An Object-Oriented Class Library for Scalable Parallel Heuristic Search

Wing-cheong Lau[1] and Vineet Singh[2]

[1] ECE Dept., Univ. of Texas, Austin, TX 78712, USA
[2] HP Labs, PO Box 10490, Palo Alto, CA 94303, USA

Abstract. In this report, we describe the design and implementation of a Parallel Heuristic Search Class Library, with emphasis on the use of object-oriented programming techniques for distributed applications. Another key consideration is the scalability of the library. We present the rationale behind major design decisions as well as the adoption of particular programming conventions. Along with an outline of the general framework of the library, we discuss implementation details of individual library classes. To demonstrate the applicability of the class library, it was used to port two existing parallel applications to an object-oriented platform. Different software metrics regarding the porting projects are used for evaluating library usability.

1 Introduction

In this report, we discuss the design and implementation of an object-oriented parallel heuristic search library. Heuristic search is a widely-used technique for solving problems which cannot be tackled through direct, algorithmic means. Typical applications include optimization, scheduling, artificial intelligence as well as VLSI design, with domains spanning various engineering and applied science disciplines.

In the search approach, the solution space of the problem is organized in the form of a decision graph, usually a tree. The task of solving the problem is therefore transformed to the search for a solution path in the corresponding decision graph (tree). This path-finding process is performed through a systematic graph traversal, in which an objective function is evaluated upon the arrival at every node to check if the solution has been reached. Usually, application-specific heuristics are used to steer the graph traversal process by prioritizing different candidate paths. Owing to the wide applicability and computation-intensive nature of heuristic search, various parallel formulations of heuristic search have been proposed [11]. The design of our library is based upon the formulation by Kumar and Rao in [16]. A brief description of this formulation is as follows: The sequential heuristic search is parallelized by dynamically partitioning the problem search-space into disjoint portions that are then distributed across individual processing entities (PE). Upon the receipt of the assigned search-space, a PE can work on it with minimal interprocess communications (except the infrequent exchange of information about the global state of the search process). Whenever a PE exhausts its own search space, it asks another PE for more work. The latter, upon executing the request, splits off part of its unexplored search space (if any) and sends it to the requester. Once a PE finds the solution, it notifies the other PEs and they respond by terminating their search. At

the beginning, the entire search space is assigned to a single PE while others have a null search space. Through the requesting process described before, the work-load is subsequently divided amongst PEs and eventually diffuses to all of them.

Below, we will describe the design of an object-oriented heuristic parallel search library based on this formulation. Two applications, VLSI Floorplan Optimization [2] and Automatic Test Pattern Generation [1], have been built by using this library in order to demonstrate its usability and practicality. Organization of this paper is as follows: Section 2 states the design goals and requirements of the library. Section 3 contains a brief description of ESP, the software programming system on which the library is built. The application of various object-oriented techniques for concurrent processing is discussed in Section 4. Section 5 gives the overall structure of the library and the specifications of the individual components. In order to demonstrate the usability of the library, two existing parallel applications were ported to ESP using the library. Experience and results about the porting process are described in Section 6, which also includes a brief evaluation of the Class Library. A retrospect of the overall library design is given in Section 7, which is followed by conclusions in Section 8.

2 Design Goals and Requirements

The objective of our project is to build a scalable parallel heuristic search library. By scalable, we mean that the usefulness of the library should be independent of the number of processors used by the application. It is a well-known fact in parallel processing that, for a fixed problem size, an increase in number of processors (N) used results asymptotically in efficiency degradation (i.e., the speedup to N ratio decreases as N increases). This is because the communications overhead increases while the total amount of time spent by all processors on useful computation remains constant. Moreover, other factors which can be safely neglected when the number of processors is small may become significant with the growth of N and require reconsideration. Problems due to design imperfections are usually aggravated by massive increase in number of processors. Typical examples include creation of communication bottlenecks caused by inefficient broadcast algorithms, computational hot-spots due to ineffective work-distribution schemes, overloading of the coordinating processor in centralized management activities, as well as excessive contention and "unfairness" in system resource sharing.

Several formal metrics have been proposed to measure scalability quantitatively but their details are beyond the scope of this report. Interested readers may refer to [12, 13, 18, 10] for further information. Scalability of the library is a function of the choice of algorithms and the type of hardware used [17] (e.g., shared memory multiprocessor system vs. distributed memory multicomputers), the connection topology amongst the processors (e.g., hypercube, mesh or ring), as well as the characteristics of interprocessor communication links (e.g., Ethernet, dedicated serial links, or integrated cross-bar switches). Although there are some generally applicable guidelines (e.g., distributed coordination schemes are more scalable than centralized ones), relevant factors are usually mutually-dependent. For instance, different connection topologies require different broadcast/reduction algorithms to attain the

most efficient result. It is, therefore, important to note that a well-designed library should support various alternatives to meet different system requirements. This may not be possible due to limited development resources and the ever-increasing hardware/interconnection combinations. However, we should at least design the library in an extensible way so that support of new hardware, algorithms, as well as connection topologies can be added into the existing library framework without large-scale restructuring. A set of generic functions including synchronization, coordination, load-balancing, search-space traversal, should be provided while the underlying implementations must be modifiable without changing the library interface. These generic functional units have to be defined in such a way that they can, on one hand, act as a stand-alone entity to provide the corresponding services independently, and, on the other hand, inter-operate with each other to provide integrated services when necessary. To further enhance the usability of the library, user-customization should be supported. By this, we mean that the library should provide "hooks" for the application programmer to override and/or replace some of its internal operations (e.g., to use his own work-request strategy while keeping other parts of the load-balancing mechanism untouched).

Readers may have already realized that lots of our design requirements fit very well with the object-oriented programming (OOP) paradigm. Through functional abstraction, independent, reusable units can be constructed in the form of generic classes. With data encapsulation using classes, a stable programming interface can be maintained while the implementation changes take place. Multiple selections of operational schemes can be supported via polymorphic functions (i.e., the combined use of class inheritance and virtual functions). Likewise, user-customization can also be achieved. By organizing the structure of the library as a hierarchy of classes, future extensions can be conveniently introduced without sacrificing program compatibility. This also improves code-reusability, amongst internal modules of the library. On the other hand, the use of the object-oriented approach is not without its cost — performance degradation due to the multiple software abstractions. Fortunately, with the maturing of optimizing compiler technology for object-oriented languages, this loss can be reduced. In conclusion, because of the considerable gain in library reusability, maintainability, extensibility as well as flexibility, we choose the object-oriented approach for our development. It is worthwhile to note that although the advantage of object-oriented design has been widely demonstrated for sequential programming, there is no extensive use of OOP in parallel/concurrent computing. Consequently, another goal of our project is to explore the potential of object-oriented design for concurrent computing.

We have already discussed the design goal of the library and the choice of using the OOP paradigm. In the next section, we will give an overview of ESP, the object-oriented concurrent computing environment upon which our library is implemented.

3 The ESP System

Our parallel heuristic search library is built on the Extensible Software Platform (ESP) system developed by MCC. ESP operates in a heterogeneous hardware environment which consists of a set of computational nodes, front-end systems as well as

parallel hardware accelerators (if available). These components are interconnected via an industry-standard network (Ethernet) and they can cooperate with each other to perform the concurrent execution of a single user application.

ESP provides a concurrent programming environment based on C++. It also contains a collection of software modules (in the form of C++ objects), which are used by the application developers to access ESP system services. Rapid development of experimental parallel and/or distributed applications is made possible due to the modular approach and the use of object-oriented design. Although our library is implemented in ESP, the design is general enough that it could be ported easily to other distributed C++ systems such as Mentat [7].

An ESP application consists of a set of C++ objects distributed across one or more hardware nodes in the system. A processing node can participate in the computation of multiple applications simultaneously. Application objects are derived from a base class supplied with the ESP software development environment while application-specific classes and methods should be provided by the application developer. In addition, the ESP C++ language includes extensions for implementing distributed, parallel operations. The ESP kernel provides resource allocation and communication services for the applications.

The **new** function in ordinary C++ is extended by ESP C++ to support dynamic object creation on remote nodes across the network. Once a remote object is instantiated, its execution can be explicitly controlled by local invocation of the corresponding methods associated with that object. When a method is invoked, the ESP system will check the location of the corresponding object. If the object resides in the local node, the necessary local execution is performed. On the other hand, if the object is on a remote node, the ESP system will automatically send a message to that object to start remote execution. This remote method call is transparent to the programmer and is implemented by the ESP system software by overloading the method invocation operator, `->()`, in ordinary C++. Both synchronous and asynchronous method calls are allowed. In addition, other synchronization primitives, e.g. `futures()`, `select()` and `wait()` are also supported.

As one may have already realized, there is no centralized control on the flow of execution; every object is an independent, message-driven, state/stateless entity (i.e., concurrency is at the object level, not at the process level). In fact, the *main()* function used in ordinary sequential C++ is not supported by ESP C++. A user initiates an ESP application by creating a host-object on the front-end system. The constructor of this host-object then spawns all the application-objects on the participating processors. Once the application-objects are in place, their constructors start the thread of execution in each object. Concurrent execution then continues with a series of local/remote method-invocations exchanged amongst various objects. Usually, the host-object will then stay idle and wait for the results of the computation. Interested readers are urged to see [9, 20, 19] for a detailed description of the ESP system.

Some researchers have suggested modifications to C++ to make it more suitable as a base programming language for distributed class libraries, especially for scientific and engineering computations. See [6] for a discussion on this topic.

4 Design Rationale of the Class Library Framework

Under the object-oriented ESP environment, the library is organized as a C++ class hierarchy. Functions that are common to multiple library components are extracted and encapsulated in a parent class. Classes corresponding to different functional units are derived from this parent class. The same technique can be used recursively amongst adjacent levels in the hierarchy to achieve code-sharing. Sharing flexibility is further enhanced by using features like multiple-inheritance and virtual-base classes supported by C++. We identify the service-providing units of the library as *library-classes* while the term *application-class* refers to the user-supplied class that defines the application object.

To provide a distributed service, such as synchronization, it is often necessary to create an instance of the corresponding library-class on every participating processor. The group of instances then coordinate with each other to support that particular service. We categorize the interactions amongst the multiple instances of the same library-class as *peer-to-peer communications*. Another type of communication takes place when library-class services are accessed by an application class. The latter type of interaction is identified as *service-access communications* and only occurs within the same processor because all inter-processor coordinations are handled through peer-to-peer communications.

There are two ways for the library components to provide their services. One approach is to instantiate stand-alone library-class objects to act as separate entities which serve the application-objects. This results in *inter-object communications* amongst the server(s) (library-class object) and client(s) (application object). A drawback of this approach is the large number of objects involved since multiple objects, which support different functionalities, have to be created on every processor. This will require the storing/book-keeping of multiple object-handle tables (one for each type of objects) on each processing node. Moreover, operating system overhead for inter-object communications is usually greater than that of *intra-object communications*. Take the ESP environment as an example. Inter-object communications, in the form of either local or remote method-invocations, involve message-passing, buffering, and message-queue processing. On the other hand, intra-object communication is simply transformed to an ordinary procedure (method) call as in a sequential program.

Due to the reasons given above, we decided to adopt an *all-in-one* approach in which only a single application-object is created on each processor. The application-object is instantiated from a derived class that has all the necessary library-classes as its parent. Through multiple-inheritance, the application-object can access all the necessary library services via inter-class, intra-object communications. In other words, all service-access communications become simple local procedure calls and the only necessary inter-object communications are the peer-to-peer ones. This approach can both reduce the message processing overheads and limit the number of object-handle tables in each processor to one because of the *one-object-per-node* policy.

According to our alternative approach, the interface of each library-class is defined (in the context of C++ classes) as follows: Methods used in peer-to-peer communications are declared as *private* because they are only invoked by objects of the same type located on other processors. Routines used for internal class im-

plementation are also defined as private methods. The type "*protected*" is used to declare methods related to library service-access so that they are inherited by the application-object. In addition, some of the service-access-type methods are also declared as *virtual* so that the application-class can overload them with application-specific routines via functional-polymorphism. Only the constructor and other monitoring/debugging routines have to be declared as public methods.

The understanding of the subtle difference between various types of communications/interactions such as inter-object, intra-object, peer-to-peer and service-access, is essential for us to design the deadlock prevention strategies in the concurrent executing environment. For instance, in the computing model of parallel ESP-C++, no more than one method of the same object can be invoked and executed simultaneously. Even if the execution of the currently active method is blocked (e.g., while waiting for responses from other objects), no other external invocations on any other method of the same object can be processed. The incoming invocation requests are queued and will not be started until the current method has been finished. To illustrate the potential problem caused by this during inter-object communications, let's consider the scenario in which two objects issue a synchronous (blocking) method call to each other simultaneously. Since the "replying" method in each object is prevented from "responding", deadlock will result due to "circular-waiting".

In order to avoid the above situation, we decided to adopt the following convention in the design of inter-object communications within the library: In all peer-to-peer (inter-object) communications, at least one side of the calling/called party should issue its method invocation request asynchronously (non-blocking). Since there is little a priori knowledge on the communication patterns amongst various objects, the above requirement usually leads to the use of asynchronous method invocations on most inter-object communications. The only exception appears in the design of the tree-synchronization mechanism. In this particular case, since the communication pattern is pre-determined by the chosen algorithm, the synchronization is initiated by a non-blocking inter-object (peer-to-peer) communication but the remaining steps are performed via inter-object, peer-to-peer, synchronous method-calls. The complete process will be described in detail in the next section of the report. Another related deadlock avoidance measure is that the library-classes do not support explicit method-locking services for the application programmers. Method-locking/enabling activities are all handled within the library-classes themselves, according to the execution state of the library-class objects. On the other hand, service-access (intra-object) communications (method invocations), unlike their inter-object counterparts, do not suffer from the circular-waiting problem since all the method invocations are performed as ordinary local procedure calls.

We have just described the interface conventions, communication "rules", service-access "protocols" as well as the general operating mechanisms used by the parallel heuristic search library. In the following section, the design and implementation of individual library-classes will be discussed. Considerations for enhancing the flexibility and extensibility of the library-classes will be emphasized.

5 Design of Library Classes

The class hierarchy of the Parallel Heuristic Search Library is shown in Figure 1. Six library classes have been defined. The first group has only one library class, `srch`, which is used to perform sequential search procedures. The other group consists of five "parallel processing classes" that are dedicated for coordinating parallel/concurrent activities. Under this layout, the parallel, load-balancing tasks are completely separated from the sequential search operations in individual PEs. Usually, an application-class object obtains work from other PEs and then calls `srch` class member functions to perform sequential search on the assigned problem space. The member functions will return with the search result either after the solution has been found or when the search space is exhausted. Note that the design of the "parallel processing classes" are general enough to support other distributed/parallel applications besides parallel heuristic search. The five "parallel processing" classes are :

- `topo_base`: manages the physical/logical connections and object-addressing among various processing entities (PE);
- `syncman`: provides synchronization primitives to coordinate the executions of the PEs;
- `gtd`: provides global termination detection;
- `distributor`: provides information broadcast and collection services; and
- `lb`: performs load-balancing by splitting the available work amongst the PEs.

The designs of individual library classes are discussed in the following subsections.

5.1 The `topo_base` Class

In the ESP environment, a physical *node-id* is assigned for each processor of the participating machines. The resultant node-id-to-processor mapping is stored in a system configuration file which is independent of the search library. With our "one-object-per-node" approach , an application comprises of a host object and a group of identical application-class objects which can be considered as *logical* processing units, PEs, of the parallel application. Readers should distinguish between physical processing nodes (the actual hardware processors) and the logical PEs. Their decoupling not only facilitates software development but also increases programming flexibility. For instance, the developer can use a uniprocessor machine to debug a parallel application without tying up expensive parallel hardware resources.

When a PE is spawned as an ESP-C++ object, its location can be specified by the physical node-id of the target processor. Or programmers can leave the decision to the ESP kernel which will choose the best site for the object according to current machine loads. Once a PE has been created, it is accessed via the object-handle returned by the `new()` function during its creation process. One can store the object-handles of all the PEs to form an address table and then use the index on the table as a *pid* of the corresponding PE. This approach is not very scalable because the table grows with the number of PEs. Moreover, the extra communication overhead in distributing object-handle tables further reduces program scalability. An alternative

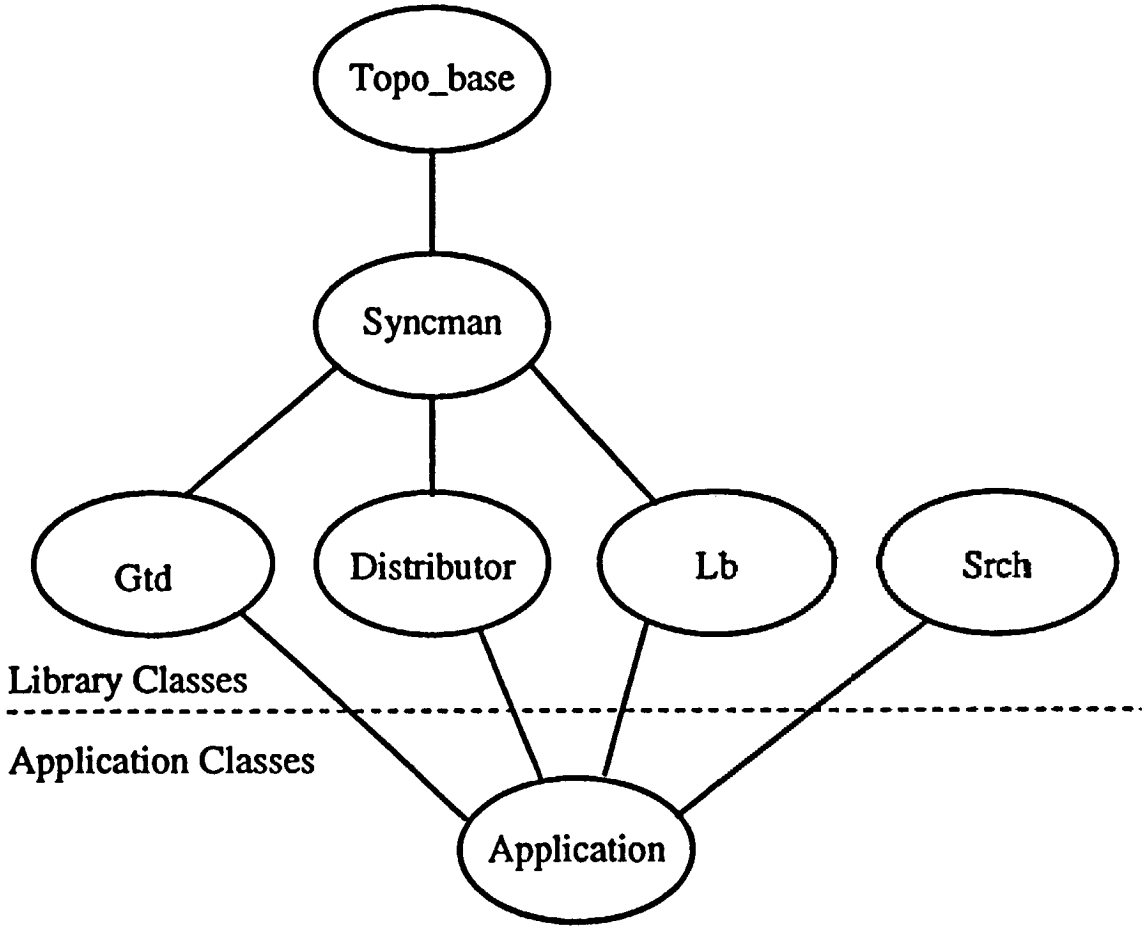

Fig. 1. Hierarchy of the class library

way to obtain the handle of an object is via the use of an ESP system function. This function takes the instance-number and the class identifier of the object and maps it to the corresponding object-handle. If this approach is used, the instance-number can be regarded as the logical pid of a PE. Comparing these two schemes, the latter seems to be more scalable, but it imposes extra system-function call overhead in every object access. This makes the latter scheme less efficient than the handle-table approach, especially when the number of PEs is small (< 100). As a result, the topo_base class support both schemes through the `gethandle()` method which takes a pid as an argument and returns the object-handle of the corresponding PE.

Another function of the `topo_base` class is to maintain logical connection topologies amongst PEs. Most distributed/parallel algorithms assume the existence of a particular connection topology between the PEs (e.g., a logical tree or ring formed by the PEs). In order to make the codes more readable, it is essential to clearly expose the inter-PE connection-relationship and hide the complicated pid/address manipulations at lower level(s) of the implementation. Let's consider the case where the target hardware is a hypercube multicomputer with one PE per processor and a tree-based synchronization algorithm is used. It is preferable to provide functions which can return the the parent or children pids of a given PE so that the tree-algorithm can be built on these functions. This not only enhances program maintainability but also improves application portability across different hardware topologies. In our example, if we decide later to port the application to a mesh-connected multicomputer, we only have to change the internal implementation of the "parent" or "children" methods while the rest of the tree algorithm can be kept untouched. Following the same line of thought, a set of "connection-relationship" methods are defined in the `topo_base` class interface for embedding various logical topologies into the underlying physical network. Currently, logical tree, ring and mesh embeddings are supported for the ESP environment. The following methods are defined for this purpose:

- `parentnode()`, `childnode()` for logical tree;
- `nextnode()` for directional-ring;
- `North/South/East/West_neighnode()` for 2-D mesh;
- `all_neighbor()` to support flooding broadcast;

In addition to the functions mentioned above, the `topo_base` class also provides a method, `broadcast()`, for users to send a message to all the PEs of an application. The decision to include broadcast service in `topo_base` class is due to the fact that knowledge about the physical system topology is usually required for implementing an efficient broadcast scheme.

5.2 The syncman Class

Usually, multiple "rounds" of computation are involved in a parallel application. Each "round" is analogous to an iteration of the outer-most loop in a sequential program. They differ in the sense that distributed mechanisms are necessary in parallel computation to determine the end of a round and notify all the participating PEs to enter the next round, while no coordination is needed between consecutive

iterations in a sequential loop. The `syncman` class is designed to support the synchronization between successive rounds of parallel computation. It also serves to keep track of the current "round number" of the computation. The current round-number is obtained through the `getroundcnt()` member function and is used for stamping every outgoing message (i.e., it is an argument for all remotely-invoked methods).

Upon the receipt of a method-invocation request, a PE checks the round-number stamped on the message. A request will only be processed if the stamp matches the current round-number, or the message will be ignored and discarded. The use of round-number stamp prevents unexpectedly delayed messages of previous rounds from causing computation-state confusion. The `syncman` class uses a tree-based synchronization mechanism. The PEs are organized as the nodes of a logical tree, with the PE #0 ($pid = 0$) as the root. When any PE decides to terminate a round (e.g., it discovers the solution), it initiates a synchronization cycle by invoking its own `lstartsync()` method which, in turn, performs a remote, asynchronous invocation of the `extstartsync()` method in the root PE. Since a non-blocking mode of invocation is used, the initiating PE can continue the execution of the local `lstartsync()` while sending the message to the root. After the local `lstartsync()` is completed, the initiating PE becomes idle, waiting to join the global synchronization process.

When the root PE receives the `extstartsync()` request, it starts the synchronization process by launching `extstartsync()` method calls to its immediate children PEs in the logical tree. When a PE's `extstartsync()` method is remotely invoked by its parent PE, it will relay a blocked `extstartsync()` invocation to all its children (if any), waiting for their reply (i.e., the return of those `exstartsync()` method calls). Once all of its children are ready, it will execute the local `lsyncaction()` method to perform initialization for the next round. When the preparation is finished, it will "reply" to its parent by returning to the parent's `extstartsync()` call. The same mechanism propagates recursively down the tree until the leaf-PEs are reached. Because of the acyclic nature of trees, circular-wait deadlocks are avoided. After all the `extstartsync()` calls issued by the root PE have been returned, it is certain that every PE in the tree is ready and the root can then initiate a new round of computation by making `startnewround()` method calls which are recursively relayed down the tree in a similar manner.

5.3 The gtd Class

The `gtd` class is used for detecting global termination — the situation where every PE is asking others for work while there is, in fact, no more work left within the system. The current `gtd` is a straightforward implementation of Dijkstra's ring termination detection algorithm whose details are beyond the scope of this report. However, it should be noted that the class interface has been carefully designed so that other termination detection algorithms can be implemented without library-interface changes. The overall computation status is monitored by "circulating an enquiry message" around the logical ring of PEs. Application class objects can ask for the current status by invoking the `gtenquiry()` method. When global termination is detected by a PE, a local dummy routine is invoked. This dummy routine, declared as a C++ virtual method, is usually overloaded by an application-class method via functional polymorphism. As a result, application-specific actions can

be carried out in response to global termination. Typical actions include new round initialization or reporting of computation results.

5.4 The distributor Class

The distributor class is used for managing global variables shared by all PEs of an application. Reduction operations on distributed data are also supported for evaluating global sum, maximum and minimum functions. At present, two types of distribution/collection mechanisms have been implemented, namely, the ring-relay and tree-based operations. Under the ring-relay approach, the value of the global variable is continuously circulated and updated around the logical PE ring, while the tree-based operations take a recursive approach similar to the one used in the syncman class.

5.5 The lb Class

When a PE has finished its previously assigned work, it requests more work from other PEs by invoking its own intworkreq() method inherited from the lb class. The local lb then negotiates with its peers to get more work by remote invocation of a target peer's extworkreq() method. Under the current implementation, the PE to be asked is chosen in a decentralized, random manner through calling the local choose() method. Once again, the selection mechanism is completely encapsulated inside the choose() method so that programmers are free to overload it with their own PE-selection method. (In general, different selection methods will be superior in scalability for different types of parallel/distributed hardware [12].)

As one may expect, the lb is also responsible for handling incoming work-request from other PEs. Following the pre-determined work-splitting guidelines, a PE responds according to the amount of work on hand. It is necessary to enforce a lower limit on the size of work being transferred. This is because if the locally available work is too small, little speedup can be gained via load-sharing as work-transfer, and setup overhead becomes dominant. If a work-request is approved by the local lb, "work" will be transferred as an unstructured byte-stream to the destination PE. When external work arrives, lb notifies its application-class object. Again, this is achieved by the same "dummy method invocation" technique as described before. The dummy virtual method extworkarrival() in the lb class interface is defined for this purpose.

5.6 The srch Class

Finishing download of external work from its lb, a PE invokes member functions of the srch class to perform the sequential search tasks. The srch class has a variable for tracking the status of the searching process and the search can be viewed as a tree-traversal process. When a new tree-node is visited, a series of "test" methods are invoked to check if certain conditions have been met. Leaftest(), prunetest(), aborttest() and foundtest() are some necessary test methods falling in this category. The results of these tests are used to update the current search status. Possible search states include FOUND, ABORT, PRUNED, LEAF, STEPFORWARD. Based

on the resultant search status, corresponding "action" methods are invoked (e.g., `leafaction()`, `finishaction()`, `backtrack()`, `stepforward()`). As usual, all of the "test" and "action" methods are declared as virtual so that they can be overloaded by application-specific routines. In order to further enhance the reusability of the `srch` class, a programmer is allowed to overload the class methods at different levels. For instance, he can change the whole backtracking mechanism by "substituting" the default `backtrack()` method with his own version or he can just replace some of the internal routines called by the default `backtrack()` method. Note that no *real* modification of library code is required in any case because all the "substitutions" and "replacements" are achieved through virtual-function overloading.

This completes the design description of individual library classes. To demonstrate the usability of the library, we have used it in the porting of two existing parallel applications from the Caltech Cosmic Environment to ESP. Results on code-reusability of the ported applications are given in the next section. Experience in the porting process will also be discussed.

6 Experience with the Parallel Search Class Library

Two existing parallel-search applications, namely, VLSI Floorplan Optimization (FP) [2] and Automatic Test Pattern Generation (ATPG) [1] have been ported to ESP using the class library. In order to simplify the transition, the original C codes written for the Caltech Cosmic environments were first ported to ESP without the use of the class library. And the integration of the class library to the applications is carried out in the second phase of the porting. Since the major goal of the porting project is to *test-drive* the class library, we decided to minimize changes whenever possible – as long as the applications can run under the ESP environment and interoperate with the class library. In other words, no full-scale re-design is made on converting the existing applications to become completely object-oriented.

Table 1. The extent of changes for various ports of applications

Application	Environment	Number of lines of code	
		Unmodified across ports	Changed while porting
FP	Cosmic	828 (15%)	1476 (85%)
	ESP w/o library	828 (29%)	2058 (71%)
	ESP with library	828 (43%)	1085 (57%)
ATPG	Cosmic	2276 (52%)	2104 (48%)
	ESP with library	2276 (60%)	1526 (40%)

As shown in Table 1, a considerable portion of the code remains untouched during porting (15% in FP, 52% in ATPG, using the size of the cosmic-version as reference). The unchanged code mainly consists of routines which implement the sequential operations within a PE. During the first phase of porting, the Cosmic version of a

routine is usually re-declared as a private method of the application class. The resultant method is then used to overload the corresponding generic method defined in the `srch` class. For those parts of the program where *non-straightforward* modifications are required, the necessary changes are by no means complicated. It is not uncommon that code segments from the original C-functions can be extracted and added to the corresponding methods in a *cut-and-paste* manner. The high percentage of code being reused across various ports also confirms the interoperability of the object-oriented parallel search class library with traditional, non-object-oriented parallel programs. This also illustrates another advantage of using ESP-C++; it allows gradual transition from the procedural programming paradigm to OOP.

However, several major re-structurings of the original program were unavoidable before it could be run in the object-oriented ESP environment. One of these changes is due to the difference in main control-flow of the program. Since a Cosmic-based program always contains an active message-polling loop, modifications are necessary in order to decompose the polling-loop into separate message-driven methods within an ESP application-object. Because of this difference in programming paradigm, the size of the FP application increases from 2304 (in Cosmic) to 2886 (in ESP) lines of codes, even before the use of the class library.

Table 2. Code savings and program size changes with the use of the parallel search library

		Name of Application	
		FP	ATPG
Number of lines of code	Cosmic version	2304	4380
	ESP w/o library	2886	5000 [a]
	ESP with library [b]	1913	3802
Code Savings % with search library [c]		34%	24%
Program size increase % with search library [d]		17%	5%

[a] This is an estimated value only. This phase is skipped in the porting of ATPG. Increase in code-size while porting FP from Cosmic to ESP (w/o library) is mainly due to the breakup of original main control-loop into separate methods of the application object. Since the main control-loop for the Cosmic version of ATPG is very similar to that of FP, it can be broken into approximately the same number of methods if the port is carried out. Thus, the expected increase in code-size for porting ATPG from "Cosmic" to its "ESP w/o library" version should be close to that of the corresponding port for FP: $4380 + (2886 - 2304) = 4962 \approx 5000$.

[b] Do not count library internal codes.

[c] Use the "ESP w/o library" version as reference.

[d] Size includes library internal codes; also use "ESP w/o library" version as reference.

For the versions which make use of the class library, the size of code can be reduced as shown in Table 2 (only if the 1470 lines of library codes are not counted). But if the library-codes are included, the overall program size generally increases. For the FP application, the resultant 17% growth is quite significant. This can be explained by the fact that FP requires relatively simple parallel-processing support (e.g., it only consists of a single round of a computation), and hence a lot of library

features, such as multi-round support, are unnecessarily included in the bulk of the program. On the other hand, for an application like ATPG, which requires sophisticated synchronization support, the increase in program size can be considered as marginal (5%).

With the porting-process completed, we try to assess the usability of the class library. Due to the lack of extensive usage statistics, *code-savings* achieved in the above ports are chosen as the metric for the usability study. In order to exclude the effects caused by software environment differences (i.e., Cosmic vs. ESP), comparisons are based on different ESP-port versions (with and w/o class library) of the same application, using the following formula:

$$Code\ savings(\%) = \frac{ESP\ with\ library\ Size - ESP\ w/o\ library\ Size}{ESP\ w/o\ library\ Size} * 100\%$$

Based on the above formula, relative code-savings of 34% and 24% has been achieved for FP and ATPG respectively. (In contrast to this definition of code savings for two implementations of the same application, Mancl [14] has defined code savings across multiple applications.) We argue that the *code-saving* metric is very conservative in evaluating the *usefulness* of our *parallel-oriented* class library. We see the significance of the library in freeing programmers from complicated, *hard-to-debug* synchronization and load-balancing codes and its *real* value cannot be fully exhibited by a line-count-based metric.

7 A Retrospect

The current design of the library is heavily biased towards the use of inheritance and virtual functions to achieve functionality aggregation. For example, classes `gtd`, `distributor` and `lb` are all derived from `syncman` only because they may need its multi-round supporting function. However, according to "mainstream" object-oriented design [3, 15], the class relationship in the above example should be specified as *using-relationship*. In other words, the class hierarchy should be reorganized such that `syncman` becomes a sibling of `gtd`, `distributor` and `lb` rather than being their parent. By flattening the class hierarchy, more flexible library uses are possible [5]. In our case, users will then be able to selectively include the multi-round support on a need-to-use basis. Our original decision of not following the "mainstream" approach is mainly based on performance considerations and the foreseeable extensive cross-coupling between class-interface specifications. Such excessive dependencies may overwhelm application programmers and hence affects library usability. After we had started the actual implementation, solution for the cross-coupling problem was proposed in [8]. They solved the problem by adding "functionoid" and "glue" classes to encapsulate inter-class dependency. With these emerging techniques, it is worthwhile to reconsider the organization of the class hierarchy in the future versions of the library. Since similar "inheriting vs. using" situations repeatedly occur throughout the library design process (e.g., interacting relationship amongst application object(s) and library components), our library can also be used as a test-bed for further studying the trade-offs between different object-oriented design approaches.

Based on some preliminary data collected during our experiments, the timings for the ESP version and the Cosmic Environment version seem to be similar. We need to conduct more experiments to validate this conclusion.

8 Conclusion

It has been a worthwhile experience to design and implement the Parallel Heuristic Search Library under the ESP environment. Starting with the goal of exploring the use of object-oriented programming techniques for distributed/parallel applications, we have found that OOP and Parallel Processing fit with each other extremely well. Perhaps this is can be attributed to the inherently modular nature shared by both of the paradigms. We also find that lots of the tasks commonly required in parallel/distributed applications can be perfectly abstracted and encapsulated in separate, functional objects. As a result, it becomes possible to use these objects as the basic building blocks for a highly reusable class library that can support *general* parallel/distributed computations. For similar results in the scientific computing area, see [4]. In an attempt to "calibrate" the usability of our class library, different quantitative metrics have been used which, we think, are unable to effectively reflect the real value of the library. This brings up the need to conduct further research in designing better metrics for future code-reusability studies.

Acknowledgements

Most of this work was done while the authors were at MCC in Austin, Texas.

References

1. S. Arvindam, V. Kumar, V.N. Rao, and V. Singh. Automatic test pattern generation on parallel processors. *Parallel Computing*, 17(12):1323–1342, December 1991. A shorter version appeared in proceedings of the 1989 International Conference on Knowledge-Based Computer Systems.
2. Sunil Arvindam, Vipin Kumar, and V. Nageshwara Rao. Floorplan Optimization on Multiprocessors. Technical Report ACT-OODS-241-89, MCC, 1989. Submitted for journal publication. Shorter version appeared in proceedings of the 1989 International Conference on Computer Design.
3. G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, Reading, MA, 1990.
4. G. Carey, J. Schmidt, V. Singh, and D. Yelton. A Scalable, Object-Oriented Finite Element Solver for Partial Differential Equations on Multicomputers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, 1992. To appear.
5. J.M. Coggins. Designing C++ Libraries. In *Proceedings of the 1990 USENIX C++ Conference*, April 1990.
6. D.W. Forslund. Experiences in Writing a Distributed Particle Simulation Code in C++. In *USENIX C++ Conference*, pages 177–190, 1990.
7. A.S. Grimshaw. An Introduction to Object-Oriented Parallel Programming with Mentat. Computer Science Report TR-91-07, University of Virginia, 1991.

8. P.R. Jossman, E.N. Schiebel, and J.C. Shank. Climbing the C++ Learning Curve. In *Proceedings of the 1990 USENIX C++ Conference*, April 1990.
9. A. Khanna et al. Experimental Systems Project at the Fifth Distributed Memory Computing Conference. Technical Report ACT-ESP-117-90, MCC, April 1990.
10. Vipin Kumar and Anshul Gupta. Analyzing the Scalability of Parallel Algorithms and Architectures: A Survey. In *Proceedings of the 1991 International Conference on Supercomputing*, June 1991.
11. Vipin Kumar and V. N. Rao. Scalable Parallel Formulations of Depth-First Search. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, 1990.
12. Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16 (6):501–519, 1987.
13. Vipin Kumar and Vineet Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem. *Journal of Parallel and Distributed Processing (special issue on massively parallel computation)*, October 1991. A short version appears in the Proceedings of the International Conference on Parallel Processing, 1990.
14. D. Mancl. Use of metrics to evaluate C++. In *Conference Proceedings: C++ at Work-'90*. The C++ Report and The Wang Institute of Boston University, 1990.
15. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
16. V. Nageshwara Rao and Vipin Kumar. Parallel Depth-First Search, Part I: Implementation. *International Journal of Parallel Programming*, 16 (6):479–499, 1987.
17. Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw Hill, NewYork, 1987.
18. V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Efficient Algorithms for Parallel Sorting on Mesh Multicomputers. *International Journal of Parallel Programming*, 20(2), 1991. Shorter version in proceedings of the 1991 International Parallel Processing Symposium.
19. K.S. Smith and A. Chatterjee. A C++ Environment for Distributed Application Execution. In *Conference Proceedings: C++ at Work-'90*, 1990.
20. E. Surma. Extensible Software Platform Development Environment Programmer's Guide. Technical Report ACA-ESP-223-88, MCC, August 1990.

This article was processed using the Latex macro package with LLNCS style