Structuring Concurrent Applications Using Active Expressions

Mauricio De Simone and Ajit Singh
{mdesimon, asingh}@etude.uwaterloo.ca
Dept. of Electrical & Computer Engineering
University of Waterloo,
Waterloo, Ontario, Canada N2L 3G1

Abstract

The paper presents the concept of *Active Expressions* and describes a model, called A_e, for structuring concurrent applications. Relying solely on the features of an object oriented language, A_e allows definition of common communication and synchronization patterns. These patterns are then combined with the user provided application code to instantiate complete concurrent applications. The approach has two novel aspects: First, it demonstrates that common patterns of concurrency can be expressed using language provided facilities. Second, the model can be implemented without requiring any complex user-interfaces, preprocessing stages or language extensions. Finally, the use of the pattern based approach for developing concurrent applications is illustrated via an example. The model has been implemented using C++/Unix on shared-memory as well as distributed memory based platforms.

Subject Areas: Parallel Programming Environments, High Performance Computing Models, Software Tools for Concurrent Processing.

1 Introduction

Given the rapid growth of the workstation environment, programmers and applications that deal with concurrency are likely to continue to grow. Therefore, concurrency is no longer the exclusive domain of operating systems developers or expert programmers of expensive parallel machines. With an ever expanding community of users who develop concurrent applications, performance and access to low level features of the underlying architecture are no longer the overriding criteria in judging the suitability of a concurrent programming tool. Other software engineering issues such as ease of use, portability, and maintainability are becoming increasingly important. Also, in a workstation environment, given its distributed nature and the associated high communication cost, programmers often focus on exploiting the medium and coarse grain concurrencies in their applications. At the same time, growing presence of shared memory multiprocessor workstations in such environments offers one the opportunity to take advantage of fine grain parallelism as well.

This paper presents a programming system, called A_e, that provides a new approach to developing concurrent application in a workstation environment. The system is based on a model that provides commonly used communication and synchronization structures for concurrent programming as building blocks. These building blocks can be combined with the user provided application code to create complete concurrent applications. The most important characteristic of the A_e model is that it allows specification of concurrent structures in terms of simple constructs called *Active Expressions*. These expressions can be composed using the existing features of an object oriented language. A system based on this model has been implemented without requiring any language extensions or preprocessors. The result is an efficient and flexible concurrent programming system that is as portable as the underlying sequential language.

The paper is organized as follows: Section 2 reviews the various approaches to concurrent programming with an emphasis on research works related to supporting commonly used communication and synchronization structures. Section 3 outlines the objectives of the A_e model. Section 4 presents the concepts of active expressions and active components used in the A_e model. Section 5 provides an example application based on the model. Section 6 briefly outlines the performance of the model in the shared memory as well as message passing environments. Finally, section 7 presents some future directions for this ongoing project and provides concluding remarks.

2 Related Research Works

Due to the complexity associated with concurrent programming using low level communication libraries [5, 13, 16, 27], several high level approaches have been proposed to simplify the development of concurrent applications.

Some of these proposed solutions have taken the approach of developing new programming languages that directly support concurrent paradigms [1, 2]. Such new languages have some clear advantages. First, the approach allows direct mapping of abstractions for concurrency into the language. Second, a compiler for such a language can provide features such as type-checking of parameters between interacting concurrent units. Third, it is also comparatively easier to achieve good performance because the solution is tailored to particular hardware platforms. The approach has several drawbacks as well: Developers have to tie themselves to a particular (not necessarily a well known) language. Auxiliary tools, such as debuggers, performance monitors, etc. to support such abstractions are often unavailable or not up to date. Portability is also hindered, given the amount of programming effort required to produce different versions of the tools for myriad of common hardware architectures.

Others have chosen to extend well-known languages [18], such as Fortran [17], C [12], or C++ [7, 9, 14, 21], to provide support for parallelism. In this approach, normally a sequential language is annotated with new keywords or compiler directives. This approach has provided interesting solutions, HPF [17] being a good example. A major advantage of this approach is that it allows reuse of a programmer's expertise in the host sequential language. The approach shares some of the drawbacks (related to auxiliary tools and portability) discussed above. In addition, a large effort may be required to keep the extended language consistent with the base language. The effect would be particularly significant in case the base language itself is evolving (as is currently the case with C++).

A more recent approach is based on integrated development environments. In this category, one can find several visual systems that support many common models for concurrency ranging from supporting components and connectors [3] to directly supporting high level patterns for synchronization and communications [4, 6, 22, 23, 25]. These systems have the distinct appeal that one can program visually and compose applications that map directly to conceptual models. The system takes the programmer's descriptions and application code and produces intermediate level code (such as C code) that is then compiled and run. The intermediate representation produced by such systems is often complex, forcing the developer to use the high level model and its tools. The concept of patterns used in such systems allows the reuse of design knowledge for the development of concurrent applications. However, these models are often inflexible in the sense that if a user's application requires certain structures that are not directly supported by the system, it may be very difficult or even impossible to develop the application using the system [24].

3 Objectives of the A_e Model

All the above-mentioned approaches have their advantages and disadvantages. The A_e model has in part been inspired by the large body of previous work on supporting common parallel structures in high level parallel programming tools. However, A_e is based on using language-based facilities to put as much functionality as possible directly in the host language, bypassing external tools. The approach emphasizes the importance of following characteristics:

- **Performance:** Often, concurrent applications are developed to achieve highly efficient parallel solutions. In such situations, performance is perhaps the most important concern of the application developers. Therefore, the extra complexity induced by concurrency must be more than balanced by the speedup achieved.
- **Higher levels of abstraction:** Common concurrent programming paradigms should be directly supported in the model to simplify and accelerate the development of concurrent applications.
- **Portability:** Modern applications are *required* to run on different platforms. Therefore, generating portable solutions is not an option but almost a requirement. Also, the programming tool itself should be available on a wide variety of platforms. Therefore, a concurrent programming tool should only rely on commonly available hardware and software to achieve its objectives.
- **Flexibility:** Although the use of higher levels of abstraction has the potential to reduce the application development time, programmers often feel caged inside these abstractions because of the inflexible nature of the underlying models or environments. Therefore, a concurrent programming approach should also allow the flexibility of "stepping out of the model" if required.
- **Familiarity:** A new approach has a higher probability of gaining acceptance if it allows its users to work with familiar methodologies and tools.

Achieving all the abovementioned objectives is challenging for any concurrent programming system. Often, there is some tension among the various objectives, e.g., a high level model may not produce very efficient solutions. The A_e model does not claim to fully meet the above-mentioned criteria, however, it has been designed with these issues in mind.

The distinctive features of A_e are: use of a common sequential programming language without any extensions, a simple model that does not depend on having graphical user interfaces or

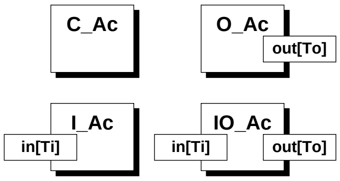


Figure 1: Active Components Category

language preprocessors, and a flexible model in which a user still programs in the familiar host language. The model does not restrict the user from mixing features of A_e with other primitives such as sockets or threads.

4 The A_e Model

Active Expressions (A_e) is a model for concurrent programming that can be implemented directly in a modern object oriented language as a library. It leverages on features of an object-oriented language and associated compiler technology to express and instantiate type-safe concurrent patterns without using external tools or graphical user interfaces.

Throughout this discussion, interfaces and examples reflect the current implementation of active expressions in C++, but other languages that provide traditional object oriented features of encapsulation, inheritance and polymorphism including operator overloading and parameterization can support the model if required.

The model consists of three abstractions: *Active Components* (A_c), *Active Expressions* (A_e), and *Shared Regions* (S_r). Active components represent the concurrent units of execution in a concurrent program, active expressions state the communication pattern among active components and shared regions permit the safe sharing of data among concurrent units of execution. The following subsections describe the A_e model with an emphasis on active components and active expressions related aspects of it.

4.1 Active Components

An active component (A_c) encapsulates a user's code that runs sequentially in its own thread of execution. A_cs run concurrently with each other. An A_c has exactly one entry point for execution, at most one input port for receiving input, and at most one output port for sending output. The entry point is the place where the user provides the component functionality and the ports permit the communication with other A_cs in active expressions (A_es). As figure 1 shows, depending on the data flow behavior of components, four distinct categories of active components (A_cs) are possible: Closed (A_{cc}), Input (A_{ci}), Output (A_{co}) and Input/Output (A_{cio}). The types of the ports associated with an A_c are specified at compile time. This provides the C++ compiler a way to validate connections between A_cs for type-safety and proper data flow when used in active expressions (A_es). The user specifies the availability of input/output ports and their types by instantiating a templated inheritance tree.

As an example, consider the following definition of an active component of Input/Output category (A_{cio}) called `Filter`. It has an input port that expects `int` and an output port that produces Color:

```
class Filter : public IO_Ac<int, Color>
{
 virtual void run() {
    int i;
     // Get int from input port
    while (pin >> i) {
         // Do something with i
        Color c(i);
         // Send it to output port
        pout << c;
    }
   ...
};
```

By publicly inheriting from `IO_Ac<Ti,To>` and providing types for the template parameters corresponding to the input port and output port respectively, the above example specifies the category of the active component and the types of its input and output ports. Similarly classes for the other possible categories (`C_Ac`, `I_Ac<Ti>` and `O_Ac<To>`) are also provided. The user expects the function `run()` associated with an active component to be called in its own concurrent thread of execution when an A_c is executed. The functionality included at this entry point is entirely up to the user, thus providing maximum flexibility.

As shown in the above example, an input port is accessed through a streaming object `pin` which is accessible only if the active component has an input port. Similarly an output port is accessed through a streaming object `pout`. These streaming objects behave similar to `cin` and `cout` objects in the `iostream` library [26]. Even though `pin` and `pout` allow the communication with other A_cs, there is no restriction on the user program with respect to using any other communication mechanisms, such as files, IPC, or sockets to communicate with the outside world. The ports are only used for communication among active components in active expressions.

4.2 Active Expressions

A mechanism based on built-in primitives of an object oriented language is required to express the communication patterns among active components (A_cs). An *Active Expression* (A_e) serves this purpose. In an A_e, operands describe A_cs and thus represent the sequential units of execution that need to operate concurrently; and the operators describe the communication primitives among its *lhs* and *rhs* A_c operands.

Consider the following example, where `p`, `w` and `c` are active components of categories Output (A_{co}), Input/Output (A_{cio}) and Input (A_{ci}) respectively. Now, let us consider the expression:

```
C_Ae exp = p | (n * w) | c;
```

where `n` is a positive integer. As discussed below, by providing appropriate semantics to the *lhs* and the operators of the above expression this construct can specify a concurrent pattern which in this case happens to be a task farm. Here `p`, `w`, and `c` represent producer, worker, and consumer respectively of a task farm that employs `n` workers. To fully understand the semantics of the above expression, let us first discuss possible operators in an A_e:

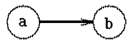
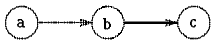
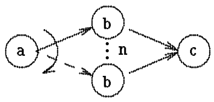
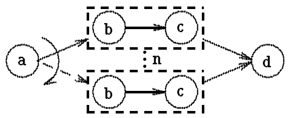
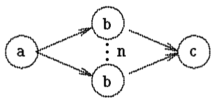
	Active Expression	Concurrency Graph
(a)	`a	b`	
(b)	`a	b	c`	
(c)	`a	(n * b)	c`	
(d)	`a	(n * (b | c))	d`	
(e)	`(a || (n * b))	c`	

Figure 2: Operators and Graphs

- **Pipe (operator |):** Pipes allow the connection of two active components in a pipeline. The output port of the *lhs* operand of the pipe (|) is connected to the input port of the *rhs* operand. Figure 2(a) and (b) show examples of its usage.
- **Replication (operator $*$):** Replication, specified as (n * A_c), converts its active component argument (A_c) into an aggregate consisting of as many elements as its integer argument (n) specifies. Such an aggregate can be used as a normal A_c in other operations. Consider figure 2(c) and (d) as examples of the replication operation and its interaction with the | operator. In case the input port is defined, the A_cs that compose the aggregate compete for the incoming data. Similarly, in case the output port is defined, the A_cs composing the aggregate active component produce outgoing data in no particular order.
- **Broadcast Pipe (operator ||):** A broadcast pipe is another communication structure which can be used in conjunction with replication of active components. In this case, the broadcast pipe provides a copy of each element produced by its *lhs* argument to every member of the replication on its *rhs*. A new element will not be received by any A_c in the replication until all have received the previous one. This effectively time steps the computation. Figure 2(e) shows an example application graph.

All these operators are defined in such a way that a compilation will only succeed if valid connections are provided. This means connections respect data flow (output ports connect only to to input ports) and are type safe (the object type produced by the output port is the same as the object type consumed by the input port). This way an A_e guarantees that a compilable program is a structurally correct one in these respects. Using these operators, a user can compose arbitrarily complex expressions.

Some additional constructs are also available that can be used in active expressions. As an example, we present the cycle construct. The cycle allows data to be recycled through the pipes, that is, the end of a piped expression can be connected to its beginning component. As an example, let us consider the expression:

```
C_Ae exp = p | cycle(a | b, cond) | c;
```

The expression enclosed in the cycle must be able to receive input at one end and produce output at the other (to allow the cycle to be meaningful!). When data reaches the end of the enclosed expression (active component `b` in the above example), `cond` is evaluated to decide whether the produced data item should be recycled (to active component `a` in this example) or it should be sent to the output port (to active component `c` in the example).

4.3 Naming and Instantiation

The A_e model uses a mechanism for defining and instantiating active expressions that allows incremental composition of arbitrarily complex expressions. The naming of active expressions and the actual construction of the concurrent structures expressed by them is controlled by an assignment statement. At the top level, A_es can also be categorized as Closed (`C_Ae`), Input (`I_Ae<Ti>`), Output (`O_Ae<To>`) or Input/Output (`IO_Ae<Ti,To>`). By assigning an expression to an appropriate type, a programmer can control naming and instantiation. If an expression is assigned to a handle `C_Ae_h` (or, depending on the category: `I_Ae_h<Ti>`, `O_Ae_h<To>`, `IO_Ae_h<Ti,To>`), no instantiation occurs and the handle points to the expression for future use. For example:

```
{
   // Name the expressions
  IO_Ae_h<int, int> handle = b | c;

   // Compose it with another
  C_Ae_h ext = a | handle | d;
}
```

If an expression is assigned to an object of type `C_Ae` (or `I_Ae<Ti>`, `O_Ae<To>`, `IO_Ae<Ti,To>`) then the structure is constructed and allowed to run. Therefore, in the following example, `handle` only names the active expression, while `structure` points to the actual running concurrent structure:

```
{
  Producer p;  // O_Ac<SomeType>
  Consumer c;  // I_Ac<SomeType>
   // Name the expression
  C_Ae_h handle = p | c;
    {
       // Instantiate it!
      C_Ae structure = handle;

       // Synchronize when structure
       // goes out of scope
    }
}
```

A synchronization barrier is present when `handle` goes out of scope. The creator thread will block until the A_e has finished. If an application requires the concurrent structure to be detached from the creator, the *bang* (!) operator is provided. A *Closed Active Expression* (`C_Ae`) can be detached from the currently running thread by applying the bang operator to it. For example:

```
{
   // Name the expression
  C_Ae_h handle = p | c;

   // Instantiate it and let it go!
  !(handle);
}
```

The application will synchronize with these detached threads when exiting `main()`.

4.4 Shared Regions

A shared region is a block of memory that can be shared safely among concurrent threads and that is keep consistent even if the threads that access it reside in disjoint address spaces.

Shared Regions use the Readers/Writer protocol for consistency. The Readers/Writer protocol states that when a thread attempts to acquire a read lock on a resource, it will succeed only if no other thread holds the lock on the resource or if all the threads that hold a lock are readers. If another thread holds a write lock, a would be reader must wait. If a thread requires a write lock, it must wait until no other thread holds a lock (read or write) on the resource.

The A_e system provides mechanisms for declaring a shared region and locking a declared shared region for read/write access. Due to the lack of space, the detailed description of these mechanisms is not provided here. An interested reader should refer to [11] for details. A similar approach can also be found in [20].

5 An Example

As an example of the use of active expressions, consider the problem of discrete image convolution. A discrete image convolution is the process of converting a source image into another through filtering [19]. Many filters can be applied to the image by replacing the *mask* that performs the filtering. Each pixel of the resulting image is weighted average of sum of all the multiplications of corresponding pixels in the source image and the mask. The mask is scanned all though the dimensions of the source image, thus completely filling the destination one.

For this example, a Gaussian filter is implemented, which is nothing more that a discrete image convolution using a Gaussian mask. This example has been implemented as a task farm of workers performing the convolution. A producer reads the image from a file and distributes it to identical workers that carry out the convolution on different regions of the image. The workers then inform a consumer of their completion and the consumer saves the resulting file.

The user's main code is:

```
int pmain(int argc, char* argv[])
{
  int n;        // Get n from argv[1]
  Mask* mask;  // Load mask from disk
   // Set up Acs
  ImageProducer p;
  ImageConsumer c;
  Convolver      conv(mask);
  Reducer       join();

   // Set up the structure and
   // Let it go
  !(p || (n * conv)) | join | c);
    ⋮
}
```

The broadcast operator is used to distribute the same image to n workers. The join stage is responsible for providing a barrier point thus forbidding the ImageConsumer from saving the file until all the workers have finished. The mask required for the convolution is the same for all workers and because it is of a small size, a copy of it is given to each of the workers in their constructor.

As shown, the entry point for the above applications is pmain. This indirection is required in order to initialize the run-time system before the application starts.

The convolution stage is:

```
class Convolver : public IO_Ac<Img, Img>
{
public:
  virtual void run() {
    Img p;
    while (pin >> p) {
      int myNumber = idInGroup();
      int group = numInGroup();
       // Do the convolution here
      pout << p;
    }
  }
⋮
};
```

In the above example, the implementation of run() for the active component Convolver is straightforward. It receives the image from the input port, processes it and sends it to the output port. The code presented above also shows the use of the methods idInGroup() and numInGroup() to obtain the worker-number and the number of workers in a replication stage. This information is used inside the loop to select the region of the original image to operate on.

The active expression used in this example application represents a reasonably complex concurrent structure. A significant amount of code and effort would be required if it was done using a low level communication mechanism such as sockets or IPC. An additional advantage of

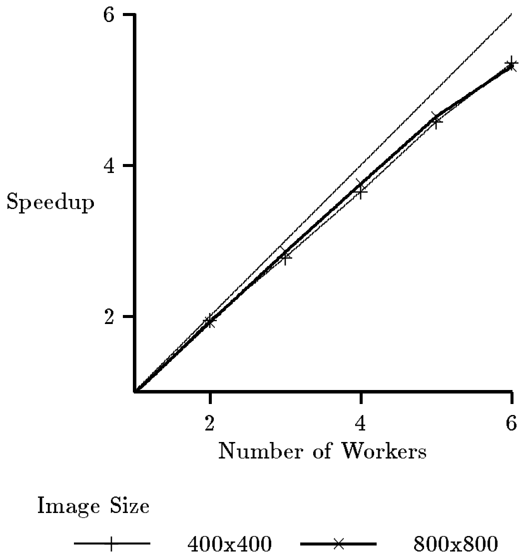


Figure 3: Discrete Convolution.

using A_e for this problem is that all the communication structure of the problem is stated in one line which aids in program understanding and maintainability . Also given that the number of workers in the task farm can be modified with a run time variable, the cardinality of the farm can be modified without recompilations or changes to the application code.

This example has been implemented on an 8 processor RS6000 J30 multiprocessor running AIX 4.0. Figure 3 presents the result of running this program on two gray-scale images (400x400 and 800x800 pixels each). The results show almost linear speed-up. This was to be expected given that there is no synchronization on the source or destination images. Each worker works on a different section of the image with no communication among them. This example shows the simplicity of implementing a complex concurrent program with A_e and that this simplicity does not imply any significant performance penalty.

6 The A_e System and its Performance

The A_e model has so far been implemented on two distinct platforms: shared memory machines and homogeneous network of workstations. The API is exactly the same for both versions and in most cases the application programmer would not feel any difference in terms of application development. The implementation has been done using C++ and Unix.

This "two-implementation" approach was conceived in order to exploit performance advantages that shared memory allows (no need to stream objects to a remote active components and the possibility of shallow copying). Of course, an application compiled for a distributed memory architecture will still run on a shared memory machine, but some extra performance overhead is to be expected. The two implementations are included in the same source code and are selected through a compilation flag. This way the user can, at compile time, use the model that best suits his/her needs, relieving the system of expensive run-time checks and decisions.

For the shared memory environment, the architecture of the system is straightforward. Active components are bound to kernel threads and input and output ports are associated with in-memory semaphore-protected queues. Thread scheduling and binding is left for the operating system and thread managements is handled though the Pthreads interface [15] thus ensuring portability. In the networked environment, active components may execute on different processors. Communication among remote active components has been implemented using the socket interface [16]. The A_e system maps the active components of an applications on machines available on the network in a round-robin manner. For both environments, the required auxiliary software functionality is provided in the form of a run-time library.

Most of the code required for realizing active components and various operators used in active expressions is implemented via a template library. The implementation makes extensive use of the object oriented features of C++ such as its support for encapsulation, inheritance and polymorphism (including operator overloading and templates).

As a result, to use the A_e system a programmer never makes any explicit library call to any function of the A_e system. Instead, based on the active expressions composed by the user, the required code is implicitly included in the application from the template library at compile time. The functions in the template library in turn make use of the run-time library.

6.1 Performance

Application independent experiments have been done to assess the relative performance of A_es. Figure 4 (a) shows the cost of an expression such as:

```
{
  Producer p;
  Consumer c;
  !(p | c);
}
```

where 10000 elements are produced by `p` and then consumed by `c`. The same functionality has also been implemented using Pthreads and a semaphore protected queue. The figure 4 (a) compares the two costs. These experiments were performed on an 8 processor RS6000 J30 workstation.

Figure 4 (b) shows the results of the experiments conducted for assessing the message passing efficiency of A_e in the distributed environment. A_e's message passing efficiency is compared with those of PVM and Sockets. In these experiments, the round-time for a message of a given size is measured. This is accomplished by using an expression such as:

```
{
  Producer p;
  Filter f;
  Consumer c;
  !(p | f | c);
}
```

where `p` and `c` are mapped on the same machine and `f` is mapped on a different machine on the network. The results show that the extra cost of executing this expression is negligible when compared with the implementation of the same functionality using sockets. It also shows that in this respect Ae performs significantly better than PVM. This is partially due to an efficient design and implementation and partially due to the low overhead requirements of the A_e model. The experiments were done using a pair of RS6000 J30 workstations connected over a 10 Mbit/s Ethernet.

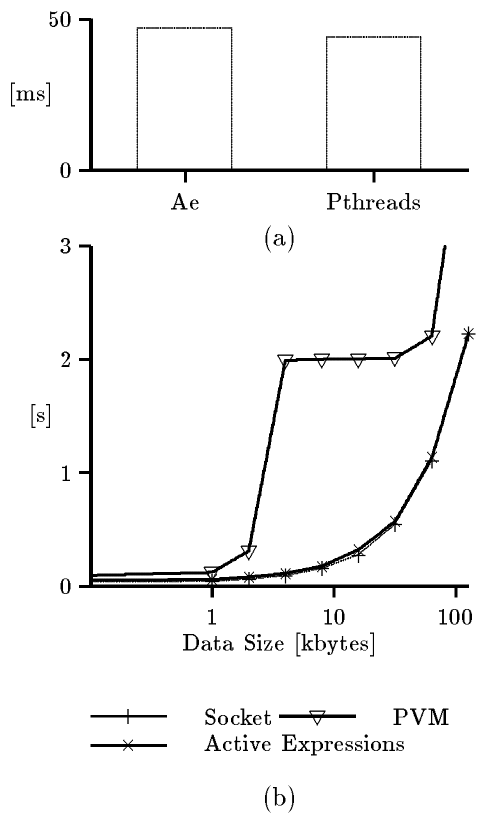


Figure 4: Application Independent Performance.

7 Summary and Future Directions

The concept of active expressions was presented as a model and a system that aims to provide an efficient approach for building concurrent applications by providing common synchronization and communication structures as building blocks. Parallel programming using commonly occurring building blocks has been advocated by several researchers in the past [8, 10]. However, systems that employ similar building blocks have previously been implemented (for example, [4, 23]) via language extensions, graphical interfaces, or language preprocessors. By discovering that new language based techniques permit the representation of higher-level concurrent constructs directly in a modern object-oriented language, the A_e model takes advantage of the inherent portability of library-based solutions. The library-based approach also makes the system inherently flexible, as the user simply views the library as a tool that can be used in conjunction with others, using the host language as a composition tool. However, unlike previous library based systems, the user does not have to insert library calls into his application code. The library-resident code is implicitly included into the application when a user composes an active expression. A_e has been implemented using C++/Unix on shared memory as well as on networks of workstations. The performance overhead of A_es was shown to be insignificant when compared with the fastest available options.

A_e is an ongoing project. Some of the future directions include looking at the issue of debugging of applications developed using this model. We also plan to integrate more commonly used interactions in concurrent programming, such as general patterns for fan-in and fan-out structures, peer-to-peer communication, etc.

References

[1] G. Andrews, R.A. Olsson, M.A. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. "An Overview of the SR Language and Implementation". *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, 1988.

[2] H. Bal, M. Kaashoek, and A. Tannenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[3] Alberto Bartoli, Paolo Corsini, Gianluca Dini, and Cosimo Antonio Prete. "Graphical Design of Distributed Applications Through Reusable Components". *IEEE Parallel and Distributed Technology*, pages 37–50, Spring 1995.

[4] Adam Beguelin, Jack J. Dongarra, G.A. Geist, Robert Mancheck, and V.S. Sunderam. "Graphical Development Tools for Network-Based Concurrent Supercomputing". In *Proceedings of Supercomputing '91*, pages 435–444, Albuquerque, New Mexico, 1991.

[5] K. Birman and R. van Renesse (editors). "Reliable Distributed Computing with the Isis Toolkit". *IEEE Computer Society Press*, 1994.

[6] J.C. Browne, Muhammad Azam, and Stephen Sobek. "CODE: A Unified Approach to Parallel Programming". *IEEE Software*, pages 10–18, July 1989.

[7] 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):137–172, February 1992.

[8] K. M. Chandy. "Concurrent Program Archetypes". In *International Parallel Programming Symposium*, 1994. Keynote Address.

[9] K.M. Chandy and C. Kesselman. "Compositional C++: Compositional Parallel Programming". In *Proc. Fourth Workshop on Parallel Computing and Compilers*, page 91. Springer-Verlag, 1992.

[10] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Programming.* MIT Press, Cambridge, Mass., 1989.

[11] Mauricio. De Simone and Ajit. Singh. "A_e: Model and Implementation". Technical Report 03-97, Electrical & Computer Eng. Dept., University of Waterloo, 1997.

[12] N.H. Genahi et al. "The Concurrent C Programming Language". *Software – Practice and Experience*, 16(9):821–844, 1986.

[13] G.A. Geist and V.S. Sunderam. "Network-Based Concurrent Computing on the PVM System". *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[14] A. Grimshaw, W.T. Strayer, and P. Narayan. "Dynamic Object-Oriented Parallel Processing". *IEEE Parallel and Distributed Technology*, 1(2):33–27, 1993.

[15] IEEE. "Information Technology - Portable Operating Systems Interface (POSIX) - Part 1: System Application Program Interface (API) – Amendment 2: Threads Extension [C Language]". IEEE Standard 1003.1c, New York, NY., 1995.

[16] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of 4.3 BSD Unix Operating System.* Addison-Wesley Publishing Company, Inc., 1990.

[17] D.B. Loveman. "High Performance Fortran". *IEEE Parallel and Distributed Technology*, 1(1):25–42, February 1993.

[18] J. Magee, N. Dulay, and J. Kramer. "Structuring Parallel and Distributed Programs". *Software Engineering Journal*, 8(2):73–82, 1993.

[19] Harley R. Myler. *The Pocket Handbook of Image Processing Algorithms in C.* Prentice Hall, 1993.

[20] William G. O'Farrell, Frank Ch. Eigler, S. David Pullara, and Gregory V. Wilson. "ABC++". In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, Scientific and Engineering Computation Series, chapter 1. The MIT Press, Cambridge, Mass., 1996.

[21] Raju Pandey and James C. Browne. "Support for Extensibility and Reusability in a Concurrent Object-Oriented Programming Language". In *Proceedings of the International Parallel Processing Symposium*, pages 241–247. IEEE, 1996.

[22] Jonathan Schaeffer, Duane Szafron, Greg Lobe, and Ian Parsons. "The Enterprise Model for Developing Distributed Applications". *IEEE Parallel and Distributed Technology*, 1(3):85–96, August 1993.

[23] Ajit Singh, Jonathan Schaeffer, and Mark Green. "A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations". *IEEE Transactions on Parallel and Distributed Systems*, 2(1):52–67, January 1991.

[24] Ajit Singh, Jonathan Schaeffer, and Duane Szafron. "Experience with Parallel Programming Using Code Templates". *Concurrency: Practice and Experience*, 1997. Accepted for Publication.

[25] Stephen Siu. "Openness and Extensibility of Design-Pattern-Based Parallel Programming System". Master's thesis, Department of Electrical and Computer Engineering. University of Waterloo, Waterloo, ON. Canada, August 1996.

[26] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, Mass., 1986.

[27] D. Walker. "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers". *Parallel Computing*, 20(4):657–673, 1994.