STAPL: A Standard Template Adaptive Parallel C++ Library

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

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 in areas such as particle transport calculations, molecular dynamics, geometric modeling, and graph algorithms, which use dynamically linked data structures. STAPL provides several different algorithms for some library routines, and selects among them adaptively at run-time. STAPL can replace STL automatically by invoking a preprocessing translation phase. The performance of translated code is close to the results obtained using STAPL directly (less than 5% performance deterioration). However, 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.**

I. 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 [9], [8], [24]. 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 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 [20], [26], [16]. 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 [9]). 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 (e.g., m_fork and parallel do for concurrency, and OpenMP synchronizations and MPI primitives for communication), and (iv) the *machine layer* which is OS, 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 [21]) 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

Dept. of Computer Science, Texas A&M University, College Station, TX 77843-3112. Email: {pinga, alinj, silviusr, sms5644, tgs7381, gabrielt, nthomas, amato, rwerger }@cs.tamu.edu. Research was supported in part by NSF by CAREER awards CCR-9624315 and CCR-9734471, by NSF grants IRI-9619850, ACI-9872126, EIA-9805823, EIA-9810937, EIA-9975018, by DOE ASCI ASAP Level 2 grant B347886, by Sandia National Laboratory, and by a Hewlett-Packard Equipment Grant. Thomas supported in part by a Dept. of Education Graduate Fellowship.

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). The STAPL `executor` is responsible for executing subranges of the pRange on processors based on the specified execution schedule. 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. In a preprocessing step at compile time, calls to STL algorithms are replaced with calls to special STAPL algorithms that create the necessary pRanges and call the appropriate pAlgorithms. 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 [4], [3] that incorporate system specific information and current run-time conditions, STAPL will predict the relative performance of the algorithmic choices for each library routine and will adaptively select an appropriate algorithm for the current situation.

II. Related Work

There is a relatively large body of work that has similar goals to STAPL. Table I 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 [27].

The Parallel Standard Template Library (PSTL) [18], [17] has similar goals to STAPL; it uses parallel iterators as a parallel equivalent to STL iterators and provides some parallel algorithms and containers. NESL [9], CILK [15] and SPLIT-C [13] 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 [25], CHAOS++ [11] 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 [23] 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.

III. 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 pContainers, 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

	STAPL	AVTL	CHARM++	CHAOS++	CILK	NESL	POOMA	PSTL	SPLIT-C
Paradigm	SPMD/MIMD	SPMD	MIMD	SPMD	SPMD/MIMD	SPMD/MIMD	SPMD	SPMD	SPMD
Architecture	Shared/Dist	Dist	Shared/Dist	Dist	Shared/Dist	Shared/Dist	Shred/Dist	Shared/Dist	Shared/Dist
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 I
RELATED WORK

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.

A. 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.

A.1 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(pC.begin(), pC.end());
```

constructs a pRange on the data in pContainer `pC`. 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 `vec` can be constructed as follows.

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

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

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 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.

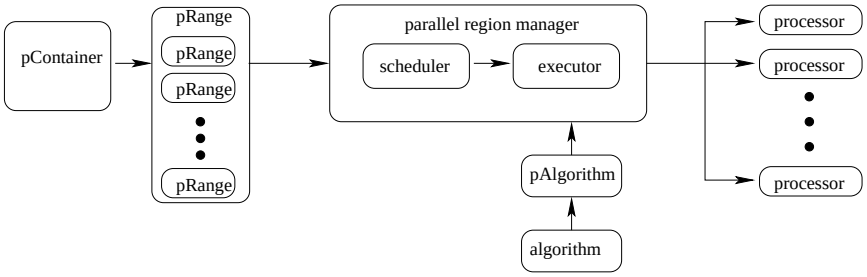


Fig. 1. STAPL Components

B. 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.

B.1 pContainer Implementation

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

```
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 pContainers and Containers (line 1) and sorting them (line 2).

The STAPL pContainer only maintains its internal pRange during operations that modify the pContainer (e.g., insertion and deletion). Any pRanges *copied from the internal pRange* or *created externally* on a portion of the container may be invalidated by changes made to the container data. The same is true of iterators in STL, where a user must take care not to use invalidated iterators. Similarly, it is the user's responsibility to avoid using a pRange that may have been invalidated.

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. pVector's 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 of method parallelization coexist and are orthogonal. A method of a pContainer can utilize both methods of parallelism simultaneously to allow for nested parallelism.

C. 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 implementor 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.

C.1 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 run-time system (implemented on top of Pthreads, native, etc.) 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 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 opera-
tor()(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

D. 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].

D.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 two types of scheduling: *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 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.

E. 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 managing directly 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 [21].

While memory allocation can create performance problems for sequential programs, it is often a source of major bottlenecks for parallel programs [28]. 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 such 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 [7]. 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 (see Section IV-C).

IV. Performance

This section presents performance results for some of STAPL's pContainers and pAlgorithms. We also use sorting as a case study to describe run-time adaptive algorithm selection in STAPL. Additional details and examples illustrating STAPL's flexibility and ease of use can be found in [6]; these include the STAPL parallelization (both automatically and manually) of a sequential C++ molecular dynamics code and a programmed from scratch in STAPL discrete ordinates particle transport code.

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.

A. STAPL pContainers: The pTree

Several containers in STL are implemented with Red-Black trees. To parallelize algorithms using sets, multisets, maps, etc., we have implemented a parallel tree container (pTree).

The parallel insertion and deletion operations for a pTree are assumed to be order-commutative, i.e., we assure the user of such a container that its final state (after the global synchronization) will contain exactly the same nodes and in the same order as the corresponding STL tree. However, we do not guarantee anything about the inner structure of the container, i.e., the results are only guaranteed to be sequentially consistent at synchronization points. The STAPL implementation of this container requires only negligible additional memory over its sequential counterpart, thus assuring scalability.

There are two types of tree operations: (i) *Bottom-up* operations start from the leaves and go upward until they reach the root, and (ii)*Top-down* operations start from the root and go down until they reach a leaf. Usually, top-down operations are Read operations and bottom-up operations are Write/Read.

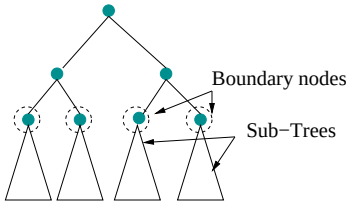
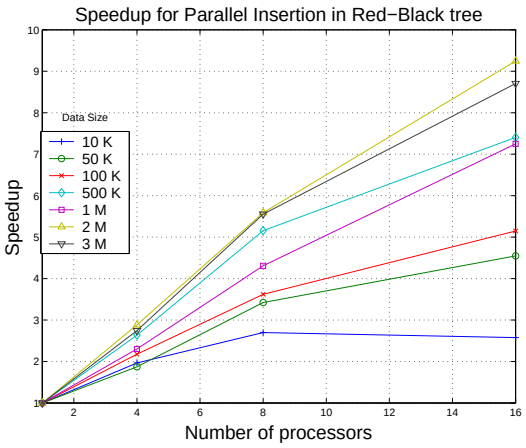


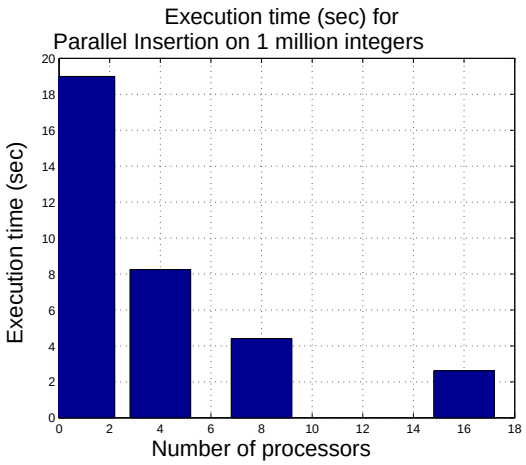
Fig. 5. pTree structure

In Fig. 5 we show the general pTree structure. Each processor locally manages one or more subtrees. The topmost tree structure, called the *base*, has the same root as the tree and its leaves (called *boundary nodes*) are roots of subtrees. The base connects the subtrees and together they form a tree. All operations performed on subtrees are local to the processor which "owns" the subtree and can be executed in parallel, while operations on the base are executed atomically. Atomic operations on the base ensure overall tree coherence and preserve tree and subtree properties. The *base* has the following properties: (i) every base leaf is the root of a subtree, and (ii) all operations performed on the base are atomic. Each processor owns one or more boundary nodes (and hence subtrees). A *base* that yields good parallel performance will have: (i) a small number of nodes to minimize the time in critical sections, and (ii) the resulting subtrees should be balanced (in their number of nodes) to assure good load balance.

We provide a generic way for parallelizing each type of tree operation. Usually *bottom-up* operations are Read/Write operations and are required to maintain tree properties. Generally, the height of the propagation (distance up from the leaf) is dynamically determined – as soon as the desired property is obtained, then the operation finishes. Propagations through (up or down) the base are atomic. Bottom-up operations on subtrees are local and independent. As soon as a bottom-up operation reaches the base, it locks it (selectively), finishes its processing, and then releases the locks. In many cases, only a small fraction of the bottom-up operations propagate information to the base. In our experimental results for randomized input data, the number of bottom-up operations that reach the base is very small, and grows extremely slowly with data size. *Top-down* operations start from the root and move down the tree until they reach a leaf (e.g., search operations). Concurrent top-down operations could create a bottleneck at the base if proper care is not taken. A naive implementation could start every operation from the root, which would make the base a hot-spot (even if the base were not locked, it would require broadcast of the base to all processors). In top-down search operations (as are the ma-



(a)



(b)

Fig. 6. (a) Speedup of parallel insertion of Red-Black trees (b) Execution time for parallel insertion in Red-Black trees for 1 million integers

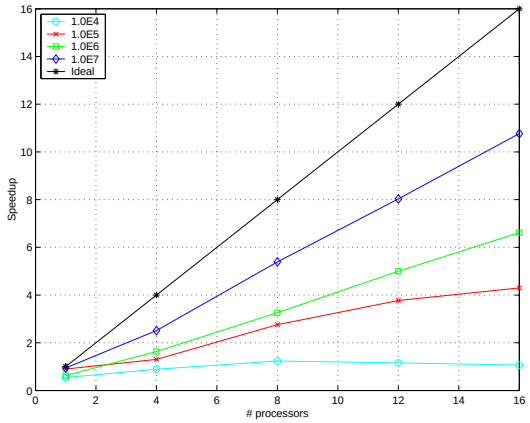


Fig. 7. Parallel Find on Integers

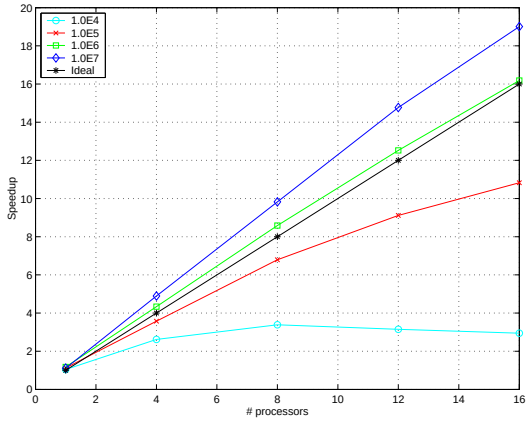


Fig. 8. Parallel Inner Product on Integers

jority of top down operations), it is not always necessary to start searching from the root. Instead, each processor could simply start searching from the roots if its subtrees, and avoid reading the base.

Typically, a parallel insertion of multiple elements into a pTree will consist of two or more fully parallel phases. Initially, each processor has multiple elements to insert into the tree (not necessarily its own subtree(s)). In the first parallel phase, each processor determines which of its original elements should be inserted into which subtrees; this requires that all processors know/read the boundary nodes of the pTree. In the next phase, the elements to be inserted into each subtree will be collected into buckets in the corresponding processors' memory. This involves a prefix computation of p elements for each bucket, and then a total exchange operation to place the elements in the correct buckets. The prefix computation may be executed sequentially or in parallel, depending on the number of processors and subtrees. In the final phase, which is executed in parallel, each processor will insert the elements in its buckets into its subtree(s). This phase requires no communication, but may involve atomic update of the base.

We used STAPL to parallelize the STL Red-Black tree. Figure 6(a) shows the speedup for parallel insertion in Red-Black trees. For these experiments the initial size of the tree was 50,000. We have chosen to start our parallel insertion in a tree which already has nodes in it, since we need to create a base first. Note that the performance is scalable as the work per processor remains constant. In figure 6(b) we show the raw execution times on different numbers of processors for the insertion of one million integers in a Red-Black tree.

B. STAPL Basic Algorithms

Figures 7 and 8 show the speedups obtained by STAPL's p_inner_product and p_find pAlgorithms over their (sequential) STL counterparts. The speedups reported for p_find are the average speedups for fifty runs, each run having the key in a different location of the input vector (to obtain a uniform distribution). If the key value is very close to the beginning of the first n/p elements (assuming block scheduling) then the STL algorithm will be faster than the STAPL one because `pfind` must wait for all processors to finish their search before gathering the results and returning the location. The closer the key value is to the end of the search space the greater the speedup of the parallel algorithm will be. We are implementing a faster version that can interrupt the parallel loop as soon as a key is found.

The speedups reported for p_inner_product are also the average of fifty executions. The algorithm is not input dependent, but multiple runs were done to ensure an accurate timing was obtained. Figure 8 shows the speedup obtained by the

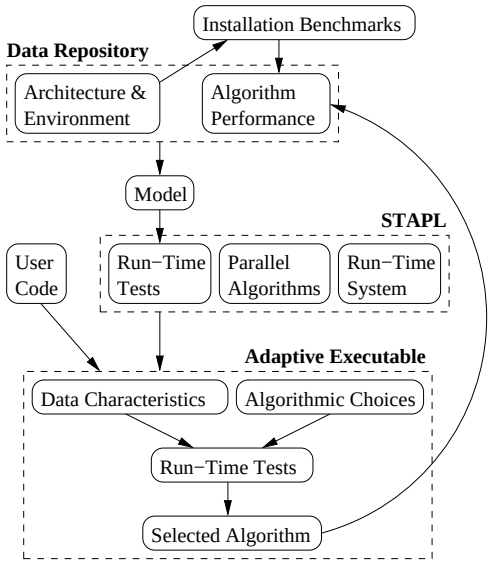


Fig. 9. Adaptive Framework

`p_inner_product` algorithm. Super-linear effects are due to cache effects. When data sizes are small it is not profitable to execute in parallel.

To evaluate the profitability of parallelization, each STAPL pAlgorithm has instrumentation code that checks data size (as an approximation of workload) and decides automatically when to use the STAPL version and when to use STL. The threshold data size for this decision is determined experimentally for each machine during STAPL's installation on the system.

C. Algorithm Adaptivity

Sequential computing benefits from the fact that the relative performance of several algorithms solving the same problem can be successfully modeled without the need to use detailed information about the environment or the problem instance. However, parallel computing greatly increases the sensitivity to these external influences, often making it less clear which approach to use for a given situation. Specifically, the best parallel algorithm to use often is sensitive to:

Architecture - processor to memory interconnection network, communication network, and available resources.

Environment - thread management and operating system policy (e.g., memory allocation and management, migration policies).

Data Characteristics - algorithmic sensitivity to input data type or layout.

To ensure maximum performance across a number of different architectures, environments and data characteristics, we must adapt our approach to the dynamic context of the problem. To this end, we have developed an adaptive framework, shown in Figure 9, to enable STAPL to select the best algorithm given the dynamic situation.

The framework begins by collecting statically available information about the architecture and environment, and storing it in a data repository to facilitate later analysis. It also requires installation benchmarks of various algorithms to collect performance characteristics. This dynamic testing can be aided by static information. For instance, if the cache sizes in the memory hierarchy are known, the data sizes to benchmark can be selected around cache boundaries.

Once this information has been collected, a model is generated to predict the best algorithm based on various execution parameters. This off-line analysis is used to generate simple and inexpensive run-time tests that are inserted into the STAPL functions. Coupling this information together with a user's code produces an adaptive executable. The executable can effectively select the most appropriate algorithm based on the dynamic data characteristics and available algorithms. The results of each execution can also be stored in the repository, allowing for periodic updates to the run-time tests for increased precision.

C.1 Application: Parallel Sorting

As an example of algorithm selection in STAPL, we have applied the adaptive framework to parallel sorting. Sorting is a fundamental function in a large number of codes, and STL contains a number of sort functions which STAPL parallelizes. We first discuss the specific framework implementation, then give an example of adaptive sort selection.

Implementation

The *data repository* is implemented as a PostgreSQL database. Information about architecture and environment, such as number of processors, the amount of main memory, and a list of the various cache levels and their respective sizes, is collected from header files and system calls. Algorithm performance includes information such as execution time, input parameters, data characteristics, and measures of statistical confidence.

For *installation benchmarks*, the number of processors and input size and data type are varied and tested for each the available algorithms. In addition, values of algorithm input parameters are varied where applicable.

We *model* the information in the database using Quinlan's ID3 algorithm to generate a decision tree model of the predictions inducted from the benchmarking runs [22]. This approach works well for sorting on the systems we have studied; however, we in general envision a toolbox of modeling approaches that allows us to choose among competing analytical techniques.

Sort	Strength	Weakness
Column	time optimal	many passes
Merge	low overhead	poor scalability
Radix	extremely fast	integers only
Sample	two passes	high overhead

Fig. 10. Parallel Sort Summary

The *run-time tests* are directly generated from the decision tree, and inserted as a series of nested if-then statements into the STAPL source code. We have implemented four shared memory parallel sorts from which to choose (See Fig. 10). Column sort is a version of one of the first parallel sorts to prove a $O(\log n)$ lower bound [19], [12]. Merge sort is an implementation of parallel odd-even merge sort [2]. Radix sort is a version of the commonly known linear time sequential sort for integers, implemented by parallel counting [5], [10]. Sample sort is a two pass technique that first samples the data to distribute it into discrete buckets, and then sorts each bucket independently [10], [14].

When STAPL, augmented with *run-time tests*, is compiled with a *user's code* (we use a simple program that generates random data and then sorts), an *adaptive executable* is produced. At run-time, the if-then tests are executed during a call to STAPL sort, and the best parallel sort is selected and executed.

Results

We first consider how to adaptively select sorts across different architectures. This experiment was run using 10 million random integers as input on three different machines. The V2200 is a Hewlett Packard system that contains 16 processors with 2MB L2 caches and 8 memory banks connected by a crossbar. The Power Challenge is an SGI system with 24 processors with 1MB L2 caches and a single shared memory connected by a bus. The SGI Origin 2000 is a DSM, configured as a hypercube, where each node contains 2 processors with 4MB L2 caches and a single memory bank. Speedup is measured as the sequential STL sort's execution time divided by a given parallel sort's time.

Figures 12, 13, and 14 demonstrate the sorts on the three systems. As expected, radix sort is the clear winner (we're sorting integers). However, given that many other data types are possible, we must also consider the other sorts' performance. Merge sort runs very well on 1-4 processors, but fails to scale well to higher number of processors. On the HP V2200, sample sort is the best choice for 5-16 processors, yet on the two SGI machines, column sort is the best. This difference is significant, with the correct choice affecting performance by up to 34%.

We also found that algorithmic scalability varied by machine. As seen in Figure 13, contention on the bus in the Power Challenge causes all the sorts to lose scalability at higher numbers of processors. Figure 14 shows a substantial loss of scalability for all sorts on the Origin 2000. This is due to the input not being properly distributed between processors, and is a problem we are addressing since it is not only limited to Origin systems.

Regardless of these machine specific features, the adaptive framework can still generate appropriate run-time tests for a specific machine. Our installation tests varied processor counts between two and eight, along with element counts ranging from 50,000 to 10 million. Ten decision tree creation iterations were run through our model on the V2200, each using a different 10 percent of the data to test a decision tree created with the remaining 90 percent.

```
if (data_type = INTEGER)
     radix_sort()
else if (num_procs < 5)
     merge_sort()
else sample_sort()
```

Fig. 11. Sample Run–Time Test

The experiment yielded an average tree accuracy of 99 percent, representing a single mis-prediction by one of the trees. Figure 11 shows code generated from a generated decision tree. This code represents exactly the adaptation desired to correctly fit the cases mentioned earlier.

Although this is a simple example, it demonstrates our framework's ability to generate inexpensive run-time tests based on offline analysis. For sorting, we are working to make our selection process more sophisticated by incorporating a run-time test

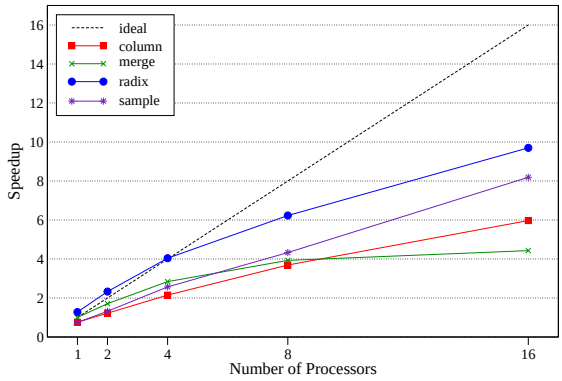


Fig. 12. Sorting on a HP V2200

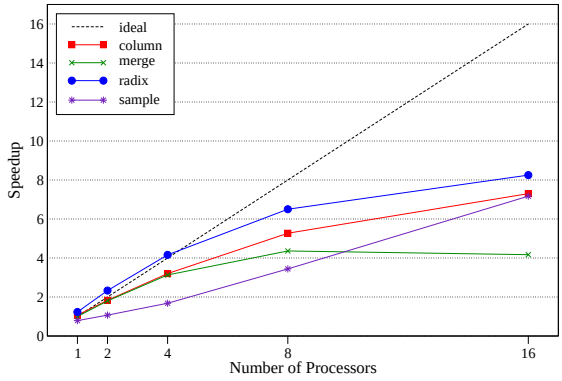


Fig. 13. Sorting on a SGI Power Challenge

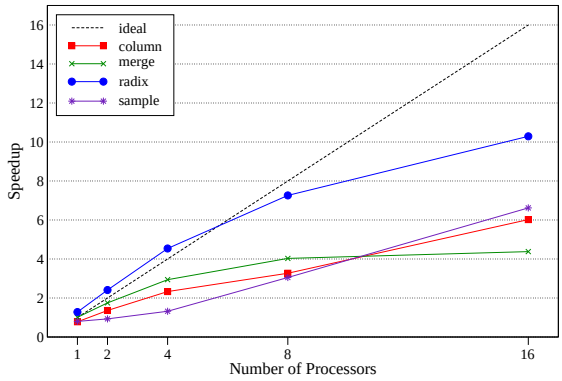


Fig. 14. Sorting on a SGI Origin 2000

of presortedness into the algorithm selection process. Our tests show that this data characteristic can greatly affect the performance of some of the comparison-based approaches.

Adaptive Parameter Selection

Once an algorithm has been selected for a given situation, it is often possible to further adapt by the selection of key parameters. One example is the number of bits to consider (r) on each pass of radix sort. Table 15 shows the optimal value of r for various ratios of number of elements (n) over number of processors (p). The discrete steps of 6, 8, and 11 correspond to radix sort using 6, 4, and 3 passes over the 32-bit integers. These steps in r correspond to overflows into lower levels of the memory hierarchy. Radix sort requires at least twice as much memory as the input, meaning 2MB are used for $250,000$ 4-byte integers

$(2 \times 4 \times 250k)$. Since this no longer fits into L2 cache, it makes sense that fewer iterations would yield better performance by producing fewer cache misses. A similar effect occurs at 4k elements.

n/p	Optimal r
1250	6
2500	6
5000	8
50,000	8
250,000	11
500,000	11
1,000,000	11

Fig. 15. Optimal Selection of r in Radix Sort

The performance gain of adaptively choosing r based on n/p as opposed to fixing r at a value of 8 yielded an average performance gain of 7%. We feel the benefit of the adaptive technique will increase even more on architectures with deeper memory hierarchies and steeper increases in latencies between levels.

As our results clearly indicate, the best sort varies across architecture and number of processors being used. Our framework provides a straightforward means of generating run-time tests to correctly select the best sort for a given situation. In addition, the framework can focus on a specific sort and produce additional tests, such as for the selection of r, to yield additional gain.

V. 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 a sequential C++ molecular dynamics application that has been an excellent test code for STAPL development.

References

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

[2] Selim Akl. *Parallel Sorting Algorithms*. Academic Press, Inc., 1985.

[3] 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.

[4] 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*.

[5] Nancy Amato, Ravishankar Iyer, Sharad Sundaresan, and Yan Wu. A comparison of parallel sorting algorithms on different architectures. Technical Report TR98-029, Department of Computer Science, Texas A&M University, January 1996.

[6] 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.

[7] 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.

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

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

[10] Guy Blelloch, Charles Leiserson, Bruce Maggs, Greg Plaxton, Stephen Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, 1991.

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

[12] Geeta Chaudhry, Thomas Cormen, and Leonard Wisniewski. Columnsort lives! An efficient out-of-core sorting program. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.

[13] 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.

[14] Andrea Dusseau, David Culler, Klaus Erik Schauser, and Richard Martin. Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, 1996.

[15] 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.

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

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

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

[19] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions On Computers*, C–34:318–325, 1985.

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

[21] C.G. Plaxton 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.

[22] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

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

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

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

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

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

[28] 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.