# Chapter 1
# Modularity, Reuse and Efficiency with Message-Driven Libraries

This research was supported in part by the National Science Foundation grants
CCR-90-07195 and CCR-91-06608.

L.V. Kale and A. Gursoy
Department of Computer Science
University of Illinois at Urbana Champaign
1304 W. Springfield Ave., Urbana, IL-61801

**Abstract**

*Software re-use* via libraries is a strategy that allows the cost of software to be amortized. A parallel programming system must support the ability to develop modules that can be "fitted together" in a variety of contexts. Although it is important to be able to reuse parallel libraries, it is also more difficult to use parallel modules in comparison to sequential module. We present a methodology for developing libraries that addresses these issues effectively. The methodology, which is embodied in the Charm system, employs message-driven execution (in contrast to traditional, receive based message passing), information sharing abstractions, the notion of branched objects, and explicit support for modules.

## 1 Introduction

*Software re-use* via libraries is a strategy that allows the cost of software to be amortized. A parallel programming system must support the ability to develop modules that can be "fitted together" in a variety of contexts. For example, one should be able to use a (previously written) parallel linear system solver module and a parallel FFT module to construct a computational fluid dynamics application program.

Parallel programs and modules are more difficult to produce than sequential ones; as a result their is a higher premium on being able to reuse parallel libraries. Yet, the nature of parallel software makes it more difficult to reuse modules developed independently: different modules may employ different data distributions, control and data transfer may occur asynchronously or may have to be explicitly synchronized, and one may have pay significant costs in efficiency for modularizing a program to use libraries.

In sequential programs, the module interfaces are simple; a module is invoked by calling a subroutine with appropriate parameters. The control flow and data flow are co-incident. In case of parallel modules, one must find suitable ways in which modules can exchange data. Some of the requirements for promising reuse in parallel programs are:

1. One should be able to decompose a program into multiple modules, without losing efficiency significantly.
2. The intermodule interfaces should allow distributed flow of data across modules to avoid sequential bottlenecks. The data-exchange protocols should be sufficiently flexible so that

modules can be used in varied contexts, even when their clients may employ different data distributions.

3. For practical promotion of re-use, it is desirable to allow modules which are distributed in object-code format only, to protect the possible proprietary nature of modules.

## 2 Why Message-Driven Parallel Libraries

In traditional approaches, library computations are invoked by regular function calls. The library call blocks the caller on all processors. After completion, the library module returns the result and control to the calling modules. Some disadvantages of libraries in this style are:

1. Idle times in the library computation cannot be utilized even if there are other independent computations
2. Caller modules must invoke the library on all processors even when only a subset of the processors provide input, and receive output.
3. Library computations must be called in the same sequence on each processor.

A message-driven system, such as Charm [1], supports multiple objects per processor, and uses a pool of messages on every processors. An object is scheduled for execution when there is a message for it. In such a system, one can invoke multiple library modules concurrently, allowing them to naturally overlap their idle times with useful computations. This is a substantial boost for encouraging use of libraries.

As an example, taken from [4], consider an application A that invokes two independent library modules B and C as shown in Figure 1. The B and C themselves are parallel computations which may have their own idle times on all processors due to message latency and dependencies among sub-computations. In the traditional approach, once the control is given to the library B, the idle times of B can not be used by other modules. However, in message-driven execution model, the control can be passed to the module C while B is idle provided that there is a message for the C (or vice versa). Therefore, the idle times across library modules can be used effectively in message-driven execution model.

## 3 Charm and Modularity

Charm is a portable object-based message-driven parallel programming system. A Charm program/computation consist of potentially small-grained processes or objects, called chares, and a special type of replicated objects, called branch-office chares. Charm supports dynamic creation of chares, by providing dynamic (as well as static) load balancing strategies.

A chare consists of local data, entry-point functions, private and public functions. Private functions are not visible to other chares, and can be called only inside the owner chare. Public functions can be called by any object on the same processor. Entry functions are invoked asynchronously by an object on any processor. Invoking an entry function in a remote object can also be thought of

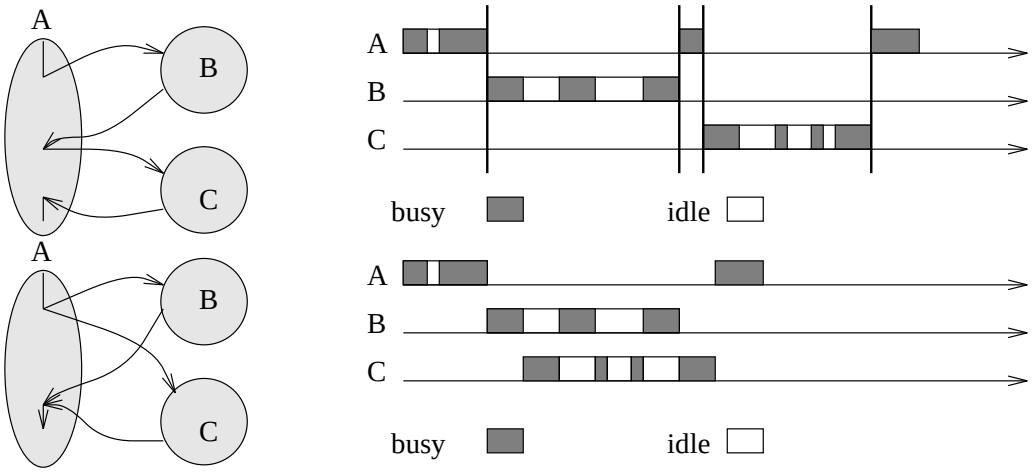


Figure 1: SPMD and MD modules

as sending a message to it. A full description of the Charm language and its C++ based version, Charm++, can be found in [2, 3].

The Charm runtime system is message driven. It repeatedly selects one of the available messages from a pool of messages in accordance with a user selected queueing strategy, restores the context of the chare to which it is directed, and initiates the execution of the code at the entry point.

The utility of message driven execution for efficient modularization has been already been described above. In addition, features supported in Charm that support modularity and reuse include branched objects, information sharing abstractions, and the module construct.

A branched object (BOC) is an object with a branch on every processor; all of the branches answer to the same name. One can call public functions of the local branch of a BOC, send a message to a particular branch of the BOC, or broadcast the message to all of its branches. BOCs provide a versatile abstraction that can be used to implement static load balancing, local services (e.g. memory management), distributed data structures, and inter-module interfaces. Two library modules, each spread over all of the processors of the system, can exchange control and data easily through public calls to each others local branches. This provides a simple mechanism for distributed flow of data across modules. Of course, function calls used in the traditional programming model also allow such exchange but they do not support encapsulation of the state of the library — so, for example, names of global variables (which *have* to be used to express state) in the two libraries can conflict in such situations.

Information is shared in a few but multiple, distinct ways in a parallel program. A "message" is not an adequate mechanism for expressing information sharing, nor is any single generic mechanism such as Linda's tuples [6]. On the other hand, a shared variable is too amorphous a mechanism — its generality is not needed in most situations and costs too much to implement. Recognizing this, Charm provides six specific information sharing abstractions in addition to messages. Further information about these abstractions can be found elsewhere [5]. Many of these mechanisms provide flexible ways of data exchange across modules. For example, a distributed table holds a collection of data items, each indexed by a distinct key. On distributed memory machines, this collection may be distributed across the processors. One can insert, delete, and find data from such tables asynchronously. Thus, a module may deposit data in a distributed table from which another module

may extract it, thus obviating the need for "hardwiring" data distribution requirements in module interfaces.

Charm supports a well developed module system. As its first simple benefit, names in one module do not conflict with those in others. Names that are exported to other modules are referred to with a module prefix. For example, an entry function `f` of a chare `c` in module `m` is invoked as: `m::c@f(msg)`. Thus, a library developer can freely use names — even including the names that they wish to export — without worrying about possible conflicts. The recent concept of "contexts" in MPI [7] eliminates conflicts on tags across modules, but other name conflicts remain. Complex libraries often require the use of callbacks: a library module invoked by its client might, during the course of its parallel computation, require further information from the client. As the library is written independent of the client, it doesn't know the names of entities in the client. The client must pass references to such entities dynamically at invocation. Charm supports such dynamic interfaces by treating function names, chare names, entry function names, and chare id's as first class objects — i.e. one can have variables in which to hold such entities and one can pass them around. All of these features are supported with separate compilation, which is essential for promotion of practical reuse in a commercial environment. Thus, multiple proprietary libraries can exist in a larger application program without requiring access to their source code.

# 4 Message-Driven Library Interface Techniques

The interface between message-driven programs and message-driven libraries are different from the interface for SPMD style. Since computations are split-phase in message-driven style, library calls must provide a return address for the result or completion. The three separate steps of a simple library interaction in message-driven style are: (a) creation of the library object, (b) invocation of a library computation, and (c) reception of the result at a later point. The caller might execute other code before the result is returned, including possibly invoking other library computations.

## 4.1 Patterns of Library Interactions

There are various ways of interaction between parallel library computations. However, most interactions exhibit patterns that can be classified into a few groups (Figure 2). One of such interactions, which we will call as distributed interface, is: when every object/node makes a request to the same library and receives the result of the library computation. An example of this case is a reduction/broadcast operation which is very common in many scientific applications. A second pattern is the client/server interaction. An object makes a request to the library. Here the library is a server. Then the library computation continues concurrently with the rest of the application. The library may involve parallel tasks on multiple processes. Eventually, the result is returned to the requester by the library. The third pattern involves a single request from a client which triggers a distributed computation and returns the result to multiple agents, (e.g., all branches of a BOC) as specified by the client.

Another important interaction between library and computations is the delayed data access by the library. In this interaction, the caller invokes the library, and then continues with some other computation. Later, the library may request some data from the caller.

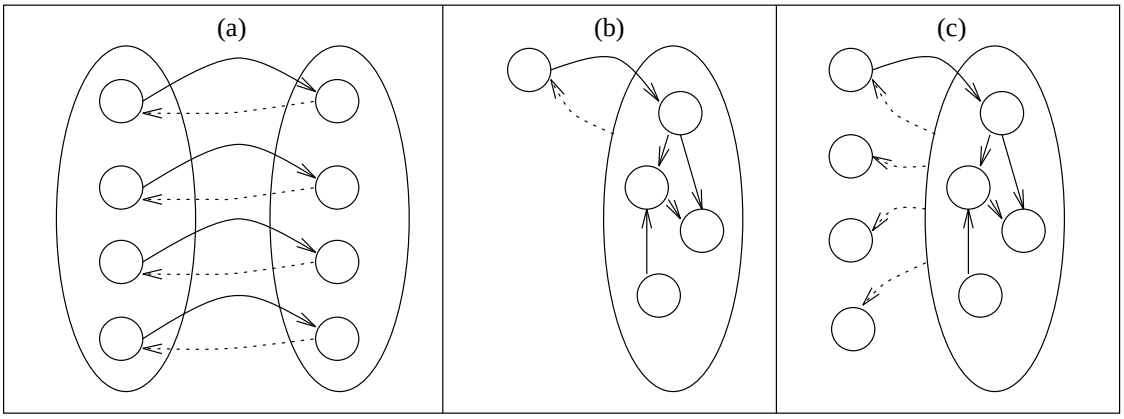


Figure 2: Pattern of Library Interactions

### 4.2 Parallel Library Interface

The first step in a typical Charm program is to create instances of library modules. Once the instance is created, it can be invoked by all the participating computations using its unique instance identifier. This requires the participating computations (or chares) to have the library's instance identifier. The creation phase, thus, consists of creating the instances of the library and distributing the instance identifier to users of the library (clients). The creation of a library module can be encapsulated into a function. The library exports this function, `create`, that handles all the steps of a creating the library (the user does not have to know the details of the library). There are two ways to invoke the create function:

```
lib_instance = LIB::create() or
LIB::create(chareid,entrypoint)
```

In the latter case, the instance id is returned asynchronously to the given chare id via the named entry function. A library can be invoked by sending a message to it. The library module exports definitions of types that clients need, such as messages and names of chares and entry points. This function needs input data from the caller (optional) and a return address. The result can be returned in a message. In this case, the library invocation call is as follows:

```
LIB::request(lib_instance,data,my_chare_instance,entry_point)
```

The library instance `lib_instance` is invoked, and the result will be returned in a message to `entry_point` of the caller chare. A second option is to receive the result by function call

```
LIB::request(lib_instance,data,result_buffer,function_ptr)
```

This call provides the library with input data, a pointer to the result area, `result_buffer`, where the library directly puts the result there, and `function_ptr` which is a public function that the library module calls when the result is ready.

Concurrent library invocations: If the same library module needs to be called multiple times and concurrently, what should the interface be? There are two options; the first one is to create multiple instances of the library module as shown below:

```
id1 = LIB::create();
id2 = LIB::create();
```

and different instances can be invoked concurrently as follows:

```
LIB::request(id1,data1,mychareid,e1);
LIB::request(id2,data2,mychareid,e2);
```

The second option is to design the library module such that it handles concurrent calls. The caller provides a reference number. The library maintains a separate environment for each reference number to service the requests concurrently. A typical usage of this scheme is to create an instance of the library and invoke the same instance with different reference numbers for each distinct request.

```
id1 = LIB::create();

LIB::request(id1,data1,REQUEST_1,mychareid,e1);
LIB::request(id1,data2,REQUEST_2,mychareid,e1);
```

The reference number of the result message is set to the REQUEST_1 or REQUEST_2. If the return is by a function call, the reference number can be passed as an additional parameter to the return function.

# 5 Example and Performance

This example [4] is abstracted and modified from a real application — a core routine in parallelized version of a molecular mechanics code, CHARMM. Each processor has an array $A$ of size $n$. The computation requires each processor to compute the values of the elements of the array and to compute the global sum of the array across all processors. Thus, the $i^{th}$ element of $A$ on every processor after the operation is the sum of the $i^{th}$ elements computed by all the processors. One can divide the array $A$ into $k$ parts, and in a loop, compute each partition and call the reduction library for each segment separately and concurrently. Table 1 shows the completion time of the traditional and message-driven implementation of this example. The advantage arises from being able to invoke multiple reductions which execute concurrently.

| | ncube/2 - Number of Processors | | | | | |
|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 |
| SPMD | 0.45 | 0.56 | 1.02 | 1.27 | 1.54 | 2.17 |
| Message-Driven | 0.46 | 0.65 | 0.93 | 0.94 | 0.92 | 0.98 |

Table 1: Completion time (sec) of concurrent reductions, $n = 40960, k = 160$

# 6 Summary

Traditional library interface techniques do not allow one overlap idle times in one library with useful computations in the client or another library. This encourages programmers to merge multiple library modules together and tune them for efficiency. Message-driven execution, in contrast, supports modularity without sacrificing efficiency. A programming system that combines message-driven execution with facilities for flexible exchange of data and control across parallel modules, and supports module with static and dynamic linkages and separate compilation, thus provides an excellent substrate for building libraries.

## References


[1] L.V. Kale, *The Chare Kernel Parallel Programming Language and System*, Proc. ICCP90, Vol II, Aug 1990, pp. 17–25.

[2] *Charm 4.3 Programming Language Manual*, Department of Computer Science, University of Illinois at Urbana-Champaign, Sep 1994.

[3] L.V. Kale and S. Krishnan, *Charm++ : A portable concurrent object oriented system based on C++*, Proc. of OOPSLA 93, Washington D.C.

[4] A. Gursoy, *Simplified expression of message-driven programs and quantification of their impact on performance*, Ph.D Thesis, University of Illinois at Urbana-Champaign, Apr 1994.

[5] L.V. Kale and A. Sinha, *Information Sharing Mechanisms in Parallel Programs*, Proc. of IPPS-94, Cancun April 1994.

[6] N. Carriero and D. Gelernter, *Linda in Context*, Comm. ACM, 32-4 (1989), pp. 444–458.

[7] Message Passing Interface Forum, *Document for a standard message passing interface*, CS-93-214, University of Tennessee, Nov, 1993.