Aroma: Language Support for Distributed Objects

Hiroshi Nishikawa and Peter Steenkiste

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Aroma simplifies the task of parallelizing large applications on multicomputers by providing applications with a shared object space. Aroma supports both traditional monolithic objects and aggregate object that can be partitioned across multiple nodes. Aggregate objects support data parallelism efficiently. An Aroma program consists of tasks that operate on shared objects. Tasks typically execute on the node on which their input data is located, thus minimizing communication. Shared data objects have synchronization properties associated with them, making it possible to parallelize a large class of applications without using explicit locks and condition variables. In this paper, we present and justify the Aroma language features, and give examples of Aroma programs. Aroma has been implemented on the Nectar multicomputer and we give performance results for several applications.

1. Introduction

Existing shared memory support for distributed memory systems typically fall in one of two categories. In the first group of systems, the programmer specifies the data distribution across the nodes using data distribution directives. A compiler implements the data mapping, compiles accesses to data on remote nodes into message passing and distributes loop iterations across processors [14, 11, 13]. These systems have proven to be effective for data parallel applications, typically in the form of FORTRAN DO loops, on tightly-coupled systems.

A second group provides more general support for distributed objects. Linda [6], Orca [3] and the Chare kernel [10] all support some form of shared object space. Amber [7] and Munin [4] support a thread-based programming model similar to that found on a shared-memory processors. These systems support a wider range of programming models than the compiler-based systems, but since accesses are resolved at runtime, they are less efficient, and exploiting data locality is harder.

This research was supported by Defense Advanced Research Projects Agency (DOD) monitored by DARPA/CMO under Contract MDA972-90-C-0035. Hiroshi Nishikawa is supported by Matsushita Co. Ltd.

Aroma combines the benefits of both categories: it supports general distributed objects, but allows efficient accesses in the case of regular data structures. This is achieved through *aggregate objects* that are distributed over multiple computing nodes using directives similar to those used by parallelizing compilers. For data parallel computations, aggregate objects support the distribution of data and associated computation, resulting in mostly local data accesses and good load balancing. Aroma data objects can also have synchronization properties associated with them. This makes it possible to write applications without explicit locks or condition variables [8] if the synchronization and data consistency requirements are simple, as is often the case. Knowing the synchronization and access properties of objects also allows an efficient implementation.

An Aroma program is organized as a set of tasks. Tasks are segments of sequential code that can read shared data when they start and write shared data when they finish. Tasks are scheduled based on the availability of their input data and they never block once they started execution. The limited interaction between tasks allows an efficient implementation and also simplifies the writing and debugging of programs: tasks are sequential pieces of code that can be developed without worrying about concurrency issues. Tasks are conceptionally similar to procedures in sequential programs. One of the motivations for organizing the program as a set of tasks is dynamic load balancing; this work is in progress.

This paper concentrates on the definition of shared data in Aroma. Sections 2 and 3 describe the Aroma object and tasks concepts. We present several examples in Section 4 and we give performance results for an implementation of Aroma on the Nectar multicomputer [2].

2. Shared Data

To allow efficient sharing of data by tasks on different nodes, Aroma requires programmers to declare ‘‘application data units’’: blocks of data that are always accessed in their entirety. The Aroma runtime system transfers application units over the network in a single block, and read and write operations on application units appear as atomic operations to the application. *Scalar* Aroma objects consist of a single application unit, while *aggregate* objects consist of a vector of units; they simplify the expression of data parallel programs. Tasks access the elements of aggregate objects independently, but for some operations, such as mapping the data, Aroma views aggregate shared data as a single entity.

Objects can be created at startup time (*static* shared data) or at runtime (*dynamic* shared data). Dynamic data items are destroyed and their resources are reclaimed when all references to them disappear. Static and dynamic data are similar to global and local data in sequential programs.

2.1. Data Placement

The placement of shared data determines what data references require network communication, and as such, it has a big impact on performance. Scalar data objects are normally placed on a single node; the programmer has the option of selecting the node. If a shared object is read by a large number of tasks, the programmer can specify that it should be replicated. The Aroma runtime system directs data read and write requests to the node or nodes that have a copy of the object.

Aggregate objects are usually distributed across the computing nodes. The distribution patterns available in Aroma are similar to those supported by parallelizing compilers: BLOCK, CYCLIC [11, 13] and RANDOM; RANDOM leaves the distribution up to the Aroma runtime system. What distribution is most appropriate depends on how the aggregate object is accessed. If tasks access adjacent elements in the object, for example, the BLOCK pattern should be selected (see examples in Section 4).

2.2. Synchronization on Shared Data

Tasks can read and write shared data. The programmer has the option of changing the semantics of the read and write operations by adding a synchronization behavior to this ‘‘basic’’ data object. The options include write-once, iterator and reduction.

Write-once data objects are similar to the single assignment variables found in dataflow languages. They can be defined exactly once and tasks that read the object before it has been instantiated block. When a task writes the data item, blocked tasks become runnable. Write-once static data roughly correspond to constants. Write-once dynamic data support single-writer/multiple-reader style communication.

The *iterator* data type can be defined multiple times and a version number is used to distinguish between different versions of the data. The version number is incremented every time the object is defined, and read operations specify what version of the data is to be read. When the specified version is not yet available, the task blocks. Version numbers are typically specified relative to the version read by the task, and to allow the runtime system to reclaim the space taken by ‘‘old’’ versions of the data, the version increment have to be positive. The iterator type directly supports iteration in scientific code (DO loops), and can realize a simple version of stream processing.

The *reduction* type allows applications to define objects that have type-specific operations associated with them. Besides the normal read and write operations, operations on reduction objects can update the object based on input parameters. The reduction type is similar to traditional object-oriented types. A simple example of the use of the reduction type is to keep track of the current minimum cost in optimization problems. The minimum cost is replicated, and the update operation replaces the cost by the minimum of the old minimum and the new minimum. Since the update operation is associative it can be performed on all nodes in parallel without the need to enforce a specific order. The type-specific operations on reduction objects can block in the same way as read operations on write-once and iterator objects.

2.3. Data Consistency

When multiple tasks can access data in parallel, it is necessary to provide mechanisms to control access to the shared data [8]. The distributed memory environment also requires us to worry about maintaining data consistency. In Aroma, read operations are implemented by fetching the data from the node that stores the object; if the object is local or if the object is replicated this is a completely local operation. Write operations are performed by sending the new value to the node that stores the object; if the object is replicated, the value is broadcasted to all the nodes. This implementation is efficient, but it does not guarantee sequential data consistency. For example, objects on different nodes can be updated in an order that is different from the order in which the updates were issued, and the multiple copies of the shared data can become inconsistent [9].

The Aroma programming model encourages a programming style that avoids these problems. For example, related information is typically combined in a single object, making updates atomic because of the semantics of the Aroma shared objects. Moreover, the write-once and iterator types encourage the users to adopt a ‘‘single writer’’ programming style where objects have at most one (potential) writer at any point in time. This makes it possible to write applications without the need for explicit locks, even though the memory consistency provided by Aroma are minimal as described above.

When more complex inter-task interactions are needed, for example in the case of multiple unsynchronized writers and replicated data, tasks can select a lock option for specific data accesses. If the lock option is specified, the object is locked as part the read operation and is unlocked when the task finishes and updates the object. For replicated objects, the lock option forces all read/lock and write/unlock operations to go through a specific node, which enforces the lock. Read, write, read/lock and write/unlock requests from any node to another node are sent and handled in order. With locking, the Aroma data consistency model is similar to weak consistency [9]. Locking makes some accesses to objects more expensive, but it provides stronger consistency guarantees if needed. Most applications do not need locks.

3. Aroma Tasks

Tasks are the unit of work in Aroma. Tasks are sections of sequential code that read shared objects when they start, and write one or more shared objects before they finish. They never block during their execution. Each node has a task queue and runs a schedule-execute loop. Scheduling is based on data availability: if all the inputs of a task are available as specified by the synchronization properties of the data, the task is executed, otherwise it remains blocked. Although tasks can be executed on any node in the system, it is clearly desirable to execute tasks on the node that holds all or most of its input data. The programmer can select the parameter that should be used to determine the execution node.

The declaration of a task consists of code (specified as a C procedure), and a list of the formal parameters. The declaration of the parameters specifies for each object whether the task will read (IN), write (OUT), or read and write (IN/OUT) the object. This information is used by the runtime system to determine what data should be fetched before the task can be started. Objects can also be passed through (PASS) to be used as parameters for other tasks.

Tasks can be created at application startup time and by other tasks at runtime. At task creation time, the shared objects that it accesses are specified by name. When the task is scheduled for execution, the names are translated into pointers that can be used by the C program that forms the body of the task. Tasks still have to address objects by name when they write the object, or when they pass it as a parameter to another task. IN parameters can also be regular C data structures. Regular parameters are passed by value, i.e. a copy of the parameters is passed into the task. Since tasks can execute on any node, it is invalid to specify a pointer as a parameter.

4. Program examples

We use three Aroma programs to clarify the notions described above. We also present performance results for the applications for a prototype Aroma implementation on the Nectar multicomputer [2]. The execution time is the elapsed time on a set of dedicated Sun4/330 workstations connected by Nectar. Speedups are calculated using the execution time of carefully coded sequential programs. The syntax of the programs could be improved substantially by adding some compiler support to Aroma.

4.1. Poisson

Figure 1 shows a poisson program that uses odd-even ordering. Two matrices named **odd** and **even** store the current and new values; they are declared as vectors of rows of type iterator and their placement follows a BLOCK distribution. The computation is done by invocations of **poisson_task**. **poisson_task** takes as input three adjacent rows of the matrix and calculates the next version of center row, i.e. it corresponds to one iteration of the innerloop of the sequential algorithm. The primitive **data()** does the name to address translation for shared data. After the computation is finished, the result is written into the shared data object, and before the task exits, a new task is created for the next iteration, if needed. Since the new and old tasks have the same format, the same task structures are reused, thus reducing the task overhead; this optimization is similar to tail recursion elimination in sequential programs [1].

The Aroma program is started by a main program that first initializes the data and then creates the tasks for the first iteration of the algorithm, one per row in the matrix. It also prints out the results after the computation is done. This is detected using the **barrier()** primitives; it returns when all task queues are empty and all data updates have been performed.

The program in Figure 1 will result in little network traffic. First, the task creation statements specify that the tasks should execute on the node that holds the middle row used by the task. This, together with the BLOCK distribution of the matrices, means that almost all accesses to shared data are local: for each iteration, each node only has to fetch at most two rows from other nodes. Moreover, since all synchronization is done through data of type iterator, it does not require any additional communication. The table in Figure 1 shows the execution time for a 202×202 matrix and 100 iterations. The single node Aroma program is slower than the sequential program because of the overhead of managing tasks. The task granularity (580 microseconds) is relatively small for a multicomputer.

4.2. Prime Sieve

A number is prime if it cannot be divided by any prime smaller than its square root. Figure 2 shows an Aroma program to calculate the prime numbers in the interval 0..LIMIT based on this rule; it is similar to the implementation in Linda in [6]. It first calculates the prime numbers smaller than the square_root(LIMIT) and places them in the replicated array **i_primes** of type write-once, so all tasks can access them locally. The interval 0..LIMIT is broken up in segments of size RANGE. Each task calculates the prime numbers in one segment and writes the prime numbers it found in the array **primes** of type basic. The tasks are distributed across the nodes in a CYCLIC way. This results in good load balancing, since testing whether a number is prime is

```
/* Declaration of shared data and tasks */
iterator { float[N] } odd[N]#BLOCK;
iterator { float[N] } even[N]#BLOCK;

task task_poisson((IN int), (IN int),
  (IN float[N]), (IN float[N]), (IN float[N]),
  (PASS float[N]), (OUT float[N]), (PASS float[N]));

/* Definition of the task poisson_task */
poisson_task(i, k, up, mid, down, n_up, n_mid, n_down)
int i, k;
iterator { float[N] } up, mid, down, n_up, n_mid, n_down;
{
  float *result;
  /* Get a new data area for the result and call the poisson
    calculation; data() does the name to pointer conversion. */
  result = allocate_version(n_mid);
  poisson(result, data(up), data(mid), data(down));
  /* Write the result and if needed, start the next iteration
    by creating new task; deal with boundary conditions */
  write_iterator(n_mid, result);
  if (k > MAX) return;
  if (i == 1)
    n_mid#task_poisson(i, k+1,
       n_up, next(n_mid), next(n_up), up, mid, down);
  else if (i == N-2)
    n_mid#task_poisson(i, k+1,
       next(n_up), next(n_mid), n_down, up, mid, down);
  else
    n_mid#task_poisson(i, k+1,
       next(n_up), next(n_mid), next(n_up), up, mid, down);
}

main() {
  /* Initialize data */
  /* For every iteration: Create a task on node where
    odd[i] is located */
  for (i = 1; i < N-1; i++)
    odd[i]#poisson_task(i, 0,
       odd[i-1], odd[i], odd[i+1],
       even[i-1], even[i], even[i+1]);
  barrier();  /* Wait until all tasks are done */
  /* Print result */
}
```

Number of nodes	seq	1	2	3	4	5	6	7
Time (sec)	11.8	13.6	6.76	4.51	3.40	2.95	2.40	2.10
Speed Up	1.00	0.86	1.74	2.61	3.46	3.99	4.90	5.60

Figure 1: Poisson equations

more expensive for larger numbers. The main routine combines the results when all tasks are finished. The speedup of the prime sieve program is close to linear.

```
#define LIMIT 1000000
#define GRAIN 2000
struct init_prime {
    struct {
        int p;  /* prime number */
        int p2; /* square of prime number */
    } init_p[LIMIT/100];  };
struct prime {
    int no;        /* number of prime found in the block */
    int p[GRAIN/5]; /* the prime numbers */  };

write_once { struct init_prime } i_primes#replicate;
basic { struct prime } primes[LIMIT/GRAIN + 1]#host;
task init_primes((OUT struct init_prime)); /* not shown */
task is_prime((IN struct init_prime), (IN int),
  (OUT struct prime));

is_prime(i_pr, i, out_pr)
write_once { struct init_prime } i_pr;
int i;
basic {struct prime} out_pr;
{
    struct prime *my_prime;
    my_prime = allocate_version(out_pr);
    my_prime->no = 0;
    calculate_prime_segment(data(i_pr), i, my_prime);
    write_basic(out_pr, my_prime);
}

main() {
    /* Calculate initial primes on the host node */
    host#init_primes(i_primes);
    /* Calculate the prime numbers */
    for (i = 0; i < LIMIT/GRAIN; i++)
        cyclic#is_prime(i_primes, i, primes[i]);
    barrier();
    /* Print results */
}
```

Number of nodes	seq	1	2	3	4	5	6	7
Time (sec)	82.1	82.8	42.0	28.3	21.7	17.2	14.6	12.7
Speed Up	1.00	0.99	1.95	2.90	3.78	4.78	5.62	6.46

Figure 2: Prime sieve

4.3. Simulation of Air Pollution in Los Angeles

We rewrote a program that tracks pollutant particles in the atmosphere of the Los Angeles area using Aroma. The sequential LA simulation program consists of about 2500 lines of FORTRAN and 300 lines of C codes and a distributed version using send and receive primitives is described in [12]. Computing the particle traces is a two phase process. Phase I consists of computing wind velocity at each point of a regular grid on the concerned geographic area, for every hour given the measurements from weather stations. In phase II, each particle is tracked as it moves about the grid. The two phases communicate through an array of type iterator. The results are shown in Table 1; the speedup of the Aroma version is similar to that of the send-receive version. In the send-receive version, phases I and II are implemented as separate processes. This mapping was chosen because it avoids the complexity of a user-level scheduler. In the Aroma version, the two phases are scheduled by the Aroma runtime system much more efficiently.

Number of nodes	seq.	1	2	3	4
Time (seconds)	60.6	61.5	32.1	21.8	17.0
Speed Up	1.00	0.98	1.89	2.80	3.56

Table 1 LA Performance

5. Conclusion

We described a system that provides support for distributed shared data objects on a distributed memory system. Aroma provides direct support for data parallelism through aggregate shared data, but it also allows programs based on other paradigms such as macro dataflow and master-slave. Shared objects in Aroma have synchronization properties associated with them that simplifies the management of shared data without the use of explicit locks or condition variables. Aroma programs are organized as a set of tasks that read shared objects when they start, and write objects when they terminate; the task itself is sequential code. Our initial experience with Aroma indicates that the performance of Aroma programs is comparable to that of send-receive based programs, but that Aroma programs are easier to write. We are in the process of adding dynamic load balancing to Aroma. This should make Aroma useful for less regular applications and in environments with shared or heterogeneous computing nodes [5].

References

1. Alfred V. Aho, Ravi Sethi and Jeffery D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1986.

2. Emmanuel Arnould, Francois Bitz, Eric Cooper, H. T. Kung, Robert Sansom and Peter Steenkiste. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ACM/IEEE, Boston, April, 1989, pp. 205-216.

3. Henri Bal, Andrew Tanenbaum, and M. Frans Kaashoek. "Orca: A Language for Distributed Programming". *SIGPLAN Notices 25*, 5 (May 1990), 17-24.

4. John Bennett, John Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, March, 1990, pp. 168-176.

5. Bernd Bruegge, Hiroshi Nishikawa, and Peter Steenkiste. Computing over Networks: An Illustrated Example. Proceedings of the Sixth Distributed Memory Computing Conference, IEEE, April, 1991, pp. 254-257.

6. Nicholas Carriero and David Gelernter. "How To Write Parallel Program: A Guide to the Perplexed". *ACM Computing Surveys 21*, 3 (September 1989), 323-357.

7. Jeffrey Chase, Franz Amador, Edward Lazowska, Henry Levy, and Richard Littlefield. The Amber System: Parallel Programming on a Network of Multiprocesses. Proceedings of the Twelfth Symposium on Operating System Principles, ACM, December, 1989, pp. 147-158.

8. Eric C. Cooper and Richard P. Draves. C Threads. Tech. Rept. CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June, 1988.

9. Kourosh Gharachorloo, Daniel Lenoskim, James Laudon, Phillip Gibbons, Anoop Gupta and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. 17th Annual International Symposium in Computer Architecture, IEEE, May, 1990, pp. 15-26.

10. L.V.Kale. The Chare Kernel Parallel Programming Language and System. International Conference on Parallel Processing, IEEE, August, 1990, pp. 17-25.

11. Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting Shared Data Structures on Distributed Memory Architectures. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, March, 1990, pp. 177-186.

12. H.T. Kung, Peter Steenkiste, Marco Gubitoso, and Manpreet Khaira. Parallelizing a New Class of Large Applications over High-Speed Networks. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, April, 1991, pp. 167-177.

13. Matthew Rosing and Robert P. Weaver. Mapping Data to Processors in Distributed Memory Computations. Proceedings of the Fifth Distributed Memory Computing Conference, IEEE, April, 1990, pp. 884-893.

14. Ping-Sheng Tseng. Compiling Programs for a Linear Systolic Array. Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM, June, 1990, pp. 311-321.