References

[1] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sunderam, *User's Guide to PVM: Parallel Virtual Machine*, ORNL/TM-11826, Mathematical Sciences Section, Oak Ridge National Laboratory, Sept. 1991.

[2] C. M. Chase, et. al., "Paragon: A Parallel Programming Environment for Scientific Applications Using Communications Structures," in *IEEE International Conference on Parallel Processing*, pp. 211-218, 1991.

[3] K. Lee and D. Gannon, "Object Oriented Parallel Programming Experiments and Results," *Supercomputing'91*, pp. 273-282, 1991.

[4] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[5] V. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience* Vol. 2, No. 4, Dec. 1990.

7 Appendix

Here we present template codes for a sample host/node program. To provide some comparison, this example is analogous to those found in the PVM User's Manual, [1]. It should be pointed out that while the master/slave model is simple to understand and useful for illustrative purposes, neither the PVM or SMP system is limited to this parallel programming model.

7.1 Host Template

```
#include "smp++.h"
const int nprocs = 4;

main()
{
    process P(nprocs);
    vector<double> result(nprocs), data(100);

    process me = whoami();

    for (i=0;i<nprocs;i++)
        P(i) = initiate("nodeprogram");

    /*-------- Begin user program -------*/


    /* broadcast data to all node programs */
    msgout(P, nprocs) << data << flush();


    /* wait for results from nodes */
    for (i=0;i<nprocs;i++)
        msgin() >> result(i);


    /* End user program */
}
```

7.2 Node Template

```
#include "smp++.h"

main()
{
    vector<double> data(100);
    process me = whoami();

    // begin user program


    msgin() >> data;
    process host = msgin.process();


    msgout(host) << user_routine(data);  // send result to host


    // end user program
}
```

at abstracting many of the low-level system dependent message passing details that program developers of distributed object-oriented libraries must contend with.

The reality is that any message-based C++ parallel programming library, however elegant, must at some point interact with the underlying system's message passing primitives. Typically, this interface has taken the form of a C or Fortran library consisting of various versions of `send()` and `receive()` . These functional interfaces, however, typically lack any standard in functionality or compatible argument lists. Coupled with the added burden of correctly tagging the data items' type for proper format conversions in heterogeneous systems, such interfaces are prone to common programming errors and safeguarding against compile-time and run-time anomalies.

The idea of SMP is to extend the concept of data streams in C++ to heterogeneous message passing systems. This is based on the fundamental notion that interprocessor communication on distributed systems is simply another form of I/O. The `msgstream` class hierarchy of SMP supports both fundamental data types, variable length vectors of such types, and extensions to integrate user-defined objects into message streams through the insertion `<<` and extraction `>>` operators. Variable length vectors of such types, and integration of user-defined objects are accomplished with the same syntax.

Efficient design and coding techniques have resulted in an insignificant run-time overhead compared to an equivalent program based on the explicit calls to low-level communication functions. The runtime overhead of implementing the SMP system without run-time checking is essentially insignificant due to the function inlining capabilities of C++ compilers. At this point, the SMP interface essentially plays the role of a sophisticated macro preprocessor which translates `msgstream` statements into the system-specific `send()` and `recv()` calls. This transformation is performed completely at compile-time, without affecting run-time performance. When run-time type checking is invoked, each message element is tagged typed; The overhead for scalar data elements can be significant (in the worst case, the message doubles in size to accommodate the information tags); however, for variable-length vectors this overhead is only encountered once, essentially hiding any measurable overhead for communicating long vectors. Typically, run-time type checking is used in the debugging stages of program development and then later turned off for final production runs.

We have illustrated how the improved run-time and compile-time type checking of SMP messages aids the monitoring and debugging phases. The responsibility of correctly tagging data items into messages is passed from the programmer to the compiler. Furthermore, at the user's option, run-time type checking can detect (and even correct) type mismatches. The user has the option of merely reporting such type-mismatch errors, or performing automatic coercion where possible. This is completely transparent to the user application.

Furthermore, we have demonstrated how its flexible interface can be extended to include the communication of user defined objects as atomic units. To integrate a user-defined object type into message streams, we need only overload the `<<` and `>>` operators with the proper action. Typically this involves communicating the object in terms of its fundamental data elements (scalars or vectors of `char` , `int` , `float` , or `double`). Because of the inheritance properties of the SMP classes, we need only define the interface for the base class `msgstream` , rather than for each separate class. In short, through function overloading, inheritance and polymorphism, communicating sophisticated user-defined objects is no different than the basic data types.

Finally, we have described an implementation of the SMP interface on the Virtual Parallel Machine (PVM) software environment for programming heterogeneous systems. This software prototype, SMP/PVM, successfully incorporates the most important features of the SMP interface on a practical platform. We are investigating further support for performance and debugging instrumentation. Extended runtime systems for SMP/PVM could support various levels of tracing message traffic, type matching of message contents, PVM event occurrences (such as process creation and information queries), and timing information.

6 Acknowledgements

The author would like to thank Adam Beguelin, Al Geist, Jack Dongarra, Chuck Romine, and Vaidy Sunderam for their helpful comments on this work.

Table 5: SMP/PVM C++ User Interface: Receiving Messages

`msgistream& msgistream()`
`msgistream& msgistream(int msgtype)`
`msgistream& msgistream(int num, int *msgtypes)`
`msgistream& msgistream (vector<int> msgtypes)`
- receives a message of specified type(s) or 'any' if msgtype is omitted (Blocking).

`msgstream >> x` - extracts values from the corresponding message stream and assigns it to object `x`.
`int rcvinfo(int *bytes, int *msgtype, process *p)` - returns the length, type, and sender of last message received or probed. Returns -1 if no message was received or probed.

`int probe()`
`int probe(int msgtype)`
`int probe(int num, int *msgtypes)`
`int probe(vector<int> &msgtypes)`
- probe for message arrival of specified types, or 'any' if msgtype is omitted or set to -1. Returns message type or -1 (not arrived).

Table 6: SMP/PVM C++ User Interface: Synchronization

`int barrier(char *barrier_name, int num)` - blocks caller until num calls with same barrier name made. Returns < 0 if error.
`void ready(char *event_name)` - sends signal with specified (abstract) name.
`void waituntil(char *event_name)` - suspends caller until specified signal name occurs.

and the message type. There are also various versions of the `probe()` function which allow one to check if a message of a specific type (or list of types) has arrived.

4.3 Synchronization

The synchronization functions are essentially unchanged from PVM and are included here for completeness. The `barrier()` primitive allows a group of processes to synchronize. A string argument identifies the barrier name and a second argument specifies the number of cooperating processes involved in the barrier. A typical use, such as `barrier("name", N)` will block until all other $N-1$ similar barrier calls have been made by other processes.

The `ready()` and `waituntil()` primitives allow for implementation simple semaphore operations. The `waituntil()` causes the process to pause until the named semaphore is set by the `ready()` call. See the PVM User's Manual [1] for more details.

5 Conclusions

We have presented an object-oriented message passing interface in C++ for heterogeneous systems. Although fully functional, this interface was not designed to be used directly at the application level. Rather, it is aimed

Table 3: SMP/PVM C++ User Interface: Process Control and Information

`process` **initiate**`(char *object_file [, char *arch])` - initiates a new process and returns its identification. If the optional architecture is not specified, the system will choose one.
`process` **initiatem**`(char *object_file [, char *machine])` - initiate a process on the specified machine and returns process identification. If the machine name is not specified, or if it is given as ".", then the initiating machine is used.
`int` **terminate**`(process &p)` - terminates a specified process. Returns < 0 if error.
`process` **whoami**`()` - returns current component name and instance.
`int` **pstatus**`(int *ncpu, int *nformats)` - returns number of hosts and data formats. Returns -1 if error.
`int` **status**`(process &p)` - returns 1 if specified process is active, 0 otherwise.

Table 4: SMP/PVM C++ User Interface: Sending Messages

`msgstream& msgstream()`
`msgostream& msgostream(process p [, int msgtype] )`
`msgostream& msgostream(vector<process> p, [, int msgtype])`
- (re)initializes message stream, with destination and optional message type information.

`mgstream& msgstream << x` - inserts object `x` into a message stream. An explicit `flush()` is required to send the contents of the message stream.
`msgostream& msgostream << flush()` - sends contents of specified message stream. Every outging message requires an explicit `flush()`.

which begins running the process "multigrid" on the specific machine "demidoff.utk.edu". Omitting a machine name from the constructor results in the PVM choosing some available machine from the current configuration. Alternately, processes can be spawned via the `initiate()` method, as in

```
process P[10];
for (int i=0; i<10; i++)
     P[i] = initiate("multigrid");
```

There are also status functions, such as `whoami()` that return the current process identification, and `status()` that report if such processes are currently running on remote nodes.

4.2 Receiving and Sending Messages

The syntax for message sending in SMP/PVM is illustrated in Table 4. The `msgstream << x` operation is predefined for `char, int, long, float, double, process` data types as well as vectors thereof.

Besides the basic mechanism for receiving messages, the SMP/PVM implementation also supports various enquiry operations such as `rcvinfo()` , which identifies the sending process, the message length (in bytes),

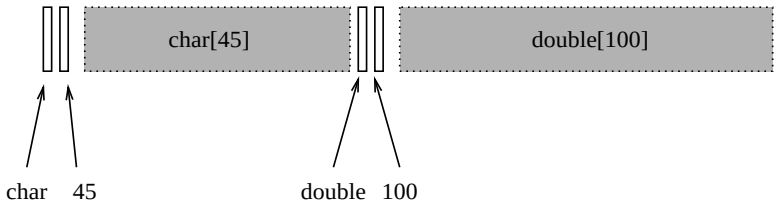

Figure 4: Schematic for encoding scalar and vector data types in messages.

When extracting data items from an incoming message, there are four possible compatibility modes for matching types between message data element and the variable type into which they will be read into. In the strictest mode, the two types must match exactly, otherwise it constitutes a runtime error; in another mode, automatic type coercion is performed whenever possible, generating a runtime warning message that the coercion took place. A third mode the last mode performs such coercions without generating messages. A finally, the fourth mode performs no type checking at all and adds no runtime overhead.

We recommend using strict type matching mode whenever possible. If absolute performance is a must, use the strict type-checking during the debugging and program development stages, and turn the runtime type-checking features off to optimize runtime performance.

Because the optional runtime type checking mechanism alters the actual contents of the message, it can be selectively turned off to be compatible with other unencoded messages sent by processes utilizing the system-dependent message passing library, but not SMP. This allows integration of SMP modules with third-party libraries and other codes written outside of the SMP paradigm.

4 An implementation of SMP using PVM

This section describes a working implementation of the SMP interface utilizing the Parallel Virtual Machine (PVM) environment [1], [5]. The PVM software environment easily configures networks of workstations and supercomputers via a simple message passing library. The PVM software package is freely available and widely used in industrial and research centers. Every feature of the abstract SMP model is implemented, with the exception non-blocking receives (currently not supported in PVM version 2.4). We will use SMP/PVM in the sequel when referring to the software implementation, to distinguish it from the underlying abstract SMP interface.

4.1 Controlling Remote Processes

For consistency, SMP/PVM utilizes the underlying PVM convention for identifying remote programs: each process is uniquely identified by a component name (a character string denoting the name of the executable) and an integer-valued instance number. Since the SMP interface deal primarily with the message-passing aspects, the SMP/PVM interface for process control has been relatively unchanged from PVM. An SMP/PVM program is identified by an alphanumeric string (denoting the name program's executable name) and an integer-valued *instance* number: 0, 1, 2, ..., for each new process with the same component name. This name/pid pair uniquely identifies any process. In the SMP/PVM interface, we bundle these two items together into a single `process` object. In this manner, we can later alter the internal representation without disturbing the interface. For example, the `process` object may also contain other relevant information, such as the machine it is running on, what architecture class the machine belongs to, and so on. A list of basic SMP/PVM processor control and information operations is given in table 3.

Processes can be spawned using object constructors as in

```
process p("multigrid", "demidoff.utk.edu");
```

```
msgstream& operator<<(msgstream &s, complex &z)
{
      s << real(z) << imag(z);
      return s;
}
```

and to receive them as

```
msgstream& operator>>(msgstream &s, complex &z)
{
      double r, i;
      s >> r >> i;
      z = complex(r,i);
      return s;
}
```

Now we can write message passing expressions such as

```
int i,
double x, y;
complex z;
...
msgout(proc, msgtype) << z << i << x << flush();
```

Because all SMP stream objects are derived from the same base class, `msgstream` , we need not specify a definition of the stream operators `<<` and `>>` for the remaining stream classes. This is possible because the basic `<<` and `>>` methods have already been defined for the basic data types. When an expression such as `msgout << z` is encountered, the system calls the `operator<<(msgostream &, complex &)` operator if it exists; otherwise the `operator<<(msgstream &, complex &)` operator is executed, and then the `operator<<(msgostream &, double &)` code (already defined by SMP) is executed.

There is some small overhead for this virtual function call (the correct version of the `<<` operator is called at run-time. Typically, this overhead is relatively insignificant, although in certain situations (such as calling the virtual operator from a tight loop) it can be noticeable. In such cases, one can explicitly define the `>>` and `<<` operators for the derived SMP stream classes and resolve the correct operator call at compile time.

3.7 Runtime Type Checking

The SMP interface library provides support for optional runtime type checking of messages. By encoding each message element with its type (for vectors we also encode the vector length), we can monitor any implicit coercions performed and identify potential type-mismatches during runtime. For example, if we send out an integer value,

```
msgout(p,t) << i << flush();
```

and read it elsewhere as a double,

```
msgin(t) >> x;
```

the value will automatically be converted at the receiving end. The extended runtime type checking is completely optional and can be dynamically turned on or off. The schematic for the data encoding is shown in fig. 4. It consists of having each message element (scalar or vector) tagged with its data type and size. (Scalars are encoded as a vector of length one.)

```
if (probe(s))  // message has arrived
    ...
```

describes how to receive an asynchronous message of `msgtype` and store the contents into the objects `x`, `y`, `z`. The message contents of non-blocking stream may only be delivered to a `msgdesc` object which has been previously defined. In essence, this `msgdesc` object acts as a list of addresses describing where to store incoming message fragments. The `probe()` function is non-blocking, returning immediately regardless whether the message has actually arrived or not. Although one can poll for this value in a busy loop, it is often more effective to utilize the `arrive()` method which will block until the complete message has arrived.

3.5 Variable Length Vectors

The SMP interface supports contiguous vectors of various data types (integer, char, byte, float, and double) as atomic data elements. This has two advantages over packing each vector element separately: (1) it avoids the overhead of utilizing the `<<` and `>>` operators in a tight loop running over the vector indices, and (2) for runtime type-checking on heterogeneous environments, the vector element can be tagged with a simple type/length pair, rather than having to type-tag each individual element of the vector (see section 3.7).

SMP uses a generic `vector` template class and overloads the `()` operator for indexing individual elements and overloads the `<<` and `>>` operators to integrate vectors into the stream interface. For example, if `X` and `I` are declared as

```
vector<double> X(100);
vector<int>    I(20);
```

then

```
msgout(p,t) << X << I << flush();
```

sends a message with 120 elements: $x_0, x_1, \ldots, x_{99}, i_0, i_1, \ldots, i_{19}$. At the receiving end, we may write

```
vector<double> a;
vector<int> b;
msgin(t) >> a >> b;
```

The vectors `a` and `b` are automatically allocated to the correct size and filled with the proper message values.

3.6 Integrating User-Defined Objects

In this section we illustrate user-defined objects are integrated with the SMP stream interface. The goal is to have the stream syntax for new objects look as simple as for the basic data types. This is accomplished by specifying only two functions: the specifications for the `msgstream` `<<` and `>>` operators. The only requirement on the programmer is that their definitions be consistent between the operator definitions.

As an example, we illustrate how to integrate a class of complex numbers with SMP. The data class `complex` defines basic operations over the complex numbers and the real and imaginary parts can be set and retrieved via the `complex::real()` and `complex::imag()` methods — both returning `double` precision values. A new `complex` object may be constructed as

```
complex z(1.4, 2.1);
```

We can describe how to send complex numbers as

Table 2: SMP syntax for receiving messages

`msgin()`	receive any next message
`msgin(int msgtype)`	receive a message of specified type
`msgin(int *msgtypes, int numtypes)`	receive any one of specified types
`msgin(vector<int> msgtypes)`	receive any one of specified types

items can be inserted and extracted from it. Thus, both `<<` and `>>` operators are defined on instances of `msgbuffer`. Furthermore, `msgbuffer` objects can be inserted into other message streams as an atomic unit. For example,

```
int i;
float x, y;
msgbuffer b;

b << x << i;
msgout(p,t) << b << y << flush();
```

sends a single message consisting of `x`, `y`, and `i`.

While insertions into the `msgbuffer` object are copied by value, the derived class `msgdesc` implements insertions by reference in order to reduce the amount of data copying. These `msgdesc` objects are useful for vectors and other large data structures as well as specifying a destination for incoming messages (see section 3.4.)

3.4 Non-Blocking Receives

Non-blocking receives are specified using the derived class `fmsgistream`. Their main advantage over blocking receives is that they can eliminate buffer copying. Specifying the final destination of the message before its arrival saves the intermediate step of buffering the message. This not only saves time, but memory space required to buffer the message as well.

This derived communication stream does not block until messages are received, but insteads returns immediately. There is a corresponding predefined stream object `fmsgin()` that plays a similar role to `msgin` and `msgout`. The `fmsgistream` object can later be examined using `probe()` to determine if the message has arrived. For example, if `x`, `y`, and `z` are user-defined objects, the code

```
int float x, y;
double z;
fmsgistream s;
msgdesc b;

b << x << y << z;
s(msgtype) >> b;

...
```

Table 1: SMP syntax for sending messages.

`msgout << data`	insert data to current message
`msgout()`	clear previous `<< operations`
`msgout(process p [, msgtype t])`	destination process and optional message type
`msgout(process *P, int n, [msgtype t])`	broadcast to given processes with optional message type
`msgout << flush()`	send contents of message to destination

```
msgout(proc,msgtype);
msgout << i;
msgout << j;
msgout << x;
msgout << flush();
```

The first line directs the `msgout` stream to its destination and specifies the message type. As before, the insertion `<<` operations on `msgout` add other data elements to this buffer, and the `flush()` operator causes the message to be sent out onto the network. This message is sent asynchronously; it does not wait for a synchronizing receive to be posted on destination processes.

3.2 Blocking Receives

The predefined stream for message input, `msgin` , has the `>>` operators defined for extracting message contents. The current SMP interface supports message typing based on integer values, although future extensions could select on arbitrary message types. An example of receiving message in SMP is given by

```
msgout(msgtype) >> i >> j >> x;
```

which is equivalent to

```
msgin(msgtype);
msgin >> i;
msgin >> j;
msgin >> x;
```

The `msgin(msgtype)` clears the last received message, initiates a new receive, and blocks until the proper message arrives. If the `msgtype` is omitted, then the next message is retrieved, regardless of its matching type. Trying to extract an item from an empty message buffer results in a runtime error. At the user's option, warning messages can be issued if there are unread data items in the message before the user requests to receive a new one. Table 1 summarizes the common usages of the `msgout` stream.

3.3 Message Buffers and Descriptors

The SMP model utilizes two separate internal message buffers for the predefined streams: `msgin` and `msgout` . They are explicitly hidden so that programmers using this simple model need not be concerned with message buffers at all. Although this design handles most of the common situations, there are cases where it is natural to have multiple message buffers.

Message buffers can be used to explicitly build messages without sending or receiving them. Objects of the message-buffer class `msgbuffer` behave much like a FIFO (first in, first out) queue data structure:

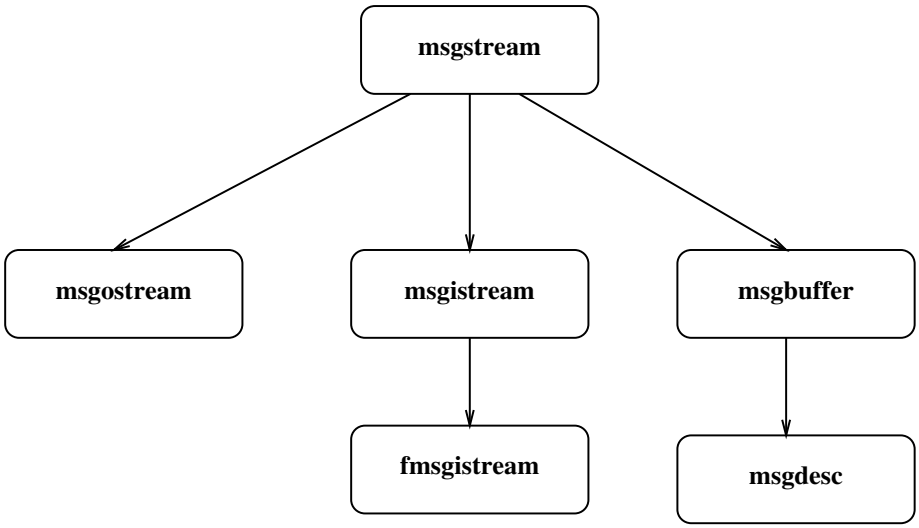

Figure 3: SMP object class hierarchy.

`msgstream` . The class `msgostream` is an output communication stream. The `msgisream` and `fmsgistream` classes control blocking and non-blocking receives, respectively. Finally the `msgbuffer` and `msgdesc` classes are in-memory message queues which are typically used to build multiple messages simultaneously, or to supply address destinations for non-blocking receives. The specific characteristics of these streams and their interactions are discussed in the following sections.

The stream object hierarchy utilizes an abstract base class `msgtream` which contains virtual methods for the `<<` and `>>` operators. In this manner, we need only to define the insertion and extraction for the base class when integrating user-defined objects. The derived classes use virtual operators to integrate user-defined objects into the SMP interface (see section 3.6).

3.1 Sending Messages

The sending of a message in SMP consists of three steps: (1) providing mailer information, destination process(es) and message type, (2) packing the information into the `msgostream` stream, and (3) explicitly flushing the buffer to move the information out onto the communications network. There is no requirement that step (1) come before (2), only that both are completed before (3).

The predefined stream `msgout` plays a similar role to standard output stream `cout` , except that information sent to `msgout` is *buffered until the stream is explicitly flushed.* Thus the message can be built piecewise while controlling message granularity for communication performance. The operator `<<` adds data elements to this buffer and the `flush()` operator physically sends the message to its destination. For example,

```
msgout(proc,msgtype) << i << j << x << flush();
```

packs three data elements (two integers and a floating point) to the process denoted by `proc` with the integer-valued `msgtype` . Message elements of fundamental data types (`char` , `int` , `float` , `double`) are typically passed by value. Therefore, the insertion operator `<<` can accept expressions of arbitrary complexity. Alternately, we may build up the message in pieces, as in

Stream I/O

C	C++
`scanf("%d %d %f ", i,j,x);`	`cin >> i >> j >> x;`

File I/O

C	C++

`read(f, &i, sizeof(int))`
`read(f, &j, sizeof(int))`
`read(f, &x, sizeof(float))`
	`f >> i >> j >> x;`

Message I/O

C	C++
?	`msgin >> i >> j >> x;`

Figure 2: Comparison of I/O capabilities between C and C++. The stream interface exploits the similarities of reading information among text files, binary files, and processor messages. This approach provides a consistent interface that focuses on *what* information is being extracted, rather than *how* the information originated.

In short, the C++ stream concept is more flexible than the standard C I/O library and corrects some of the common problems encountered with the C interface by (1) providing an integrated interface via operator overloading, (2) avoiding the common mismatch errors in `scanf()`/`printf()` for matching the `%` descriptors in the format string and the corresponding argument list, (3) providing an extendable I/O interface for file abstractions and user defined data objects (i.e. one can format other entities besides integers, floats, and characters).

3 Abstract Heterogeneous Message-Passing Model

In this section we present the abstract message-passing model employed in SMP. We will describe a working implementation based on the Parallel Virtual Machine (PVM) environment in section 4.

The SMP stream interface is designed to have a hierarchy similar to that of the stream I/O interface in C++. Our basic message-passing model supports both blocking and non-blocking sends and receives. The SMP model supports integer-based message types. Such message-passing interfaces seem to be the most common, and are similar to those of Intel, NCUBE, Caltech, PICL, and Express.

The stream object hierarchy is shown in figure 3. It consists of various classes derived from the base class

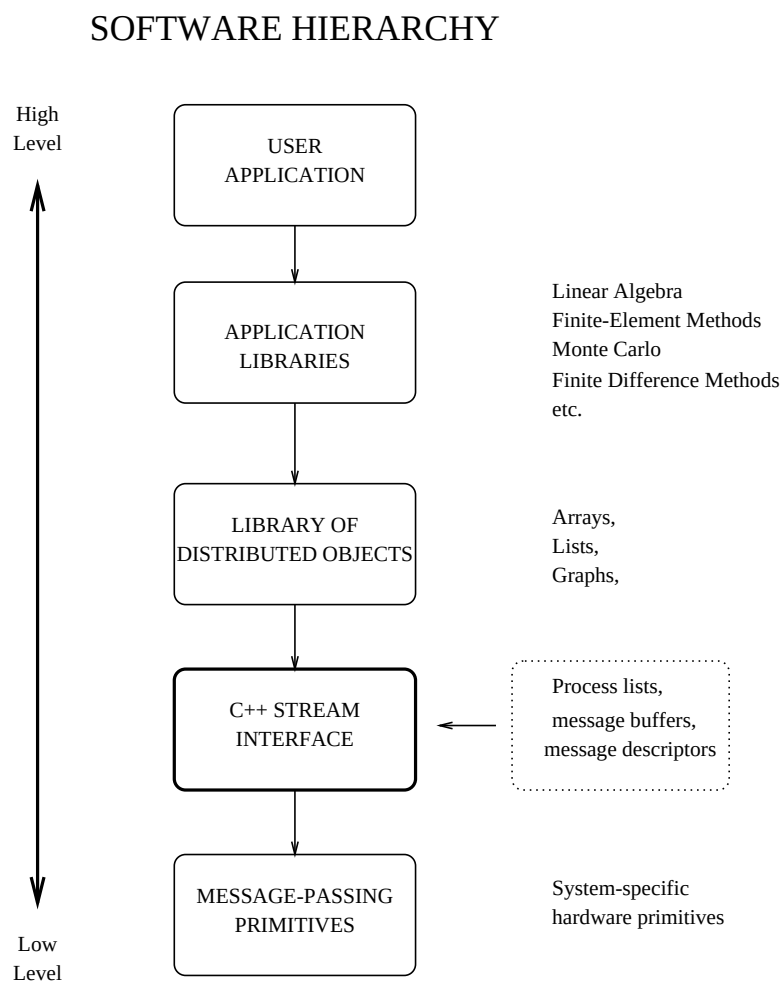

Figure 1: Hierarchical view of various programming levels for message passing systems.

responsibility. Not only does it remove the burden from the programmer, but also eliminates many of the common programming errors arising from mismatched data types in message contents.

In short, our goals are to create a portable, efficient heterogeneous computing interface in C++ that provides

- a simple function-free syntax for expressing interprocessor communication without explicit `send()` or `recv()` calls,
- an insignificant run-time overhead compared to an equivalent program based on the explicit calls to low-level communication functions,
- improved run-time and compile-time type checking of message contents,
- a flexible interface that can be extended to include the communication of user defined objects as atomic units,
- support for performance and debugging instrumentation.

Although this interface defines a fully functional communication library, it is not intended to be directly utilized at the user-application level. Rather, it is seen as a fundamental communication kernel on which further object-oriented extensions and application-specific libraries can be built on. Common object-oriented parallel programming models include data parallel (SPMD) computation models, where a large data structure (list, graph, array, etc.) is broken up over several processors. Figure 1 depicts one possible hierarchical decomposition of the various software levels for a distributed application. The SMP interface, however, supports other parallel programming models as well.

Because many of the architecture-specific details of the underlying communication kernel have been hidden via data encapsulation and object-oriented designs, this approach is applicable to almost any heterogeneous communication system with a C interface. To illustrate these concepts with a working system, we have implemented a prototype of this object-oriented interface on top of the Parallel Virtual Machine (PVM) [5] programming environment for heterogeneous networks. PVM is freely available, and runs on various architectures, including Sun-3s, SPARCstations, RS/6000's, Crays, Silicon Graphics, and NEXT machines [1]. No source code modification to the underlying PVM system was required to implement the SMP interface. The software package is called SMP/PVM and is discussed in section 4. Versions of this prototype have been tested using GNU g++ version 1.37, and the AT&T C++ translator, version 2.0.

2 Byte Streams and Message Passing

The motivation for this work arose as a natural extension to the stream I/O library of C++ [4]. The concept of a *stream* is a sink or source where information flows to or from. Examples of commonly used streams are `cin` (standard input), `cout` (standard input), and `cerr` (standard error). The stream operators `<<` and `>>` denote the direction of information flow, as in expressions such as "`cout << x`" or "`cin >> x`". In figure 2 we provide a comparison between the standard C I/O functions and streams in C++. The example shows how three values, two integers and a floating point number, are read from (a) the standard input, (b) a binary file, and (c) an incoming message. A stream can be a file, a pipe, a socket, or any other connection that allows to one to extract or insert information. The important thing to note is that stream syntax remains identical, regardless of the origination of the data. The syntax " `>> i >> j >> x`" literally means to "extract three values from some input stream". The types of these values are defined by the types of the variables. If the text file input is switched to a binary file input, we need no modification the I/O statment. The information about the stream is kept by the internal data representation of the stream object, not by the expression.

C++ I/O streams can be easily integrated with user-defined data structures. By overloading the insertion `<<` and extraction `>>` stream operators, the I/O of complex structures or objects looks as natural as that for the fundamental data types.

A Stream-Based Interface in C++ for Programming Heterogeneous Systems

Roldan Pozo
Department of Computer Science
University of Tennessee
pozo@cs.utk.edu

Abstract

We present an object-oriented interface in C++ for message passing in heterogeneous parallel architectures. This interface provides the foundation for higher-level abstractions of distributed objects. By abstracting the notion of interprocessor communication to resemble byte stream input/output, we can extend the familiar notion of C++ streams to include interprocessor communication. This new approach simplifies the message passing syntax and allows for runtime type-checking of messages. Various programming examples are presented and we discuss the relative strengths and weaknesses of this approach.

1 Introduction

We present the design of an object-oriented interface, SMP (**S**tream **M**essage **P**assing), to program heterogeneous systems. In contrast to other object-oriented solutions for homogeneous systems [2], [3], this interface consists of a low-level communication interface in C++ that serves to abstract, rather than replace, the basic message-passing communication model.

This interface supports no `send()` or `recv()` calls; instead communication is accomplished by inheritance of basic message stream classes and the overloading of the standard stream *insertion* operator `>>` and the *extraction* operator `<<` for input/output (I/O) in C++. We implement this interface as a simple library in C++ which can built upon conventional heterogeneous message passing systems.

Our approach is based on the notion that *interprocessor communication is simply another form of I/O.* There is a strong analogy between how programs interact with one another via files and how processeses communicate with one another via messages. Thus, the syntax used to denote the reading of a data value from a generic stream, for example, should appear the same whether the data originates from a file, a pipe, a message from another processor, or as input from the user's terminal. The result is that message-passing and stream I/O are united at a higher-level of abstraction.

This simple, yet powerful idea allows us to create an extendable heterogeneous programming interface. This may be achieved without changing the basic layouts of existing distributed-memory programs or sacrificing run-time performance.

One of the fundamental concerns with message passing in heterogeneous systems (in contrast to homogeneous systems) is coping with the various machine-specific data formats for bytes, integers, and floating point numbers (e.g. Cray, VAX, and IEEE floating point representations, and big-endian vs. little-endian byte orders). Thus, conventional message passing functions which require only a starting address of the message and its length are not longer sufficient to adequately describe an outgoing message. Each distinct data element in a message must be tagged so that it can be properly converted when necessary. With most C `send()` /`rev()` interfaces, the responsibility of correctly labeling message elements falls on the programmer. In SMP, we utilize object-based methods, inheritance, and operator overloading to make this the *compiler's*