Shared Memory Multiprocessors and Sequential Programming Languages: A Case Study

Lawrence A. Crowl

Computer Science Department
University of Rochester

Abstract

The Chrysalis operating system for the BBN Butterfly Parallel Processor provides a flexible, object-oriented, shared-memory environment for parallel programming. C++ is a safe, sequential, object-oriented programming language. The pairing of these environments would seem a natural fit. The development of Chrysalis++, an interface between C++ and Chrysalis, indicates that subtle assumptions within each system lead to surprising inconsistencies. Any combination of sequential programming language and shared-memory multiprocessor operating system is likely to be plagued by these inconsistencies because they center around differing assumptions on the nature of the computational environment. This paper describes some of the inconsistencies that arose during the development of Chrysalis++ and suggests possible solutions.

1 Introduction

The Chrysalis operating system [2] for the BBN Butterfly Parallel Processor [1] provides flexible, object-oriented facilities based on shared memory that support very efficient parallel programs. However, this environment is both cumbersome and error-prone. The C++ programming language [9, 8] provides a flexible, statically typed, efficient, concise, and object-oriented notation for sequential programs. Intuition suggests that an intelligent combination of the two would lead to a good parallel programming environment. The combination approach has practical advantages in that the resulting parallel programming environments are simpler to develop and construct. However, our experience shows that creating a consistent semantic view out of this particular combination is surprisingly difficult. Any similar combination is likely to provide corresponding difficulties.

This work explores the combination of a shared memory, multiprocessor operating system with a good sequential programming language in an attempt to provide a good parallel programming environment. The resulting environment adopts the same model of parallelism as Chrysalis. Specifically, heavyweight processes interact through the use of shared memory and system objects. The combination of Chrysalis and C++ led to Chrysalis++ [3, 4], an interface between the two. Chrysalis++ is not a reimplementation of C++, it is a "library" for accessing Chrysalis functions. The C++ compiler and the Chrysalis operating system have not been modified, however the C++ run-time routines were modified in porting C++ from Unix to Chrysalis. Because reimplementation of the compiler was not an option, this paper does not distinguish between assumptions in the C++ language and assumptions in its standard implementation.

Chrysalis++ provides impressive reduction in user code complexity over equivalent Chrysalis/C code. However, the effort was not as completely successful as hoped, despite the fact that Chrysalis and C++ are both "object-oriented". The difficulties center around differing assumptions in the computational model of C++ and Chrysalis. Specifically: the mechanism for naming Chrysalis objects is distinct from the mechanism for naming C++ objects; C++ provides insufficient mechanisms to support type checking between processes; shared system objects introduce a notion of scope and object lifetime beyond the procedure and process notions supported by C++; shared objects require a new mechanism for object allocation which C++ does not support; parallel systems have a need for dynamically sized structures which C++ does not adequately support; and shared objects place heavy demands on weak generic definition facilities. After a description of Chrysalis and the Chrysalis++ approach, this paper discusses each of these difficulties in turn. Because of the inherent nature of these difficulties, any combination of a sequential programming language and a shared memory multiprocessor will have to address these difficulties.

2 Chrysalis

Shared memory multiprocessors encourage a programming style in which processes access vast amounts of memory. Multiprocessors provide the necessary memory, however, their usual implementation with standard microprocessors limits the address space of any one process. The Butterfly is representative of the class of multiprocessors in which memory is shared among processes whose virtual address spaces are smaller than the available physical memory. For instance, the Butterfly may contain up to one gigabyte of physical memory, but any one process can address only 16 megabytes at a time. Even microprocessors with 4 gigabyte virtual address spaces may provide insufficient virtual address space for large problems on large machines.

Since the virtual address space of a process on such machines is insufficient to directly address all memory a process may need, the operating system must provide a mechanism for mapping segments of memory to regions of the virtual address space as needed. For instance, Chrysalis allocates each processor node's local memory into a number of memory objects. Operations exist to create, delete, map and unmap a memory object. The map operation places any memory object in the system into the virtual address space of the process performing the map.

Because virtual addresses cannot name these segments of memory, a second naming facility must be present. Chrysalis names these segments of memory with *object identifiers* (`OID`s). The system also names non-memory objects with `OID`s. Processes communicate by passing each other `OID`s and then acting on the objects to which they refer. The system provides microcode support for process synchronization objects such as atomic queues and synchronization event managers. The Chrysalis design allows explicit programmer control over system operations.

The Chrysalis programming environment is based on the C programming language [5]. This environment requires the user to write a substantial amount of code that relies heavily on the explicit use of pointers, size calculations, and type casting, which in turn severely reduces the already minimal effectiveness of the compiler type checking provided by C. As a result, the programming process is error-prone. Since most programming mistakes on the Butterfly manifest themselves late in program development as "bus errors", tracking down these mistakes can be extremely time consuming.

3 The Chrysalis++ Approach

Many of the problems programming in the Chrysalis environment can be attributed directly to the error-prone sequential programming language C, rather than parallelism. C++ is a more expressive, error-resistant programming language than C. It is a statically-typed and object-oriented superset (mostly) of C. In addition to the C types, C++ allows programmers to encapsulate data within objects. Each object is an instance of a class. A class definition statically defines the component variables and types of each instance. It also defines the functions and operations that may be performed on instances of the class. Such definitions may be inherited from a single base class. Creation of instances of a class automatically invokes a user-defined *constructor* operation, and deletion automatically invokes a user-defined *destructor* operation. Since creation and deletion are integral parts of declaring a variable, these functions allow explicit programmer control over an object's lifetime. Other facilities include programmer control of object allocation, reference parameters, inline functions, default parameters, operator overloading, and generic definitions.

The principle idea behind Chrysalis++ is to use C++ objects to encapsulate details of Chrysalis object management and to use implicit C++ object lifetime management facilities to manage Chrysalis system object lifetimes. The automatic calling of object constructors and destructors in C++ provides a powerful mechanism for insuring proper creation and destruction of data objects and allows Chrysalis++ to merge creation of and access to Chrysalis objects with the declaration of the variable used to represent them. This means that programmers cannot improperly create or access Chrysalis objects. Chrysalis++ objects also merge object deletion into the automatic C++ variable deletion mechanisms. Because of this, the programmer need not explicitly handle Chrysalis object deletion. Since the principal difficulty in the Chrysalis environment is creation, access, and destruction of Chrysalis objects, defining C++ classes to control access to Chrysalis objects creates a more concise, error-resistant programming environment. This association between language object and system object greatly reduces the amount of coding required of programmers and simultaneously increases the likelihood of correct code at run time because programmers cannot improperly create or access Chrysalis objects.

4 Object Naming

Programming languages, such as C++, name objects at run-time by their addresses. Because of address space limitations, shared-memory multiprocessors often introduce a second naming facility beyond addresses. Programming languages generally provide no support for this second level of naming. Managing the translation between one level of naming and the other presents a problem when trying to provide a consistent interface. In translating Chrysalis names (`OID`s) into C++ names (addresses), Chrysalis++ must distinguish between creating a new C++ name that represents a new Chrysalis object versus a new C++ name that provides access to an existing Chrysalis object, and it must control object initialization so that it initializes only new Chrysalis objects.

The distinction between creating a C++ name for a new Chrysalis object versus an existing object is straightforward. Chrysalis++ achieves the distinction by overloading the constructor operation. When the first parameter to a variable constructor is an `OID`, the variable represents access to an existing object referenced by the `OID`. Otherwise, the constructor represents creation of a new object.

When C++ creates a new name (or address) for an object, it initializes that object by calling the appropriate constructor. Under the standard sequential programming environment that C++ assumes, this is reasonable behavior because all new names refer to new objects. However, in mapping an `OID` of an existing object to an address, the new name (address) does not refer to a new object. Chrysalis++ must avoid initializing the object a second time.

Controlling the initialization of most system objects is straightforward. Since most Chrysalis object types have a single representation as a (possibly generic) C++ class, the implementation of the C++ class can implement two different constructors. One constructor performs the actual creation and initialization, and stores the `OID` within a representation object that the C++ code references. The other constructor just stores the `OID` within the representation object. Chrysalis++ translates operations on the representation object into the appropriate Chrysalis system operation on the `OID`. There is never an explicit address for the system object.

Memory objects contain C++ objects that are shared among many processes, so an address must be present for the user code to operate on the data contained within the memory object. On the Butterfly, programs generally cannot have all memory objects they reference mapped into the address space at all times because the hardware address mapping registers are a scarce resource. Since the mapping operation is too expensive to perform for each use of the memory object, automatic management of the mapping registers is a non-trivial task. Programmers must explicitly manage the duration over which the object is mapped in. Chrysalis++ achieves this by associating a C++ object with each memory object that is mapped in. The creation of the C++ object maps the memory object in, and the deletion of the C++ object maps the memory object out. The lifetime of the C++ object is equivalent to the time the memory object is visible.

Controlling the initialization of memory objects is more difficult because forcing the programmer to provide two different constructors does not reduce the object management code. This is contrary to the goal of Chrysalis++. It is relatively easy to avoid initialization of memory objects. The addresses generated from mapping the memory object into the process's address space are cast into pointers of the appropriate types. This avoids initialization when not wanted, but removes the capacity to easily initialize an object when desired because there exists no mechanism in C++ to call a constructor at a given address. The Chrysalis++ solution is to force programmers to provide an initialization routine that performs the work of the constructor. These problems argue for the separation of providing a name (address) for an object and its initialization.

Another problem is that system memory objects may be mapped to different addresses in different processes, so C++ objects contained within system memory objects must not contain addresses, *i.e.* pointers. This limits the flexibility of the programmer in providing dynamic data structures within a shared object.

5 Type Checking

C++ associates a type with each object name and statically enforces type consistency. In particular, pointers have types consistent with the objects to which they point. However, Chrysalis `OID`s are untyped. Even if `OID`s had types, the C++ compiler could not correctly associate the `OID` types with object types as it does with pointers because processes pass `OID`s between themselves outside the normal C++ parameter passing mechanisms.

Chrysalis++ provides a run time solution to type consistency. It hashes the type name of an object into a byte code and then tags the object with the byte code when creating the object. A constructor setting up access to such an object will check the expected byte code of the object against the actual byte code stored with the object.

There are some drawbacks to this mechanism. First, the hashing mechanism may map two different type names to the same code, although the probability is low. Second, type hashing is not fully compatible with the C++ type scheme because it uses the type name. For instance, a `typedef` from one type to another will generate different codes because the names are different, even though they are the same type under C++. Third, because the type hashing mechanism uses the C preprocessor to convert the type name into a C++ manipulable string, Chrysalis++ programs have greater lexical restrictions than normal C++ programs.

When fully integrated into the compiler, the type hashing scheme can be effective [7]. This allows type-safe dynamic configurations of statically typed processes [6]. If the language processor does not provide for run-time type checking between processes, the language processor should either compile all processes together, or provide access to its notion of type identity so that the program can perform its own type checking.

6 Object Lifetimes

C++ provides three notions of object lifetime: static, automatic, and free. Static objects are created and initialized at program initialization time. Such objects are destroyed when the program dies. Automatic objects are created and initialized upon activating a function and destroyed when exiting a function. Static and automatic object lifetimes are the same if one views the static variables as belonging to a program function which is entered before the call to the main function. Free objects are explicitly created and initialized with the `new` operator, and explicitly destroyed with the `delete` operator.

In practice, Chrysalis provides two notions of object lifetime, depending on the ownership of an object. An object may own other objects. By default, the process object that creates an object owns that object. When an object is deleted (or a process terminates), any objects it owns are also deleted. A process which owns an object may choose to assign ownership to another object. Generally, processes only assign ownership of an object to the system. This is called disowning an object and allows the created object to outlive the process that created it. Thus, in practice, programs use only two notions of object lifetime: *owned* objects are bound to the lifetime of the process which creates the object and *disowned* objects are free to exist indefinitely.

Since Chrysalis++ uses C++ objects to represent Chrysalis objects, it can use the lifetime of the C++ object to infer the desired lifetime of the Chrysalis object. Since both Chrysalis owned objects and C++ static or automatic objects are implicitly destroyed by a dying parent process (or activation), Chrysalis++ represents owned objects with static or automatic C++ objects. Also, since both Chrysalis disowned objects and C++ free objects must be explicitly destroyed, Chrysalis++ represents disowned objects with free C++ objects. This is possible because C++ provides a mechanism to allow an object constructor to determine whether or not an object is being created in free store. This binding of Chrysalis object lifetime to C++ object lifetime represents a conceptual cleanliness because users do not have to keep track of whether or not a variable references a valid object—it always refers to a valid object.

For example, if a parent process creates a child process as a variable local to a function, the child process is owned by the parent process. The parent process will automatically wait upon completion of the child process when exiting the scope of the function. On the other hand, if a parent process creates a child process out of free store (*i.e.* with the `new` operator), the child process is disowned. The parent process will wait upon completion of the child only if the `delete` operator is explicitly applied.

However, there are some quirks to this approach. Because Chrysalis processes cannot disown the synchronization event managers they create, the Chrysalis++ event mechanism does not recognize different lifetimes for events. Also, Chrysalis atomic queues borrow buffer memory from the creating process, so the termination of the creating process will result in the loss of the queue even when the queue itself has been disowned. It is possible to reprogram the Chrysalis mechanism to behave consistently. However, the run-time cost to implement this under Chrysalis is substantial. Therefore, the Chrysalis++ atomic queue mechanism does not recognize the distinction in lifetimes.

A further inconsistency resides in the deletion of Chrysalis memory objects. Deletion of memory objects is delayed until no process has the object mapped into its address space. Chrysalis++ cannot invoke the user's destructor for a shared object residing within a memory object because another process may have the object mapped into its address space. Otherwise, the other process would be able to use the shared object after its data structure had been invalidated by the destructor. Because of these delayed deletion semantics, Chrysalis++ will never call the user's shared class destructor.

This is an instance of a highly inconsistent result arising from creating an interface between an object-oriented programming language and an object-oriented operating system. This highlights a fundamental question concerning the meaning of shared objects in a parallel programming environment. *When does a delete take effect?* Object-oriented systems should carefully consider object lifetimes and apply them uniformly.

7 Dynamic Sizing

Programs for multiprocessors have a great need for dynamically sizing data structures because they need to map varying problem sizes onto varying resources. For instance, Chrysalis programs usually determine the number of available processors at run time and then start one process per available processor. This approach minimizes the context switch overhead in comparison to an approach that generates a fixed number of processes.

The intuitive approach to creating multiple processes is to allocate an array of processes. This is what naturally comes to mind when considering a set of processes of the same type. Since the program dynamically computes the number of processes, the program must dynamically compute the size of the process array. However, the number of elements in C++ local arrays is determined at compile time. So, local arrays can only support a fixed number of processes.

When sequential programs in C++ need dynamically sized data structures, they use free store. There are two drawbacks to this approach. First, for sequential programs as well as parallel programs, this approach requires more programmer management than do statically sized local arrays because the programmer must explicitly manage array deletion. Second, the previously outlined approach to linking system object lifetimes to C++ object lifetimes would imply that the system objects must be disowned upon creation. Creating disowned objects and processes can be dangerous because failures in a program may force the end user to manually delete hundreds of processes and objects.

The best solution to this problem is for the programming language to support determination of the number of elements in a local (*i.e.* automatic) array upon allocation. This requires the compiler to implement a level of indirection transparent to the user. An alternative approach might take advantage of the generic facilities in C++ to define an array which is sized at allocation.

Programmers often use Chrysalis memory objects to implement varying sized message buffers and similar structures. These structures usually have a few fixed header variables and a dynamic amount of buffer space past the header. C++ cannot represent such structures directly because C++ classes are statically sized. This problem can be solved with the concept of an open structure. An open structure is one in which the last component is an array whose number of elements is not bound until allocation. Both C and C++ programmers simulate open structures by defining a structure with a one element array as its last component and then requesting the space appropriate to the number of elements actually needed from `malloc()`. This approach requires considerably more programmer management than statically sized structures because the programmer must explicitly calculate the structure size, call the underlying allocator, and abandon automatic lifetime management. Open classes have an additional problem in that computing the number of elements needed in the tail array of an open class is best left to the class itself. This in turn requires the ability to do some user-specified computation before the actual allocation of space. This concern leads to the desire for the separation of size determination from allocation.

In programming languages, the determination of object size is not always independent of object lifetime. However, in operating systems, they usually are independent. The more restrictive programming language limits the semantics and flexibility of the combination of such systems. In particular, common Chrysalis idioms have no straightforward representation in C++.

8 Object Allocation

C++ provides three forms of variable allocation: static allocation, where the linker allocates space; stack allocation, where function activation allocates space; and free store allocation, where `malloc()` allocates space from within a process's address space. Programs implicitly handle static and stack allocation, but explicitly handle free store allocation. Chrysalis introduces a fourth notion where the system allocates space outside the address space of the process.

One obvious approach to system allocation is to have a variant of the `new` operator that allocates a memory object instead of using `malloc()`. C++ provides a mechanism to replace the free store allocator, but assumes that there is exactly one such mechanism. One must redefine the `new` operator for all uses or none. Since Chrysalis++ programs must still use the normal free store allocation, Chrysalis++ cannot redefine allocation to use only system allocation. There is no room within the C++ design for two different explicit allocators.

C++ does allow a class constructor to take over and provide its own allocation in the place of the standard free store allocator, but because the allocation is explicit within the constructor, the constructor must have full knowledge of where and how to do the allocation. This scheme works well for such things as object-specific allocation caches, but does not work well when the programmer wishes to use different allocators without class specific knowledge. One approach is for each class to accept the allocator as a parameter to the constructor. Unfortunately, the class programmer must insert the code to explicitly handle the allocation and remember the allocator for later explicit deallocation.

Languages that support multiple allocators must allow the separation of allocation from initialization. This allows a programmer to independently define allocation and initialization. The `new` operator could then specify the appropriate allocator.

9 Generic Definitions

Generic definitions provide a mechanism for writing complex code exactly once. Chrysalis++ uses generic definitions to hide the details of interaction with Chrysalis for a broad class of system object usages. For instance, the initial atomic queue approach provided a separate queue type for each of the primary types of enqueued values: integers and `OID`s. This forces users to abandon type checking with explicit casting when enqueuing objects of different types. The final approach implements queues as a generic class taking the type of the element to be enqueued as an argument. The generic definition hides the details of atomic queue manipulations in one abstraction.

C++ provides a mechanism for generic class definitions based on C preprocessor macros, [8]. A generic class is defined as a macro that takes the parameter class as an argument. The instantiation of a generic class results in a specific class declaration. Programmers then use objects defined with this specific class declaration. Basing the generic class mechanism on the C preprocessor causes several problems including lexical requirements more restrictive than normal C++ conventions, restrictions on the forms of comments, and the lack of assurance of proper argument semantics. This leads to the conclusion that the C++ compiler should implement generic classes directly.

Chrysalis++ uses generic definitions to set up access to memory objects. Because Chrysalis++ does not actually use the class contained within the memory object, it does not need to know the number and types of the contained class's constructor parameters. However, the generic definition facilities support a known and fixed number of parameters for the contained class's constructors. The consequence is that all contained classes must conform to a parameter convention that is not strictly necessary. A more general approach would allow generic functions with an unspecified number of arguments that would in turn be passed to the contained class's constructor.

Most C++ programs can avoid generic definitions, but Chrysalis++ programs generally cannot because many of the basic facilities, such as the atomic synchronization objects, are generic. The result is that programmers are forced to deal with the weaknesses in the C++ generic facility.

10 Conclusions

Combining a good sequential language with a flexible multiprocessor operating system does not necessarily lead to a good parallel programming environment, even when both are object-oriented. While Chrysalis++ represents a substantial improvement over the Chrysalis programming environment, it does not completely hide inconsistent assumptions between C++ and Chrysalis. The problems in Chrysalis++ are inherent in the combination and are likely to beset any similar combination.

The binding of Chrysalis system objects to C++ language objects provides simpler programming within Chrysalis++. This binding is effective in large part because of the automatic invocation of object constructors and destructors that comes with C++ variable declaration. This capacity allows automatic management of objects. Languages which support constructors and destructors help to support shared memory environments.

In the near term, multiprocessors may continue to have virtual address spaces smaller than physical address spaces. Programming languages (and their implementation) should provide mechanisms to deal with this. In particular, languages should not assume that creation of an address for an object is also creating the object. In addition, languages should provide a mechanism for initializing an object at a known location.

The programming language processor should either perform inter-process type checking, or be able to provide its notion of type (*e.g.* as a numeric code) to the program so that the program can perform inter-process type checking. Languages that compile all processes together can avoid run-time type checking at the cost of more statically configured programs.

Shared-memory systems introduce additional notions of object lifetime. Programming languages that recognize this will support multiprocessing better. In particular, the C++ ability to determine a variable's lifetime and use it to set the corresponding system object's lifetime is a useful capability. On the other hand, inconsistent system object lifetimes lead to inconsistent program object lifetimes. Operating system designers should provide consistent system semantics in order to allow a consistent programming environment to be built upon the operating system.

Dynamic object size determination and object lifetime in operating systems tend to be independent, but not so in programming languages. In order to support the full range of size determination and lifetime combinations supported by the operating system, the programming language should support dynamically sized local arrays and structures. This leads to the separation of object size determination from variable lifetime and memory allocation.

Operating systems introduce yet another mechanism for object allocation with memory objects. In order to prevent object programmers from coding every possible allocation mechanism within a potentially shared object, the language should support the separation of memory allocation from object initialization. This separation also allows the specification of a memory allocator to be independent of any particular object definition.

Parallel systems place heavy demands for good abstraction facilities on the programming language. The abstractions necessary to interface with the operating system will stress the generic definition facilities such that inadequacies in the facilities will be apparent to the programmer. Languages that support good generic definition facilities will provide good software leverage.

Chrysalis++ represents a substantial improvement over the Chrysalis programming environment. However, Chrysalis and C++ have subtle inconsistencies that prevent merging them into a clean, consistent programming environment. Similar inconsistencies doom any attempt to build a parallel programming environment out of a pre-existing sequential programming language and a pre-existing shared memory operating system. A single, unified design for a parallel programming environment is more likely to provide a consistent, usable system.

Acknowledgements

Special thanks to Neal M. Gafter, Thomas J. LeBlanc, John M. Mellor-Crummey, and César A. Quiroz for their many helpful comments in the development of this paper.

This work was supported by United States Army Engineering Topographic Laboratories research contract number DACA76-85-C-0001 and National Science Foundation Coordinated Experimental Research grant number DCR-8320136.

References

[1] BBN Laboratories, Cambridge, Massachusetts. *Butterfly Parallel Processor Overview*, June 1985.

[2] BBN Laboratories, Cambridge, Massachusetts. *Chrysalis Programmer's Manual*, June 1985.

[3] Lawrence A. Crowl. Chrysalis++. Butterfly Project Report 15, Computer Science Department, University of Rochester, December 1986.

[4] Lawrence A. Crowl. An interface between object-oriented systems. Technical Report 211, Computer Science Department, University of Rochester, April 1987.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, 1978.

[6] Michael L. Scott. Language support for loosely-coupled distributed programs. *IEEE Transactions on Software Engineering*, SE-13(1):88–103, January 1987.

[7] Michael L. Scott and Raphael A. Finkel. A simple mechanism for type security across compilation units. Technical Report 541, Department of Computer Science, University of Wisconsin at Madison, May 1984. A later version will appear in the IEEE Transactions on Software Engineering.

[8] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[9] Bjarne Stroustrup. An overview of c++. *ACM SIGPLAN Notices*, 21(10):7–18, October 1986.