New OpenMP directives for irregular data access loops

J. Labarta[a], E. Ayguadé[a,*], J. Oliver[a] and
D.S. Henty[b]

[a]*European Center for Parallelism of Barcelona,
Technical University of Catalunya, Barcelona 08034,
Spain*

[b]*Edinburgh Parallel Computing Center, The
University of Edinburgh, Edinburgh EH93JZ, UK*

Many scientific applications involve array operations that are sparse in nature, ie array elements depend on the values of relatively few elements of the same or another array. When parallelised in the shared-memory model, there are often inter-thread dependencies which require that the individual array updates are protected in some way. Possible strategies include protecting all the updates, or having each thread compute local temporary results which are then combined globally across threads. However, for the extremely common situation of sparse array access, neither of these approaches is particularly efficient. The key point is that data access patterns usually remain constant for a long time, so it is possible to use an inspector/executor approach. When the sparse operation is first encountered, the access pattern is inspected to identify those updates which have potential inter-thread dependencies. Whenever the code is actually executed, only these selected updates are protected. We propose a new OpenMP clause, `indirect`, for parallel loops that have irregular data access patterns. This is trivial to implement in a conforming way by protecting every array update, but also allows for an inspector/executor compiler implementation which will be more efficient in sparse cases. We describe efficient compiler implementation strategies for the new directive. We also present timings from the kernels of a Discrete Element Modelling application and a Finite Element code where the inspector/executor approach is used. The results demonstrate that the method can be extremely efficient in practice.

*Corresponding author: E. Ayguade, CEPBA, Jordi Girona, 1-3. Modul D6, 08034 Barcelona, Spain. Tel.: +34 934 01 5951; Fax: +34 934 01 7055; E-mail: eduard@ac.upc.es.

1. Introduction

Many codes in science and engineering involve array operations that are sparse in nature, i.e. array elements are updated based on the values of relatively few elements of the same or another array. Examples include Molecular Dynamics (MD) simulations with short-ranged interactions (the force depends on the positions of a few nearby particles) and Finite Element (FE) calculations (eg the values of a node in an unstructured mesh depending on those nodes directly connected to it). As these array operations often comprise the most computationally intensive part of the code it is essential to parallelise them efficiently.

A typical example is computing the total force `force(i)` on each particle `i` in an MD code where the neighbouring particle pairs are stored in a list. Figure 1 illustrates the main loop for a code such as the DEMONS application (see Section 5). Similar access patterns occur in FE codes where the main loop is over the edges, faces or elements of a mesh, but the actual computation involves updating the nodes attached to them. The unstructured nature of the mesh again means that there are potential inter-thread dependencies, and that it is difficult for the programmer to predict in advance where they will occur for an arbitrary number of threads.

The inter-thread dependencies require that the individual array updates are protected in some way to ensure program correctness. If the loop in Fig. 1 is split across threads then, although threads will always have distinct values of `ipair`, the values of `i` and `j` may simultaneously have the same values on different threads. As a result, there is a potential problem with updating the `force` array. Simple solutions to this problem include making all the updates atomic, or having each thread compute temporary results which are then combined across threads (in cases like Fig. 1 this amounts to an array reduction which is currently supported in OpenMP 2.0 [6]). However, for the extremely common situation of sparse array access neither of these approaches is particularly efficient.

```
do ipair = 1, npair
   i = pairlist(1,ipair)
   j = pairlist(2,ipair)
   fij = pairforce(x(i), x(j))
   force(i) = force(i) + fij
   force(j) = force(j) - fij
end do
```

Fig. 1. Irregular reduction in the main loop of the DEMONS program.

Due to the sparse access pattern, the majority of updates can actually take place with no protection; making every update atomic therefore incurs an unnecessary overhead. If temporary arrays are created for each thread, there is a bottleneck when they are combined across threads; even if this is implemented in parallel (i.e. not using the naive approach of a critical section) the heavy load on the memory system may produce poor scaling. When using many threads there will also be a very high memory requirement for the temporary arrays, most of which will in practice be filled with zeroes.

The key point is that the irregular access patterns usually remain constant for a long time (sometimes the whole simulation) so it is possible to use a two-pass or inspector/executor approach. In the first pass, the data access pattern is inspected to identify those updates which have potential inter-thread dependencies and their locations are stored in a lookup table. When the calculation is actually executed, only these selected updates are protected (e.g. by an atomic update).

Unfortunately, the only way to do this in OpenMP is to write the code by hand. Even if the array reduction features of OpenMP 2.0 could be used, a compiler would be forced to use an inefficient method as there is no way of knowing at compile time that the data access patterns are constant (there is little benefit in using the inspector/executor approach if the lookup tables cannot be reused). We therefore propose a new OpenMP directive, `indirect`, for loops that have irregular data access patterns. This is trivial to implement in a conforming way (e.g. by making every update in the MD loop atomic) but also allows for a two-pass compiler implementation which will be more efficient in sparse cases. The directive will provide an efficient alternative to array reduction (as in the above MD example), as well as being of use in other situations such as the use of ordered sections.

2. Related work

Similar problems have previously been been addressed in the context of High Performance Fortran.

The simplest constructs, eg specific cases of irregular array reductions, are dealt with by special routines in the HPF library such as SUM_SCATTER and MAXVAL_SCATTER. Directives for the specification of more general irregular data access loops have been used previously in some versions of HPF [1]. These directives have influenced the one proposed in this paper. The main difference is, however, that in the context of HPF this directive is used to compute long-lived communication patterns, while our proposal focuses on iteration scheduling in the context of shared-memory multiprocessors.

3. OpenMP extensions

This section presents the two proposed extensions to OpenMP: the `indirect` clause and the `schedule` directive.

3.1. The `indirect` clause

The problem of irregular data access patterns situations in loops can be described in simple words: they are access patterns in which, given a loop, two different iterations of the loop modify the same data. This situation prevents the loop from being parallel, and thus serializes its execution. OpenMP provides two mechanisms that help in the parallelization of such loops: `atomic` and `critical` directives. These mechanisms, however, are excessively expensive in codes where not all the accesses need to be done in mutual exclusion. This situation arises also in two other situations, which are irregular reductions with low level of shared updates, and loops in which only some iterations need to be executed in sequential ordered (and thus, the `ordered` clause is too restrictive). Such codes can benefit from the use of the new proposed clause (`indirect`). The `indirect` clause can be applied to the `parallel do` directive in one of the three following situations:

1. In the presence of the `reduction` clause in the directive: this tells the compiler that the reduction that is being computed in the body of the loop has an irregular data access pattern, and that some different iterations can modify the same data.
2. In the presence of the `ordered` clause in the directive: this tells the compiler that the loop is partially ordered; that is: some iterations access the same data, and need to be executed sequentially in order to guarantee sequential consistency, but some other iterations access completely different data, and thus can be executed in parallel.

3. In the presence of some `critical` or `atomic` directive in the body of the loop: this is quite similar to the case in the `ordered` clause: not all the elements need to be computed in mutual exclusion; only those being accessed by different iterations (in this case, order is not important).

The `indirect` clause accepts a list of expressions as parameters. These expressions are those causing the irregular data access pattern:

```
!$omp parallel do [reduction|
ordered]
!$omp& indirect([expr1,...exprN])
```

3.2. Irregular reductions

Reduction operations are frequently found in the core of scientific applications. Simplest reductions are those in which the final result can be computed as a combination of partial results (ie an associative/commutative operation). As result, these computations can be computed in parallel, since each thread can compute its own partial result which will be combined later (this can be also done in parallel) with the partial results from the other threads.

Some scientific applications, however, need to perform reduction operations which are not directly parallelizable. These kind of reductions are what we call irregular reductions. The point that makes these reductions non-parallelizable is that the update index for the element is not the induction variable of the loop, but a function (f) of it, having the property that for two given values of the induction variable ($i,j,\ i \neq j$), it can happen that $f(i) = f(j)$.

The code in Fig. 3 shows an example of such a case. As can be observed, the updated elements of `force` depend on the lookup table `pairlist`, which can give the same values for `i` or `j` for different values of `ipair`. The only way to parallelize this code using OpenMP is to protect the update of the `force` vector either with `atomic` or with `critical` directives. These solutions, however, are excessively expensive in codes where not all the accesses need to be done in mutual exclusion. The inclusion of the `indirect` clause in the `parallel do` directive in the presence of a `reduction` clause tells the compiler that the reduction being performed in the parallel loop has an irregular data access pattern, but some parts of it can be executed in parallel, thus enabling the compiler to generate code to deal with this situation. The naive implementation of this this `indirect` clause is to sur-

```
!$omp parallel do
!$omp& private(i,nn1,nn2,nn3,nn4)
!$omp& reduction (+:f)
!$omp& indirect(nn1,nn2,nn3,nn4)
do iel = ieli, ielf
    nn1 = ix(1,iel)
    nn2 = ix(2,iel)
    nn3 = ix(3,iel)
    nn4 = ix(4,iel)
    do i=1,3
       xn(i,1) = x(i,nn1)
       xn(i,2) = x(i,nn2)
       xn(i,3) = x(i,nn3)
       xn(i,4) = x(i,nn4)
    end do
    call mforce(force,xn)
    do i=1,6
       f(i,nn1) = f(i,nn1) + force(i,1)
       f(i,nn2) = f(i,nn2) + force(i,2)
       f(i,nn3) = f(i,nn3) + force(i,3)
       f(i,nn4) = f(i,nn4) + force(i,4)
    end do
end do
!$omp end parallel do
```

Fig. 2. Irregular reduction in the main loop of the crash kernel.

```
!$omp parallel do
!$omp&      reduction (+:force)
!$omp&      indirect(i,j)
do ipair = 1, npair
    i = pairlist(1,ipair)
    j = pairlist(2,ipair)
    fij = pairforce(x(i),x(j))
    force(i) = force(i) + fij
    force(j) = force(j) - fij
end do
!$omp end parallel do
```

Fig. 3. Extended OpenMP directives for the irregular reduction in the main loop of the DEMONS application.

round the updates of `force` either with `critical` or `atomic` directives, but more complex implementations can generate code for a two-pass implementation of the reduction.

Another example taken for a crash kernel is shown in Fig. 2. In this code, the computation of the forces is done from the displacements using an indirect addressing to an structure that setups a highly complex structured mesh. Again, the only way to parallelize this loop using OpenMP is to protect the update of the `force` vector either with `atomic` or with `critical` directives. The inclusion of the `indirect` clause in the `parallel do` directive in the presence of a `reduction` clause tells the compiler that the reduction being performed in the parallel loop has an ir-

```
!$omp do indirect(j)
do i = 1,n
   j = jindex(i)
!$omp critical
   a(j) = compute(a(j), ...)
!$omp end critical
end do
!$omp end do
```
<div align="center">(a)</div>

```
!$omp do ordered indirect(j)
do i = 1,n
   j = jindex(i)
!$omp ordered
   a(j) = b(i) op expression_1
!$omp end ordered
end do
!$omp end do
```
<div align="center">(b)</div>

Fig. 4. Two synthetic examples in which the INDIRECT clause can be applied to relax the synchronization imposed by CRITICAL or ORDERED in OpenMP.

regular data access pattern which enables some some iterations to be executed in parallel.

3.3. Relaxed Mutual Exclusion

As said in the previous subsection, a naive implementation of the `indirect` clause for irregular reductions can be done with the use of the critical sections (achieved using the `critical` directive). This also implies that the use of the `indirect` clause can also help in the code generation for cases in which the `critical` directive has been used to parallelize a loop without a reduction, but with some shared updates between iterations executed by diffferent processors. An example of this case is shown in Fig. 4(a).

3.4. Partially ordered loops

A special case of the previous example occurs when the shared updates need not only to be performed in mutual exclusion, but also in an ordered way. The use of the `indirect` clause in this case tells the compiler that the only iterations that actually need to be executed in an ordered way are those which are updating the same data. An example of code using the `indirect` clause in this manner is shown in Fig. 4(b).

```
!$omp schedule(S)
...
C outer sequential loop
do step = 1,nsteps
...
!$omp parallel do
!$omp&   reduction (+:a)
!$omp&   indirect(j:S)
   do ipair = 1, npair
      j = table(i)
      ...
      a(j) = a(j) - update
   end do
!$omp end parallel do
...
!$omp parallel do
!$omp&   reduction (+:a)
!$omp&   indirect(j:S)
   do ipair = 1, npair
      j = table(i)
      ...
      a(j) = a(j) * another_update
   end do
!$omp end parallel do
   ...
   if (some_condition(x)) then
      recalculate(table)
!$omp reset(S)
   end if
   ...
end do
```

Fig. 5. Definition of a user–level schedule and its initialization, reuse and reinitialization in a sequence of loops with indirect clauses.

3.5. Scheduling re-use

The typical structure of most scientific applications follows an iterative outer loop which implies a number of parallel computations, most of them over the same data. In that scenario, it could be useful to communicate scheduling information from one parallel region to others. This idea has been applied previously to data communication in HPF [1], with the use of the schedule clause, and can be also applied to OpenMP.

The following mechanisms are provided to name/define/undefine user-level schedules originated from the use of the `indirect` clause:

– Schedule naming:

```
!$omp schedule(schedule_name)
```

This simply defines a symbolic name `schedule_name` for a user-defined schedule.

– Schedule definition:

```
!$omp& indirect([expr1,...exprN]
!$omp& [:schedule_name])
```

```
! v(1:20)=(1,2,3,10,20,4,2,5,6,8,9,
!          3,10,3,11,12,13,14,15,16)
!$omp parallel do schedule (static)
!$omp&  reduction(*:a) indirect (v(i))
do i = 1, 20
   a(v(i)) = a(v(i)) * 2
end do
```

(a)

```
nintervals(1:4) = (3,3,3,1)
shared(1) = (false,true,false)
shared(2) = (false,true,false)
shared(3) = (false,true,false)
shared(4) = (false)
lower(1) = (1,2,4)
lower(2) = (6,7,8)
lower(3) = (11,12,15)
lower(4) = (16)
upper(1) = (1,3,5)
upper(2) = (6,7,10)
upper(3) = (11,14,15)
upper(4) = (20)
```

(b)

Fig. 6. a) Synthetic example with irregular reduction. b) Internal data structures initialized during the inspector phase.

The symbolic name schedule_name included in the indirect clause informs the compiler that the parallel loop uses the a user-level scheduling identified by the symbolic name schedule_name. When such a definition is found, the code generated by the compiler first checks if the schedule has already been computed; if not, then the inspector is invoked. The re-use of an already defined schedule amortizes the overhead associated with the inspector phase over a large number of loops or instantiations of the same one.

– Schedule invalidation:

```
!$omp reset(schedule_name)
```

This directive tells the compiler that the schedule associated to the symbolic name schedule_name is not valid any more.

Figure 5 shows an example of the use of the schedule directive combined with the indirect clause and reset directive. In this case, the schedule is defined once and re-used while condition some_condition(x) evaluates to false. At this point, the original algorithm modifies the contents of table and, as a consequence, the schedule reset.

4. Compiler implementations

This section outlines some implementations for the indirect clause in the context of an irregular reduction or non-complete mutual exclusion (using intervals) and reviews some proposed implementations that can be applied in the context of a partially ordered loop.

4.1. Intervals

The proposed implementation is based on the inspector/executor model. The first time that a parallel loop marked with an indirect clause is reached, the program starts the execution of the inspector code, which has been generated by the compiler. This code basically contains the same loop header, all those statements in the original loop body that lead to the evaluation of the expressions included in the indirect clause, and the additional code that builds the schedule data structure. Later executions of the same parallel loop (or other loops with the same named schedule) make use of that data structure.

The code in Fig. 6(a) presents a simple example of parallel loop with an irregular reduction operation that has been parallelized using the indirect clause. Notice that there are some iterations of the loop that modify the same data (positions 2, 3 and 10 of the vector v). But most of the iterations do not modify the same data, and thus they can be executed in parallel.

Figure 7 shows an example of a data structure generated by the compiler during the inspector phase. For each processor, there is a bitmap that shows which elements it is accessing in the v vector. After that, the inspector computes the shared elements vector, which is a bitmap that tells, for each element in the v vector whether it is shared or not (a position of v is shared if the bitmap count for the column is more than one; that is: it is being modified by more than one processor). Once this bitmap is computed, the inspector can build the final data structure that will be used by the executor phase: the intervals for each processor. The iteration space of each processor is split up into intervals. An interval is a range of consecutive iterations that are all either completely shared or not shared. The data structures for the previous example result in the initialization shown in Fig. 6(b). Vector nintervals is used to indicate the number of intervals that each processor will execute. Three additional vectors are used to define the lower and upper bound for each interval and a boolean indicating if the interval is shared or not.

Data(i)

P0: 1,2-3(shared),4-5
P1: 6,7(shared),8-10
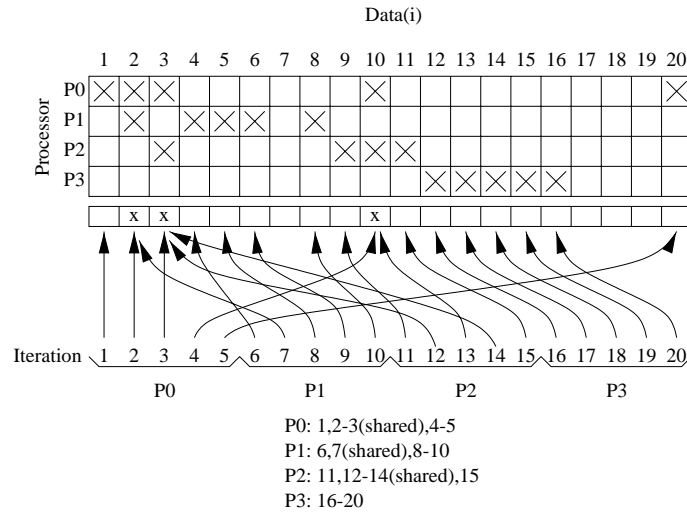P2: 11,12-14(shared),15
P3: 16-20

Fig. 7. Construction of the intervals during the inspector phase using a bitmap for the example in Fig. 6.

```
!$omp parallel do schedule (static)
do p = 1, numprocs
 my_proc = omp_get_threadnum()+1
 do interval = 1,nintervals(my_proc)
  if (shared(my_proc,interval)) then
   do i = lower(my_proc,interval),&
          upper(my_proc,interval)
!$omp critical
     a(v(i)) = a(v(i)) * 2
!$omp end critical
   end do
  else
   do i = lower(my_proc,interval),&
          upper(my_proc,interval)
    a(v(i)) = a(v(i)) * 2
   end do
  end if
 end do
end do
```

Fig. 8. Executor for the loop in Fig. 6.

Subsequent executions of the parallel loop will use the compiler's generated executor code, which makes use of the previous data structures to execute the parallel loop. This code executes in mutual exclusion (or with atomic access) only those iterations (actually intervals) that conflict with some other processor. Those intervals that are accessed only by one processor are executed without protection. The executor's code for the previous example is shown in Fig. 8.

It is important to point out that this method can be further optimised by improving the executor phase in order to classify the shared segments depending on the processors that are accessing them (shared sets), and utilizing different locks to access each one of the sets, in order to reduce contention when *hot spots* are detected.

4.2. Partially ordered loops

Partially ordered loops have been a case of study in many previous works. One possible approach to the parallelization of non-completely parallel loops is to detect data dependencies at runtime, again using an inspector/executor model. The basic idea is to move data dependence detection from compile-time (where data available for analysis is restricted) to runtime. In most of the reviewed implementations, at runtime, the inspector builds data dependence graphs based on the addresses accessed by the loop and, based on those graphs, schedules iterations in *wavefronts* (sets of iterations which are dependence-free among them) [2,7]. Many of these implementations can be applied by the compiler to implement the semantics of the `indirect` clause applied to an `ordered` directive.

5. Experiments

In this section we present experimental results for two codes whose whose major computational loop is of the same form as the ones shown in Figs 2 and 3. The first application, named DEMONS – Discrete Element Modelling on SMP clusters –, is a Fortran 90 test code used to evaluate the efficiency of hybrid message-passing and shared-memory parallelism on clusters of shared-memory machines [4]. Here we do not utilise the MPI capabilities of the code, and run on a single pro-

cess to investigate the performance of the pure OpenMP implementation with varying numbers of threads T. The second one is a kernel extracted from an industrial crash simulator.

5.1. DEMONS application

Discrete Element Models (DEMs) simulate the behaviour of systems of particles which interact via pairwise forces. The particle positions are evolved in time using an iterative loop comprising many small, discrete time-steps. A typical DEM might study the formation of "sand piles" as grains of sand are dropped onto a solid surface. The general properties of a DEM are: there are many particles; particles remain relatively static; the inter-particle force is very short-ranged; the time taken to compute the forces dominates the simulation. It is therefore very important to parallelise the force calculation efficiently.

To avoid having to loop through all pairs of particles, the DEMONS code maintains a list of pairs of all those particles which are close together and therefore likely to interact. As the force is short-ranged, this simple and commonly used technique reduces the complexity of the calculation from $O(N^2)$ to $O(N)$. As particles remain relatively stationary, this list only requires to be recalculated fairly infrequently; in a real simulation, the same list can typically be used for many hundreds of iterations. Calculating the forces involves looping over this list of pairs, computing the inter-particle force for each pair, and updating the force on the two particles accordingly. Since the loop is over particle pairs (not the particles themselves), and each particle belongs to many pairs, care must be taken when updating the forces on each particle due to potential inter-thread dependencies.

5.1.1. Implementation of force calculation

Three basic approaches were investigated for resolving the inter-thread dependencies in the force calculation

- making every force update atomic
- using array reduction
- using a two-pass inspector / executor approach

Since array reduction is not part of the OpenMP 1.0 standard we implemented it by hand in several ways

- first accumulating into private temporary arrays and performing the final global array sum across threads in a critical region

- accumulating into private arrays and performing the global sum in T sections, striped across threads and separated by barriers, so that there are no inter-thread dependencies
- accumulating into different rows of a shared matrix of dimension $T \times N$, then performing the global sum in parallel over the transpose column direction

These are common approaches for performing irregular reductions in OpenMP [5]. The use of critical sections gave very poor performance; the other two methods gave very similar timings. Here we quote results from the third "transpose" approach. We implement the inspector / executor approach in a very straightforward manner. Whenever a new list of pairs is created, we identify particles receiving force updates from more than one thread. When calculating the force, only these updates are protected by an ATOMIC directive.

5.1.2. Timings

The code for computing the force is almost identical to that in Fig. 3 except that the calculation is done in three dimensions so there are three position coordinates x rather than just one. For the tests we use a million particles, with the force law requiring one floating-point inverse and one square root per call. All calculations were performed in double precision.

The benchmark platforms used to run this application were a Sun HPC 3500 (eight × 400 MHz UltraSPARC-II CPUs) and a Compaq ES40 (four × 500 MHz Alpha EV6 CPUs). For compilation on the Sun we used the Kuck and Associates (KAI) Guide system version 3.7, which uses source-to-source translation with calls to a parallel runtime library; on the Compaq, OpenMP is part of the standard f90 compiler. Sun's own native OpenMP compiler, part of the recently-released Workshop Forte 6.0 environment, was not available in time for this paper.

We plot the parallel efficiencies of the three implementations of the force calculation on the Sun and Compaq in Figs 9 and 10. The timings for the serial code, $t(T = 1)$, were 4.16 and 1.89 seconds respectively.

Using the KAI compiler on the Sun, the atomic updates are done using software locks which are rather expensive. Contention for these locks makes using atomic updates for all force calculations completely infeasible. However, using the two-pass approach is more effective than using array reduction despite the high cost associated with atomic locks.

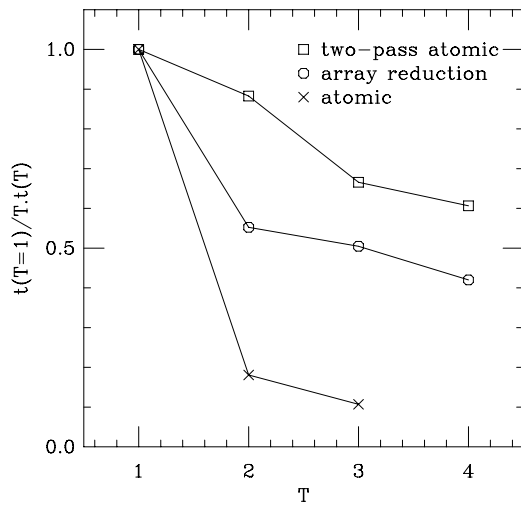On the Compaq, atomic updates are much more efficient but they still incur a significant overhead. For

Fig. 9. Parallel efficiency against threads T on Sun for the DEMONS code.

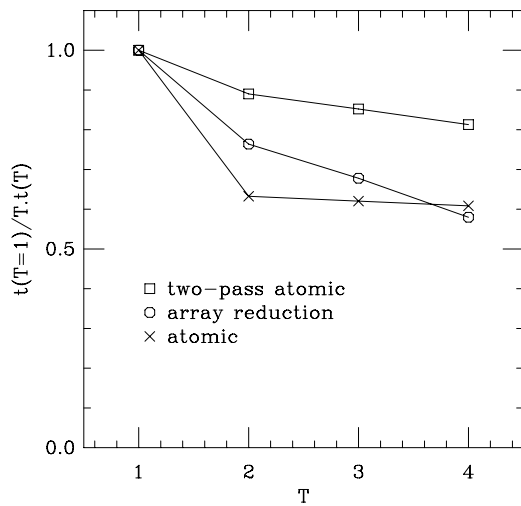Fig. 11. Parallel efficiency against threads T on SGI Origin 2000 for the Crash Kernel.

Fig. 10. Parallel efficiency against T on Compaq for the DEMONS code.

$T < 4$ the cost is such that array reduction is more efficient, although the atomic approach is slightly better for $T = 4$. However, using the two-pass approach is always the most efficient method.

5.2. Crash Kernel

This kernel has been extracted from an industrial crash simulator. It is basically composed of a loop nest that iterates over 1000 times. At each iteration of the sequential loop, the computation of the forces is performed from the displacements. The kernel uses indirect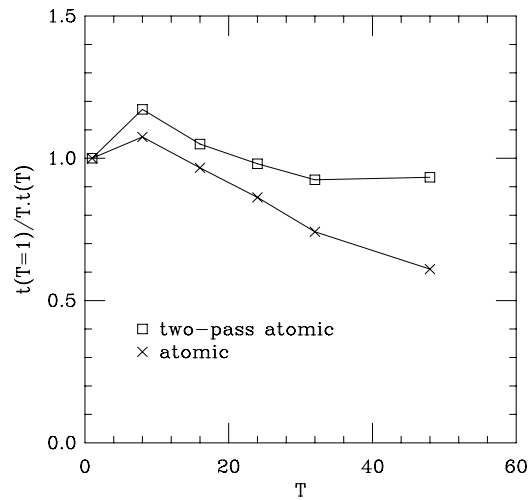 addressing to setup a highly complex structured mesh and performs memory accesses in a highly irregular way. The code for computing the force is similar to the one shown in Fig. 2. For the tests we use a synthetic case with a mesh composed of 40×640 points. All calculations were performed in double precision.

The benchmark platform used for this kernel was an SGI Origin 2000 system with 64 R10k processors, running at 250 MHz with 4 Mb of secondary cache each. For all compilations we use the native f77 compiler. On this machine, atomic updates are also implemented in hardware.

Two versions of the code are evaluated. The first one protects all force updates by an ATOMIC directive. In the second one only those updates identified during the inspector phase are protected (i.e. updates from more than one thread). For these two versions, Fig. 11 plots the parallel efficiencies when using 8, 16, 24, 32 and 48 processors. The timing for the serial code, $t(T = 1)$, was 322.5 seconds. Notice that the two-pass approach is always the most efficient method.

6. Conclusions

In this paper we have presented a set of extensions to OpenMP and a possible efficient implementation inside the compiler that ease the parallelization of applications with irregular data accesses. Two examples are used to prove the efficiency of the proposal, demonstrating that the two-pass technique for irregular reductions outperforms the alternative methods in a real code and a kernel extracted from an industrial application.

For the DEMONS code, the calculation of the force is relatively expensive; it was possible to use a very naive approach to implement the two-pass method where the decision as to whether or not to use `atomic` was taken for each individual update. In other situations the amount of work per update may be much smaller, for example a single addition, and reducing the overhead of the implementation will be important. Here, the technique of dividing the loop into intervals, as proposed in Section 4, will be extremely beneficial to performance. In a compiler implementation, the `schedule` directive could be used to re-run the inspector phase whenever the list of particle pairs was updated. For the crash kernel, the parallelization using the proposed directives and their translation using the inspector/executor model produce a speed-up up to 1.5 times better than using `ATOMIC` for each individual update.

We conclude that our proposed new OpenMP directives would enable straightforward and efficient automatic parallelisation of a wide range of scientific applications. The approach is currently being implemented in the framework of the OpenMP NanosCompiler [3] for Fortran77.

7. Acknowledgements

References

[1] S. Benkner, K. Sanjari, V. Sipkova and B. Velkov, Parallelizing irregular applications with the Vienna HPF+ Compiler VFC. Proc. of HPCN'98, Amsterdam (The Netherlands), April 1998.

[2] D.K. Chen, P.C. Yew and J. Torrellas, An efficient algorithm for the run-time parallelization of doacross loops, In proceedings of Supercomputing '94, Nov. 1994.

[3] M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro and J. Oliver, NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. Concurrency: Practice and Experience. **12** (12), pp. 1205–1218, October 2000. (Extended version of paper presented at the 1st European Workshop on OpenMP, Lund (Sweeden)).

[4] D.S. Henty, Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling, In proceedings of Supercomputing 2000, Nov. 2000.

[5] D. Hisley, G. Agrawal, P. Satya-narayana and L. Pollock, *Porting and Performance Evaluation of Irregular Codes using OpenMP*, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sep. 1999, pp. 47–59.

[6] OpenMP Organization, OpenMP Fortran Application Interface, v. 2.0, www.openmp.org, November 2000.

[7] L. Rauchwerger, N.M. Amato and D.A. Padua, Run-time methods for parallelizing partially parallel loops, In proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995.