The STAPL pArray[*]

Gabriel Tanase, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger
Parasol Laboratory
Texas A&M University
College Station, TX, 7783
{gabrielt,bmm,amato,rwerger}@cs.tamu.edu

ABSTRACT

The Standard Template Adaptive Parallel Library (STAPL) is a parallel programming framework that extends C++ and STL with support for parallelism. STAPL provides parallel data structures (`pContainers`) and generic parallel algorithms (`pAlgorithms`), and a methodology for extending them to provide customized functionality. STAPL `pContainers` are thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They provide `views` as a generic means to access data that can be passed as input to generic `pAlgorithms`.

In this work, we present the STAPL `pArray`, the parallel equivalent of the sequential STL `valarray`, a fixed-size data structure optimized for storing and accessing data based on one-dimensional indices. We describe the `pArray` design and show how it can support a variety of underlying data distribution policies currently available in STAPL, such as blocked or blocked cyclic. We provide experimental results showing that `pAlgorithms` using the `pArray` scale well to more than 2,000 processors. We also provide results using different data distributions that illustrate that the performance of `pAlgorithms` and `pArray` methods is usually sensitive to the underlying data distribution, and moreover, that there is no one data distribution that performs best for all `pAlgorithms`, processor counts, or machines.

1. INTRODUCTION

Parallel programming is becoming mainstream due to the increased availability of multiprocessor and multicore architectures and the need to solve larger and more complex problems. To help programmers address the difficulties of parallel programming, we are developing the Standard Template Adaptive Parallel Library (STAPL) [2]. STAPL is a parallel C++ library with functionality similar to STL, the ANSI adopted C++ Standard Template Library [14]. STL is a collection of basic algorithms, containers and iterators that can be used as high-level building blocks for sequential applications. Similar to STL, STAPL provides a collection of parallel algorithms (`pAlgorithms`), parallel containers (`pContainers`), and `views` to abstract the data access in `pContainers`. These are building blocks for writing parallel programs. An important goal of STAPL is to provide a high productivity environment for the development of applications that can execute efficiently on a wide spectrum of parallel architectures, from shared memory to distributed memory.

In this work, we present the STAPL `pArray` container. The STAPL `pArray` is the parallel equivalent of the sequential STL `valarray`, a fixed-size data structure optimized for storing and accessing data based on one-dimensional indices and iterators. We describe the `pArray` design and describe how it can support a variety of underlying data distributions, such as blocked or blocked cyclic. A *data distribution* for a `pContainer` specifies how its elements are mapped to the memory of the underlying physical machine. Data distribution is an important issue in parallel computing because it has a major impact on the cost of accesses, i.e., a local access vs. a remote access requiring communication. The `pArray` follows the STAPL approach for data distributions for `pContainers` – it has a default data distribution manager designed to provide good performance for naive users, but it also allows more advanced programmers to specify particular distribution policies if desired. We include experimental results that show that `pAlgorithms` using the STAPL `pArray` scale well for large numbers of processors (more than 2,000). We also provide results obtained using data distribution policies currently supported in STAPL that illustrate that different `pAlgorithms` and `pArray` methods are more or less sensitive to the underlying data distribution, and perhaps more importantly, that there is no one data distribution that performs best for all `pAlgorithms`, processor counts, or machines.

2. RELATED WORK

There are several parallel languages and libraries that have similar goals as STAPL[12, 3, 4, 6, 8, 15]. The PSTL (Parallel Standard Template Library) project [10, 11] explored the same underlying philosophy as STAPL of extending the C++ STL for parallel programming. PSTL provided distributed arrays and associative containers with support for specifying data distributions (e.g.,*ContainerRatio*) and local and global iterators for data access. Unfortunately, the PSTL project is no longer maintained and its source code is not available

[*]This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, ACI-0326350, CRI-0551685, by the DOE and HP.

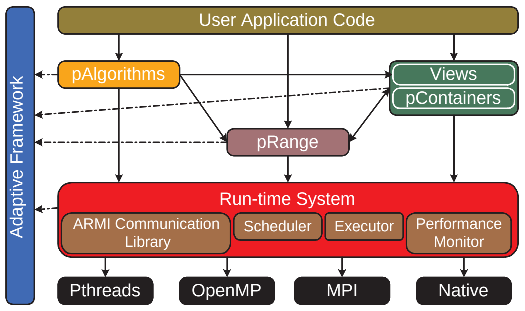

Figure 1: STAPL components

for comparison. STAPL differs from PSTL by providing an integrated framework for all `pContainers`, which also allows users to customize the default behavior, such as specifying different data distributions.

Intel Threading Building Blocks (TBB) [9] provide thread-safe containers such as vectors, queues and hashmaps for shared memory architectures. In the same spirit as STAPL, TBB provides arrays and associative containers, and the interfaces provided resemble those of STL containers. Our work is distinguished from TBB in that we target both shared and distributed systems, and it is a design choice in STAPL that all containers should provide both STL compatible interfaces and additional interfaces optimized for parallelism. Being exclusively for shared memory TBB, does not provide support for data distribution.

STAPL's objective is to provide an integrated framework for a wide range of data structures for parallel applications: array-based (e.g., `pArray`, `pVector`), associative [18] (e.g., `pMap`, `pHashMap`), and relational (e.g., `pList`, `pGraph`). There is a large body of related work in the field of distributed data structures, and many programming languages and libraries provide solutions for defining data structures and data distributions. Much work has been dedicated to array-based data structures, see, e.g., [16, 1, 7, 5]. Parallel programming languages typically supply constructs to specify how (multidimensional) arrays are mapped onto the underlying architecture.

3. STAPL OVERVIEW

STAPL consists of a set of components that include `pContainers`, `pAlgorithms`, `views`, `pRanges`, and a runtime system (see Figure 1). `pContainers`, the distributed counterpart of STL containers, are thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. While all `pContainers` provide *sequentially equivalent interfaces* that are compatible with the corresponding STL methods, individual `pContainers` may introduce additional methods to exploit the performance offered by parallelism and by the runtime system. `pContainers` have a *data distribution* manager that provides the programmer with a *shared object view* that presents a uniform access interface regardless of the physical location of the data. Thread-safety is guaranteed by providing mechanisms that guarantee all operations leave the `pContainer` in a consistent state. Important aspects of all STAPL components are *extendability* and *composability*, e.g., the `pContainers` implemented within the framework allow users to extend and specialize them, and to operate on `pContainers` of `pContainers`.

`pContainer` data can be accessed using `views`, which can be seen as generalizations of STL iterators, that represent sets of data elements and are not related to the data's physical location. `views` provide `iterators` to access single elements of `pContainers`. Generic parallel algorithms (`pAlgorithms`) are written in terms of `views`, similar to how STL algorithms are written in terms of iterators. The `pRange` is the STAPL concept used to represent a parallel computation. Intuitively, a `pRange` is a task graph where each task consists of a work function and a `view` representing the data on which the work function will be applied. The `pRange` provides support for specifying data dependencies between tasks that will be enforced during execution.

The runtime system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation [17]) provide the interface to the underlying operating system, native communication library and hardware architecture. ARMI uses the remote method invocation (RMI) communication mechanism among computing processes to hide the lower level implementations (e.g., MPI, OpenMP, etc.). A remote method invocation in STAPL can be blocking (`sync_rmi`) or non-blocking (`async_rmi`). ARMI implements several optimizations for improving performance, such as the aggregation of RMI requests to the same destination to amortize latency. For more details on ARMI please see [17, 19].

4. THE STAPL PARRAY

The STL `valarray` container is a fixed size data structure optimized for storing and accessing data based on one dimensional indices and iterators. The STAPL `pArray` is the parallel equivalent of the STL `valarray`, providing an efficient interface to access data elements using indices and views. An important property of the `pArray` is that it is a static data structure, i.e., the number of elements is known at instantiation and doesn't change during execution. As described in Section 4.2, this enables a number of optimizations such as closed form solutions for partitions and partition mappings.

4.1 pArray specification

The STAPL `pArray` provides the following generic specification (data types and methods):

Data Types:

- `value_type`: the type of elements stored in the `pArray`.
- `view_type`: the default view type used to access `pArray` elements.

STL **compatible methods:**

- `constructor(size_t _s [, value _val])`: Allocate a `pArray` of size `_s` and optionally initialize it with `_val`. Defaults are used for all aspects related to the data distribution.
- `reference operator[](size_t)`: Square bracket operator used to access the elements. Return value is a proxy class that behaves like a reference; this allows us to reference elements that are possibly remote.

- `p_array& operator+(const p_array& _other)`: Add the elements of the current `pArray` with the elements of `_other`. Similar for all arithmetic and logic operators.

STAPL **methods:**

- `value_type get_element(size_t _idx)`: return the value corresponding to the index `_idx`.
- `value_type set_element(size_t _idx, value_type _val)`: Set the value corresponding to the index `_idx` to be `_val`.
- `view_type get_view([domain][,partition])`: Return a `view` over the `pArray` data to be used when calling `pAlgorithms`. With no arguments, the view includes the entire container. When a domain is specified, the view includes the elements specified by it. Optionally, the user can provide a partition to specify how data will be grouped into blocks to be operated on in parallel.

The `pArray` provides additional methods that are not shown above due to space constraints. The `get_view()` method will be elaborated on in more detail in Section 4.2.

4.2 pArray design

The STAPL `pContainer` framework provides a set of base concepts and a common methodology for the development of thread-safe, concurrent data structures that are extendable and composable. STAPL provides basic data structures such as `pArray`, `pMap`, `pHashMap`, `pSet`, `pList`, `pGraph`, etc. The major concepts in the `pContainer` framework are *global identifier (GID), domain, distribution manager, partition, partition mapper, pContainer component,* and *views.* Below, we describe these modules for the `pArray`.

Global Identifiers (GIDs) and Domains: In the STAPL `pContainer` framework, each element is uniquely identified by its GID. This is an important requirement that allows us to provide a shared object view. The `pContainer` employs a *data distribution manager* to maintain a mapping from the GID to the location where the element is stored. For the `pArray`, the GIDs are the corresponding indices. The `pArray` *domain* is the universe of GIDs that identify its elements and is represented as an integer range corresponding to the indices of the elements (e.g., `Domain(0,10) or Domain(5,15)`). A domain also specifies an order that defines how elements are traversed by *iterators.* The order of a domain is specified by implementing two methods: `GID get_first_gid()` which returns the first GID/index of the domain and `GID get_next_gid(GID)` which returns the GID that follows the one provided as input to the method. `pArray` constructors accept a domain as an argument and the resulting index space and order of the elements will be as specified by the domain. For example `p_array<>(Domain(5,15))` declares a `pArray` whose first and last elements are `pa[5]` and `pa[14]`, respectively.

Data Distribution Manager: The Data Distribution manager is responsible for determining the location where an element associated with a GID is located. A *location* is a component of a parallel machine that has a contiguous memory address space and has associated execution capabilities (e.g., threads). A location can be identified with a process address space. Threads of one process executing within one location will share the memory associated with the location. In STAPL, we provide data distributions by using a `partition`, and a `partition-mapper`.

Partition: The `partition` is a policy class used to specify how a domain is decomposed into disjoint sub-domains. The main functionality provided by a `partition` is a mapping from a GID to the sub-domain that contains it. For example, for the `pArray` a user might use blocked or block cyclic partitions as in HPF [13].

```
1. typedef domain<int> Domain;
2. Domain dom(0,10);
3. partition_blocked  p_block(dom,block_size=5);
4. partition_balanced p_bal(dom, num_subdomains=2);
5. partition_explicit p_exp(dom,
6.                            Domain(0,5),
7.                            Domain(5,10));
8. //p_array with a block partition
9. p_array(p_block);
10. //p_array with a balanced partition
11. p_array(p_bal);
...
```

Figure 2: Specifying and using partitions of the domain [0,10).

Figure 2, shows how a user can specify different partitions for the domain $[0, 10)$. *Blocked partitions* (Figure 2, line 3) are specified using a block-size as an argument. Assuming N is the size of the domain to be partitioned this strategy will create $\lceil N/block\_size \rceil$ sub-domains of size `block_size`, except the last one which may be smaller. *Balanced partitions* (Figure 2, line 4) will divide the elements of the domain into the specified number of sub-domains, each of whose size is either $\lceil N/num\_sub\_domains \rceil$ or $\lfloor N/num\_sub\_domains \rfloor$. *Explicit partitions* (Figure 2, line 5) are built by explicitly enumerating the sub-domains. STAPL `pArrays` can be built with any of these partitions as depicted in Figure 2, lines 9,11. The `pArray` will associate with every sub-domain of a partition a `component` for data storage; currently, the components are implemented as STL `valarrays`. An important feature of STAPL is that the well-defined partition interface enables advanced users to implement their own partitions.

Partition Mapper: A partition is mapped onto a set of locations using a `partition-mapper`, which maps a sub-domain identifier (from 0 to $m-1$) to a location (from 0 to $L-1$). In this paper, we will consider two partition mappers currently available in STAPL: `cyclic_mapper`, where sub-domains are distributed cyclically among locations, and `blocked_mapper`, where m/L consecutive sub-domains are mapped in a single location.

Views: `views` are the means of accessing data elements stored in a `pArray` or any other STAPL `pContainer`. `pAlgorithms` in STAPL are written in terms of `views`, similar to how STL algorithms are written in terms of iterators. A `view` of a `pArray` is defined by an index domain, and a `partition` of that domain into `sub-views`. Elements included in a view or sub-view can be accessed using view iterators which are semantically equivalent to STL iterators. The partition of the `view` into `sub-views` is necessary for parallel processing; sub-views can be further partitioned to support nested parallelism. The `pArray` provides a default `view` that has the same partition and the same mapping as the data distribution of the `pArray` itself. Other `views` with different partitions and different access orders can be created using the `pArray get_view(Domain, Partition)` method. This method allows us to provide flexible parallel access to the entire `pArray` data or to sections of it.

```
1. value p_array::set(GID, value){
2.   D_ID = partition.map(GID)
3.   location = partition_mapper.map(D_ID)
4.   if location is local
5.     return component(D_ID).set(GID, value)
6.   else // set remotely the element
7.     return async_rmi(loc,&set(),GID,value);
```

Figure 3: Implementation of `pArray` `set()` method

An implementation of a `pArray` method is provided in Figure 3 to illustrate how the `pArray` modules interact. The runtime cost of the methods in the `pArray` interface has three constituents: the time needed to decide the location and the component in which the element is stored (Figure 3, lines 2-3), the communication time to get/send the required information (Figure 3, line 7), and the time it takes to perform the operation within a component, which is currently an STL `valarray` (Figure 3, line 5).

4.3 pAlgorithm and pArray Interaction

Generic `pAlgorithms` are written in terms of `views`. The partition of a `view` into `sub-views` impacts load balancing and data locality. Load balancing is impacted since the size of the `sub-views` dictates the amount of work in each task. Data locality is impacted since there is no guarantee that the elements in a `sub-view` are in the same location or contiguous in memory. Hence, there is a trade-off between the flexibility provided to the programmer by `views` and the potential performance loss when the view is not aligned with the data distribution.

In Figure 4, we show how `pArray`, `views` and `pAlgorithms` interact in STAPL. Fig. 4(a) shows a `pArray` with eight elements as seen by the user. Fig. 4(b) shows how the `pContainer` is distributed across two locations with a `blocked_partition` and `cyclic_mapper`. In Fig. 4(c) the default `view` (aligned with the data distribution) and the corresponding `sub-views` are depicted, and in Fig. 4(d) we show another `view` of the same `pArray` but with a different partition. Finally, we show a `p_find` algorithm executed on the `view` in Fig. 4(d). The `pAlgorithm` executes two sequential `find` algorithms in parallel, one on each `sub-view`, and a reduce operation to return the result to the caller. Note that two of the four elements of each `sub-view` are in remote locations and thus will require a remote method invocation, which is potentially expensive.

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the `pArray` container using a set of generic parallel algorithms and methods. We analyze the scalability of the `pAlgorithms` and look at how different data distributions impact performance on different systems.

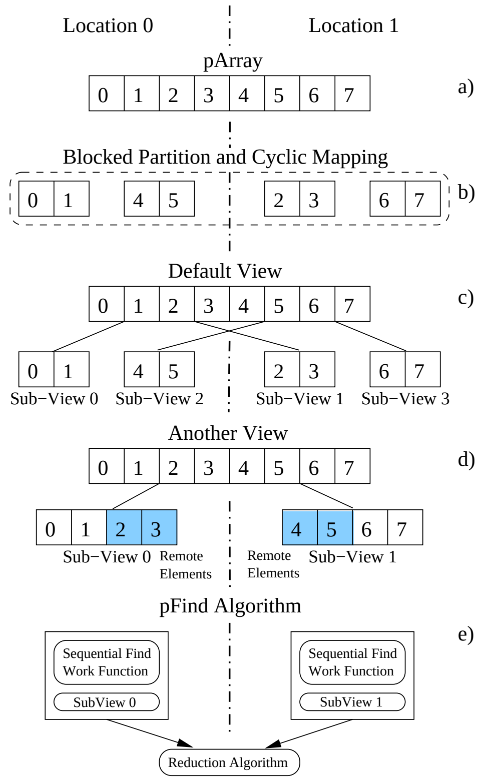

Figure 4: Example of data distribution, `views`, and `pAlgorithms`. a) the data structure as seen by the user; b) the data distributed in two locations and split in four blocks; c) the default `view` aligned with the data; d) another `view` with different distribution not aligned with the data; e) data dependence graph for `pfind` algorithm.

5.1 pAlgorithms

The `pAlgorithms` considered are `p_generate()`, `p_find()`, `p_accumulate()`, `p_for_each()`, and `p_copy()`. Also we analyze the performance of the `pArray set_element()` method. The functionality of the `pAlgorithms` included is similar to the corresponding STL algorithms but the interfaces are extended to accommodate STAPL concepts such as the `view`. We briefly describe the algorithms next:

- `p_generate(view, generator)`: Initialize all elements contained in the input view according to the provided generator. This is a simple doall type of `pAlgorithm` where every thread operates in parallel on different `sub-views`.

- `value p_for_each(view, functor)`: This algorithm is similar to `p_generate()`, but it applies the specified `functor` to all elements of the view.

- `iterator p_find(view, element)`: Searches for `element` in the `view` and returns an iterator pointing to the first occurrence of the element or the end of the `view` if the element is not found. The performance of `p_find` depends on where in the sequence the element is located, with the worst case being when the element is not in the sequence.

`- value p_accumulate(view, init_val):` Accumulate the elements of the view and add them to `init_val`.

`- p_copy(view_source,view_destination):` Copy elements from a source to a destination. `p_copy` works on two different views from possibly two different `pContainers`. The performance of it depends on the alignment of the underlying containers.

`- p_array::set_element(GID, _value) method:` This `pContainer` method sets the value of the element corresponding to the given GID (index) to be `_value`.

5.2 Architectures

An important goal of STAPL is to provide portable performance across different execution environments. To support this claim we performed our experiments on three different architectures. The first system, used for strong scaling studies, is a 6,656 processor IBM RS/6000 SP system operated by NERSC that consists of 416 SMP nodes, each with 16 Power3+ CPUs and where processors on each node have a shared memory pool of between 16 and 64 GBytes. The second system, referred to as IBM-CLUSTER, is an IBM HPC cluster consisting of 40 p5-575 nodes, each node with 8 Power5+ chips (dual core, 1.9GHz, 64-bit PowerPC architecture) and 32GB of memory per node. The third system, referred to as OPTERON-CLUSTER, is a 712-CPU Opteron (2.2 GHz) cluster running the Linux operating system and which is operated by NERSC. Processors are organized two on a node with 6GB of memory per node. The nodes are interconnected with a high-speed InfiniBand network. For all architectures we have used GNU GCC v4 and the O3 optimization level.

5.3 pAlgorithm performance using pArrays

A strong scaling study for `p_generate()`, `p_find()`, `p_accumulate()` and `p_for_each()` on an the IBM RS/6000 SP system is shown in Figure 5. A `pArray` with 400 million elements and a balanced partition is used, as shown in Figure 2, line 4, 10, 11. The number of sub-domains is equal to the number of processors available. The default view (aligned with the physical distribution) is used to access the `pArray` data. The number of processors is varied from 64 to 2048 and the scaling is computed relative to 64 processors; linear scaling is also shown on the figure for reference.

We observe that the `pAlgorithms` studied provide good scalability up to a large number of processors. `p_generate()` and `p_for_each()` incur no communication when the views are aligned with the physical distribution, while `p_find()` and `p_accumulate()` perform a simple reduction that does not impact overall performance for the particular input size considered. This study highlights the fact that STAPL and the `pArray` design provide the necessary infrastructure to deliver scalable `pAlgorithms` that can benefit from the large number of processors available in current and future high performance computers.

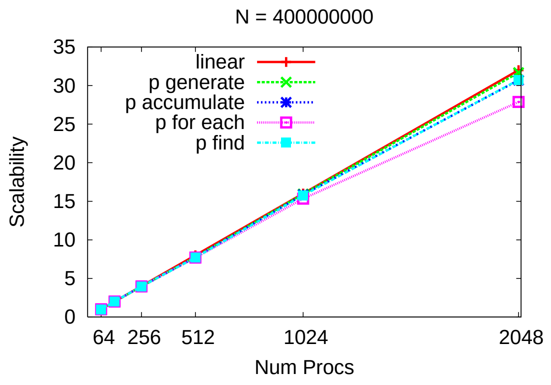

Figure 5: Strong scaling study (fixed problem size) of `pAlgorithm` performance using `pArray` containers on an IBM SP RS/6000. Note scaling is computed relative to 64 processors; linear scaling is shown for reference.

5.4 The impact of data distribution on pAlgorithms and pArray methods

In this section, we evaluate the performance of simple STAPL `pAlgorithms` when different distributions are used for the underlying `pArray`. Our findings show that no particular data distribution performs best in all situations, motivating the need for a flexible framework where different `pAlgorithms` can use different distributions [20].

```
stapl_main(argc, argv){
 partition_blocked<domain> p_block(n_elements,block);
 p_array<int> pc_a(p_block);
 p_array<int> pc_b(p_block);
 view_type view_a = pc_a.get_view();//default view
 view_type view_b = pc_b.get_view();
 // 1 - put random numbers in 'a'
 p_generate(view_a,gen_rand);
 // 2 - find _value in 'a'
 val = p_find(view_a, _value);
 // 3 - copy elements from view 'a' to 'b'
 p_copy(view_a, view_b);
 ...
}
```

Figure 6: STAPL code used to evaluate `pAlgorithms` performance.

Figure 6 shows the STAPL code used to evaluate the performance of the `p_generate()`, `p_find()`, and `p_copy()` algorithms and the `pArray` `set_element()` method. The `pArray` size considered was N = 100M (100 million) elements, the number of processors varied from 1 to 64, and the block sizes considered for the partitions were 2K (2000), 4K, 8K, 10K, 100K and 1M. For these experiments, we use the cyclic mapper where sub-domains are distributed cyclically among locations. Each experiment was repeated 3 times, and the execution times reported are averages.

A summary of the results is provided in Table 1 which shows the block size that leads to the best performance for a particular `pAlgorithm` and number of processors on the IBM-CLUSTER and OPTERON-CLUSTER systems. In some cases, a range of block sizes provided comparable perfor-

Algorithm	Number of Processors						
	1	2	4	8	16	32	64
IBM-CLUSTER							
p_generate	≥4K	≥4K	≥4K	≥4K	≤100K	≤100k	≤100k
p_find	1M	1M	1M	100K	100K	100K	100K
p_copy	≥4K	≥4K	≥4K	≥4K	≤100K	≤100k	≤100k
set_element	1M	1M	1M	100K	1M	1M	1M
OPTERON-CLUSTER							
p_generate	1M	1M	1M	100K	100K	100K	100K
p_find	1M	100K	100K	100K	100K	100K	100K
p_copy	1M	1M	1M	1M	100K	100K	100K
set_element	1M	1M	1M	1M	1M	1M	1M

Table 1: The best block size for a particular algorithm, platform and number of processors

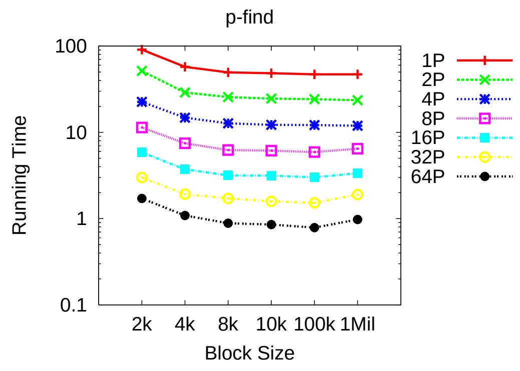

Figure 7: IBM-CLUSTER: **p_find 100M elts**

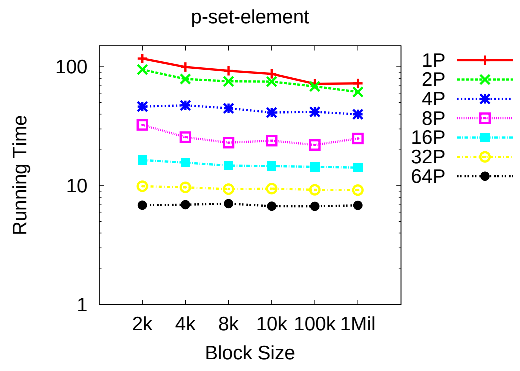

Figure 9: IBM-CLUSTER: **p_set_element 100M elts**

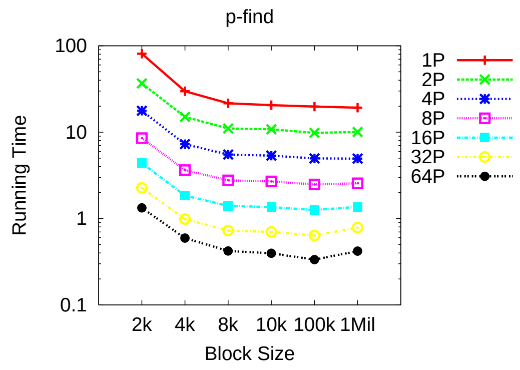

Figure 8: OPTERON-CLUSTER: **p_find 100M elts**

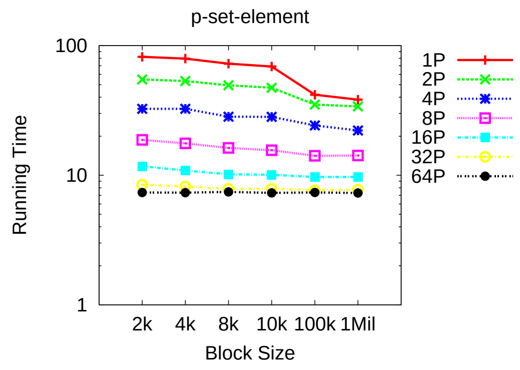

Figure 10: OPTERON-CLUSTER: **p_set_element 100M elts**

mance (e.g., `p_generate` performs similarly for any block size 4K or greater for one processor on the IBM-CLUSTER), while in others a particular block size performed best. In general, the best block size for each `pAlgorithm` depends on the machine and the number of processors used thus making it hard for a user to make the right decision about which distribution to use. To alleviate this problem, STAPL proposes an adaptive algorithm selection framework [20] to automate the decision making process. We can, however, make some general observations. For example, we can conclude that, for the `pAlgorithms` considered, small block sizes (e.g., 2K, 4K, or 8K) never provide superior performance to larger block sizes. The reason for this is that the `pContainer` allocates a sequential container (component) for each block specified by the partition and hence overhead and memory fragmentation increases as we increase the number of blocks.

In the following, we examine a few cases in more detail. For the graphs, we show times in seconds using a logarithmic scale on the y-axis, so small variations in the graph correspond to large variations in running times. In Figure 7 and Figure 8 we show the running times for `pfind` on IBM-CLUSTER and OPTERON-CLUSTER, respectively. We report the times for the worst case scenario when the el-

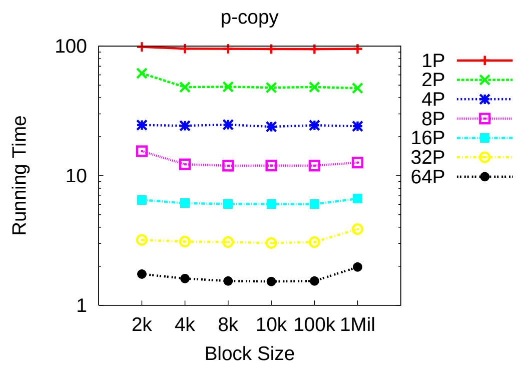

Figure 11: IBM-CLUSTER**: p_copy 100M elts**

ement sought is not in the `pArray`. For small numbers of processors, a large block size of 1M performs well, while for more processors (P≥ 8 on IBM-CLUSTER and P≥ 2 on OPTERON-CLUSTER), a smaller block size of 100K leads to better performance. In Figure 9 and Figure 10 we show the running times for the `set_element` method. For this test, each thread randomly generates N/P indices and sets the corresponding elements in a concurrent fashion generating an all-to-all like communication pattern. The scalability is not as good as for the `pAlgorithms` analyzed previously due to the amount of communication performed. The small performance improvement when going from one thread to two is because there is no communication overhead for one thread. Relative to two threads, the scalability is close to linear. Figure 11 shows that on IBM-CLUSTER the performance of `p_copy` is less sensitive to the block size and that a range of block sizes provides comparable performance, though that range does differ for different processor counts; similar behavior is observed for `p_generate`.

5.5 The impact of partition mapping on pAlgorithms performance

In this section, we evaluate the performance of STAPL `pcopy` when different blocked partitions and partition mappers are used for the underlying source and destination `pContainer`. The destination `pArray` has one sub-domain per location and we vary the number of sub-domains per location from 1 to 5 for the source `pArray`. In Figure 12(top) we depict the destination `pArray` with one sub-domain per location. A source `pArray` with two sub-domains per location and `blocked mapping` is shown in Figure 12(middle) while a `cyclic mapping` is shown in Figure 12(bottom). For the cyclic mapping, we observe that threads on location zero will copy the data corresponding to sub-domain zero locally into sub-domain zero of the destination, while the data corresponding to sub-domain three of the source will have to be copied remotely to location one, sub-domain one of the destination. When the blocked mapping is used, all data is copied locally.

Figure 13(a) shows execution times (seconds) for `pcopy` when a `blocked_mapping` is used for source, while Figure 13(b) shows the times when a `cyclic_mapping` is used. The number of processors has been varied from 1 to 64 and we include results only for OPTERON-CLUSTER, similar behavior being observed on IBM-CLUSTER. When the number of sub-domains per location increases, the cyclic mapper's performance suffers as the `pContainers` are no longer aligned and data has to be copied remotely. If advanced users specify their own partitions and mappings, then they have to pay attention to the impact of the decisions made.

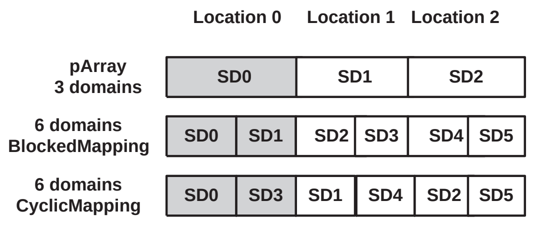

Figure 12: `pArrays` **with different partition mappers**

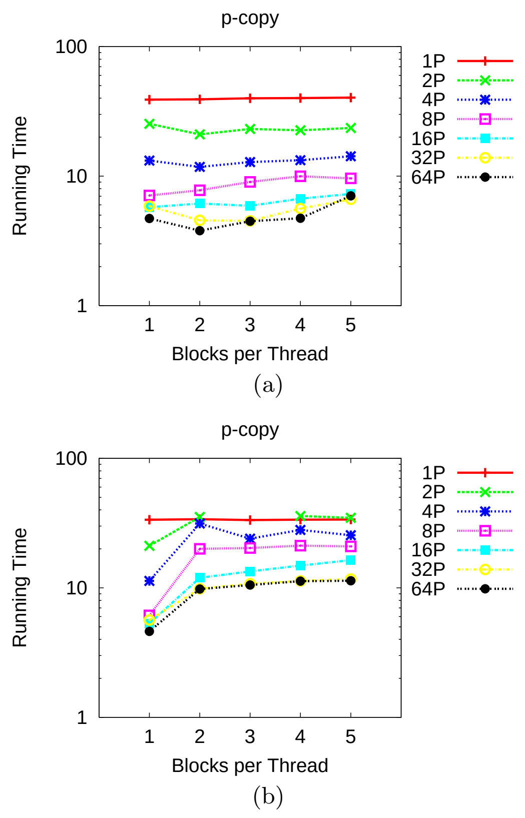

Figure 13: OPTERON-CLUSTER**: p_copy 100M elts (a) blocked mapper or (b) blocked cyclic mapper.**

6. CONCLUSION

In this paper, we have described the design and evaluated the performance of the STAPL `pArray` container. We included experimental results that show that `pAlgorithms` using the STAPL `pArray` provide good scalability up to a large number of processors and that different `pAlgorithms` and `pArray` methods benefit differently from various data distributions currently supported by STAPL. We are currently extending the STAPL framework to support more `pContainers`, partitions and data distributions.

7. REFERENCES

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The fortress language specification version 0.707. Technical report, Sun Labs Programming Language Research Group, http://research.sun.com/projects/plrg/fortress0707.pdf, 2005.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A standard template adaptive parallel C++ library. In *Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, Jul 2001.

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

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

[5] D. Callahan, Chamberlain, B.L., and H. Zima. The cascade high productivity language. In *The Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 26, pages 52–60, April 2004.

[6] A. Chan and F. Dehne. CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters, 2003.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[8] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40(10):423–437, 2005.

[9] Intel. *Intel. Reference for Intel Threading Building Blocks, version 1.0, April 2006.*

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

[11] E. Johnson and D. Gannon. HPC++: Experiments with the parallel standard library. In *International Conference on Supercomputing*, 1997.

[12] L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[13] D. Loveman. High performance fortran. *IEEE Parallel and Distributed Technology*, 1:25–42, 1993.

[14] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[15] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming in C++*, chapter 14, pages 547–588. MIT Press, 1996.

[16] H. Richardson. High performance fortran: history, overview and current developments. Technical Report TMC-261, Thinking Machines Corporation, April 1996.

[17] S. Saunders and L. Rauchwerger. ARMI: An adaptive, platform independent communication library. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, June 2003.

[18] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger. Associative parallel containers in STAPL. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Urbana-Champaign, to appear 2007.

[19] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger. ARMI: A high level communication library for STAPL. *Parallel Processing Letters*, 16(2):261–280, Jun 2006.

[20] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 277–288, 2005.