Optimising Shared Reduction Variables in MPI Programs

A.J. Field, P.H.J. Kelly and T.L. Hansen

Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, U.K.
{ajf,phjk,tlh}@doc.ic.ac.uk

Abstract. CFL (Communication Fusion Library) is an experimental C++ library which supports shared reduction variables in MPI programs. It uses overloading to distinguish private variables from replicated, shared variables, and automatically introduces MPI communication to keep replicated data consistent. This paper concerns a simple but surprisingly effective technique which improves performance substantially: CFL operators are executed lazily in order to expose opportunities for run-time, context-dependent, optimisation such as message aggregation and operator fusion. We evaluate the idea using both toy benchmarks and a 'production' code for simulating plankton population dynamics in the upper ocean. The results demonstrate the software engineering benefits that accrue from the use of the library and show that performance close to that of manually optimised code can be achieved automatically in many cases.

1 Introduction

In this paper we describe an experimental abstract data type for representing shared variables in SPMD-style MPI programs. The operators of the abstract data type have a simple and intuitive semantics and hide any required communication. Although there are some interesting issues in the design of the library, the main contribution of this paper is to show how lazy evaluation can expose run-time optimisations that may be difficult, or even impossible, to spot using conventional compile-time analysis. The paper makes the following contributions:

- We present a simple and remarkably useful prototype class library which simplifies certain kinds of SPMD MPI programs
- We discuss some of the design issues in such a library, in particular the interpretation of the associated operators
- We show how lazy evaluation of the communication needed to keep replicated variables consistent can lead to substantial performance advantages
- We evaluate the work using both toy examples and a large-scale application

This paper extends our brief earlier paper [2] in providing better motivation and further experimental results, as well as a more thorough description of the technique.

```
double s1, s2 ;

void sum( double& data ) {
  double s = 0.0 ;
  for ( j=jmin; j<=jmax; j++ ) {
    s += data[j] ;
  }
  MPI_Allreduce(&s,&s1,1,MPI_SUM,..) ;
}

void sumsq( double& data ) {
  double s = 0.0 ;
  for ( j=jmin; j<=jmax; j++ ) {
    s += data[j] * data[j] ;
  }
  MPI_Allreduce(&s,&s2,1,MPI_SUM,..) ;
}

for( i=0; i<N; i++ ) {
  sum(a[i]) ;
  sumsq(a[i]) ;
  var[i] = (s2 - s1*s1/N)/(N-1) ;
}
```

```
CFL_Double s1(0), s2(0) ;
/* Note: Initial values assumed
   consistent across procs. */

void sum( double& data ) {
  s1 = 0.0 ;
  for ( j=jmin; j<=jmax; j++ ) {
    s1 += data[j] ;
  }
}

void sumsq( double& data ) {
  s2 = 0.0 ;
  for ( j=jmin; j<=jmax; j++ ) {
    s2 += data[j] * data[j] ;
  }
}

for( i=0; i<N; i++ ) {
  sum(a[i]) ;
  sumsq(a[i]) ;
  var[i] = (s2 - s1*s1/N)/(N-1) ;
}
```

Fig. 1. Variance calculation using MPI (left) and CFL (right).

2 The Idea

Figure 1 illustrates the basic idea. This is a toy C++ application which computes the sample variance of N batches of M data items, stored in an $N \times M$ array. The data is replicated over P processors and each processor computes its contribution to the sum and sum-of-squares of each batch of data using appropriately defined methods. An MPI reduction operation sums these contributions. The main loop fills the variance array (`var`).

This program suffers two drawbacks. Firstly, the code is convoluted by the need to code the communication explicitly—an artefact of all MPI programs. Secondly, it misses an optimisation opportunity: the two reduction operations can be fused (i.e. resolved using a single communication to sum the contributions to `s1` and `s2` at the same time) since the evaluation of `sumsq` does not depend on that of `sum`. If the two reduction operations are brought out of the methods `sum` and `sumsq` and combined into a single reduction over a two-element vector in the outer loop a performance benefit of around 43% is achieved using four 300MHz UltraSparc processors of a Fujitsu AP3000 with `N=M=3000`. Further results for this benchmark are reported in Section 5.

Spotting this type of optimisation at compile time requires analysing across method boundaries. While perfectly feasible in this case, in general these operations may occur deep in the call graph, and may be conditionally executed, making static optimisation difficult. The alternative we explore in this paper is to attempt the optimisation at run-time, requiring no specialist compiler support.

We have developed a prototype library called CFL (Communication Fusion Library) designed to support shared reduction variables. The library can be freely mixed with standard MPI operations in a SPMD application. C++ operator overloading is used to simplify the API by using existing operators (e.g. `+, *, +=` etc.). Where an operation would normally require communication e.g. when a shared reduction variable is updated with the value of a variable local to each processor, the communication is handled automatically.

Figure 1 shows how the CFL library can be used to model the shared quantities `s1` and `s2` in the previous example. This eliminates all the explicit communication, in the spirit of shared-memory programming. However, the main benefit comes from CFL's lazy evaluation: just prior to the assignment to `var[i]` no communication has yet taken place. The assignment forces both delayed communications and resolves them using a single reduction operation, akin to the manual optimisation outlined above. There are some overheads associated with the maintenance of these shared variables, so we would not expect to achieve the performance of the manually optimised code. Nonetheless, this very simple example, with scope for just two fusions per iteration, yields a performance improvement of around 37% when compared to the original code on the same platform. Again more detailed results are presented in Section 5.

In the remainder of this paper we present some relevant background to the work (Section 3), discuss the semantics of shared variables in the context of MPI programs (Section 4) and present some performance benchmarks for both contrived test programs and a production oceanography simulation (Section 5). The conclusions are presented in Section 6.

3 Related work

The idea of delaying execution in order to expose optimisation opportunities has appeared before. POOMA [4] uses expression templates in C++ to support explicit construction and then evaluation of expressions involving arrays and communication. A delayed-evaluation self-optimising (DESO) numerical library for a distributed memory parallel computer is described in [7]. By delaying the evaluation of operations, the library is able to capture the data-flow graph of the computation. Knowing how each value is to be used, the library is able to calculate an optimised execution plan by propagating data placement constraints backwards through the DAG. This means that the library is able to calculate a very efficient initial distribution for the data across the processors, and hence fewer redistributions of the data will be necessary.

A related idea, which is exploited in BSP [3] and KeLP [10], is to organise communication in a global collective operation. This allows multiple small

messages to be aggregated, and also provides the opportunity to schedule communication to use avoid network and buffer contention.

A shared-memory programming model can be supported on distributed-memory hardware using a page-based consistency protocol; sophisticated implementations such as TreadMarks [5] support some run-time adaptation, for example for pages with multiple writers. However, Treadmarks offers no special support for reductions.

4 Shared variables in SPMD programs

In a data-parallel SPMD program, a large data structure is distributed across each processor, and MPI is used to copy data to where it is needed. In contrast, we focus in this paper on the program's global state variables. In a distributed-memory implementation, each processor holds its own copy of each shared variable. When the variable is updated, communication is needed to ensure that each processor's copy is up to date.

In the context of this paper we focus exclusively on scalar double-precision floating-point variables. The semantics of arithmetic operations on private variables are very well-understood, but are not so straightforward for shared variables as an operation on a shared double will be executed on several processors.

The interesting case concerns the assignment of the result of an arithmetic expression to a variable. In what follows, `x,y` and `z` will refer to (global) shared variables (i.e. of type `CFL_Double`) and `a, b` to local variables private to each processor. Each processor maintains a local copy of each shared variable and the library must ensure that after each operation these copies are consistent.

If the target variable of an assigment is local, as in `a = x - b` then the assignment can be performed concurrently on each processor without (additional) communication. However, if the result is stored in a shared variable then the behaviour depends on the operator arguments. If both operator arguments are shared, as in `x = y * z` then again the assignment can be effected locally. However, if one of the arguments is local and the other shared, as in `x += a` or `x = y + a`, then our interpretation is that each processor contributes its own update to `x`, implying a global reduction operation, with the rule that `x -= a` is interpreted as `x += (-a)`. Because CFL is lazy, one or more of the shared variables on the right-hand side of an assignment may already contain a pending communication, either from an earlier assignment or an intermediate expression on the same right-hand side, as in `x = y + a - z`. Any new required communication is simply added to those currently pending.

Similar rules apply to the other operators `-, *, /` etc. and combined operations like `+=` have the same meaning as their expanded equivalents, e.g. `x += a` and `x = x + a`.

Assignment and reduction Note that the way the assignment `v=e` is implemented now depends on the nature of `v` and `e`. It is tempting to think that any potential confusion can be overcome by using a different operator symbol when a global

reduction is intended, for instance `x ++= a` instead of `x += a`. However the assignment `x = x + a` should have the same meaning so we would also need special versions of `+` (and the other operators) to cover all combinations of argument types. We thus choose to stick to the familiar symbols using the overloading, but propose the use of naming conventions to distinguish shared from local variables where any confusion may arise.

An attempt to assign a local variable to a shared variable either directly (e.g. `x = a`) or as a result of a calculation involving only local variables (e.g. `x = a - b`) is disallowed.

4.1 Delaying communication

The parallel interpretation of some operator uses such as `x += a` means that at any point a shared variable may need to synchronise with the other processors. Because each processor sees the variable in the same state every processor will know that the variable needs synchronisation. Moreover, as operations are executed in the same order on all the processors (the SPMD model), shared variables will acquire the need for synchronisation in the same order on every processor. This means that, in order to delay communication, we need only maintain a list of all the variables that need to be synchronised, and in what way. When communication is forced (see below) these synchronisations are piggybacked onto a single message with an associated reduction operator. An alternative would be to initiate a non-blocking communication instead; although this might be appropriate for some hardware, little or no computation/communication overlap is possible in most current MPI implementations.

An assignment of a shared variable to a local variable constitutes a *force point*. At this point a communications manager marshalls all CFL variable updates into a single array and performs a single global reduction operation over that array. On completion, the resulting values are used to update all `CFL_Doubles` which were previously pending communication.

In principle, the synchronisation of a shared variable may be delayed until its global value is required (force point), but in practice the synchronisation may be forced earlier than this, e.g. when another shared variable synchronisation is forced before it. Forcing *any* delayed synchronisation will force *all* such synchronisations.

Limitations In the prototype implementation of CFL only 'additive' operators (`+, -, +=, -=`) are handled lazily at present. This is sufficient for experimental evaluation of the basic idea. The other operators (and the copy constructor) are all supported but they force all pending communication. Implementing the remaining operators lazily requires a little more work to pack the data for communication and to construct the associated composite reduction operation, but is otherwise straightforward. This is left as future work.

N	AP3000 (P=4) Execution time(s)		
	Original	Hand optimised	CFL
500	0.341	0.157 (53.9%)	0.192 (43.6%)
1000	0.748	0.347 (53.5%)	0.433 (42.0%)
1500	1.159	0.574 (50.5%)	0.724 (37.5%)
3000	2.544	1.463 (42.5%)	2.119 (16.7%)

Table 1. AP3000 execution times (in seconds) for Figure 1 for various problem sizes, with percentage speedup relative to the original, unoptimised code.

5 Evaluation

Our performance results are from dedicated runs on three platforms: a Fujitsu AP3000 (80 nodes, each a 300MHz Sparc Ultra II processor with 128 RAM, with Fujitsu's 200MB/s AP-Net network), a Quadrics/COMPAQ cluster (16 nodes, each a Compaq DS20 dual 667MHz Alpha with 1GB RAM), and a cluster of dual 400MHz Celeron PCs with 128MB RAM on 100Mb/s switched Ethernet. In each case there was one MPI process per node.

5.1 Toy Benchmark

Table 1 shows the execution times for the toy benchmark of Figure 1 for four problem sizes for the AP3000 platform using 4 processors. Here the data matrix is assumed to be square, so the problem size defines both M and N. The figures in parentheses show the reduction in execution time, relative to the original unoptimised code. The results show that a significant performance improvement can be achieved by fusing the communications, even though only two such communications can be fused at any time. The results also show, as one would expect, diminishing returns for the CFL library, relative to the hand-optimised code, as the problem size increases. This is because larger problems incur a smaller communication overhead, so the overhead of maintaining the shared variable state takes greater effect.

We would intuitively expect the performance of the CFL library to improve, relative to the hand-optimised code, for platforms with slower communication networks (measured by a combination of start-up cost and bandwidth) and vice versa. This is borne out by Table 2 which shows the performance of the same benchmark on our three reference platforms, using 4 processors in each case and with a problem size of 3000. Relative to the hand-optimised code, the CFL library performs extremely well on the PC cluster. However, on the COMPAQ platform, which has a very fast communication network, the overheads of supporting lazy evaluation outweigh the benefits of communication fusion.

The example of Figure 1 enables exactly two reductions to be fused on each iteration of the loop. In some applications (see below, for example) it may be possible to do better. The variance example was therefore generalised artificially

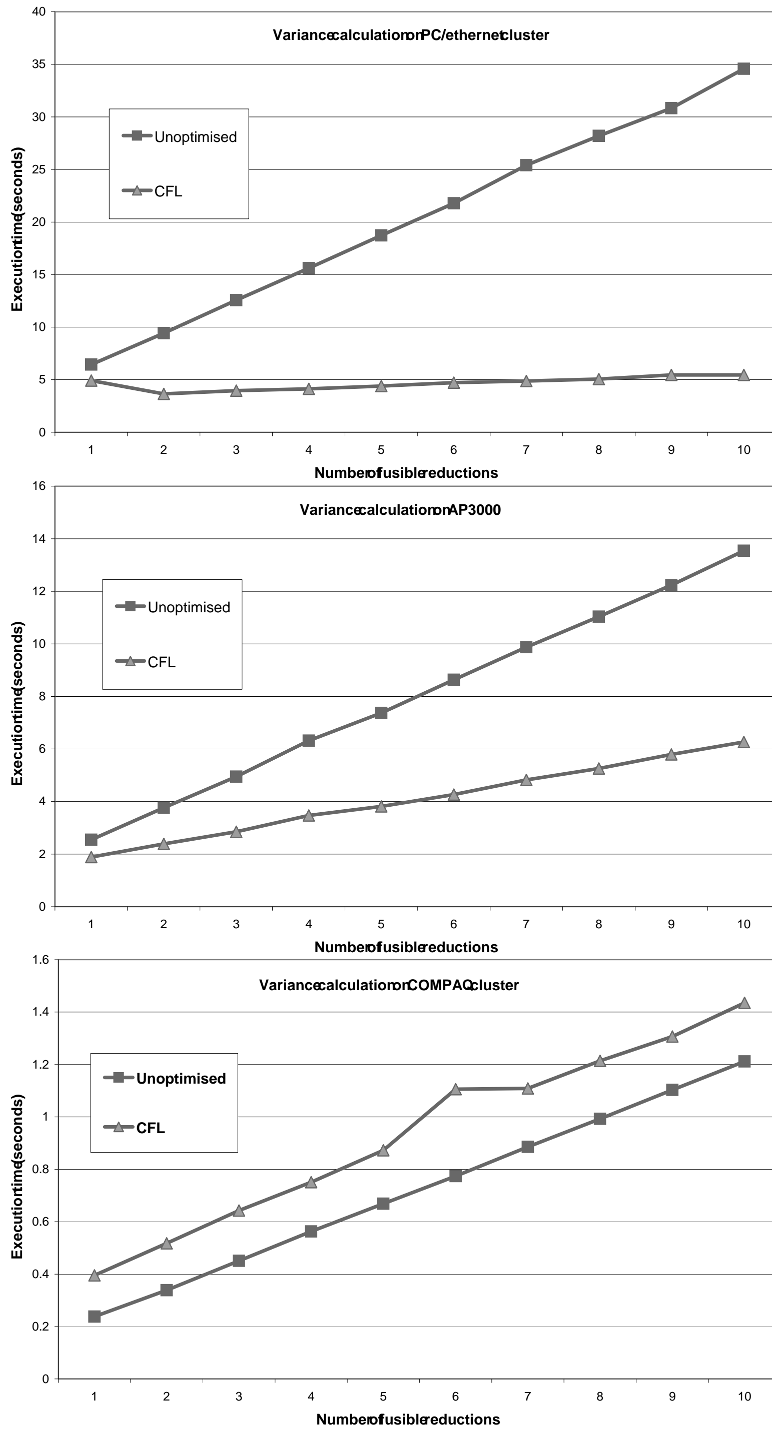

Fig. 2. Variance calculation (3000×3000 array) on 4 processors: performance of original MPI code versus CFL library.

Platform	Execution time(s) (N=3000)		
	Original	Hand optimised	CFL
AP3000	2.544	1.463 (42.5%)	2.119 (16.7%)
Cluster	7.154	3.670 (48.7%)	3.968 (44.5%)
COMPAQ	0.263	0.161 (38.9%)	0.418 (-58.8%)

Table 2. Execution times for Figure 1 for various platforms.

by introducing an extra loop that called the `sum` function (only) a given number of times, n, on each iteration of the outer (`i`) loop. The results were stored in an array and later summed (again arbitrary, but this has the effect of forcing communication in the CFL case). The objective was to measure the cost of performing repeated (explicit) MPI reduction operations relative to the cost of fusing them within CFL. The results for 4 processors on each platform with $N = 3000$ are shown in Figure 2. Note that the slope of the two curves (original MPI vs. CFL) in each case expose these relative costs and we can see why CFL wins out on both the AP3000 and PC cluster. Conversely, on the COMPAQ platform no amount of fusion opportunity can buy back the performance overheads inherent in the current CFL implementation.

5.2 Oceanography Simulation

We now present the results obtained when the CFL library was used to model shared variables in a large-scale simulation of plankton population dynamics in the upper ocean using the Lagrangian Ensemble method [9]. Some discussion of the code structure is in order.

The simulation is based on a one-dimensional water column which is stratified into 500 layers each 1m deep. The plankton are grouped into *particles* each of which represents a sub-population of identical individuals. The particles move by a combination of turbulence and sinking/swimming and interact with their local environment according to rules derived from laboratory observation.

The simulation is built by composing modules each of which models an aspect of the physics, biology or chemistry. The exact configuration may vary from one simulation to the next. To give a flavour for the structure of a typical code, the dominant (computationally speaking) component of a particular instance called "ZB" models phytoplankton by the sequential composition of four modules: `Move()`, `Energy()`, `Nutrients()` and `Evolve()` (motion, photosynthesis, nutrient uptake and birth/death). A similar structure exists for zooplankton. The model essentially involves calling these (and the many other) modules in the specified order once per time-step.

In parallelising the model a vertical partitioning strategy is used to divide the plankton particles among the available processors. The processors cooperate through *environment variables* which represent the chemical, physical and biological attributes of each layer. The parallelisation strategy requires that each processor sees the same global environment at all times.

Procs	Execution time(s)		
	Unoptimised	Hand-optimised	Using CFL
1	3721	3721 (0%)	3738 (-0.5%)
2	1805	1779 (1.5%)	1790 (0.8%)
4	934	869 (7.5%)	866 (7.9%)
8	491	433 (13.4%)	418 (17.5%)
16	317	244 (29.9%)	257 (23.3%)
32	292	191 (52.9%)	182 (60.4%)

Table 3. Execution times for the plankton code (320,000 particles). In brackets we show the speedup relative to the unoptimised implementation.

The various modules have been developed independently of the others, although they must fit into a common framework of global variables, management structures etc. Within these modules there are frequent updates to the shared variables of the framework and it is common for these to be assigned in one module and used in a later one. This relatively large distance between the producer and consumer provides good scope for message aggregation. However, manual optimisation will work only for that particular sequence of modules: adding a new module or changing the order of existing modules changes the data dependency. This is where the CFL library is particularly beneficial: it will automatically fuse the maximum possible number of reduction operations (i.e. those that arise between force points).

We began with the original (parallel) version of ZB and then hand-optimised it by manually working out the data dependencies between the global shared quantities and identifying force points. The fusion was actually achieved by building a lazy version of `MPI_All_Reduce` [1]. This simplified the implementation significantly but introduced some overheads, very similar in fact to those in CFL. The MPI code was then rewritten using the CFL library, simply by marking the shared environment variables as `CFL_doubles`. The original code uses exclusively MPI reduction operations so the immediate effect of using CFL is to remove *all* explicit communication from the program. The effect of the message aggregation (both manual and using CFL) is to reduce the number of synchronisations from 27 to just 3 in each time step. In one of these no less than 11 reduction operations were successfully fused between force points.

AP3000 timing results for the execution of the ZB model before and after CFL was incorporated are shown in Table 3 for a problem size of 320,000 particles. Both hand-optimised and CFL versions of the model have very similar performance but this is not surprising given the way the hand-optimisation was done.

Remarks In order to use the CFL library in this case study, we had to turn off one feature of the model. During nutrient uptake the required reduction operation is actually *bounded* in the sense that the `-=` operator would not normally

allow the (shared) nutrient variable to become negative; instead the uptake would be reduced so as to exactly deplete the nutrient. It is perfectly possible to build such bounded reductions into CFL but they are not currently supported.

6 Conclusions

This paper presents a simple idea, which works remarkably well in many cases. We have built a small experimental library on top of MPI which enables shared scalar variables in parallel SPMD-style programs to be represented as an abstract data type. By implementing the library in C++ and using C++'s operator overloading, the familiar arithmetic operator symbols, such as `+`, `-`, `*=` etc. can be used on shared variables. Some operators have a parallel reading when the target of an assignment is another shared variable. Because the operations are abstract, dynamic run-time optimisations can be built into their implementation. We have shown how delayed evaluation can be used to piggyback communications on top of earlier, as yet unevaluated, parallel operations. This means that the communication associated with a global reduction, for example, may actually take place as a side-effect of another reduction operation in a different part of the code. This avoids reliance on sophisticated compile-time analyses and can exploit opportunities which arise from dynamic data dependencies. Using a contrived test program and a realistic case study we have demonstrated very pleasing performance improvements on some platforms. Unsurprisingly, the greatest performance benefits are seen on platforms with slower communication networks.

In essence, what we have done is to implement an application-specific cache coherence protocol, in the spirit of, among others, [11]. This hides consistency issues, and the associated communication, from the programmer, with obvious benefits in software engineering terms.

Could we achieve the reduction fusion optimisation by executing standard MPI functions lazily? Not without compiler support, since the results of the MPI operation are delivered to normal private data so we don't know when to force communication.

The library is currently very much a prototype. Nonetheless, the current implementation is robust and has proven to be of surprising utility, both in performance and ease of use. We are now seeking to extend the library (for example to handle arrays as well as scalars) and to focus on internal optimisations to reduce management overheads.

References

1. S. H. M. Al-Battran: *Simulation of Plankton Ecology Using the Fujitsu AP3000*, MSc thesis, Imperial College, September 1998
2. A J Field, T L Hansen and P H J Kelly: *Run-time fusion of MPI calls in a parallel C++ library.* Poster paper at LCPC2000, The 13th International Workshop on Languages and Compilers for High-Performance Computing, Yorktown Heights, August 2000.

3. J.M.D. Hill, D.B. Skillicorn: *Lessons learned from implementing BSP*. Journal of Future Generation Computer Systems, Vol 13, No 4–5, pp. 327-335, March 1998.
4. Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William J. Humphrey, John Reynders, Stephen Smith, Timothy Williams: *Array Design and Expression Evaluation in POOMA II.* ISCOPE'98 pp.231-238. Springer LNCS 1505 (1998).
5. P. Keleher, A. L. Cox, S. Dwarkadas, W. Zwaenepoel: *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.* In Proceedings of the 1994 Winter Usenix Conference, pp. 115-131, January 1994
6. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennesey: *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.* In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 15-26, May 1990
7. O. Beckmann, P. H. J. Kelly: *Efficient Interprocedural Data Placement Optimisation in a Parallel Library*, LCR '98
8. O.B. Beckmann and P.H.J. Kelly, *A Linear Algebra Formulation for Optimising Replication in Data Parallel Programs.* In LCPC'99, Springer Verlag (2000).
9. J. Woods and W. Barkmann, *Simulation Plankton Ecosystems using the Lagrangian Ensemble Method*, Philosophical Transactions of the Royal Society, B343, pp. 27-31.
10. S.J. Fink, S.B. Baden and S.R. Kohn, *Efficient Run-time Support for Irregular Black-Structured Applications*, Journal of Parallel and Distributed Programming, Vol 50, No. 1, pp 61–82, 1998.
11. Andrew J. Bennett and Paul H. J. Kelly, *Efficient shared-memory support for parallel graph reduction.* Future Generation Computer Systems, V.12 No.6 pp.481–503 (1997).