Smart Messages: An Object-Oriented Communication Mechanism

Eshrat Arjomandi, York University, Toronto
William G. O'Farrell, Gregory V. Wilson, IBM Canada Ltd.

Smart Messages: An Object-Oriented Communication Mechanism for Parallel Systems

Eshrat Arjomandi
Dept. of Computer Science
York University
Toronto, Ontario
eshrat@cs.yorku.ca

William G. O'Farrell & Gregory V. Wilson
Center for Advanced Studies
IBM Canada Ltd.
Toronto, Ontario
{billo,gvwilson}@vnet.ibm.com

Abstract

ABC++ is a portable object-oriented type-safe class library for parallel programming in C++. It supports active objects, synchronous and asynchronous object interactions, and object-based shared regions on both shared- and distributed-memory parallel computers. ABC++ is written in, and compatible with, standard C++: no language extensions or pre-processors are used. This paper focuses on its use of an object-oriented technique called *smart messages* to support object interactions. Smart messages demonstrate the effectiveness of object-oriented programming in encapsulating low-level details of concurrency and in improving software portability.

1 Introduction

While massively-parallel computers and networks of workstations (NOWs) are now widely available, good programming systems for them are not. Rapid architectural change has meant that the levels of code re-use which are taken for granted in conventional computing environments are still only dreamt of by parallel programmers. Many groups are now trying to insulate users from such changes using the abstraction and polymorphism facilities of the object-oriented programming (OOP) paradigm.

Language designers have three options in integrating OOP and concurrency: create a new language, add new features to an existing language, or construct libraries to work with an existing language. The first two approaches give much more freedom to experiment with new ideas, but experience shows that application programmers are very reluctant to translate existing codes, or develop new ones, to make use of a non-standard environment. It is also very difficult for any but the largest of research groups to develop the "boring" (i.e. sequential) portion of a non-standard language system, and keep its capabilities and performance competitive with that of standard systems. A library-based approach, on the other hand, must find a way to accommodate language features which were designed (or which "just happened") long before parallelism was an issue.

C++ is rapidly replacing C as the language of choice for systems programming, and is starting to be adopted by scientific programmers, who have traditionally been the largest users of parallel computers. For this reason, numerous attempts have been made to add concurrency to C++ [2, 4, 5, 6, 7, 8, 9, 10, 12]. Most such systems require extensive compiler extensions and/or preprocessors. Attempts using purely a class library approach have often not fully utilized OOP in their design and implementation, or have imposed unreasonable limitations on the users. For example, some libraries [2, 7, 8] limit the height of the user's class hierarchy to one, require explicit use of *wait* and *alert* routines for synchronization, or require explicit manipulation of message queues to manage object interaction.

ABC++ is a class library to support parallel programming in C++. Its initial implementation [1] demonstrated that many of the limitations listed above can be eliminated without resorting to language extensions or pre-processors. ABC++'s concurrency model supports *active objects*—objects which have their own threads of control, and which can run simultaneously with other active objects—and remote method invocation for inter-object communication. However, the first version of ABC++ had several limitations, such as compiler and architecture dependencies and a lack of type-safety checks.

One of the difficulties in the design and implementation of a concurrent class library for C++ is constructing a flexible, architecture-independent communication mechanism for object interaction. For example, while the implementation of method invocation is straightforward in a sequential environment, it can be problematic on distributed-memory machines. For the sake of efficiency and simplicity, most message-passing systems require the data to be sent to occupy a contiguous block of memory. Since programmers often want to pass logically-unconnected values as parameters to a single method invocation, some way to *marshal* these values into a contiguous block of bytes must be found. Furthermore, the type of the class whose method is being invoked, and the name of the invoked method, must somehow be communicated along with these parameters.

The implementation of object interaction in ABC++ uses the data abstraction, genericity, and polymorphism facilities of C++ to solve these problems. Based on *active messages* [14], we introduce an object-oriented communication mechanism, *smart messages*, that captures the required data and typing information. Active messages [14] is an asynchronous communication model designed to minimize network latency by allowing communication and computation to be overlapped. In this model, a process sends a message to another remote process which contains not only data, but also the address of a user-defined function, called a *handler*. When the message arrives, this handler is invoked. Its duty is to extract the message from the network and integrate it with the ongoing computation.

A smart message is an object which carries not only data, but also the typing information and all the other functionality provided by its class. This paper shows how smart messages are used and implemented in ABC++ to provide a portable, object-oriented solution to active object creation and interaction. Section 2 provides a brief overview of ABC++. Section 3 discusses the implementation of smart messages. Active object creation and interactions are covered in Sections 4 and 5 respectively. An appendix discusses issues related to automatic type conversion during method invocation and object creation.

2 A Brief Overview of ABC++

This section provides a brief introduction to major components of ABC++. For more detail, see [13].

ABC++ is a class library written in standard C++ to support parallel programming in C++. Its run-time system provides portability across uniprocessors, shared-memory multiprocessors, homogeneous NOWs, and massively parallel machines. It is presently available for IBM RISC System/6000 workstation networks and SP supercomputers.

ABC++ supports concurrency through *active objects*. An active object possesses its own thread of control and can be created on any processor. To allow active instances of a class to be created, the class must publically inherit from class `Pabc` provided in the header file `ABC++.h`.

The thread of an active object executes method `main()`. `Pabc` provides a default `main()`, which repeatedly accepts method invocations from other active objects; classes derived from `Pabc` may redefine `main()`. An active object terminates only when the whole program terminates.

An active object can control which of its methods are invocable using the functions `Paccept` and `Paccept_any`. The arguments to `Paccept` are the names of methods that the object is willing to serve. The use of `Paccept_any` signals that the object is willing to serve any of its public methods. Each call to `Paccept` or `Paccept_any` matches exactly one method invocation. The present implementation of ABC++ only allows a call to `Paccept` or `Paccept_any` from within the `main()` routine of active objects. A call to these functions elsewhere causes an exception to be thrown.

An active object is created in two steps. First, a handle is declared using the template `Pabc_pointer`. This handle is then used to reference the active object.

```
class C_Active : public Pabc {
  body of active class
};
...
Pabc_pointer<C_Active> active_p;
```

Next, the function `Pabc_create` is called to create an active object and bind a reference to it to the handle. `Pabc_create` is in fact a family of overloaded function templates with varying numbers of arguments. The first (optional) argument to `Pabc_create` allows users to specify a particular processor on which the active object should be created. If this argument is not provided, by default, the ABC++ run-time system determines where to create the new active object. The next argument is a handle for the active object being created. The remaining zero or more arguments are arguments to the active object class constructor. For example, the following code creates an active instance of `C_Active`. The value `12345` is passed to `C_Active`'s constructor.

```
Pabc_create(active_p, 12345);
```

Active objects in ABC++ communicate through remote method invocation (RMI). Both synchronous and asynchronous RMIs are supported. ABC++'s template functions Pvalue and Pvoid perform blocking invocations of methods returning or not returning a value, respectively. For example, the following line of code invokes the method foo of the active object bound to active_p, passing the integer 456 as an argument:

```
int i = Pvalue(active_p, C_Active::foo, 456);
```

ABC++ requires the specification of fully qualified method names (as in C_Active::foo) as an argument to Pvalue and other template functions used in active object communication. This is sometimes tedious; however, since this argument is a function pointer, there is no interference with the virtual function mechanism. In particular, if active_p is declared as a base class pointer but is pointing to an object of a derived class, the foo method of the "right" class will be invoked, even if the Pvalue call is made using a base class function pointer.

Ppar_void and Ppar_value implement asynchronous RMI. Their arguments are identical to those taken by Pvoid and Pvalue. However, neither Ppar_void nor Ppar_value block the caller. Instead, the calling object proceeds with its activity as soon as the arguments to the call have been copied to a safe place. Ppar_void is used for asynchronous invocation of void methods; Ppar_value is used when a result is expected.

Futures [11, 6, 12] are used to receive results of asynchronous RMIs. A future is an instance of the Pfuture class template. When Ppar_value is called, the future is marked as pending. The future is resolved when a value becomes available for it; any attempt to read its value before it is resolved blocks its reader. The following code shows how ABC++ futures are used:

```
Pfuture<int> iF;
iF = Ppar_value(active_p, C_Active::foo, 456);
int i = iF; // block until result becomes available
```

The mechanisms presented so far allow for direct communication among active objects. ABC++ supports a second model of communication and synchronization which allows for indirect object interactions. *Parametric shared regions*, or PSRs, are used to provide the illusion of shared memory. A PSR may be any C++ object. ABC++'s run-time system provides copies of PSRs to other processors when and as they are needed. Consistency of shared regions are guaranteed by ABC++'s run-time system. Detailed discussions of how PSRs are used and implemented are outside the scope of this paper. For more detail on PSRs see [13].

3 Smart Messages

In order to support the model described in the previous section, ABC++'s run-time system must provide a flexible, architecture-independent communication mechanism. For example, a message requesting object creation must carry any constructor arguments required to create that object, the request to call new, and the type of the class whose constructor must be called. The arguments to the constructor must be automatically marshalled into a contiguous block of memory. Similarly, a remote method invocation (RMI) must contain some identification of the object whose method is being invoked, the method itself, and any arguments that method requires.

ABC++ utilizes data abstraction, genericity[1], and polymorphism to create a unified framework called *smart messages* to handle both remote object creation and remote method invocation. A smart message is an instance of a *smart class*. The instance variables of each particular smart class contain the information required to carry out the desired operation. A designated method of the smart class, called do, encapsulates the requested operation. For example, if the request is for object creation, ABC++ will automatically create a smart class by template expansion. This class's instance variables will have the same types as its constructor arguments; upon instantiation, its instance variables will be initialized by copying the user-supplied constructor arguments. This class's do method will be defined to invoke new using its instance variables. When an instance of this class is received at a remote processor, that processor will invoke its do method to create an object of the required class.

Since a request for object creation or method invocation may involve a varying number of arguments, ABC++ provides a family of smart classes with varying number of instance variables. Each of these classes has its own unique name. However, upon the arrival of a smart message at a destination processor, ABC++ must be able to invoke its do method. Polymorphism is used to allow this. All smart classes inherit from an abstract base class SmartMsg which provides a deferred method (called a *pure virtual function* in C++):

```
class SmartMsg {
 public:
  virtual void do() = 0;
};
```

Descendents of this class define do to perform either object creation or method invocation. The remainder

[1] Provided in C++ by templates.

of this section deals with smart classes implementing remote method invocation; Section 4 covers how smart messages implement remote object creation.

3.1 Marshalling Data

As mentioned earlier, ABC++ must marshal the arguments to remote operations. Smart messages allow this to be done in a type-safe way. Every smart message used for an RMI has at least two data members: a pointer to the method to be invoked, and a pointer to the object which is to execute the method. This pointer is defined in the address space of the processor on which that object executes; as will be seen in Section 4, such a pointer is embedded in an active object handle by `Pabc_create`. The remaining instance variables store the arguments to be passed to the invoked method. The `do` method of the smart message invokes the desired method using the values stored in the instance variables.

Smart messages should be able to encode the invocations of methods with varying number of parameters. This is handled by defining a family of class templates. The present implementation of ABC++ provides class templates with zero to 16 parameters. The following segment of code demonstrates the 1-parameter version for a method returning a value.

```
class C_Obj
{
  ···user-defined object class···
};

template<class C_Obj, class C_Ret, class C_Arg1>
class SmartMsg1: public SmartMsg
{
 public :
  C_Obj * obj_p;
  C_Ret (C_Obj::*meth_p)(C_Arg1);
  C_Arg1 arg1;

  SmartMsg1(Pabc_pointer<C_Obj> objHndl,
            C_Ret (C_Obj::*method)(C_Arg1),
            const C_Arg1 & a1)
  : obj_p(objHndl.object),
    meth_p(method),
    arg1(a1)
  {}

  void do()
  {
    C_Ret r = (obj_p->*meth_p)(arg1);
  }
};
```

The data member `obj_p` caches the object pointer from the active object handle given as a constructor argument. The data member `meth_p` caches a pointer to the method being invoked, while `arg1` stores a copy of the argument to be passed to that invocation. `SmartMsg1`'s `do` method takes a particular object as an argument, and calls the specified method on that object with the actual argument. This method can only be called safely in the address space of the processor on which the specified active object is running.

The most important aspect of `SmartMsg1` is its role in encapsulating the necessary information for invoking a method into a single object, so that the data is guaranteed to be contiguous in memory. Copying `sizeof(SmartMsg1)` bytes from `&sm1` (where `sm1` is a particular instance of the smart message class) is therefore guaranteed to copy the whole of the smart message.

3.2 Creating Smart Messages

Smart message classes are used by a series of overloaded function templates to create smart messages. The following function demonstrates how smart messages are created from the `SmartMsg1` class:

```
template<class C_Obj, class C_Ret, class C_Arg1>
C_Ret Pvalue(Pabc_pointer<C_Obj> & objHndl,
             C_Ret (C_Obj::*method)(C_Arg1),
             const C_Arg1 & a1)
{
  // create smart message
  SmartMsg1<C_Obj, C_Ret, C_Arg1>
    smartMsg(objHndl, method, a1);

  ···send the smart message to the desired destination···

  C_Ret r; // temporary storage for result
  ···assign value returned to r···
  return r;
}
```

The first statement in this function creates a smart message. This message is then sent to another processor and the calling object blocks until the result is returned. The mechanism used for sending the message is discussed in Section 4.

3.3 Remote Invocation

ABC++ provides smart classes in order to accommodate the invocation of methods taking up to 16 parameters. Therefore on the receiving side, a correct handle must be used to invoke method `do`. This is achieved with the help of polymorphism. As mentioned earlier, ABC++ provides an abstract base class, `SmartMsg`, with a single pure virtual function `do`. All smart classes publically inherit from this class. By obtaining a handle to SmartMsg on the receiving side, the virtual function mechanism of C++ will guarantee the invocation of the "right" do method.

4 Active Object Creation

Section 2 presented a brief introduction to how users can create active objects. In this section we show how smart messages are used in the implementation of remote active object creation.

As mentioned earlier, the function Pabc_create is called to create active objects. Pabc_create is a family of overloaded function templates with varying number of arguments. The first argument is a handle to the active object being created. The remaining zero or more arguments are arguments to the active object class constructor. A skeletal implementation of Pabc_create for active object constructors taking one argument is shown below:

```
template<class C_Obj, class C_Arg1>
void Pabc_create(Pabc_pointer<C_Obj> & objHndl,
                 const C_Arg1 & a1)
{
  ···check for pointer arguments···
  Proc p = procSet.select();
  SmartMsg_Create1<C_Obj, C_Arg1> smartMsg(a1);
  P__send(p, &smartMsg, sizeof(smartMsg));
  P__createReply<C_Obj> reply;
  P__recv(&reply, sizeof(reply));
  P__abcSetPtr(objHndl, reply.data, p);
}
```

The two template arguments to this function are the class of the active object being created, and the class of the object's constructor's argument. The function's formal parameter is the argument to use when constructing the active object. SmartMsg_Create1 is the smart class version for constructors requiring one argument. Versions of this function for constructors taking up to 16 arguments are provided, as are versions which allow a processor to be selected by the user, rather than by using the automatic load-balancing method of the procSet class.

After some (sequential) code to force the compiler to check that the constructor argument is not a pointer, Pabc_create creates a smart message containing the constructor argument. The data transfer function P__send is then called to ship the smart message to a daemon on the designated processor. The actual transport mechanism may be TCP/IP, MPI, or whatever else is convenient.

Next, Pabc_create creates a reply buffer, which will be used to store a pointer to the object generated on the remote processor, and then blocks its caller until a reply is received. Since inter-object communication can only be done through handles, this ensures that an active object never tries to send a request to another object which has not yet completed its own initialization. The value in this reply is a pointer to the active object which has just been created. The final line of Pabc_create extracts this pointer, and stores it in its handle argument. This pointer can then be extracted in subsequent RMIs.

Section 3.1 showed how smart classes marshal the arguments to a RMI in a type-safe manner. Smart classes for remote object creation are very similar to smart classes presented in Section 3.1 for RMI. These classes also inherit from the abstract base class SmartMsg. The instance variables (if any) of these smart classes store the constructor arguments needed to create the active object. The definition of the do() method simply calls new using its instance variables as the constructor arguments. The following code demonstrates the 1-argument version:

```
template<class C_Obj, class C_Arg1>
class SmartMsg_Create1: public SmartMsg
{
 public:
  Proc srcProc;
  C_Arg1 arg1;
  SmartMsg_Create1(const C_Arg1 & a1) :
    arg1(a1)
  {
    srcProc = procSet.this();
  }
  void do()
  {
    P__reply<C_Obj> reply;
    // create active object
    reply.data = new C_Obj(a1);
    P__send(srcProc, &reply, sizeof(reply));
  }
};
```

5 Data-Based Synchronization Using Futures

As stated in Section 2, ABC++ uses *futures* to implement data-based synchronization of remote method invocations. Futures are implemented by defining a template class to hold the results of asynchronous remote method invocations which return values. Each instance of the class Pfuture holds a reference to the future return value of an asynchronous invocation. Type conversion from the future type to the base type blocks until the return value actually arrives. A future can also be initialized with an actual object, in which case it acts just like a holder for that value without blocking. Assignment of futures to each other is well-defined. A partial signature of `Pfuture` is:

```
template<class T>
class Pfuture
{
 private:
  P__future_result<T> * future_result;

 public:
  // create futures
  Pfuture();
  Pfuture(const Pfuture<T>& fut);
  Pfuture(const T& value);
  // alias and overwrite futures
  Pfuture<T>& operator=(const Pfuture<T>& fut);
  Pfuture<T>& operator=(const T& value);
  // delete futures
  ~Pfuture();
  // convert to base type
  operator T() const;
  // test for completion
  int resolved() const;
};
```

The class `P__future_result` contains data and member functions to handle the future-resolution protocol. The behavior of instances of this class is independent of the data type encapsulated in any particular future. The methods of `Pfuture` itself allow futures to be created, assigned values, converted to their base type, and tested for completion.

Given this structure, the template function `Ppar_value` performs an asynchronous remote method invocation. When invoked, it marks its future argument as unresolved. Subsequent attempts to access the future's data will block until the future becomes resolved. As with `Pvalue`, a smart message is created to carry argument values and a method function pointer to the remote method whose operation is being invoked. The following segment of code illustrates the major components of `Ppar_value`. `SmartMsgFuture1` is the 1-argument version of smart classes defined for future interactions.

```
template<class C_Obj, class C_Ret, class C_Arg1>
Pfuture<C_Ret>
Ppar_value(Pabc_pointer<C_Obj> objHndl,
           C_Ret (C_Obj::*method)(C_Arg1),
           const C_Arg1 & a1)
{
  ···check validity of arguments···

  // create future token for later reference
  Pfuture<C_Ret> token;

  // create smart message
  SmartMsgFuture1<C_Obj, C_Ret, C_Arg1>
    smartMsg(token, objHndl, method, a1);

  ···send message···

  return token;
}
```

The future object, `token`, created inside this function is a placeholder containing synchronization control information. It is returned to the caller so that references to it may block or complete, and a pointer to it embedded in the smart message so that the reply from the remote method invocation will have a way to identify its future.

6 Conclusion

This paper focussed on *smart messages*, an object-oriented technique in support of active object interactions in ABC++. We showed how to utilize the data abstraction, genericity, and polymorphism facilities of the object-oriented paradigm to create a unified framework to handle both remote object creation and remote method invocation.

ABC++ provides several commonly-used abstractions of parallelism within standard C++. It supports type-safe parameter marshalling, remote method invocation, and object-sized distributed shared memory without any pre-processors, post-processors, or language extensions. As a result, ABC++ is very portable, and does not require users to commit themselves to non-standard or poorly-supported tools. Future directions for ABC++ may include the incorporation of multiple consistency protocols as exemplified in Munin [3], and support for group operations in the form of data parallelism, or its higher-level counterpart, method parallelism.

Acknowledgments

ABC++ was developed at the Center for Advanced Studies at IBM Canada's Toronto Laboratory, in collaboration with researchers from several universities, including York, Syracuse, Toronto, and McGill. We wish to thank Frank Eigler for his work on implementing the current version of ABC++, and Howard Operowsky for his support and suggestions. We also wish to thank Tim Brecht of York University for his suggestions toward making the implementation of ABC++ more portable. Finally we thank the many people who contributed to the prototyping and testing of ABC++, including Young-il Choo, Jagdeep Dhillon, Ali Ghobadpour, Stephen Howard, Ivan Kalas, Gita Koblents, Henry Lee, Peter Milley, Fernando Nasser, S. David Pullara, Ilene Seelemann, and Susan Sim.

Appendix: Checking and Converting Argument Types

The templates shown in this paper are unnecessarily restrictive, as they do not allow type conversion during invocation. For example, because the types of both the formal argument to the method in `Pvalue`, and the actual argument given as `Pvalue`'s third parameter, are specified as `C_Arg1`, an invocation such as:

```
Pvalue(tHndl, TestClass::methodTakingInt, 'a');
```

would fail with a type-matching error.

ABC++ circumvents this with a slightly more convoluted definition of `Pvalue`:

```
template<class C_Obj, class C_Ret,
         class C_Formal1, class C_Actual1>
C_Ret Pvalue(Pabc_pointer<C_Obj> & objHndl,
             C_Ret (C_Obj::*meth)(C_Formal1),
             const C_Actual1 & a1)
{
  // check legality of type conversion
  C_Actual1 * actual_p = NULL;
  C_Formal1 * formal_p = actual_p;

  ···check for pointer arguments···

  ···rest of body as before···
}
```

This version of `Pvalue` takes three class parameters: the object's class, the type of the formal argument to the method being invoked, and the type of the actual argument being passed to the invocation. The first line of `Pvalue` creates a `NULL` pointer of the actual argument type; the second line then tries to assign from this pointer to a pointer of the formal type. If the actual type cannot be converted to the formal type, this conversion will fail. A modern optimizing C++ compiler, such as IBM's CSET++, can determine that these values are not subsequently used, and delete them during optimization. This technique therefore has no run-time cost.

The next statements in this modified `Pvalue` checks to ensure that the actual argument being passed is not a pointer. ABC++ does not (presently) allow pointer arguments to be passed during remote method invocation because there is no guarantee that the thing pointed to at the receiving end will bear any resemblance to the thing pointed to at the sending end. ABC++ forces the compiler to check for pointer parameters by providing templates that will match all pointer and non-constant reference arguments, plus one that will match immediate and constant reference arguments. We begin by noting that if the declared type of a formal argument is `const X&` (for some type `X`), then `C_Actual1` will be bound to the whole of `const X&` and not just to `X`. If the user attempts to pass a pointer or reference argument to a remote method invocation, the compiler will find two ways to unify this attempt with these templates. Since this ambiguity is illegal, the compiler will generate an error message using the mock argument name `no_pointer_argument_allowed`, and fail. These templates are:

```
template<class T>
void _P_ptr_invalid(
  T* no_ptr_arg_allowed
) {}

template<class T>
void _P_ptr_invalid(
  T* const no_ptr_arg_allowed
) {}

template<class T>
void _P_ptr_invalid(
  const T* no_ptr_arg_allowed
) {}

template<class T>
void _P_ptr_invalid(
  const T* const no_ptr_arg_allowed
) {}

template<class T>
void _P_ptr_invalid(
  const T& no_ptr_arg_allowed
) {}
```

For example, if a program attempts to pass an `int*` as a parameter to a remote method invocation, then during compilation, the compiler will unify `int` with `T` in the first template, but also unify `int*` with `T` in the last template (trying to create a function with a `const int * &` argument).

References

[1] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F.C. Eigler, and G. Gao, "ABC++:

Concurrency by Inheritance in C++", *IBM Systems Journal*, Vol. 34, No. 1, (1995), pp. 120-136.

[2] AT&T C++ Language System Release 2.0: Product Reference Manual, Select Code 307-146, AT&T Bell Laboratories, Murry Hill, NJ 07974 (1989).

[3] John K. Bennett, John B. Carter, and Willy Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", in *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming*, ACM Press, 1990.

[4] B. Bershad, E. D. Lazowska and H. M. Levy, "PRESTO: A System for Object-Oriented Parallel Programming", *Software–Practice and Experience*, Vol. 18(8), (August 1988) pp. 713-732.

[5] P.A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke, "μC^{++}: Concurrency in the Object-Oriented Language C++", *Software–Practice and Experience*, 22(2), (1992), pp. 137-172.

[6] R. Chandra, A. Gupta and J. Hennessy, "COOL: a Language for Parallel Programming", in *Languages and Compilers for Parallel Computing*, edited by D. Gelernter, A. Nicolau, D. Padua, MIT Press, (1990).

[7] T.W. Doeppner Jr., and Alan J. Gebele, "C++ on a parallel machine", Report CS-87-26, Department of Computer Science, Brown University, (November 1987).

[8] P. Gautron, "Porting and Extending the C++ Task System with the Support of Lightweight Processes", *USENIX* C++ *Conference Proceedings*, (1991), pp. 135-146.

[9] N.H. Gehani and W.D. Roome, "Concurrent C++: Concurrent Programming with Class(es)", *Software–Practice and Experience*, Vol. 18(12), (1988), 1157-1177.

[10] D. Grunwald, A User's Guide to AWESIME: An Object-Oriented Parallel Programming and Simulation System, Technical Report CU-CS-552-91, Department of Computer Science, University of Colorado at Boulder (1991).

[11] R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation", *ACM Transactions on Programming Languages and Systems*, October 1985.

[12] D. Kafura and K.H. Lee, "ACT++: Building a Concurrent C++ with Actors", *Journal of Object-Oriented Programming*, Vol. 3, No. 1, (1990), pp. 25-37.

[13] W. G. O'Farrell, F. Ch. Eigler, I. Kalas, and G.V. Wilson, ABC++ User Guide, "An Introduction to the IBM Parallel Class Library for C++", ABC++ Version 1, Release 1, IBM Canada, abc++@vnet.ibm.com, (1995).

[14] T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation", *Proceedings of the 19th International Symposium on Computer Architecture*, ACM Press, Gold Coast Australia, (May 1992).