Design for Interoperability in STAPL: pMatrices and Linear Algebra Algorithms*

Antal A. Buss, Timmie G. Smith, Gabriel Tanase, Nathan L. Thomas, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science, Texas A&M University
{abuss,timmie,gabrielt,nthomas,bmm,amato,rwerger}@cs.tamu.edu

Abstract. The Standard Template Adaptive Parallel Library (STAPL) is a high-productivity parallel programming framework that extends C++ and STL with unified support for shared and distributed memory parallelism. STAPL provides distributed data structures (`pContainers`) and parallel algorithms (`pAlgorithms`) and a generic methodology for extending them to provide customized functionality. To improve productivity and performance, it is essential for STAPL to exploit third party libraries, including those developed in programming languages other than C++. In this paper we describe a methodology that enables third party libraries to be used with STAPL. This methodology allows a developer to specify when these specialized libraries can correctly be used, and provides mechanisms to transparently invoke them when appropriate. It also provides support for using STAPL `pAlgorithms` and `pContainers` in external codes. As a concrete example, we illustrate how third party libraries, namely BLAS and PBLAS, can be transparently embedded into STAPL to provide efficient linear algebra algorithms for the STAPL `pMatrix`, with negligible slowdown with respect to the optimized libraries themselves.

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) [1,15,16,18]. STAPL is a parallel C++ library with functionality similar to STL, the ANSI adopted C++ Standard Template Library [14] that provides a collection of basic algorithms, containers and iterators that can be used as high-level building blocks for sequential applications.

* This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, EIA-9810937, ACI-0326350, CRI-0551685, CNS-0615267, CCF 0702765, by the DOE and Intel. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Buss is supported in part by Colciencias/LASPAU (Fulbright) Fellowship.

J.N. Amaral (Ed.): LCPC 2008, LNCS 5335, pp. 304–315, 2008.

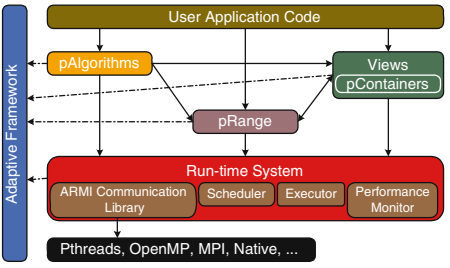


Fig. 1. STAPL software architecture

STAPL consists of a set of components that include `pContainers`, `pAlgorithms`, `views`, `pRanges`, and a run-time system (see Figure 1). `pContainers`, the distributed counterpart of STL containers, are provided to the users as shared memory, thread-safe, concurrent, extendable, and composable objects. `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 necessarily related to the data's physical location, e.g., a row-major view of a matrix that is stored in column major order. Generic parallel algorithms (`pAlgorithms`) are written in terms of `views`, similar to how STL algorithms are written in terms of iterators. Intuitively, `pAlgorithms` are expressed as task graphs (called `pRanges`), where each task consists of a work function and `views` representing the data on which the work function will be applied. STAPL relies on the run-time system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation [17]) to abstract the low-level hardware details of the specific architectures.

An important goal of STAPL is to provide a high productivity environment for developing applications that can execute efficiently on a wide spectrum of parallel and distributed systems. A key requirement for this is that STAPL must be interoperable with third party libraries and programs. First, STAPL programs must be able to take advantage of well known, trusted, highly optimized external libraries such as BLAS [13], PBLAS [5,6], LAPACK [2], parMETIS [12], etc. Second, it is equally important for a programming tool like STAPL to provide the ability to be used by other packages, including programs written in other languages (e.g., FORTRAN). For example, STAPL must provide interfaces to make `pAlgorithms` callable on third party data structures, such as FORTRAN/MPI distributed matrices. In this paper, we present a methodology for making STAPL interoperable with external libraries and programs. We describe how to *specialize* `pAlgorithms` to use third party libraries, and how to *invoke* `pAlgorithms` from other programming languages. Our strategy exploits unique features of `pContainers` and `views` that provide generic access to data and of `pAlgorithms` that allows them to be specialized to use optimized third party libraries when possible.

Since `pAlgorithms` are defined in terms of `views`, the same `pAlgorithm` can be used with multiple `pContainers` each with arbitrary physical distributions.

However, the genericity provided by `views` can complicate the use of third party libraries. For example, the PBLAS library [5,6] for matrix multiplication requires that the data be of a PBLAS recognized type (e.g., `float`) and laid out in local memory in contiguous chunks and in a block-cyclic manner across the processes executing the parallel application. Thus, a generic matrix multiplication `pAlgorithm` can only be specialized to use PBLAS if the data corresponding to the `pAlgorithm` input `views` can be shown to already satisfy these properties, or can be efficiently converted to do so. Our methodology for optimizing `pAlgorithm` performance is designed to address these problems. We do this by providing algorithm developers the tools to describe the problem input conditions under which tuned external libraries can legally be called and the mechanisms to properly invoke them.

We illustrate the process by showing how third party libraries, namely BLAS and PBLAS, can be transparently embedded into STAPL to provide efficient `pAlgorithms` for the `pMatrix`, a `pContainer` providing two-dimensional random access dense arrays with customizable data distributions. We also describe how `pAlgorithms` can be used by external codes. Our results on two architectures, a 640 processor IBM RS/6000 with dual Power5 processors and a 19,320 processor Cray XT4, show that generic `pAlgorithms` operating on `pMatrices` can be: (i) specialized to exploit BLAS and PBLAS and provide performance comparable to the optimized libraries themselves, and (ii) used by external codes with minimal overhead.

2 Related Work

The interoperability of software components and libraries is a broad field of study in software engineering. For the purposes of this paper, we will focus our discussion of related work to research covering library integration for high performance computing as well as recent work for library development and composition done within the context of generic programming in C++.

Breuer et al. [3] employ the current generic programming mechanisms in C++ to invoke external eigensolver routines on their distributed graph data structure. Specifically, their graph is mapped to the eigensolver's matrix concept via an adapter code module.

In [7], Edjlali et al. present Meta-Chaos to address the interoperability problem in a parallel environment. The approach is to define an application independent, linearized data layout that the various software components must use. It is useful for cases where the expected and actual layout of a data structure are very different.

Jarvi et al. [10] adapt the flood-fill algorithm in Adobe's generic image library to use the Boost graph library's sequential graph algorithms. This work uses a compiler that implements a prototype of the C++0x concept proposal [9]. They show that with minimal changes to either existing code base, concepts can adapt data structures from the image library to reuse generic Boost graph algorithms. This static (i.e., compile-time) adaptation incurs no runtime overhead.

Many active parallel language and library projects list interoperability as a key design goal, and some give the specification of calling conventions for the invocation of mainstream language (e.g., FORTRAN) libraries. However, it does not appear that much development effort has yet been spent demonstrating how the various projects' constructs can be used to promote code reuse and interoperability.

3 The pMatrix pContainer

In this section we briefly describe the `pMatrix`, a `pContainer` that implements a two-dimensional dense array. We concentrate on the aspects necessary to present our interoperability methodology. More details about the `pContainer` framework can be found in [15,16].

The declaration of a `pMatrix` requires the following template arguments:
`template <VType,Major=Row,BlocksMajor=Row,Partition=Default,Traits =Default > class p_matrix;`
The `VType` defines the type of elements stored in a `pMatrix`. A `pMatrix` is partitioned into sub-matrices (`components`) as specified by a `partition` class that defines in which sub-matrix each element of the `pMatrix` is stored. A total order over the elements of the matrix is defined by the `Major` and `BlocksMajor` types which specify the order among and within, respectively, the sub-matrices; the majors can be `Row` or `Column`.

Each sub-matrix is stored in a *location*, which is assumed to have execution capabilities, e.g., a location can be identified with a process address space. The mapping between sub-matrices and locations is performed by a `partition-mapper` (defined in the `Traits` template argument). Hence, together the `partition` and `partition-mapper` define a data distribution in STAPL. This can define a location/processor grid, as in [5]. For the `pMatrix`, we provide a generic *block cyclic* data distribution which can be customized to obtain other `partitions`, such as *blocked*, *blocked row-wise*, or *blocked column-wise*.

The `pContainer` framework is designed to allow the reuse of existing containers. This is supported by a component interface that can be implemented by `pContainer` developers as a light wrapper around their existing containers. For example, for `pMatrix` we support sub-matrices implemented as MTL [8], Blitz++ [19], or `malloc` allocated buffers.

The interface of the sub-matrix component must provide methods to allocate and manipulate the data and iterators to *natively* access the elements, which enables optimized data access for specialized `pAlgorithms`. Another interface requirement is to flag if the iteration order provided by the sub-matrix component iterators is the same as iterating through memory (e.g., `is_contiguously_stored =true/false`). This is necessary, for example, when the data of the sub-matrix component has to be passed to external libraries like BLAS.

A `view` over a `pMatrix` is a two-dimensional random access array typically representing an arbitrary sub-matrix of the `pMatrix`. A `view` can be partitioned into `sub-views` using a `partition` and a `partition-mapper`. A `sub-view` is

in every respect a `view`. The *default* `view` provided by a `pMatrix` matches the partition and the mapping of the `pMatrix` data. This view provides the most efficient data access since all the elements in a `sub-view` are in the same physical location. Starting from the default `view`, the user can obtain other `views`, such as `views` over a sub-matrix, `views` over rows (columns), `views` that provide the transpose, or `views` with an arbitrary block cyclic partition.

4 pAlgorithms

`pAlgorithms` in STAPL are specified as *task graphs*, where tasks are function objects (called *workfunctions*) that operate on `views` passed to them as arguments. Typically the workfunctions express the computation as operations on `iterators` over views. STAPL provides parallel versions of the STL algorithms, `pMatrix` algorithms, and `pGraph` algorithms, that operate on linearized `views`, `pMatrix` `views` and `pGraph` `views`, respectively.

Once the workfunction has been specified, the views passed as arguments may have to be rearranged to match the algorithmic requirements. This step is called `view` *alignment*. The workfunctions have *traits* that can specify requirements of the algorithm, e.g., the return type of the work function, the reduction operator to be used on it, the way the input `views` are accessed, such as read-only (`R`), write-only (`W`), or read-write (`RW`), etc. The latter traits are used by STAPL to allocate tasks to locations to improve performance.

STAPL provides support for querying and exploiting locality of `views`. Since a `view` describes a logical layout and partition of the data of a `pContainer`, accessing a data element through a `view` can involve a remote memory access which, in general, adds some overhead, even to local accesses. To mitigate this potential performance loss, STAPL can verify if a given `view` (i) is completely contained in a single address space, and (ii) if its iteration space is the same as the `pContainer`'s layout. Based on what conditions are matched, accesses to the data in the `views` can be optimized. For this reason, the `views` that are actually passed to the workfunctions may be of a type different than the one originally specified. For instance, it may be a `local_view`, whose data is in the local address space. This mechanism is exploited by the specializations described in the next section.

We now provide a `pAlgorithm` for matrix-matrix multiplication called `p_matrix_multiply_general` that given a view of an $m \times k$ matrix `A` and one of a $k \times n$ matrix `B`, computes $A \times B$ in the view of an $m \times n$ matrix C. The algorithm rearranges the input views corresponding to A, B, and C as blocked matrices, where blocks are compatible for matrix multiplication and accessed in column-major. The tasks of the algorithm perform the block-by-block multiplications necessary to compute the final result. A sketch of the workfunction is provided in Figure 2. The triple loop algorithm is written to access the views according to the view's major to exploit possible locality/access opportunities.

Note the `typedefs` in the class public interface that define the type of access performed in the `views`. Each `view` in the argument list has a corresponding

```
struct mat_mult_wf {
  typedef void result_type; // workfunction returns void
  // vA and vB are read-only, vC is write-only
  typedef access_list<R,R,W> view_access_types;

  template<class ViewA, class ViewB, class ViewC>
  void operator()(ViewA& vA, ViewB& vB, ViewC& vC) {

    // Triple loop rearranged to exploit possible locality
    for(size_t j = 0; j < cols_of_B; ++j)
      for(size_t k = 0; k < rows_of_B; ++k)
        for(size_t i = 0; i < rows_of_C; ++i)
          vC(i,j) += vA(i,k) * vB(k,j);
  }
};
```

Fig. 2. Example of workfunction for `p_mat_mul` algorithm showing traits for return type and access type of arguments (`R` for read-only, `W` for write only), and the templated function operator

flag indicating if it is read-only (`R`), write only (`W`), or read-write (`RW`). These access specifications are used to decide where the tasks will be executed, how the data will be accessed during the execution of the algorithm, and what types of optimizations can be applied.

5 Interoperability for Linear Algebra Computations

In this section, we describe our methodology for interoperability between STAPL's `pAlgorithms` and other libraries. Here, we apply it to parallel matrix multiplication; however, the approach is general and indicative of how other code bases interact with STAPL. We first look at how `p_matrix_multiply` transparently invokes PBLAS and BLAS routines when a set of compile time and runtime constraints are satisfied. We show how these constraints are specified and subsequently enforced. Finally, we show how an application not written in STAPL can invoke `p_matrix_multiply` through an interface provided by the `pMatrix`.

Algorithm 1. p_matrix_multiply(A, B, C)

```
1. if input conforms to PBLAS then
2.    call PBLAS
3. else
4.    decision =  redistribute | general
5.    if decision == redistribute then
6.       p_copy A,B,C to temporary PBLAS conformable storage
7.       call PBLAS
8.       p_copy temporary result to C
9.    else
10.      call general_matrix_multiply (use BLAS in sequential sections if possible)
11.   end if
12. end if
```

```
template<typename ViewMatA,typename ViewMatB,typename ViewMatC> void
p_matrix_multiply(ViewMatA& vA, ViewMatB& vB, ViewMatC& vC) {

  algorithm_impl::pdblas_conformable<ViewMatA, ViewMatB, ViewMatC> check_conformability;
  if (check_conformability<COLUMN_MAJOR, COLUMN_MAJOR, COLUMN_MAJOR>(vA, vB, vC) ||
      check_conformability<COLUMN_MAJOR, ROW_MAJOR, COLUMN_MAJOR>(vA, vB, vC) {
    p_matrix_mult_pblas(vA, vB, vC);
  } else {
    p_matrix_multiply_general(vA,vB,vC);
  }
}
```

Fig. 3. Specializing the p_matrix_multiply algorithm

5.1 Optimizing `p_matrix_multiply` with PBLAS and BLAS

Algorithm 1 shows pseudocode for `p_matrix_multiply`. First, the input is tested to determine if PBLAS can be called. If it cannot, the algorithm may employ an approach outlined in [18] to temporarily redistribute the data so that it is amenable to PBLAS invocation. This has been shown [6] to often be the best approach (assuming memory is available). Otherwise, STAPL's `p_matrix_multiply_general`, described in the previous section, is invoked and BLAS is used in serialized sections of the computation when the input conforms to BLAS interfaces.

Specializing the Parallel Computation. For brevity, we only show the test for invoking `pdgemm`, the PBLAS routine for double precision data. Specializations for other types follow the same approach. The conditions to use `pdgemm` are partially tested at compile time and partially at runtime. The conditions are the following: 1) the type of the data elements has to be `double`, 2) the input `views`, after aligning, have the property that each `sub-view` is contained in a single address space, 3) the `partition` of the `pMatrices` has to be block-cyclic and the majors of the blocks and within the blocks have to be the same, 4) the `partition-mappers` define a common computing grid for all the matrices, and 5) that the block parameters of the distributions make the three matrices distributions compatible with what `pdgemm` requires. The latter conditions depend on the type of the major of the matrices, and a proper invocation of `pdgemm` has to be picked up to cover the eight combinations of the transposition flags for A, B, and C.

Figure 3 shows the specialization of `p_matrix_multiply` to invoke PBLAS. The code shows only the condition checking for two specializations, one when the input `pMatrices` will be passed as-is to `pdgemm`, and another when the transposition flag for matrix B has to be set appropriately since B is stored row-major and `pdgemm` accepts by default column-major. `check_comformability` checks all the conditions listed above. The template parameters are used to select the proper set of checks for condition 5.

Specializing the Sequential Computation. For BLAS, we use a different approach than PBLAS for algorithm customization. There is still a runtime constraint for BLAS conformability. It shares the requirement with PBLAS that the subviews refer only to elements on the execution location (contiguous storage and traversal sequence requirements are checked statically as shown below).

The runtime check, however, is not explicitly located within the sequential workfunction but instead is implicitly performed by STAPL before each workfunction invocation. The workfunction invocation can determine the result of this test by checking whether the types of the views passed to it are `local_views` (see Section 4). This approach employs additional C++ language constructs to provide a more structured approach to incremental algorithm specialization. It allows new cases to be added over time without the need to modify existing code.

This relies on C++'s class template partial specialization mechanisms and the `enable_if` [11] template utility. Partial specialization allows one to define a *primary template* (i.e., the general matrix multiplication algorithm) which is used unless a template specialization is defined which is a better fit for the given template arguments (i.e., view types). `enable_if` allows us to specify when a specialization is appropriate to use based on *type traits*, by exploiting the *Substitution failure is not an error* (SFINAE) condition in C++.

```
struct matmul_wf {
  //function operator invoked by STAPL task graph
  template<typename VA, typename VB, typename VC>
  void operator()(VA& vA, VB& vB, VC& vC) {
    mat_mult_algorithm<VA, VB, VC> algorithm;
    algorithm(vA, vB, vC);
  }

  //Define nested class template struct mat_mult_algorithm
  //and specialize behavior as desired.

  //The generic (default) algorithm
  template<typename VA, typename VB, typename VC, typename Enable = void>
  struct mat_mult_algorithm {
    void operator()(VA& vA, VB& vB, VC& vC)
    { ... } // General algorithm
  };

  //1st specialization
  template<typename VA, typename VB, typename VC>
  struct mat_mult_algorithm<VA, VB, VC,
    typename enable_if<and_<dblas_capable_view_set<VA, VB, VC>
                            column_major<VA>, native_traveral<VA>,
                            column_major<VB>, native_traveral<VB>,
                            column_major<VC>, native_traveral<VC>
                          > >::type > {
    void operator()(VA& vA, VB& vB, VC& vC)
    { ... } // Setup matrix descriptors and CALL BLAS dgemm
  };

  //2nd specialization
  template<typename VA, typename VB, typename VC>
  struct mat_mult_algorithm<VA, VB, VC,
    typename enable_if<and_<dblas_capable_view_set<VA, VB, VC>
                            column_major<VA>, native_traveral<VA>,
                            column_major<VB>, transposed_native_traveral<VB>,
                            column_major<VC>, native_traveral<VC>
                          > >::type > {
    void operator()(VA& vA, VB& vB, VC& vC)
    { ... } // Setup matrix descriptors and CALL BLAS dgemm
  };
};
```

Fig. 4. Specializing the matrix multiplication workfunction to use BLAS

Note that our workfunction's function operator forwards the input views to a nested class template for execution. This is because only class templates (and not function templates) support partial specialization in C++. As with PBLAS specialization, the approach used to specialize the sequential matrix multiplication will also benefit from new concept features likely to be part of the next language standard.

Figure 4 shows the pseudo-code for BLAS specialization. `dblas_capable_view_set` checks, at compile time, if data is local and contiguous in memory. BLAS assumes matrices are stored in a column-major fashion. Hence, with these guarantees, it can be invoked directly with its standard parameters. The second specialization is similar, but handles the case when a column-major view of B represents a transposed traversal over the container's native order (i.e., the container is row-major). Here, the transposition flag must be set for matrix B when invoking the BLAS `dgemm` routine.

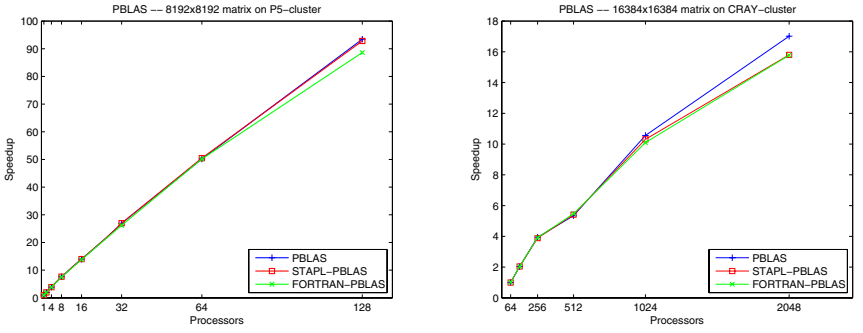
5.2 External Invocation of p_matrix_multiply

STAPL supports the use of `pAlgorithms` by applications that have been developed outside STAPL by providing a wrapper function for each `pAlgorithm`. The wrapper accepts pointers to the calling program's data instead of the `views` required by the `pAlgorithm` interface, and then constructs a `pContainer` for each argument using a special constructor that explicitly takes pre-allocated memory. The `views` obtained from these `pContainers` are then passed to the STAPL `pAlgorithm`. The `pAlgorithm` transparently accesses the external data through the `view` interface. When the `pAlgorithm` returns, the destructors of the `pContainers` do not attempt to free the memory since they are aware that it is externally managed, and control is passed back to the calling application.

6 Experimental Results

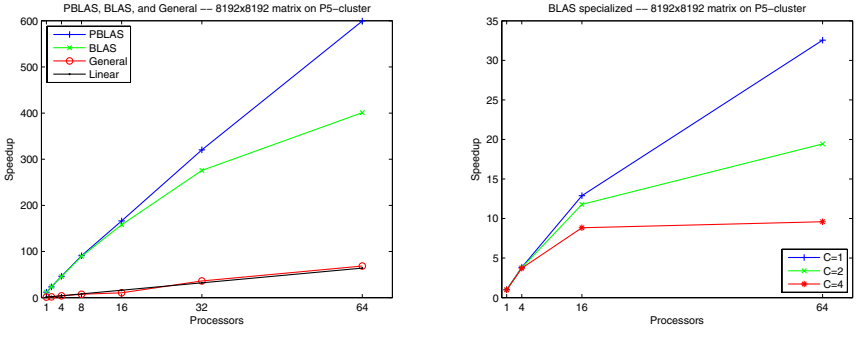
In this section, we present experimental results to evaluate the performance of our specialization methodology. We run experiments on two different architectures: a 640 processor IBM RS/6000 with dual Power5 processors available at Texas A&M University (called P5-CLUSTER), and a 19,320 processors Cray XT4 at NERSC (called CRAY-CLUSTER). More extensive results are available in [4].

The first set of experiments compares the performance of the STAPL `p_matrix_multiply` algorithm in the case where PBLAS specialization can be used (STAPL-PBLAS line) with a direct invocation of PBLAS (PBLAS line). We also show the performance of a FORTRAN/MPI program that allocates the data and invokes the STAPL matrix multiplication algorithm (FORTRAN-STAPL). The results are shown in Figure 5(a) for P5-CLUSTER, and Figure 5(b) for CRAY-CLUSTER. The plots show the speed-up with respect to running PBLAS sequentially on P5-CLUSTER, and with respect to 64 processors on CRAY-CLUSTER (to fit the large input in memory). The plots show that the overhead of the runtime check to determine if the PBLAS specialization can be invoked is negligible



(a) (b)

Fig. 5. Comparison of speed-ups for a direct PBLAS invocation (PBLAS), a PBLAS specialized STAPL `pAlgorithm` (STAPL-PBLAS), and a FORTRAN/MPI program invoking a STAPL `pAlgorithm` (FORTRAN-PBLAS). Results are shown for two architectures/data sizes: (a) P5-CLUSTER with 8192×8192 matrices and (b) CRAY-CLUSTER with 16384×16384 matrices. The baseline is direct PBLAS.



(a) (b)

Fig. 6. P5-CLUSTER with 8192×8192 matrices: (a) Speedups of the unspecialized (General) and BLAS specialized (BLAS) versions of the algorithm, and (b) speed-ups of the BLAS specialized workfunction when varying the number of `components` per location

up to a thousand processors. For larger processor counts, for which the actual multiplication takes less than a second, STAPL exhibits a visible overhead. The knee in Figure 5(b) arises since `pdgemm` does not scale as well if the number of processors is not a perfect square.

As mentioned in Section 3, the partition of a `pContainer` is decoupled from its actual distribution across the address spaces (locations) of the processes carrying out the computation (done by the `partition-mapper`). Thus, more than one `pContainer component` can be placed into a single address space. Since PBLAS forbids the use of such a data layout, when the number of sub-matrices is greater than the number of locations, then the runtime check for using PBLAS fails and the specialization for using BLAS within the workfunction is then tested.

Next we analyze the performance of the BLAS specialization. Figure 6(a) shows the speed-ups of the STAPL algorithm when no specialization is used (General line) and when the BLAS specialization is used (BLAS line). The baseline is the execution time of the matrix-matrix multiplication algorithm implemented in the workfunction of Figure 2. The performance of the unspecialized algorithm is dramatically slower than the BLAS specialization, showing that major performance gains are possible by using highly optimized third party libraries. The general algorithm starts exhibiting super-linear speed-up for 32 processors, when data fits in cache. The plots include the execution of PBLAS whenever the data layout allows us to use it, using the same baseline for the speed-up. It can be seen that the performance of the BLAS specialized algorithm is comparable to the PBLAS specialized algorithm, which is an interesting result from a productivity point of view.

Finally, Figure 6(b) shows the speed-up achieved by the BLAS specialized workfunction with respect to the execution of PBLAS on one processor. The plots report three experiments varying the number C of components per location (for $C = 1$ we forced the BLAS specialization to be invoked instead of PBLAS). We show results for perfect square processor grids, since this allows us to make fair comparisons. As can be seen, the speed-up decreases as the number of components increases. This is due to the increased number of memory copies and communications executed by the algorithm.

7 Conclusion

In this paper, we addressed the problem of interoperability in STAPL. We showed how STAPL can take advantage of third party parallel and sequential libraries by combining compile time and runtime checks. We illustrated the methodology by implementing a matrix-matrix multiplication algorithm that can exploit the availability of PBLAS and BLAS when the proper conditions are met. Our results show that the overhead of specialization is negligible, and, when proper specialization can be utilized, that STAPL performance is comparable to that of PBLAS. We also showed how STAPL can be used by other languages by providing the proper constructors for `pContainers` to embed foreign data structures within STAPL with negligible overhead.

References

1. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., Rauchwerger, L.: 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 (2001)
2. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. Society for Industrial and Applied Mathematics, 3rd edn. Philadelphia, PA (1999)

3. Breuer, A., Gottschling, P., Gregor, D., Lumsdaine, A.: Effecting parallel graph eigensolvers through library composition. In: 20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006, p. 8 (March 2006)
4. Buss, A.A., Smith, T.G., Tanase, G., Thomas, N.L., Olson, L., Fidel, A., Bianco, M., Amato, N.M., Rauchwerger, L.: Design for interoperability in STAPL: pMatrices and linear algebra algorithms. Technical Report TR08-003, Dept. of Computer Science, Texas A&M University (August. 2008)
5. Choi, J., Dongarra, J.J., Ostrouchov, L.S., Petitet, A.P., Walker, D.W., Whaley, R.C.: Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Scientific Programming 5(3), 173–184 (Fall, 1996)
6. Demmel, J., Dongarra, J., Parlett, B.N., Kahan, W., Gu, M., Bindel, D., Hida, Y., Li, X.S., Marques, O., Riedy, E.J., Vömel, C., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., Langou, J., Tomov, S.: Prospectus for the next lapack and scalapack libraries. In: PARA, pp. 11–23 (2006)
7. Edjlali, G., Sussman, A., Saltz, J.: Interoperability of data parallel runtime libraries with meta-chaos. Technical Report CS-TR-3633, University of Maryland (1996)
8. Gottschling, P., Wise, D.S., Adams, M.D.: Representation-transparent matrix algorithms with scalable performance. In: ICS 2007: Proceedings of the 21st annual international conference on Supercomputing, pp. 116–125. ACM, New York (2007)
9. Gregor, D., Stroustrup, B., Widman, J., Siek, J.: Proposed wording for concepts. technical report n2617=08-0127. ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++ (2008)
10. Järvi, J., Marcus, M., Smith, J.: Library composition and adaptation using c++ concepts. In: GPCE 2007: Proceedings of the 6th international conference on Generative programming and component engineering (October 2007)
11. Järvi, J., Willcock, J., Hinnant, J., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal 21, 25–32 (2003)
12. Karypis, G., Schloegel, K., Kumar, V.: ParMeTis: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0. University of Minnesota, Dept. of Computer Science (September 1999)
13. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for fortran usage. ACM Trans. Math. Softw. 5(3), 308–323 (1979)
14. Musser, D., Derge, G., Saini, A.: STL Tutorial and Reference Guide, 2nd edn. Addison-Wesley, Reading (2001)
15. Tanase, G., Bianco, M., Amato, N.M., Rauchwerger, L.: The STAPL pArray. In: Proceedings of the 2007 Workshop on Memory Performance (MEDEA), Brasov, Romania, pp. 73–80 (2007)
16. Tanase, G., Raman, C., Bianco, M., Amato, N.M., Rauchwerger, L.: Associative parallel containers in STAPL. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 156–171. Springer, Heidelberg (2008)
17. Thomas, N., Saunders, S., Smith, T., Tanase, G., Rauchwerger, L.: ARMI: A high level communication library for STAPL. Parallel Processing Letters 16(2), 261–280 (2006)
18. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Chicago, IL, USA, pp. 277–288. ACM, New York (2005)
19. Veldhuizen, T.L.: Arrays in blitz++. In: Caromel, D., Oldehoeft, R.R., Tholburn, M. (eds.) ISCOPE 1998. LNCS, vol. 1505, pp. 223–230. Springer, Heidelberg (1998)