Efficient Parallelization of Unstructured Reductions on Shared Memory Parallel Architectures*

Siegfried Benkner[1] and Thomas Brandes[2]

[1] Institute for Software Technology and Parallel Systems
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria
sigi@ieee.org
[2] Institute for Algorithms and Scientific Computing (SCAI)
German National Research Center for Information Technology (GMD)
Schloß Birlinghoven, D-53754 St. Augustin, Germany
brandes@gmd.de

Abstract. This paper presents a new parallelization method for an efficient implementation of unstructured array reductions on shared memory parallel machines with OpenMP. This method is strongly related to parallelization techniques for irregular reductions on distributed memory machines as employed in the context of High Performance Fortran. By exploiting data locality, synchronization is minimized without introducing severe memory or computational overheads as observed with most existing shared memory parallelization techniques.

1 Introduction

Reduction operations on unstructured meshes or sparse matrices account for a large fraction of the total computational costs in many advanced scientific and engineering applications. Typical examples of such applications include crash simulations, fluid-dynamics codes, weather forecasting models, electromagnetic problem modeling, and many others. In order to fully exploit the potential of parallel computers with such applications using high-level parallel languages like OpenMP [11] or HPF [8], it is of paramount importance to apply efficient parallelization strategies to reduction operations performed on irregular data structures. Unstructured reductions are usually implemented by means of loops containing indirect (vector-subscripted) array accesses. When parallelizing such loops it is crucial to avoid high synchronization or serious communication overheads, respectively. Several different parallelization techniques targeted either for distributed or for shared memory parallel architectures have been described in the literature [10, 2, 7, 1] and have been integrated in parallelizing compilers.

In this paper we present a new parallelization method for unstructured array reductions on shared memory parallel computers and compare it to existing parallelization strategies. Without loss of generality we discuss these techniques in the context of finite-element methods (FEM). Section 2 describes existing shared memory parallelization techniques for unstructured reductions and their support in OpenMP. Section 3 briefly discusses parallelization for distributed

* The work described in this paper was supported by NEC Europe Ltd. as part of the ADVICE project in cooperation with the NEC C&C Research Laboratories and by the Special Research Program SFB F011 AURORA of the Austrian Science Fund. Published in: Workshop Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2000.

memory machines and high-level language support for unstructured reductions as provided in HPF. Section 4 presents an optimized parallelization method for unstructured array reductions on shared memory machines. This method is strongly related to an efficient handling of irregular reductions on distributed memory machines [1]. By exploiting data locality, synchronization is minimized without introducing severe memory or computational overheads as observed with most existing shared memory parallelization techniques. We have implemented this parallelization method in a compilation system [4] that translates high-level data parallel programs into shared memory parallel programs utilizing OpenMP to realize thread parallelism and synchronization. Performance results presented in Section 5 for a typical FEM application kernel verify the effectiveness of our approach and its superiority compared to existing techniques.

Figure 1 shows an example of an unstructured reduction operation on a finite-element mesh. The mesh consists of `NELEMS` elements and `NNODES` nodes, whereby each element comprises four nodes. The arrays `NODE` and `ELEM` store certain physical quantities[1], e.g. positions for each node or forces for each element, respectively. The integer array `IX` captures the connectivity of the mesh, i.e. the array section `IX(1:4,I)` contains the four node numbers of element `I`.

```
integer, parameter :: NNODES=...,NELEMS=...
real,    dimension (NNODES)    :: NODE
real,    dimension (NELEMS)    :: ELEM
integer, dimension (4, NELEMS) :: IX

! unstructured reduction loop
do I = 1, NELEMS
   VAL = Work(ELEM(I))
   do K = 1, 4
      NODE(IX(K,I)) = NODE(IX(K,I)) + VAL
   end do
end do
```

```
      real, dimension (NNODES) :: NTMP
          ...
!$omp parallel, private (VAL, K, NTMP)
      NTMP = 0.0
!$omp do
      do I = 1, NELEMS
         VAL =  Work(ELEM(I))
         do K = 1, 4
           NTMP(IX(K,I)) = NTMP(IX(K,I)) + VAL
         end do
      end do
!$omp critical
      NODE = NODE + NTMP
!$omp end critical
!$omp end parallel
```

Fig. 1. Unstructured reduction on a finite-element mesh: sequential version (left) and OpenMP version (right).

In each iteration of the unstructured reduction loop shown in Figure 1 a value `VAL` (e.g. an elemental force) is computed for an element of the mesh by means of a function (`Work`) and added to all nodes comprising this element. Since the addition is assumed to be an associative and commutative operation, the order in which the loop iterations are executed does not change the final result (except for possible round-off errors). However, when parallelizing the loop it has to be taken into account that different loop iterations may update the same node. This is caused by the fact that neighboring elements of the mesh may share one or more nodes, and thus `IX(:,I)` may have the same values for different iterations. If the reduction loop is parallelized on a shared memory machine by partitioning the loop iterations among concurrent threads, synchronization will be required to ensure that two distinct threads do not update one node at the same time. On

[1] To simplify presentation, only one physical quantity is stored per node and element.

distributed memory machines, where the mesh is to be partitioned with respect to the local memories of the processors, communication will be required for nodes at processor boundaries.

2 Unstructured Reductions on Shared Memory Machines

In this section we briefly discuss existing techniques for parallelizing unstructured reductions on shared memory machines by means of thread parallelism, as for example offered by OpenMP.

Array Privatization The central idea of the array privatization technique is that every processor gets an own private copy of the reduction array and performs its part of the loop iterations independently of other processors. Subsequently, the private results of all processors are combined to yield the final result. In order to ensure correct results, this ultimate step requires synchronization. Since the reduction clause of OpenMP [11] supports only reductions on scalar variables, parallelization of array reductions based on array expansion has to be programmed explicitly as shown in the right hand side of Figure 1. The original loop is enclosed in a `parallel` region and the temporary array `NODE_TMP` is declared as `private` to enforce that each thread gets its own copy. The `I`-loop is parallelized by relying on the default work sharing mechanism of OpenMP. As a consequence, a chunk of loop iterations is assigned to each thread, and all threads execute their chunk of iterations independently of each other. After parallel execution of the loop, the array assignment statement, protected by a `critical` section, ensures that each thread adds its local result of the reduction operation stored in `NODE_TMP` to the shared array `NODE`.

Parallelizing compilers for shared memory architectures like POLARIS [2] or SUIF [7], automatically translate sequential reduction loops as outlined in Figure 1 but utilize a thread library instead of OpenMP to implement multithreading. Array privatization is a good solution for reductions on scalar variables and for small arrays. For our FEM example, `NNODES << NELEMS` should be fulfilled in order to keep both memory and execution overhead for the final array assignment reasonably small.

Array Expansion The array expansion technique splits the reduction loop into two separate loops, whereby the first loop only evaluates the function `Work` for each element of the mesh. In order to store the results of `Work` for each element, the variable `VAL` has to be expanded into an array of size `NELEMS`. The second loop solely performs the reduction operation by reading the expanded `VAL` variable. The first part is executed in parallel without requiring any synchronization, whereas the second part has to be executed serially by the master thread only in order to avoid synchronization.

In our example, array expansion is a feasible solution only if `NNODES >> NELEMS`, since the memory overhead and the execution time for the serial part of the computation increase with the number of elements.

Atomic Reductions OpenMP provides the atomic directive for enforcing the atomic updating of a specific memory location, rather than exposing it to multiple, simultaneously writing threads. Using this directive, the reduction loop of

Figure 1 can be parallelized by means of a `parallel do` directive and by inserting an `atomic` directive prior to the assignment of `NODE`. This technique does not require any additional memory but may cause high synchronization overheads.

Although the atomic directive may be replaced by a `critical` directive, the atomic directive permits better optimizations. In contrast to a critical section, more than one thread may execute the assignment at the same time as long as different memory locations are updated.

3 Unstructured Reductions on Distributed Memory Machines

Parallelization of unstructured reductions on distributed memory machines is more complex than on shared memory machines. Since a distributed memory architecture provides no global address space, the reduction arrays have to be distributed to the local memories of the processors and accesses to array elements on other processors require communication. As a consequence of the indirect array accesses, analysis of array access patterns, which is a pre-requisite for communication generation, cannot be performed at compile time and, therefore, runtime parallelization techniques have to be applied. In the following the parallelization of unstructured array reductions for distributed memory machines is discussed in the context of High Performance Fortran (HPF) [8].

```
!hpf$ distribute(block) ::  ELEM, NODE
!hpf$ align IX(*,i) with ELEM(i)
      ...
!hpf$ independent, on home (ELEM(I)), new(VAL, K), reduction(NODE)
      do I = 1, NELEMS
         VAL =  Work(ELEM(I))
         do K = 1, 4
            NODE(IX(K,I)) = NODE(IX(K,I)) + VAL
         end do
      end do
```

Fig. 2. Unstructured Reductions in High Performance Fortran.

HPF provides high-level directives for specifying the distribution of arrays to abstract processors according to various formats (`block, cyclic`, etc.). The `independent` directive may be used to assert that a loop does not contain loop-carried dependences and therefore may be parallelized. In this context, temporary variables that are (conceptually) private for each loop iteration may be specified by means of a `new` clause. Moreover, a `reduction` clause may be used to indicate that dependences caused by associative and commutative reduction operations can be ignored. As opposed to OpenMP, in HPF also array variables may appear within a reduction clause. With the `on home` clause the loop iteration space may be partitioned according to the distribution of an array.

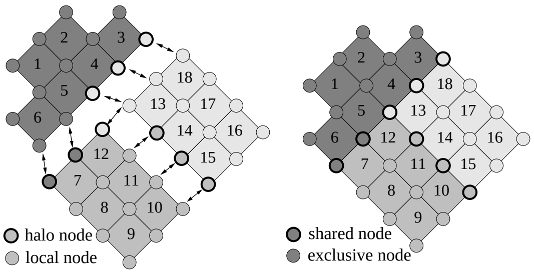
Inspector/Executor Parallelization Technique In order to parallelize the code shown in Figure 2, an HPF compiler distributes the arrays `ELEM, NODE`, and `IX` as specified by the distribution and alignment directives to the local memories of the processors. Parallel execution of the reduction loop is usually performed in two phases based on the inspector/executor strategy [10]. During the inspector phase each processor determines for its share of iterations the set of non-local elements of `NODE` it needs to access and derives the corresponding communication schedules (i.e. gather/scatter schedules). In the executor phase, on each processor non-local data to be read are gathered from the respective owner processors according to the gather-schedules by means of message-passing communication and are stored in local buffers. This is followed by a local computation phase where each processor executes, independently of the other processors, its share of loop iterations on its local part of the `NODE` array or the local buffers, respectively. Finally, a global communication phase based on the scatter-schedules takes place combining all those elements of `NODE` that have been written by processors not owning them. Since the inspector phase may be very time-consuming, it is essential to amortize the preprocessing overhead over multiple executions of a loop by reusing communication schedules [9, 1, 3] as long as communication patterns do not change. This is possible since unstructured reductions are performed in many codes within a serial time-step loop and the communication patterns are invariant for all (or at least many) time-steps.

Some HPF compilers also employ alternative strategies akin to array privatization or array expansion as discussed in Section 2. However, these techniques usually exhibit a larger memory and/or communication overhead.

Non-Local Access Patterns The main task of the inspector phase when preprocessing a loop with irregular array accesses (vector subscripts) is to determine, on each processor, the set of non-local array elements to be accessed. Once the non-local access pattern has been determined, the required communication can be derived. Recently, the concept of halos [1] has been proposed for HPF, enabling the explicit specification of non local data access patterns for distributed arrays. A *halo*, which in its simplest form comprises a list of global indices, specifies the set of non-local elements to be accessed at runtime for each abstract processor participating in the execution of an HPF program. The information provided by a halo significantly reduces the overheads of the inspector phase and alleviates computation and reuse of communication schedules. Figure 3 sketches a mesh partitioned in a node-based way, and the corresponding halo describing the set of non-local nodes to be accessed on each processor. By making the required communication explicit, the size of the halo area provides an appropriate measure for data locality. In the next section we show how the concepts of data distribution and halos can be utilized in order to parallelize irregular reductions efficiently for shared memory parallel architectures.

4 Exclusive Ownership Technique

In this section we present a new parallelization method for unstructured reductions on shared memory machines. This strategy is an extension of the atomic reduction technique described in Section 2 that avails itself with the concepts of data distribution and halos in order to minimize synchronization overheads. It can be employed for compiling an HPF program for multithreaded execution on shared memory parallel computers or for parallelizing irregular array reductions with OpenMP directly. We outline this technique for the FEM reduction loop shown in Figure 1.




```
!$omp parallel, private (K, VAL, N)
!$omp do
      do I = 1, NELEMS
         VAL = Work(ELEM(I))
         do K = 1, 4
           N = IX(K,I)
           if (EXCLUSIVE(N)) then
              NODE(N) = NODE(N) + VAL
           else
!$omp atomic
              NODE(N) = NODE(N) + VAL
           end if
         end do
      end do
```

Fig. 3. Distributed mesh and halo (left); virtually distributed mesh with exclusive ownership (middle); OpenMP code based on exclusive ownership (right).

As a starting point, a *virtual* distribution is determined for the arrays `ELEM`, `NODE` and `IX` with respect to abstract processors. Data distribution is referred to as virtual, since the arrays are not actually distributed but allocated in an unpartitioned manner in shared memory. Based on the virtual distribution, each array element is associated with a unique abstract processor which becomes the *owner* of this element. Ownership is then used to determine the work sharing for the reduction loop with respect to abstract processors, whereby each abstract processor will be implemented by a separate thread. In our example, the loop iteration space is partitioned such that each iteration `I` is assigned to the abstract processor owning `ELEM(I)` of the mesh. Assuming a block distribution for `ELEM`, the loop iteration space can be partitioned by relying on the standard work sharing mechanism of OpenMP. In order to minimize synchronization overheads for the assignment to `NODE`, we introduce, based on halos, the concept of *exclusive ownership*. An element of array `NODE` is exclusively owned by an abstract processor (thread) if it is owned by that processor and not contained in the halo of any other processor. Synchronization via atomic updates is necessary only for loop iterations that access nodes not exclusively owned by the executing thread (shared nodes), while exclusively owned nodes can be handled like private data requiring no synchronization.

In Figure 3 the halo and exclusive ownership information is shown for a simple mesh together with the resulting OpenMP code. In the code, exclusive ownership information is represented by means of a logical array `EXCLUSIVE` which can be easily derived from the halo of the array `NODE`. We assume that the halo is either supplied by a domain partitioning tool or explicitly computed before the reduction loop is executed by analyzing the indirection array `IX`. In the latter case, the analysis of the indirection array is similar to an inspector phase as applied in the context of distributed memory parallelization, yet much simpler, since due to the shared address space, no communication is required. Exclusive ownership information can be reused employing techniques for communication schedule reuse [9, 1, 3], as long as the indirection array is not changed.

Gutiérrez et al. [6] presented a parallelization method for irregular reductions on shared memory machines that exploits locality similar to our method. The iteration space of a reduction loop is partitioned among threads in such a way that conflict-free writing on the reduction array is guaranteed and no synchronization is required. For this purpose, *loop index prefetching* arrays are built before the loop is executed by employing techniques similar to those applied in an inspector/executor strategy. The construction of the loop-index prefetching arrays becomes very complex and the algorithms presented in [6] work only for one or two reductions within one loop iteration. As opposed to our technique, some loop iterations have to be executed by more than one processor, since more than one element of the reduction array may have to be updated during each iteration. In the context of the FEM example presented previously, all iterations that manipulate elements on the distribution boundary of the mesh would have to be executed by all threads that own a node of this element. As a consequence, redundant computations are introduced, with an overhead depending on the computational costs of the function `Work`.

5 Performance Results

For the evaluation of the exclusive ownership technique we used a kernel from an industrial crash simulation code [5]. The kernel is based on a time-marching scheme to perform stress-strain calculations on a finite-element mesh consisting of 4-node shell elements. In each time-step elemental forces are calculated for every element of the mesh (cf. function `Work`) and added back to the forces stored at nodes by means of unstructured reduction operations. Besides the computation of elemental forces, the unstructured reduction operations to obtain the nodal forces represent the most important contribution to the overall computational costs. Table 1 shows the elapsed times measured on an SGI Origin 2000 (MIPSpro Fortran compiler, version 7.30) for different variants of a crash kernel performing 100 iterations on a mesh consisting of 25600 elements and 25760 nodes. In the table the entry *halo (DM)* refers to an HPF version parallelized with the Adaptor compiler [4] for distributed memory according to the inspector/executor strategy, *privatization (SM)*, *expansion (SM)* and *atomic (SM)* refer to the different shared memory parallelization strategies discussed in Section 2, *redundant (SM)* refers to the method based on loop-index prefetching, and *exclusive (SM)* to our exclusive ownership technique. All versions marked with (SM) utilize thread parallelism based on OpenMP, while the HPF version (DM) is based on process parallelism and relies on MPI for communication.

The irregular mesh used in this evaluation exhibits a high locality. There are only 160 non-exclusive (shared) nodes for two processors, 324 for three, and 480 nodes for four processors, respectively. As a consequence, both the distributed memory version and the shared memory versions that exploit data locality (i.e. exclusive ownership technique and loop index prefetching) show very satisfying results. The versions based on array privatization and array expansion achieve some speed-up, yet they exhibit very poor scaling due to the high synchronization overhead or computational overheads introduced by the serial code section, respectively. The version using atomic updates for all assignments to the node array scales but exhibits an overhead of about a factor of two. The best performance is obtained with the exclusive ownership strategy. The version based on

	NP = 1	NP = 2	NP = 4	NP = 8	NP = 16	NP = 32
halo (DM)	6.39	3.58	1.76	0.99	0.61	0.40
privatization (SM)	5.57	3.81	4.33	8.53	16.83	37.12
expansion (SM)	6.43	6.03	5.28	4.91	5.04	5.68
atomic (SM)	11.39	6.51	3.48	2.06	1.41	1.27
redundant (SM)	5.35	2.95	1.53	0.82	0.65	0.38
exclusive (SM)	5.10	2.79	1.47	0.74	0.55	0.34

Table 1. Execution times (secs) for crash simulation kernel on the SGI Origin 2000.

index prefetching is slightly worse since for redundant computations of elemental forces more time is required than for atomic updates with our strategy.

6 Summary and Conclusion

An efficient handling of unstructured reductions is crucial for many scientific applications. The usual methods for implementing reductions on shared memory parallel computers based on privatization, array expansion, or atomic updates, may not yield satisfying results for unstructured array reductions as they do not exploit data locality. The parallelization technique presented in this paper exploits data locality in order to minimize synchronization. The performance results verify that the concept of ownership in a shared memory programming model is essential for an efficient realization of unstructured reductions.

References

1. S. Benkner. Optimizing Irregular HPF Applications Using Halos. In *Proceedings of IPPS/SPDP Workshops*. LNCS 1586, 1999.
2. W. Blume, R. Doallo, and R. Eigenmann et.al. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.
3. T. Brandes and Germain C. A Tracing Protocol for Optimizing Data Parallel Irregular Computations. LNCS 1470, pages 629–638, September 1998.
4. T. Brandes and F. Zimmermann. Adaptor – A Transformation Tool for HPF Programs. In *Programming environments for massively parallel distributed systems*, pages 91–96. Birkhäuser Boston Inc., April 1994.
5. J. Clinckemaillie, B. Elsner, and G. Lonsdale et al. Performance issues of the parallel PAM-CRASH code. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(1):3–11, Spring 1997.
6. E. Gutiérrez, O. Plata, and E.L. Zapata. On Automatic Parallelization of Irregular Reductions on Scalable Shared Memory Systems. In *Euro-Par'99 Parallel Processing, Toulouse*, pages 422–429. LNCS 1685, Springer-Verlag, September 1999.
7. M. Hall and al. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–90, 1996.
8. High Performance Fortran Forum. High Performance Fortran Language Specification. Vers. 2.0, Rice University, January 1997.
9. Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings Supercomputing '93*, pages 361–370, 1993.
10. J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.
11. The OpenMP Forum. OpenMP Fortran Application Program Interface. Technical Report Ver 1.0, SGI, October 1997.