g., post/await) will be used to inform dependent tasks the results are ready. This process continues until all tasks have been completed. STAPL can support MIMD parallelism by, e.g., assigning each processor different DDGs, or partial DDGs, and work functions. Nested parallelism is achieved by nested pRanges, each with an associated executor.

3.5 STAPL Run-Time System

The STAPL run-time system provides support for parallel processing for different parallel architectures (e.g., HP V2200, SGI Origin 2000) and for different parallel paradigms (e.g., OpenMP, MPI). We have obtained the best results by directly managing the Pthread package. The STAPL run-time system supports nested parallelism if the underlying architecture allows nested parallelism via a hierarchical native run-time system. Otherwise, the run-time system serializes the nested parallelism. We are in the process of incorporating the HOOD run-time system [18].

While memory allocation can create performance problems for sequential programs, it is often a source of major bottlenecks for parallel programs [24]. For programs with very dynamic data access behavior and implicit memory allocation, the underlying memory allocation mechanisms and strategies are extremely important because the program's performance can vary substantially depending on the allocators' performance. STL provides a dynamic-behavior framework through the use of the memory heap. STAPL extends STL for parallel computation, and therefore relies heavily on efficient memory allocation/deallocation operations. The memory allocator used by STAPL is the HOARD parallel memory allocator [5]. Hoard is an efficient and portable parallel memory allocator that enhances STAPL's portability.

All algorithms and containers whose characteristics may change during execution have been instrumented to collect run-time information. For now we collect execution times of the different stages of the computation and "parallel behavior", e.g., load imbalance, subrange imbalance (suboptimal distribution). The output of the monitors is used as feedback for our development effort and, in a few instances, as adaptive feedback to improve performance.

4 Performance

This section examines STAPL performance and shows its flexibility and ease of use through several case studies. More details and results illustrating the performance of pContainers (a ptree) and adaptive algorithm selection can be found in [4].

All experiments were run on a 16 processor HP V2200 with 4GB of memory running in dedicated mode. All speedups reported represent the ratio between the sequential algorithm's running time and its parallel counterpart.

4.1 Molecular Dynamics

To illustrate the performance and the ease of coding with STAPL, we used STAPL to parallelize a molecular dynamics code which makes extensive use of STL, including algorithms and data encapsulated in containers. We have used both semi-automatic and manual parallelization modes. This code was written by Danny Rintoul at Sandia National Labs.

Algorithm Description The algorithm is a discrete event simulation that computes interactions between particles. At the initialization phase, each particle is assigned an initial coordinate position and a velocity. According to their attributes (coordinates and velocity) these particles interact according to specified physics laws.

At each discrete time event, interactions between particles are computed that determine the evolution of the system at the next time step. Due to obvious flow data dependences at the event level we have parallelized the code at the event processing level. We have applied two parallelization methods with STAPL: (i) automatic translation from STL to STAPL, and (ii) manual modification to STAPL constructs.

The section of code parallelized with STAPL uses STL algorithms (e.g., for_each, transform, accumulate) and uses parallel push_back[1] operations on vectors. This section of the code represents 40% to 49% of the sequential execution time.

The rest of the program uses a set[2] for ordering the particles according to their next interaction time. Insertion and deletion operations are performed on this set.

Automatic Translation STAPL provides two ways for automatically translating STL code to STAPL code: (i) *full translation* (the entire code is translated), and (ii) *partial translation* (only user defined sections of the code are translated).

There is a fundamental condition in both full and partial translation: the user must decide which sections of the STL code are safely parallelizable. It is not possible for STAPL to determine if the code is inherently sequential due to the potential existence of data dependencies. Compiler support is needed for detecting such dependencies. For full translation, the only change necessary is adding the STAPL header files. For partial translation, the sections to be parallelized are enclosed by the user inserted STAPL preprocessing directives $\#include < start\_translation >$, and $\#include < stop\_translation >$. Figure 5 and Figure 6 give an example of the manual code instrumentation needed to start automatic translation from STL to STAPL. For the molecular dynamics code we have used partial translation.

Manual Translation This method takes full advantage of STAPL, by allowing the user to explicitly replace STL code with STAPL, thus reducing the run-time work STAPL needs to perform. Table 8 gives the execution times and Figure 7 plots the speedups obtained from our experiments, which show a scalable performance.

STAPL exhibits the best performance when used directly through manual modification. However, our experimental results indicate that even though automatic translation incurs some run-time overhead, it is simpler to use and the performance is very close to that obtained with manual modification (less than 5% performance deterioration).

[1] A push_back operation is a container method which some STL containers have. The push_back method appends an element to the end of a container.

[2] Most STL implementations implement a set with Red Black trees. We are in the process of creating a parallel set.

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

Fig. 5. 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
```

Fig. 6. STL to STAPL code

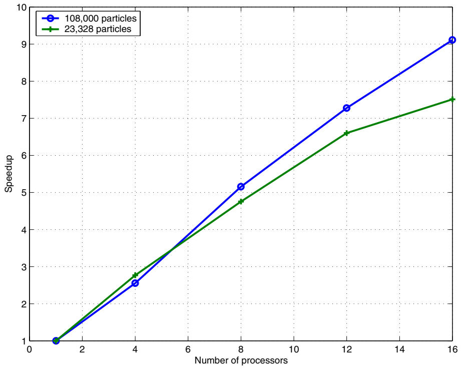


Fig. 7. Molecular Dynamics Partial Speedup for manually STAPL programmed code

Number of	*Number of processors*				
Particles	*1*	*4*	*8*	*12*	*16*
108,000	2815	1102	546	386	309
23,326	627	238	132	94.2	86.4

Fig. 8. Molecular Dynamics Execution (*sec*)

4.2 Generic Particle Transport Solver

Within the framework of the DOE ASCI project we have developed, from scratch, a particle transport code. 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 grids. Its central algorithmic component is a discrete ordinate parallel sweep across a spatial grid. This involves the traversal of the spatial grid in each direction of particle travel [11, 12]. This part of the computation takes 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.

Using STAPL 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. The application defines five entities: (1) **pContainers** to store data in a distributed fashion, (2) **pRanges** to define the domain on which to apply algorithms, (3) execution **schedules** to specify data dependencies, (4) **pFunctions** to be applied to every element of the specified domain, and (5) an **executor** to manage parallel execution. In the following we will refer to three important routines: particle scattering computation (fully parallel), sweep (partially parallel) and convergence test (reduction).

The **pContainer** that receives the input data is called the `pGrid` and represents a distributed spatial discretization grid. It is derived from STAPL's predefined pVector. All parallel operations use the same **pRange** because they operate on the same pGrid.

For every parallel operation we specify an execution **schedule** (by default the execution schedule is empty, i.e., no predefined order). The schedule is a directed graph whose vertices are points in the range described by the pRange built above. The scattering routines are fully parallel loops and thus have empty schedules. For the partially parallel sweep loop, the schedule is computed by the application and is given to STAPL as a Data Dependence Graph (DDG) which STAPL incorporates into the pRange.

A **pFunction** is a work function that takes as arguments elements in the pRange built above. For the scattering routines, the pFunction is a routine that computes scattering for a single spatial discretization unit (Cell Set). For the sweep, the pFunction is a routine that sweeps a Cell Set for a specific direction of particle travel. For the convergence routines, the pFunction is a routine that 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).*

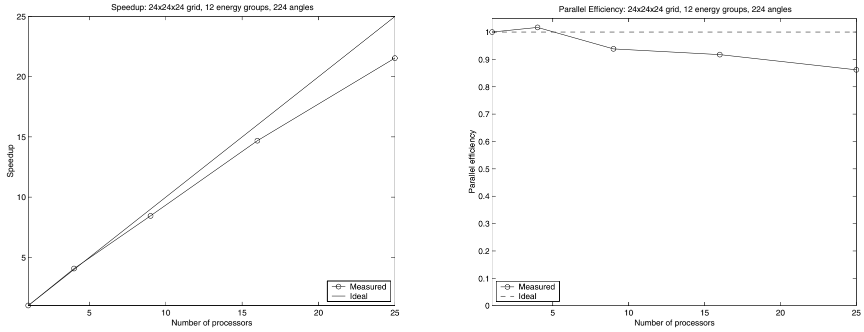
The **Executor** is created by combining a set of pRanges with a set of pFunctions. Parallel execution, communication and synchronization are handled by the executor. It first maps subranges in the pRange to execution threads based on knowledge about machine, OS and underlying RTS. Nodes in the DDG stored in the pRange correspond to execution vertices and edges specify communication and synchronization. The executor ensures that the corresponding message for every incoming edge is received before it starts to execute the work function on a specific node. After the execution of the work function, the executor sends out messages for all outgoing edges. For the scattering routines (fully parallel), the executor will not initiate any communication because the DDG given as the schedule has no edges. In the convergence routines, the executor performs a reduction by using a lower-level MPI call `MPI_Allreduce`. It should be noted that an application can define its own executor class as long as it presents the same interface as the STAPL base executor. *In our case, creating the executor class required only 1 line of code: an instantiation of the executor constructor taking the pRanges and the pFunctions as actual parameters.*

Experimental Results 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 2. Speedups are for 16 processors in the experimental setup described above.

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

Table 2. Profile and Speedups on 16 processor SGI Origin 2000



(a) (b)

Fig. 9. (a) Speedup and (b) parallel efficiency (averaged across two runs.)

Our results are shown in Figure 9. The particular scheduling scheme used in this experiment (KBA) [16] requires that the number of processors be a perfect square, so we had to stop at 25 processors.

5 Conclusions and Future Work

STAPL is a parallel programming library designed as a superset of STL. STAPL provides parallel equivalents of STL containers, algorithms, and iterators, which allow parallel applications to be developed using the STAPL components as building blocks. Existing applications written using STL can be parallelized semi-automatically by STAPL during a preprocessing phase of compilation that replaces calls to STL algorithms with their STAPL equivalents. Our experiments show the performance of applications that utilize the automatic translation to be similar to the performance of applications developed manually with STAPL. The automatic translation of STL code to STAPL, the handling of the low level details of parallel execution by the parallel region manager, and the adaptive run-time system allow for portable, efficient, and scalable parallel applications to be developed without burdening the developer with the management of all the details of parallel execution.

STAPL is functional and covers almost all of the equivalent STL functionality. However much work lies ahead: Implementing several algorithmic choices for each function, full support of the recursive pRange on very large machines, a better RTS and its own, parallel memory manager are only a few of the items on our agenda.

Acknowledgements We would like to thank Danny Rintoul of Sandia National Laboratories for providing us with the molecular dynamics application presented in Section 4.1.

References

[1] *The CHARM++ Programming Language Manual*. http://charm.cs.uiuc.edu, 2000.

[2] N. M. Amato, J. Perdue, A. Pietracaprina, G. Pucci, and M. Mathis. Predicting performance on SMPs. a case study: The SGI Power Challenge. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 729–737, 2000.

[3] N. M. Amato, A. Pietracaprina, G. Pucci, L. K. Dale, and J. Perdue. A cost model for communication on a symmetric multiprocessor. Technical Report 98-004, Dept. of Computer Science, Texas A&M University, 1998. A preliminary verson of this work was presented at the *SPAA'98 Revue*.

[4] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Stapl: An adaptive, generic parallel programming library for c++. Technical Report TR01-012, Dept. of Computer Science, Texas A&M University, June 2001.

[5] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. HOARD: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operatings Systems (ASPLOS)*, 2000.

[6] Guy Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[7] Guy Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[8] C. Chang, A. Sussman, and J. Saltz. Object-oriented runtime support for complex distributed data structures, 1995.

[9] 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.

[10] 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.

[11] Adolfy 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.

[12] Adolfy 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.

[13] International Standard ISO/IEC 14882. *Programming Languages – C++*, 1998. First Edition.

[14] Elizabeth Johnson. *Support for Parallel Generic Programming*. PhD thesis, Indiana University, 1998.

[15] Elizabeth Johnson and Dennis Gannon. HPC++: Experiments with the parallel standard library. In *International Conference on Supercomputing*, 1997.

[16] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of the 3D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65:198–199, 1992.

[17] David Musser, Gillmer Derge, and Atul Saini. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[18] C.G. Plaxtion N.S. Arora, R.D. Blumofe. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, June 1998.

[19] J. Reynders. Pooma: A framework for scientific simulation on parallel architectures, 1996.

[20] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.

[21] Thomas Sheffler. A portable MPI-based parallel vector template library. Technical Report RIACS-TR-95.04, Research Institute for Advanced Computer Science, March 1995.

[22] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.

[23] Gregory Wilson and Paul Lu. *Parallel Programming using C++*. MIT Press, 1996.

[24] Paul Wilson, Mark Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, September 1995.