The Execution Instance Overloading Pattern

Douglas Gregor and Andrew Lumsdaine
Open Systems Lab, Indiana University
Bloomington, IN 47405
{dgregor, lums}@osl.iu.edu

1 Intent

This pattern facilitates the creation of parallel libraries from sequential libraries, reusing the interface and implementation of the sequential library and providing near-seamless transition to the parallel library.

2 Motivation

Software libraries typically represent substantive investment into a single code base, often capturing the best-known practices, algorithms, and data structures for a given problem domain. Popular libraries become more than a collection of code; they become a language in which programs and algorithms can be expressed. Just as a parallelizing compiler can take program code written in a general-purpose, sequential programming language and produce a parallel program, a parallel library can implement the language of a sequential library to parallelize existing programs. The typical barriers to adoption of a parallel library—unfamiliar interfaces and the need to rewrite a large amount of application code—no longer apply or are lowered substantially.

The Execution Instance Overloading pattern facilitates the construction of parallel libraries that inherit the syntax, semantics, and implementation of sequential libraries. Execution Instance Overloading may be applied to algorithms or data structures (collectively called "components"), using three steps: *adapt* the sequential component with an interface-preserving instance of the Adapter pattern [9, pp. 139–150], *overload* the sequential component with the parallel one, and *optimize* cases where the combination of adapter and adaptee would produce suboptimal performance. For instance, consider parallelizing a function that computes the inner product of two vectors and has the following signature:

double inner_product(const vec_double& u, const vec_double& y);

A parallel library could implement a distributed vector type ***dist_vec_double*** as a collection of local ***vec_double*** instances, one per processor. A distributed ***inner_product()*** function could be implemented by first calling ***inner_product()*** on the local vector, then combining the results from all processors. Moreover, by using the following signature for the distributed inner product, function overloading will select the appropriate algorithm based on the type of the vectors involved:

double inner_product(const dist_vec_double& u, const dist_vec_double& y);

The intuition behind the Execution Instance Overloading pattern has been used in many libraries, including [1, 3, 17]. This paper provides a concrete pattern to aid this intuition and offers

implementation guidance for high-performance parallel libraries that integrate seamlessly with sequential libraries. Examples are drawn from the Parallel Boost Graph Library [11], a generic library for parallel and distributed computation on graphs. The Parallel BGL builds upon the interface and implementation of the Boost Graph Library (BGL) [18, 19], a generic, sequential graph library written in C++.

3 Applicability

The Execution Instance Overloading pattern can be applied when:

- The work an algorithm performs or the storage of a data structure can be divided into meaningful parts that can be distributed to different processors. For instance, a data structure may be stored in distributed memory.
- The sequential algorithm or data structure that is being parallelized can perform the majority of the work (or handle the storage) on each processor.
- There exists a way to combine the work or storage from the various processors into a semantically coherent algorithm or data structure.

4 Participants

The primary participants in the Execution Instance Overloading pattern are as follows:

- ***Adaptee***: the sequential data structure or algorithm that provides the interface and functionality that we want to introduce into the parallel library. It is also the functionality that will be used on individual processors.
- ***Execution instance***: defines the context in which a data structure will be used, e.g., sequential programs, processes using MPI, or processes using threads.
- ***Adapter***: the parallel or distributed component built on the Adaptee.

5 Implementation

Implementing Execution Instance Overloading requires three steps:

1. Construct an adapter that wraps a sequential component and provides a parallel (e.g., thread-safe, distributed, or both) analogue with an identical interface.
2. Overload the sequential interface to select between the sequential and parallel implementations as necessary. By "overload", we refer to any method of selecting the most appropriate implementation. Function overloading is one form of "overloading" we consider, but C++ class template partial specialization and run-time, factory-based techniques also meet our requirements.
3. Provide optimized parallel implementations when the combination of adapter and adaptee results in suboptimal performance.

This description of the implementation will focus on parallel and distributed data structures. The implementation techniques apply analogously to parallel and distributed algorithms.

```
template<typename ExecInst, typename Queue>
class distributed_queue {
public:
  typedef typename Queue::value_type value_type;
  void enqueue(const value_type& x) {
    if (owner(x) == exec.my_rank()) q.enqueue(x);
    else exec.send(owner(x), x, ...);
  }
  optional<value_type> dequeue() {
    while (exec.have_message()) { /* receive and enqueue elements */ }
    return q.dequeue();
  }
private:
  Queue q; // Local queue
  ExecInst exec; // Execution instance
};
```

Figure 1: Skeletal implementation of a distributed queue adapter.

5.1 Adaptation

The adapters in the Execution Instance Overloading pattern are a special case of the general Adapter pattern where the original interface and the target interface are the same. These adapters are therefore not merely syntactic transformations that route from one function to another, but instead adapt the semantics of these operations to a parallel environment. For instance, in a shared-memory environment the adapter may synchronize access to the underlying data structure by locking a mutex, whereas in a distributed-memory environment the adapter may send messages to other processors or perform remote method invocation. The behavior of a particular adapter should reflect the parallel environment and the semantics of the underlying interface.

Figure 1 illustrates the implementation of a simple distributed queue adapter. It is a C++ class template with two template parameters: one for the execution instance (***ExecInst***) and one for the type of the adaptee (***Queue***), which will be used as the local queue. The ***enqueue()*** operation queries the incoming element: if it is owned by the executing process, the element is placed in the local queue; otherwise, it is sent in a message to the actual owner. Conversely, ***dequeue()*** receives any messages and enqueues them locally first, then returns the result of ***dequeue()*** on the local queue. Figure 2 illustrates the wrapper architecture.

The distributed queue adapter works with any underlying queue, regardless of its concrete representation or semantics. We have opted to represent adapters as C++ class templates because this form of abstraction introduces no overhead; the pattern could also be implemented using C++ abstract classes or Java interfaces.

5.2 Overloading

Overloading refers to the automatic replacement of sequential components with their parallel counterparts. By overloading a sequential component with its parallel counterpart, the parallel component will be automatically selected when there is an appropriate execution instance, offering seamless parallelization. The effect is most obvious with algorithms in a library, where overloading

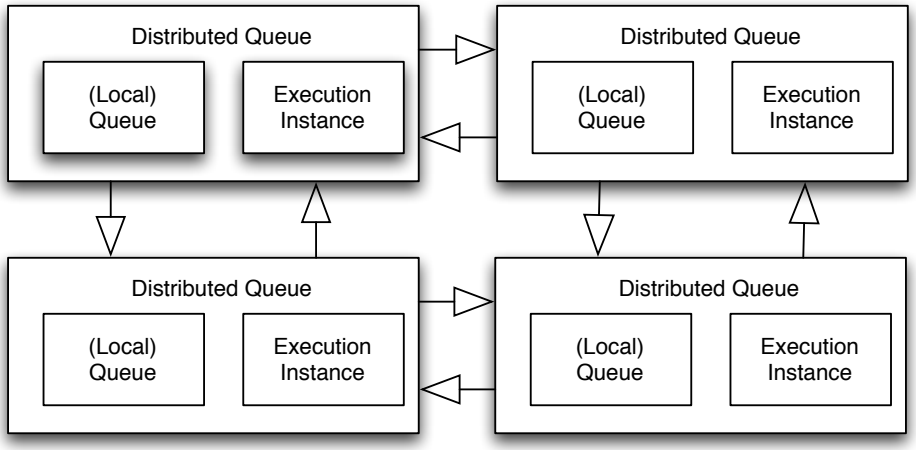

Figure 2: Distributed queue adapter

```
/* Sequential , local queue */
template<typename T>
class queue
{
public :
  typedef T value_type;

  void enqueue(const value_type &);
  optional<T> dequeue();
};
```

```
/* Overload for automatic distribution */
template<typename ExecInst, typename T>
class queue<distributed_value<ExecInst, T> >
  : public distributed_queue <ExecInst,
                local_queue <distributed_value <T> >
{
  // Inherits distributed operations
};
```

Figure 3: A generic ***queue*** class template along with a specialization that overloads uses of ***queue*** to use distributed queues when the underlying values are distributed.

is accomplished by function overloading. The following example illustrates how simple function overloading can be used to select between parallel and sequential versions of the same algorithm. User code written with a non-distributed ***graph*** that calls ***dijkstra_shortest_paths()*** need not be changed when the graph type is changed to ***distributed_graph***, because overload resolution will select the appropriate algorithm.

```
// Sequential version of shortest paths algorithm
void dijkstra_shortest_paths(const graph& g, graph::vertex start);
// Parallel version of shortest paths algorithm for distributed memory
void dijkstra_shortest_paths(const distributed_graph& g, distributed_graph::vertex start);
```

Overloading for data structures can be implemented in several ways, depending on the desired binding time. Data structures can be overloaded at run time with an implementation of either the Abstract Factory or Factory Method pattern [9]: the factory determines whether the underlying data structure is sufficient or whether an adapted, parallel version should be returned.

```
template<typename T>
class shared_queue<fifo_queue<T> >
{
  // lock−less queue implementation
};
```

Figure 4: An optimized, lock-free queue implementation will automatically be used when a FIFO queue is made thread-safe.

Figure 3 illustrates the interface to a typical ***queue*** class whose element type is the template parameter ***T***. The second definition of class template ***queue*** is a partial specialization that is used whenever the type ***T*** is actually a value that has been distributed over a particular execution instance, e.g., a vertex in a distributed graph. In this case, we know that the values have actually been distributed, so the class merely inherits the behavior of a distributed queue (see Figure 1). Thus, user code will receive a local queue for a local value type and a distributed queue for a distributed value type.

5.3 Optimization

In some cases the combination of an adapter and an underlying, sequential component results in suboptimal performance. For instance, an adapter ***shared_queue<Queue>*** could easily be written that makes a sequential FIFO queue thread-safe by locking a mutex in the ***enqueue()*** and ***dequeue()*** operations. However, the overhead of locking can be unacceptable, which could be remedied by using certain lock-less techniques [22]. C++ class template partial specialization can again be employed to recognize this particular case—a locking version of a shared queue—and instead provide a more efficient lock-free queue. Figure 4 illustrates one such partial specialization.

The ability to introduce optimizations without unnecessary overhead is crucial to the Execution Instance Overloading pattern. Without this ability, a parallel adapter would not be able to match the performance of a hand-coded parallel data structure, and a library of these parallel adapters would be abandoned in favor of hand-coded parallel libraries.

6 Examples

The parallel and distributed queue adapters presented in the previous section are simplified versions of the queue adapters used in the Parallel BGL. This section describes other examples of the Execution Instance Overloading pattern from the Parallel BGL.

6.1 Distributed Adjacency List

The distributed adjacency list in the Parallel BGL builds on the distributed adjacency list from the (sequential) BGL. The Execution Instance Overloading pattern applies in this case because a distributed adjacency list can be realized by dividing the vertex set V into disjoint subsets $V_1, V_2, \ldots, V_p$ and the edge set E into disjoint subsets $E_1, E_2, \ldots, E_p$, where $E_i = \{(u, v) | u \in V_i\}$.[1] A distributed adjacency list is implemented by placing the (non-induced) subgraph $G_i = (V_i, E_i)$ on processor i. Figure 5 illustrates the storage of the distributed adjacency list.

[1] For undirected graphs, both (u, v) and (v, u) will be stored in the distributed adjacency list.

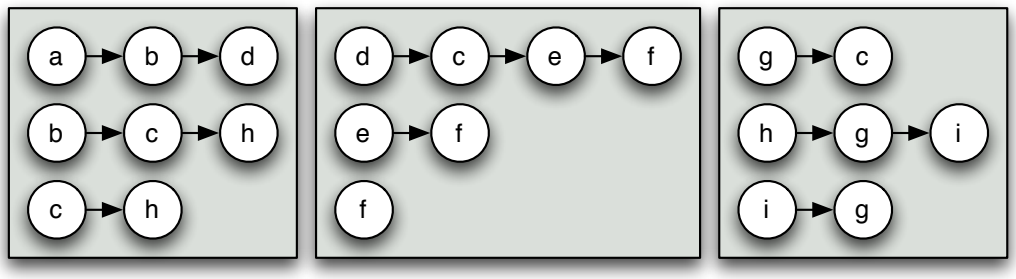

Figure 5: The storage of a distributed adjacency list: individual boxes represent the local adjacency lists on each processor.

The (sequential) BGL adjacency list is a highly-parameterized class template, permitting the user to specify the precise data structures it uses. The first two (of seven) template parameters specify the storage mechanism:

- ***OutEdgeListS***: A "selector" that describes the data structure that will be used to store the out-edges of each vertex. May be ***vecS*** (for vector storage), ***listS*** (for linked-list storage), etc.
- ***VertexListS***: A selector that describes the data structure that will be used to store the vertices in the graph.

Thus, ***adjacency_list<listS,listS>*** builds a graph type that stores vertices and edges in a list for fast insertion/removal of both whereas ***adjacency_list<vecS,vecS>*** permits fast iteration over vertices and edges and potentially reduces the amount of storage overhead for the graph. Although this parameterization introduces complexity for the implementation of the adjacency list, that complexity does not affect the ***distributed_adjacency_list*** adapter: the adapter need only forward its template parameters to the underlying adjacency list. The resulting distributed adjacency list is highly configurable, but the implementation of the adapter need only implement the communication layer.

The parameterization of the BGL adjacency list on its data structures permits an interesting way to implement the Overloading aspect of the pattern. In addition to the sequential selectors ***vecS***, ***listS***, etc., the Parallel BGL provides a templated selector ***distributedS<ProcessGroup,Selector>*** that indicates that the adjacency list should be distributed across the execution instance named ***ProcessGroup*** with local storage based on the selector ***Selector***. The BGL ***adjacency_list*** is then overloaded via C++ partial specialization as follows:

```
template<typename OutEdgeListS, typename ProcessGroup, typename VertexListS>
class adjacency_list<OutEdgeListS, distributedS<ProcessGroup, VertexListS> >
  : distributed_adjacency_list<ProcessGroup, OutEdgeListS, VertexListS>
{
  // ...
};
```

Thus, ***adjacency_list<listS, vecS>*** retains its former meaning as a local adjacency list but ***adjacency_list<listS, distributedS<mpi_bsp_process_group, vecS> >*** refers to an adjacency

list distributed across a particular execution instance[2], whose vertices are stored in a vector on each processor.

6.2 Distributed Property Maps

Most interesting graph algorithms deal with both the structure of the graph and with other attributes (or "properties") of the graph, e.g., the weight of an edge or the color of a vertex. The property map abstraction decouples *access* to a particular property for an edge or vertex from the *representation* of that property. A given property map ***pm*** has two primary operations:

- ***get(pm, key)***: Retrieves the property value of a given vertex or edge (described by ***key***) from the property map ***pm***.
- ***put(pm, key, value)***: Replaces the property value of a given vertex or edge (described by ***key***) in the property map ***pm***.

There are many implementations of property maps in the BGL, but all adhere to the same interface. Algorithms in the BGL are parameterized by the property maps they will operate on, e.g., the implementation of Dijkstra's algorithm accepts property maps for edge weights, distance to each vertex from the source, and predecessor of each vertex in the shortest paths tree.

Distributed property maps are a prime candidate for the Execution Instance Overloading pattern. The distributed property map will distribute the property values across the processors in the execution instance, but will still provide the ***get()*** and ***put()*** operations with similar semantics:

- ***get(pm, key)***: If the key represents a property value stored locally, returns that value. Otherwise, returns the value from a ghost cell. When no ghost cell is available, it is created with a suitable default, defined by the distributed property map.
- ***put(pm, key, value)***: If the key represents a property value stored locally, updates that value. Otherwise, a message containing the new value is sent to the owner of the key.

Local storage of properties can be achieved with a local property map. The distributed property map adapter handles communication, the storage of ghost cells, and reconciliation when multiple ***put()*** operations refer to the same key.

There are several sources of property maps in the BGL that require overloading for distributed property maps. The primary source of property maps is the graph itself: *interior* properties can be attached to vertices or edges in the graph and will be stored inside the graph data structure. Property maps for these properties are accessed using functions on the graph. The distributed graph adapter will therefore wrap the property maps returned from the underlying graph with the distributed property map adapter. The user always receives distributed property maps from distributed graphs.

The secondary source of property maps in the BGL is the property map library, which defines several kinds of property maps that adapt various existing data structures (vectors, deques, etc.) to the property map interface. These property maps permit *exterior* properties that are allocated and manipulated outside of the graph. The most widely used property map is the ***iterator_property_map***, which has the following interface:

[2]The instance in question is implemented over MPI [16] using the Bulk Synchronous Parallel (BSP) communication model [21].

```
template<typename RAIter, typename OffsetMap>
  class iterator_property_map
  {
    iterator_property_map(RAIter first, OffsetMap id);
  };
```

The ***iterator_property_map*** represents property values as an indexed set, using integer indices in the range $[0, n)$ where n is the number of keys. The i^{th} value is accessed using the C++ iterator syntax $*(first + i)$. An auxiliary property map, ***OffsetMap***, maps from keys to offsets. The offset map typically comes from a graph: the ***vertex_index*** property map for a graph ***g*** maps from the vertices of ***g*** to offsets whereas the ***edge_index*** property map maps from the edges to offsets. For instance, we can create a property map of vertex predecessors with:

```
Graph g;
std::vector<Vertex> preds_vec(num_vertices(g));
iterator_property_map<Vertex*, VertexIndex> pred(&preds_vec[0], get(vertex_index, g));
```

Even though the previous code segment makes no mention of the execution instance, the execution instance can be propagated to the ***iterator_property_map*** through the vertex and edge index property maps. The distributed graph adapter uses a special kind of property map, ***local_property_map***, for the index property maps. ***iterator_property_map*** can then be overloaded via C++ partial specialization to automatically adapt to a distributed execution instance as follows:

```
template<typename RAIter, typename ProcessGroup, typename LocalOffsetMap>
class iterator_property_map<RAIter,
                            local_property_map<ProcessGroup, LocalOffsetMap> >
  : distributed_property_map<ProcessGroup,
                             iterator_property_map<RAIter, LocalOffsetMap> >
{
  // ...
};
```

Note how the recursion unravels in the type system: ***iterator_property_map*** is provided with an iterator referring to storage for the property map and a ***local_property_map*** with an embedded execution instance. It then inherits its implementation from a distributed property map communicating via that execution instance, whose property map for local storage is again an ***iterator_property_map***, but this property map is restricted to the local storage.

The Parallel BGL applies the same implementation of the Execution Instance Overloading pattern to each of the property map types from the BGL. In each case, the execution instance instilled in the distributed graph type is used to provide distributed property maps for both interior and exterior properties.

7 Effect of overloading on users

The use of C++ template (partial) specialization for overloading permits greater integration of the parallel library with the sequential library than alternative approaches. Only when introducing a new execution instance does the user need to change existing code to parallelize it, and in many cases this need only occur in one place: the definition of a data structure or invocation of an algorithm. The following (complete) example program illustrates the effect of overloading on users of the Parallel BGL:

```
typedef adjacency_list<listS,
                       VertexListS, // See below
                       directedS,
                       no_property, // Vertex properties
                       property<edge_weight_t, double> // Edge properties
                       > Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;

Graph g(first_edge, last_edge, weights, num_nodes);

// Keeps track of the predecessor of each vertex
std::vector<Vertex> p(num_vertices(g));
// Keeps track of the distance to each vertex
std::vector<double> d(num_vertices(g));

Vertex s = vertex(0, g);
dijkstra_shortest_paths
  (g, s,
   predecessor_map(
     make_iterator_property_map(p.begin(), get(vertex_index, g))).
   distance_map(
     make_iterator_property_map(d.begin(), get(vertex_index, g)))
   );
```

In this example, three different instances of the Execution Instance Overloading pattern are used:

- The ***adjacency_list*** may or may not be distributed, depending on the (user-determined) type ***VertexListS***; see Section 6.1. This is the only place that the user needs to customize the program.
- The ***iterator_property_map***s generated by the calls to ***make_iterator_property_map()*** will be either sequential or distributed depending on the graph itself; see Section 6.2.
- The call to ***dijkstra_shortest_paths()*** will apply the sequential Dijkstra's algorithm (if the graph is not distributed) or the distributed Dijkstra's algorithm due to Crauser et al. [7] (if the graph is distributed).

8 Performance evaluation

The implementation of the Execution Instance Overloading pattern introduces several additional layers of abstraction to generate a parallel data structure or algorithm from a sequential one. There is a common misconception that introducing additional layers of abstraction necessarily degrades performance, scalability, or both: this is not the case. Certain types of abstractions—most notably, the widespread use of virtual function calls—do in fact have a nontrivial effect on performance. Figure 6 illustrates the performance of the sequential BGL and the Parallel BGL relative to other graph libraries (LEDA [15] for sequential results, CGM*graph* [6] for parallel results), which use some of these more-costly abstractions. In the sequential case, we see that the BGL outperforms LEDA by 6–12 times, depending on the compiler and size of the problem. In the parallel case, the Parallel BGL outperforms CGM*graph* by 27–47 times, primarily because CGM*graph* uses virtual functions in its abstraction of the communication layer.

Figure 6 illustrates that the abstractions of the BGL and Parallel BGL are significantly less costly than similar abstractions in other publicly-available libraries. In a previous study of the

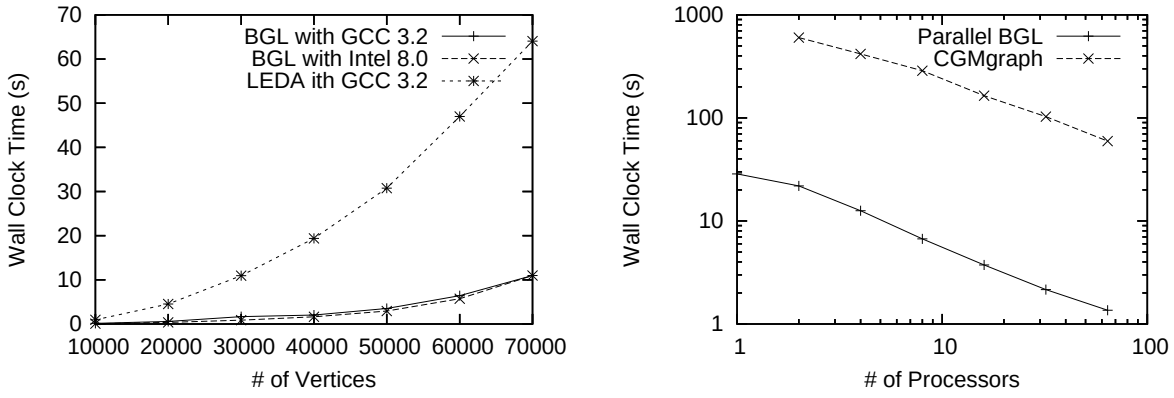

Figure 6: Performance of the sequential BGL relative to the sequential LEDA library and the Parallel BGL relative to the parallel CGM*graph* library, for connected components.

performance of the BGL [13], it was found to be competitive with hand-optimized Fortran code for sparse matrix ordering.

The Parallel BGL contains implementations of several graph algorithms [7, 8, 10]. Figure 7 illustrates the performance of these algorithms on undirected Erdös-Renyi graphs with $100k$ vertices and $15M$ edges. Although we have not yet compared the Parallel BGL to existing, hand-optimized parallel graph libraries (due to lack of availability), the fact that it is built upon and uses the same abstractions as the BGL leads us to believe that it will perform favorably. Moreover, the single-processor results of several of the Parallel BGL algorithms (especially Minimum Spanning Tree and Connected Components) reduce trivially to calls to the sequential versions of those algorithms, with comparable performance. Additional performance data is available on the Parallel BGL web page at `http://www.osl.iu.edu/research/pbgl`.

We executed our parallel performance tests on AVIDD, which consists of two identical clusters each with 96 compute nodes. Each node contains 2.5GB main memory with a 400Mhz front-side bus and two 2.4GHz Prestonia processors, but for our tests we have left one processor idle on each node. The nodes are connected via Myrinet 2000-C cards (one per node) in 100 MHz PCI-X slots and a Myrinet M3-E128 switch. The Parallel BGL tests were compiled using Boost 1.32.0 [4] (containing the sequential BGL) and the current development version of the Parallel BGL. All programs were compiled with version 3.3.1 of the GNU C++ compiler using optimization level $-\boldsymbol{O3}$ and LAM/MPI 7.1.1 [5] with version 1.6.5 of the ***GM*** driver.

9 Known Uses

Several parallel libraries have patterned their interfaces after popular sequential libraries. The following libraries follow this pattern to some degree.

- ScaLAPACK [3] is a library for linear algebra computation with distributed memory. The library resembles LAPACK [2] as closely as possible and is implemented on top of LAPACK. Full overloading is not implemented in ScaLAPACK; instead, ScaLAPACK provides routines with the same name and signature as in LAPACK, but prefixes the name with a "P".

- PBLAS is another parallel library associated with ScaLAPACK that implements the interfaces

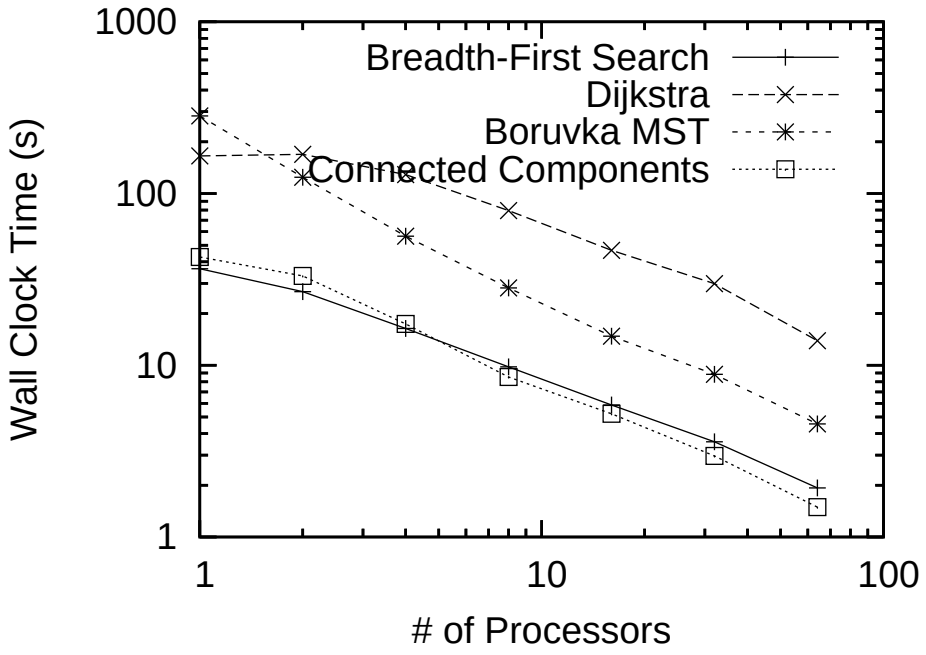

Figure 7: Performance of the Parallel BGL on a fixed graph size.

of the Basic Linear Algorithm Subroutines (BLAS) library [12] and uses the native BLAS library for the majority of its computation. PBLAS adopts the same overloading scheme as ScaLAPACK, prefixing each parallel routine name with a "P".

- The Standard Template Adaptive Parallel Library (STAPL) [1] is a parallel library that implements the data structures and algorithms of the C++ Standard Template Library (STL) [20]. Overloading in STAPL requires using a preprocessor to transform calls to STL algorithms into calls to the equivalent STAPL algorithms.

- The Parallel Boost Graph Library [11] is a library for parallel and distributed computation on graphs. It implements the Execution Instance Overloading pattern exactly as specified here, building on the sequential Boost Graph Library [18, 19].

10 Related Patterns

- **Adapter** [9, pp. 139–150]: The Execution Instance Overloading pattern uses a special case of the Adapter pattern. While an Adapter typically wraps an object to give it a different interface, the Execution Instance Overloading pattern wraps an object to give it the same interface with parallel or distributed semantics.

- **Proxy** [9, pp. 207–217]: A distributed adapter created with the Execution Instance Overloading pattern acts like a Proxy for the remote instances of the adapter. Execution Instance Overloading builds on this proxy notion to include overloading, efficient implementation, and optimization.

- **Distributed Array** [14, §5.10]: The Distributed Array pattern is a specific instance of the Execution Instance Overloading pattern, focusing solely on arrays and in particular the distribution of arrays. The Distributed Array pattern, along with the Shared Data [14, §5.8] and Shared Queue [14, §5.9] patterns, can clearly be helpful when constructing the adapter and optimized implementations for the Execution Instance Overloading pattern.

11 Conclusion

The Execution Instance Overloading pattern aids in the development of parallel libraries that reuse the implementation and interface of existing sequential libraries, then integrate with the sequential library to support a near-seamless transition to parallel programs. Through examples taken from the Parallel BGL, a parallel and distributed graph library built on the (sequential) BGL, we have illustrated the use of this pattern. Finally, although this pattern relies heavily on abstractions, performance does not necessarily degrade when the pattern is used: the Parallel BGL performs and scales well despite the heavy use of abstractions.

References

[1] 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 *Int. Wkshp on Adv. Compiler Technology for High Perf. and Embedded Processors*, page 10, July 2001.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, editors. *LAPACK User's Guide*. SIAM, 3rd edition, 199.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: a linear algebra library for message-passing computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997)*, 1997.

[4] Boost. *Boost C++ Libraries*. `http://www.boost.org/`.

[5] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In J. W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[6] A. Chan and F. Dehne. CGM*graph*/CGM*lib*: Implementing and testing CGM graph algorithms on PC clusters. In *PVM/MPI*, pages 117–125, 2003.

[7] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science (MFCS)*, volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. Springer, 1998.

[8] F. Dehne and S. Götz. Practical parallel algorithms for minimum spanning trees. In *Symposium on Reliable Distributed Systems*, pages 366–371, 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[10] S. Goddard, S. Kumar, and J. F. Prins. Connected components algorithms for mesh connected parallel computers. In S. N. Bhatt, editor, *Parallel Algorithms*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–58. American Mathematical Society, 1997.

[11] D. Gregor, N. Edmonds, B. Barrett, and A. Lumsdaine. The Parallel Boost Graph Library. `http://www.osl.iu.edu/research/pbgl`, 2005.

[12] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[13] L.-Q. Lee, J. Siek, and A. Lumsdaine. Generic graph algorithms for sparse matrix ordering. In *ISCOPE'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[14] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[15] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[16] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[17] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349. ACM Press, 1994.

[18] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[19] J. Siek, A. Lumsdaine, and L.-Q. Lee. *Boost Graph Library*. Boost, 2001. `http://www.boost.org/libs/graph/doc/index.html`.

[20] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[21] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[22] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1996.