11th International Parallel Processing Symposium (IPPS'97), April 1-5, 1997, Copyright 1997 IEEE Computer Society

Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing

Paul Lu
Dept. of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 3G4
paullu@sys.utoronto.ca

Abstract

We introduce the all-software, standard C++-based Aurora distributed shared data system. As with related systems, it provides a shared data abstraction on distributed memory hardware. An innovation in Aurora is the use of scoped behaviour *for per-context data sharing optimizations (i.e., portion of source code, such as a loop or phase). With scoped behaviour, a new language scope (e.g., nested braces) can be used to optimize the data sharing behaviour of the selected source code. Different scopes and different shared data can be optimized in different ways. Thus, scoped behaviour provides a novel level of flexibility to incrementally tune the parallel performance of an application.*

1. Introduction

Parallel programming systems based on shared memory and shared data models are becoming more popular and widespread. Accessing local and remote data using the same programming interface (i.e., reads and writes) is often more convenient than mixing local accesses with message passing. Consequently, the desire to support logically shared data, even on distributed memory hardware, has resulted in a variety of *distributed shared memory* (DSM) [15, 5, 1] and *distributed shared data* (DSD) [3, 20, 12] systems.

DSM systems are usually based on fixed-sized units of sharing, often a page, and DSD systems usually have a variable granularity of sharing. *False sharing*, when performance is degraded by the interference between updates to logically independent data located in the same unit of data management, is a potential problem with DSM systems [17]. In DSD systems, false sharing is avoided by managing independent data as separate objects.

Another challenge for distributed systems is to overcome the network latency and bandwidth limitations of typical distributed memory platforms. In particular, remote accesses of shared data can be expensive on general-purpose networks.

The Aurora DSD system provides a familiar shared data programming model on distributed memory hardware. Aurora addresses the false sharing problem by allowing a per-object granularity of sharing. Aurora addresses the latency and bandwidth problems by providing a set of data sharing optimizations: The *owner-computes* technique co-locates computation and data to avoid remote read accesses. *Data placement* mechanisms in Aurora enhance the effectiveness of owner-computes. *Caching read-only data* in local memory further eliminates remote read accesses. A special form of *prefetching* into a read cache is also supported. As well, Aurora provides a form of *release consistency* [10] to amortize the cost of remote write accesses by batching updates.

Aurora is unique in its use of the *scoped behaviour* mechanism for applying optimizations. With scoped behaviour, a new language scope (e.g., nested braces in C++) around selected source code can be used to optimize the data sharing behaviour. Using a standard C++ compiler, the optimizations can be applied on a per-context (i.e., portion of source code) and per-object (i.e., specific shared data) basis to incrementally tune the performance of a parallel application. Thus, different loops and computational phases can be optimized in different ways, according to the needs of the application.

2. Aurora: a distributed shared data system

Aurora is a C++ class library and run-time system that provides a shared data programming model on distributed memory architectures. Scalar and vector data can be distributed across, and accessed from, different address spaces.

Aurora's mixed-parallelism process model includes both task parallelism and data parallelism. The various process models can be mixed, in reasonable ways, within a single application. Currently, the task parallelism is provided by the underlying ABC++ library through active objects [18]. Active objects are normal C++ objects that contain their own thread of control. Data parallelism is provided by a simple `for_all()` construct that invokes a parallel method

```
 1:extern int dotProd( int * a, int * b, int j, int n );
 2:
 3:void mmultiply(    // SPMD thread and shared data among parameters
 4:  ProcessTeam * myTeam,
 5:  GVector<int> mA, GVector<int> mB, GVector<int> mC,
 6:  const int m, const int n, const int p )
 7:{
 8:  int i, j;
 9:
10: { // Begin scoped behaviour
11:  NewBehaviour( mA, GVOwnerComputes, int );
12:  NewBehaviour( mB, GVReadCache, int );
13:  NewBehaviour( mC, GVReleaseC, int );
14:
15:  while( mA.doParallel( myTeam ) )   // Owner-computes
16:  {
17:    for( i = mA.begin();i < mA.end();i += mA.step() )
18:      for( j = 0; j < p; j++ )
19:        mC[i][j] = dotProd( &mA[i][0], mB, j, n );
20:  } // while
21: } // End scoped behaviour
22:}
```

Figure 1. Matrix multiplication, SPMD-style

on all distributed data. Aurora also provides a fairly standard single-program, multiple-data (SPMD) process model (similar to the PCP system [7]). The model includes teams of threads, sequential sections (i.e., code to be executed by only one thread, usually the team's "master"), and synchronization barriers.

2.1. Shared data objects

Aurora's data model, which is our main focus, requires that shared scalar and vector objects be created using Aurora's templated C++ classes `GScalar` and `GVector`:

```
GScalar<int>     scalar1;
GScalar<VNode>   scalar2;            // User-defined type
GVector<float>   vector1( vsize );   // vsize elements
```

`scalar1` is a C++ integer that is shared across address spaces. By default, updates to `scalar1` are immediately visible to other processors, which may require a message send for each update. Immediate updates may be expensive, but it is the overall least error-prone default behaviour. Any of the C++ built-in types or any user-defined concrete type [8], as with `scalar2`, can be an independent unit of sharing.

Shared vectors, such as `vector1`, are similar to shared scalars, except that the vector elements can be distributed across nodes and the subscript operator is overloaded to support indexing. Parameters to the constructor specify the size of the vector and (optionally) data placement on the nodes. Currently, the vector elements can only be block distributed across the specified nodes.

2.2. Illustrative example: matrix multiplication

Consider the parallel dense matrix multiplication function in Figure 1. The Aurora implementation uses the SPMD process model, distributed matrices[1], and three different data sharing optimizations applied using scoped behaviour. At the heart of the function are the nested loops and a C-style dot product function.

Three elements of the SPMD process model are shown in this example. First, the `myTeam` (line 4) parameter points to the local thread executing the function. Each thread has a unique number and each thread is part of a larger team of threads that is executing the same function in the application.

Second, the `NewBehaviour()` with `mA` (line 11), which uses a macro provided by Aurora, is an example of scoped behaviour and owner-computes. Note that this macro exists within a new language scope (i.e., the braces on lines 10 and 21). Outside of the scope, matrix `mA` continues to behave as a object of type `GVector<int>`. Inside the scope, the macro changes the type of matrix `mA` to `GVOwnerComputes<int>`. This changes matrix `mA`'s behaviour such that only local data (i.e., the portions of `mA` co-located with the thread) can be accessed and it can be accessed using C-style indexing. Since the scoped behaviour is known at compile-time, `NewBehaviour()` actually changes the code generated by the compiler to use a faster access technique. The source code for accessing the shared data is unchanged.

Third, the combination of `doParallel()`, `begin()`, `end()`, and `step()` method calls (lines 15 and 17) guarantees that the computation will span all of the local data. These methods are *not* defined for class `GVector` so they can only be used with owner-computes scoped behaviour. For a given thread in `myTeam`, the `doParallel()` method guarantees to return `true` until all of the local data for the object has been processed. For each call to `doParallel()`, methods `begin()`, `end()`, and `step()` will return the correct parameters to cover a portion of the local data. Using owner-computes for matrix `mA` improves performance by eliminating a large number of remote read accesses and it is a convenient way to partition the parallel work among the different SPMD threads.

However, even after optimizing the accesses to matrix `mA`, performance problems remain because matrices `mB` and `mC` are also distributed. Therefore, the behaviour of `mB` within the scope is modified so that each thread makes a local copy of the entire matrix before entering the loops (line 12). Since `mB` is re-used many times and since its values do not change, it is cached in local memory for read accesses. Again, since `mB` is of type `GVReadCache<int>` within the scope, the code generated by the compiler is different than without the scoped behaviour.

The behaviour of `mC` within the scope is modified (line 13) so that all updates to the matrix are not immediately sent, but are buffered until a full network message can be sent or

[1] As in C/C++, multi-dimensional matrices in Aurora are implemented using one-dimensional vectors and a simple indexing function. For clarity, matrix indexing is shown using matrices of static size.

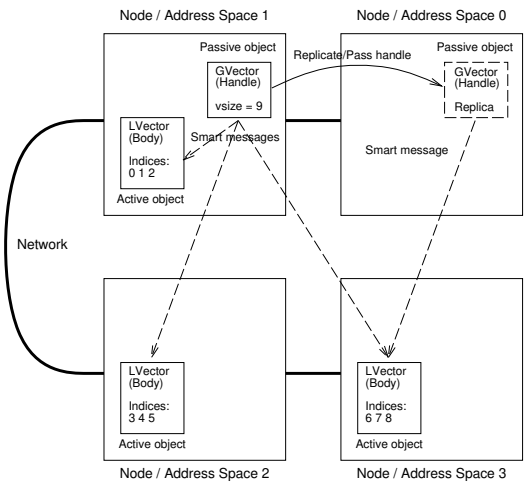

Figure 2. Basic handle-body architecture

the scope is exited (line 21). Consequently, the cost of sending a message is amortized over many updates to improve performance. This behaviour is functionally equivalent to the release consistent memory model for writes. But, in contrast, the synchronization point for remote updates is at the end of the scope and only the updates to the specific data object need to be communicated.

Note that the scoped behaviours of the matrices are all different and are independent of each other (i.e., per-object). They only share the lifetime of the language scope. In fact, different combinations of scoped behaviour can be correctly used. For example, owner-computes could be applied to matrix `mC` and both matrices `mA` and `mB` could be cached for reading, albeit with greater resource overheads. This strategy could be tried with minimal changes to the code and should achieve comparable performance. The looping code for owner-computes needs to be changed, but the user's code to access shared data is usually the same. The standard C++ compiler generates the correct access code using the Aurora class library.

2.3. Basic architecture and implementation

Both shared scalars and vectors in Aurora are implemented through the *handle-body* idiom [8]. The handle class implements the appropriate overloaded operators and type constructors. The handle also provides the location and distribution transparency required to access remote data. In contrast, the body objects contain the actual data. The separation of handle and body is important to Aurora's flexibility.

Consider a vector of nine elements, say `GVector<int> a( 9 )`, block distributed across three (of four) nodes, as illustrated in Figure 2. Assuming block distribution, the bodies of the vector are distributed across three *local objects*, each (potentially) located in a different address space and on a different physical processor. Handles can be replicated cheaply and passed as parameters, but the local objects are shared among the handles.

Currently, Aurora implements handles as passive objects, but the local objects are implemented as ABC++ active objects. Operations on the member data of an active object are handled by the member functions, as is the case with passive objects. However, unlike passive objects, the member functions of an active object can also be *remotely* invoked with a smart message [2]. In ABC++, smart messages resemble normal method invocations and automate parameter marshalling and de-marshalling. Distributed memory and shared memory ABC++ run-time systems are available to Aurora applications.

By default, a read or write to a shared object is translated into a smart message from the handle to a local body object. Thus, Aurora uses smart messages, in part, for `put()`s to and `get()`s from remote memory.

2.4. Scoped behaviour: per-context optimization

Scoped behaviour exploits the compiler's implementation of the scoped visibility (i.e., hiding) of methods and overloaded operators, the hiding of variable names in nested scopes, and the extra level of indirection provided by the handle-body approach. The indirection overhead can also be avoided, in controlled ways, such as with type constructors that return a C-style pointer (e.g., `int *`) to local data and read caches (e.g., calling `dotProd()`, line 19, Figure 1). Aurora also uses the automatic construction and destruction of scoped objects in C++ to properly initiate the required coherence and data movement actions. Therefore, scoped behaviour is a combination of compile-time and run-time techniques.

Consider the following code, where the local objects of the shared vector are explicitly placed on nodes 0 and 1:

```
  GVector<int> vector1( vsize, Nodes( 0, 1 ) ); // Shared
{                                                // New
  NewBehaviour( vector1, GVReleaseC, int );      // New

  for( i = 0; i < vsize; i++ )     // Update intensive
    vector1[ i ] = someFunction();
}                                                // New
```

Given that updates to `vector1` are immediately communicated by default, the loop would normally generate `vsize` individual update messages. However, if the semantics of the program allow it, a single message to each node could be sent instead. Thus, if the update buffers are sufficiently large, all of the updates to nodes 0 and 1 can be communicated with two messages. Message batching to exploit packet sizes and amortize per-message overheads is a well-known performance optimization.

Scoped Behaviour	Comment
Owner-computes	Threads access only co-located data.
Caching for reads	Create local copy of data.
Release consistency	Buffer write accesses.
Special-purpose prefetching	Used with owner-computes for *specific* applications.

Table 1. Current scoped behaviours

The scoped behaviour requires three new elements. Two of the new elements are simply opening and closing curly braces that define the context of the intended optimization. These braces delimit a language scope within which objects can be created and are then automatically destroyed at the end of the scope. Furthermore, objects within a nested scope can be given the same *name* as an object in an outer scope, thus effectively hiding the previous object.

Specifically, the `NewBehaviour()` macro (the third new element) creates a new handle of type `GVReleaseC<int>`, called a *scoped behaviour object*. The macro calls the constructor of class `GVReleaseC` with the old handle to `vector1` as a parameter. Without copying the bodies of `vector1`, the new handle now controls access to the shared vector elements within the scope. The behaviour of the new handle is identical to the original handle except that updates are now batched. When the scope is exited, the scoped behaviour object is destroyed and the batched updates are flushed by `GVReleaseC`'s destructor.

The new handle is also called `vector1`, thus it lexically hides the old handle, and all references to `vector1` within the new scope are compiled to use the new handle without changes to the user's source code. Consequently, it can be easy to add, change, and experiment with scoped behaviour without requiring many error-prone lexical changes to the code.

There are three important aspects to Aurora's mechanisms and architecture:

1. Operator overloading supports location and distribution transparency across address spaces.
2. The handle-body architecture and scoped behaviour substitutes handles for the lifetime of a language scope, thus altering the behaviour of data accesses.
3. The lexical hiding semantics of nested scopes in C++ (and similar block-structured languages) causes the compiler to generate the appropriate scoped behaviour code without changes to the programmer's code.

Although the current set of scoped behaviours is small (Table 1), the behaviours can be combined on a per-context and per-object basis, and enhanced with explicit data placement, to support a variety of optimization strategies.

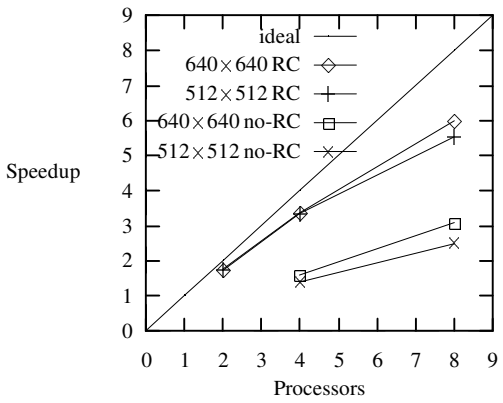

Figure 3. Matrix multiplication

3. Experience with parallel programs

We discuss the use of scoped behaviour in actual parallel programs and provide some initial performance results. Note that all speedups are computed against a sequential C language implementation of the same algorithm.

The hardware platform is a cluster of workstations, assembled as part of the University of Toronto's Parallelism on Workstations project. Each workstation has a single 133 MHz PowerPC 604 processor and 64 MB of memory. The software includes IBM's AIX 4.1 operating system, POSIX threads (pthreads), the xlC C++ compiler, a prototype version of ABC++, and the MPICH [9] implementation of MPI for inter-workstation communication. A single 100 Mbit/s Ethernet network is used for these experiments.

3.1. Matrix multiplication

Figure 3 shows the speedup for matrix multiplication as implemented in Figure 1. Speedups are given for multiplying two 512×512 matrices and for two 640×640 matrices. The real times for 8 processors are about 12 seconds and 22 seconds, respectively. The data sets labelled RC correspond to Figure 1, with all three scoped behaviours.

Since matrix multiplication is known to be easy to parallelize, the speedups are, not surprisingly, good. However, as more processors are added, there is greater contention for the Ethernet network when matrix `mB` (see Section 2.2) is loaded into the read cache, and speedup declines.

An advantage of the Aurora approach is that one can often experiment with different optimization strategies. For example, the program can be executed without the release consistency optimization for matrix `mC` by simply commenting out the `NewBehaviour()` (line 13, Figure 1). No other code changes are needed. Thus, by default, `mC` is updated immediately with the performance labelled no-RC in Figure

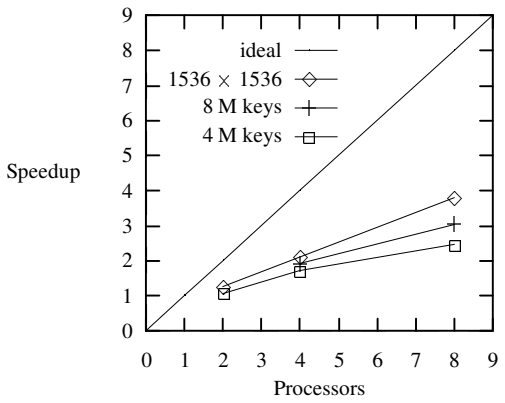

Figure 4. 2-D diffusion and PSRS

3. Given the relatively high latencies and medium access contention of the network, the performance decrease is substantial.

In this example, the key to good performance is the use of C-style pointers in `dotProd()`, which avoid the overheads of operator overloading. In effect, Aurora orchestrates the necessary data movement via scoped behaviour and then provides a convenient way to bypass the overheads through type constructors. Admittedly, once C-style pointers are introduced in the source code, it becomes more difficult to simply "comment out" a scoped behaviour. However, a function template version of `dotProd()` can be used to allow either C-style pointers or Aurora handles as parameters. Although optimizing to C-style pointers is important for compute-intensive loops and functions, it is not always critical to performance. Specifically, when debugging a program and for non-compute-intensive contexts the overheads are tolerable. Aurora allows the programmer to optimize the program when and where it matters most.

3.2. 2-D diffusion

Simulating the diffusion of matter in two dimensions involves data sharing patterns that are archetypical of a class of applications. In our *simplified* simulation of 2-D diffusion, each floating point element of a matrix represents the density of matter at a given coordinate. For each time-step, the new density at a coordinate is computed by taking the average of the previous density and its eight nearest neighbours. This computation is known as a 9-point stencil operation.

Assuming a block-distributed matrix, the normal form of owner-computes is a problem because each thread needs to access some matrix elements from a different thread's local body when a stencil spans local bodies. In particular, at the top and bottom borders of the block, the nearest neighbours above or below the current coordinate can reside in a different local body.

Aurora addresses this form of data sharing by supporting block-distributed matrices with border padding and a modified form of owner-computes. These matrices are identical to normal block-distributed matrices except for the extra storage overhead required for the padding. Furthermore, when the scoped behaviour is invoked, the new handle automatically prefetches the contents of the border from the appropriate local bodies. Therefore, the computationally intensive 9-point stencil operation can be performed using a C-style function on the local block plus padding.

A total of 32 time-steps are simulated on a 1536×1536 grid, with the performance given in Figure 4. A single processor requires about 48 seconds for the simulation, which is a low 1.5 seconds per time-step. On 8 processors, which requires a barrier synchronization after each time-step in the current implementation, the simulation takes about 12.6 seconds. A larger granularity of work would help improve the speedups for this parallel program.

Of course, the padding strategy used by Aurora is not new. However, Aurora provides it in the context of scoped behaviour, it abstracts the implementation details of the padding, and it automates the prefetching behaviour required for this archetypical data sharing pattern.

3.3. Parallel sorting by regular sampling

The Parallel Sorting by Regular Sampling (PSRS) application kernel [22, 16] has been implemented using Aurora. PSRS is a general purpose, comparison-based sort with key exchange. It can be communication intensive since the number of keys exchanged grows linearly with the problem size.

The basic PSRS algorithm consists of four distinct phases. Assume that there are p processors and the original vector is block distributed across the processors. Phase 1 does a local sort (usually via quicksort) of the local block of data. No interprocessor communication is required for the local sort. Then, a small sample (usually p) of the keys in each sorted local block is gathered by each processor. In Phase 2, the gathered samples are sorted by the master process and $p - 1$ pivots are selected. These pivots are used to divide each of the local blocks into p partitions. In Phase 3, each processor i keeps the ith partition for itself and gathers the ith partition of every other processor. At this point, the keys owned by each processor fall between two pivots and are disjoint with the keys owned by the other processors. In Phase 4, each processor merges all of its partitions to form a sorted list. Finally, any keys that do not reside in the local data block are sent to their respective processors. The end result is a block-distributed vector of sorted keys.

A multi-phase algorithm with several shared data objects, like PSRS, is particularly suited for the per-context and per-object data sharing optimizations in Aurora. The optimiza-

Phase	Algorithm	Main Optimizations
1	Sort local data and gather samples.	Sort local data using owner-computes. Samples are co-located with master and are gathered using release consistency.
2	Sort samples, select pivots, and partition local data.	Master sorts samples using owner-computes. Pivots are accessed from a read cache.
3	Gather partitions.	Partitions are gathered into a read cache using `distmemcpy()`.
4	Merge partitions and update local data.	Merge partitions from the read cache.

Table 2. Per-context and per-object optimizations in PSRS

tions required, and the objects that are optimized, differ from one phase to another phase. Table 2 summarizes the phases of PSRS and the main Aurora optimizations. Note that the data movement in Phase 3 is implemented with a `memcpy()`-like construct (called `distmemcpy()`) with full location and distribution transparency.

Also, the programmer's knowledge, that the samples will be sorted by the master in Phase 2, can be exploited by placing that local object on the master's node. For this multiple-producer and single-consumer sharing pattern, the updated samples are sent directly to where they are needed for the next phase. Notably, the combination of using owner-computes with explicit data placement was incrementally added only after the program was debugged and it required no changes to the code that accessed the shared data.

The performance of PSRS is given in Figure 4. For the (limited) range of processors used, adding processors results in increased performance. Informal experiments with 16 non-dedicated workstations indicate that the speedup continues to grow, which is an encouraging trend. With 8 processors, sorting 8 million random keys takes about 16 seconds, with 44% of the time spent in communication-intensive Phase 3. Similarly, sorting 4 million random keys on 8 processors takes about 10 seconds, with 42% of the time spent in Phase 3.

Clearly, the bottleneck is the underlying communications software, including Aurora, and the hardware. Adding more processors and more keys result in more contention for the Ethernet, especially during Phase 3. Experimenting with different software and hardware techniques to deal with contention will be the topic of future work.

4. Discussion and related work

Since Ivy [15], the first DSM system, a large body of work has emerged in the area of DSM and DSD systems. These systems vary in their strategies to overcome false sharing, tolerance for high communication costs [5, 1], reliance on compiler [4, 3, 6] and hardware support [13, 19], implementation techniques [20, 12], and their extensibility. Related work in parallel array classes (for example, [14]) has also addressed the basic problem of transparently sharing data.

One refinement in Aurora's implementation of release consistency is that only the affected data objects are made consistent at the end of a scope, instead of an entire page or all of memory. We refer to this as *per-object release consistency*. In this way, Aurora partitions the consistency namespace, as does scope consistency [11] and entry consistency [6]. However, Aurora does not require the hardware support of scope consistency and Aurora has a per-object granularity of coherence, instead of per-page. Also, Aurora does not require the special compiler support of, say, the Midway implementation of entry consistency.

The basic ideas behind scoped behaviour are not new. The notion of nested scopes is fundamental to block-structured sequential languages. The association of actions with C++ constructors and destructors is also not new. For example, ABC++ also associates data movement actions with the scoped lock handles of its parametric shared regions [18]. Aurora's innovation is the application of scoped behaviour, with the handle-body architecture, to the problem of per-context and per-object data sharing optimization.

In a largely C context, shared regions [20], SAM [21], and CRL [12], associate run-time coherence actions with programmer-inserted code annotations (i.e., function calls). One advantage of the automatic invocation of constructors and destructors in Aurora is that it is impossible to omit an annotation and miss a coherence action. Aurora is also able to exploit compile-time information to generate different code for different behaviour.

More generally, scoped behaviour, as an interface and mechanism, is similar to compiler directives and annotations. However, as an alternate implementation technique, scoped behaviour does not require complex compiler support and its class library approach is easier to extend than modifications to a compiler.

5. Concluding remarks

We have introduced an all-software, standard C++-based DSD system called Aurora. The system is unique in its combination of a handle-body architecture for shared data and the novel use of scoped behaviour for data sharing optimizations. The advantages of Aurora's approach include its per-context and per-object flexibility and the relative ease with which applications can be incrementally tuned.

We have experimented with Aurora's performance using two small parallel programs and a more substantial parallel sorting application kernel. For the network of workstations

platform, the performance has been encouraging. Although the current set of scoped behaviours is small, the ability to combine them on a per-context and per-object basis has made them sufficient for these programs. The development of Aurora is on-going and future research will include more application experience, implementing more scoped behaviours, and experiments using other network technologies.

6. Acknowledgments

Thank you to Ben Gamsa, Kate Keahey, Bill O'Farrell, Eric Parsons, Karen Reid, John Reynders, Jonathan Schaeffer, Ken Sevcik, Michael Stumm, Roel van der Goot, Greg Wilson, Songnian Zhou, and the anonymous referees for their comments and support during this work. Thank you to the Department of Computer Science and NSERC for financial support. Thank you to ITRC and IBM for their support of the POW Project.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] E. Arjomandi, W. O'Farrell, and G. Wilson. Smart Messages: An Object-Oriented Communication Mechanism for Parallel Systems. In *Proc. USENIX 1996 Conference on Object-Oriented Technologies*, Toronto, Ontario, Canada, June 1996.

[3] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[4] M. Beltrametti, K. Bobey, and J. Zorbas. The Control Mechanism for the Myrias Parallel Computer System. *Computer Architecture News*, 16(4):21–30, September 1988.

[5] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. 1990 Conference on Principles and Practice of Parallel Programming*. ACM Press, 1990.

[6] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Proc. 38th IEEE International Computer Conference (COMPCON Spring'93)*, pages 528–537, Feb. 1993.

[7] E. Brooks III. PCP: A Parallel Extension of C That is 99% Fat Free. Technical Report UCRL-99673, Lawrence Livermore National Laboratory, Livermore, CA, 1988.

[8] J. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison–Wesley, 1992.

[9] N. Doss, W. Gropp, E. Lusk, and A. Skjellum. A Model Implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1993.

[10] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[11] L. Iftode, J. Singh, and K. Li. Scope Consistency: a Bridge between Release Consistency and Entry Consistency. In *Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[12] K. Johnson, M. Kaashoek, and D. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 213–228, Dec. 1995.

[13] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[14] M. Lemke and D. Quinlan. P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications. In *Proc. CONPAR 92–VAPP V*. Springer-Verlag, September 1992.

[15] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. 1988 International Conference on Parallel Processing*, volume II, pages 94–101, Aug. 1988.

[16] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19:1079–1103, 1993.

[17] H. Lu, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Message-Passing vs. Distributed Shared Memory on Networks of Workstations. In *Proc. of Supercomputing'95*, December 1995.

[18] W. O'Farrell, F. Eigler, S. Pullara, and G. Wilson. ABC++. In G. V. Wilson and P. Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[19] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. 21st International Symposium on Computer Architecture*, pages 325–336, Apr. 1994.

[20] H. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions Approach to Software Cache Coherence. In *Proc. Symposium on Principles and Practices of Parallel Programming*, pages 229–238, May 1993.

[21] D. Scales and M. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proc. 1st Symposium on Operating Systems Design and Implementation*, pages 101–114, Nov. 1994.

[22] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.