Facing the Multicore-Challenge III, Lecture Notes in Computer Science Volume 7686,
Springer, 2013, pp 25–36
http://dx.doi.org/10.1007/978-3-642-35893-7_3

Array-based reduction operations for a parallel adaptive FEM

Martina Balg[1], Jens Lang[2], Arnd Meyer[1], Gudula Rünger[2]

[1] Department of Mathematics, Chemnitz University of Technology, Germany
{*martina.balg|arnd.meyer*}*@mathematik.tu-chemnitz.de*
[2] Department of Computer Science, Chemnitz University of Technology, Germany
{*jens.lang|gudula.ruenger*}*@cs.tu-chemnitz.de*

Abstract For many applications of scientific computing, reduction operations may cause a performance bottleneck. In this article, the performance of different coarse- and fine-grained methods for implementing the reduction is investigated. Fine-grained reductions using atomic operations or fine-grained explicit locks are compared to the coarse-grained reduction operations supplied by OpenMP and MPI.
The reduction operations investigated are used for an adaptive FEM. The performance results show that applications can gain a speedup by using fine-grained reduction since this implementation enables to hide the reduction between calculation while minimising the time waiting for synchronisation.

1 Introduction

For applications of parallel scientific computing, reduction operations play an important role. In many cases, the performance of reduction operations, which aggregate data located on different processors to a common result datum using a specified operation, is crucial to the overall performance of the application. One example is the adaptive Finite Element Method (FEM) [1] applied to deformation problems (1) which is considered in this article.

$$\mathrm{div}(\sigma(u)) + \rho\, f = 0 \quad \textit{with appropriate boundary conditions.} \tag{1}$$

In the adaptive FEM, the created mesh is refined stronger around critical points which allows, compared to total refinement, more exact results within the same execution time. In order to being able to process large problems, a fast reduction operation is needed. The basic concept of this FEM is the discretisation of the domain Ω with hexahedral elements el and the approximation of all functions as a linear combination of linear or quadratic ansatz functions Ψ_k:

$$u(X) \approx u_h(X) = \textstyle\sum_{k=1}^{N_X} u^{(k)} \Psi_k(X) \quad . \tag{2}$$

Hence, every element el consists of 6 faces, 12 edges and 8, 20 or 27 element nodes $X^{(k)}$ and all $\Psi_k(X)$ are defined by the degrees of freedom of each element, i.e. by their function value in each element node. Applying this discrete ansatz to (1) leads to a discrete linear system of equations:

$$A\underline{u} = \underline{b} \tag{3}$$

where the vector $\underline{u}$ consists of all $u^{(k)}$ and represents the solution u. A denotes the system matrix, called stiffness matrix, and $\underline{b}$ the right-hand side, called load vector. Using the restriction of the ansatz for each element, the given problem decomposes into a sum of element-wise contributions:

$$A = \sum_{el} L_{el}^t A_{el} L_{el} \qquad \text{and} \qquad \underline{b} = \sum_{el} L_{el}^t \underline{b}_{el} \tag{4}$$

with appropriate projections L_{el} and L_{el}^t. So, it is convenient to just compute all A_{el} and $\underline{b}_{el}$ instead of assembling the whole stiffness matrix and load vector.

The adaptive FEM investigated executes the following steps consecutively until a given accuracy is reached: (I) adaptive, instead of total, mesh refinement, (II) assembly of the stiffness matrices, (III) solution of a linear system of equations with the conjugate gradient method which involves a reduction operation, and (IV) error estimation for next refinement. Steps (II) and (III) are available in a parallel implementation. Step (III) is investigated in this article.

The main goal of this article is to optimise the execution time of the FEM. The contribution of this article is the optimisation in the reduction phase of step (III) by introducing fine-grained reduction. Several implementations of the reduction are investigated in isolation, as well as in the context of the adaptive FEM. Examining the background shall enable to generalise the findings for transfer to related problems. Section 2 describes the parallel solution of linear systems of equations. Section 3 proposes variants of the fine-grained reduction. Their implementations in OpenMP are given in Sect. 4. Section 5 shows experimental results. Section 6 discusses related work and Sect. 7 concludes the article.

2 Solution of linear systems of equations

Step (III) of the adaptive FEM is performed as follows [1]: The conjugate gradient method [9] is used for solving the linear system of equations (3). This iterative method minimises the residuum $\underline{r}^{[n]} := A\underline{u}^{[n]} - \underline{b}$ along a corresponding search direction in each step starting from an initial solution $\underline{u}^{[0]}$. Each iteration produces a correction term which generates an improved approximated solution $\underline{u}^{[n]}$.

According to Formula (4), the matrix A is composed of the element stiffness matrices A_{el}. This leads to a sparse structure that is exploited to compute products of the type

$$\underline{y}^{[n]} = A\underline{u}^{[n]} \quad . \tag{5}$$

The projector L_{el} converts $\underline{u}^{[n]}$ into a vector $\underline{u}_{el}$ containing only those entries that belong to the nodes of el, i.e.:

$$\underline{u}_{el} = L_{el}\underline{u}^{[n]} \quad . \tag{6}$$

A_{el} is applied to $\underline{u}_{el}$ in order to create the element solution vector $\underline{y}_{el}$:

$$\underline{y}_{el} = A_{el}\underline{u}_{el} \quad . \tag{7}$$

The vector $\underline{y}_{el}$ is then interpolated to the whole length of $\underline{y}$ by applying the transposed projector L_{el}^t and added to the overall solution vector $y^{[n]}$:

$$\underline{y}^{[n]} = \sum_{el} L_{el}^t \underline{y}_{el} \quad . \tag{8}$$

2.1 Data structures

Each element stiffness matrix A_{el} is symmetric and is stored column-wise as a packed upper triangular matrix (BLAS format *TP* [2]). The values for the nodes are stored consecutively. Each node needs n_{dof} values if there are n_{dof} degrees of freedom (dof). When considering three-dimensional deformation problems with 3 degrees of freedom in 27 element nodes, the element stiffness matrix has a size of 81×81. Correspondingly, the size of the vectors $\underline{x}_{el}$ and $\underline{y}_{el}$ is 81. While the size of the element-related data structures A_{el}, $\underline{x}_{el}$ and $\underline{y}_{el}$ is constant over the whole runtime of the FEM, the size of the overall data structures $\underline{u}$, $\underline{b}$ and $\underline{y}$ increases with each refinement step as the number of elements increases. The size of the overall data structures is proportional to the number of elements and can be up to some hundreds of thousands.

The element solution vectors $\underline{y}_{el}$ are added to the overall solution vector $\underline{y}$ according to Formula (8) where the projector L_{el}^t defines to which entry of $\underline{y}$ an entry of $\underline{y}_{el}$ is added. For memory efficiency, the implementation does not store the projector as a large matrix but uses a separate array for each element el for this assignment. In Fig. 1, which illustrates this principle, this array is represented by the arrows from the source entry in $\underline{y}_{el}$ to the target entry in $\underline{y}$. The figure also illustrates that each node, represented by a square, consists of 3 degrees of freedom. When calculating Formula (8), the corresponding location in $\underline{y}$ is looked up in the array for each entry of $\underline{y}_{el}$. To each entry of $\underline{y}$, entries from two $\underline{y}_{el}$ are added if and only if this node is part of these two elements.

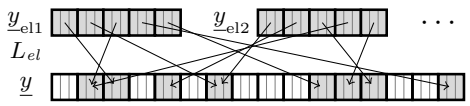


Figure 1: Summation of element solution vectors $\underline{y}_{el}$ to the overall solution vector $\underline{y}$ using the projection of L_{el}

2.2 Parallelisation

The parallelisation of the FEM method described exploits that the calculations of Formula (7) are independent of each other. Each element is assigned to one processor p of the set of processors P which calculates (6) and (7). For the result summation in (8), a local solution vector $\underline{y}^{\circ}$, which has the same size as $\underline{y}$, is used on each processor. The part of non-null entries in $\underline{y}^{\circ}$ is greater than $\frac{1}{p}$ and they need much less memory than the element stiffness matrices so that they can be stored in a dense format. After Formula (8) has been calculated on all processors $p \in P$, the local solution vectors $\underline{y}^{\circ}$ are added to the overall solution vector $\underline{y}$, i.e.

$$\underline{y} = \sum_{p \in P} \underline{y}^{\circ,p} \quad , \tag{9}$$

where $\underline{y}^{\circ,p}$ denotes the local solution vector of processor p in this summation. Formula (9) is the reduction being optimised in this work.

The SPMD-style parallel algorithm calculating Formula (5), i.e. one iteration of approximating the solution of (3), is shown in Alg. 1. The algorithm receives the element stiffness matrices A_{el} and the projectors L_{el} for all elements el as well as the approximation $\underline{u}$ for the solution of the linear system of equations (3) as input and returns the vector $\underline{y}$ as output. In this algorithm, $\underline{y}$ is a shared variable; all other variables are private, i.e. only accessible by the processor owning them. After setting $\underline{y}$ to zero (Line 1), the calculation of Formulas (6) to (8) is executed for each element (Lines 4 to 7) in a parallel section. The reduction operation at the end of the parallel section (Lines 8 to 10) adds the local solution vectors $\underline{y}^\circ$ to the overall solution vector according to (9). In order to avoid conflicts when accessing $\underline{y}$, this addition is performed within a critical section. The absence of conflicts could also be ensured by other methods, for example by a global barrier in Line 8, followed by an arbitrary reduction algorithm. In any case, after the barrier operation, only reduction, and no computation, is performed.

Input: A_{el}, L_{el} for all el, $\underline{u}$
Output: $\underline{y}$

```
1  y := O // shared vector y
2  begin parallel
3      y° := O
4      foreach element el do
5          x_el := L_el u
6          y_el := A_el x_el
7          y° := y° + L_el^t y_el
8      begin critical section
9          y := y + y°
10     end critical section
11 end parallel
```

Algorithm 1: Parallel calculation of (5)

Input: A_{el}, L_{el} for all el, $\underline{u}$
Output: $\underline{y}$

```
1  y := O // shared vector y
2  begin parallel
3      foreach element el do
4          x_el := L_el u
5          y_el := A_el x_el
6          y° := L_el^T y_el
7          foreach entry i of y° with
           y°[i] ≠ 0 do
8              atomic_add(&y[i], y°[i])
9  end parallel
```

Algorithm 2: Parallel calculation of (5) with atomic addition of $\underline{y}$

3 Fine-grained reduction

In contrast to the coarse-grained reduction in Alg. 1, which locks the whole vector $\underline{y}$, the reduction can also be implemented in a fine-grained way. Fine-grained reduction means that each update of a vector entry is synchronised separately. This method allows to interleave reduction with computation and enables multiple threads to access $\underline{y}$ concurrently if they are processing different entries. Blockings due to concurrent write accesses to an entry of $\underline{y}$ may only occur for nodes shared by elements stored on different processors. This is true only for a small part of the nodes. Furthermore, the implementation does not need to store $\underline{y}^\circ$ explicitly. In contrast to a sequential implementation, the order of writes to $\underline{y}$ is not defined. However, the addition is commutative and $\underline{y}$ is only read after finishing the reduction. Hence, the order in which the elements are processed is irrelevant.

Two methods for the fine-grained synchronisation of updates of the overall solution vector $\underline{y}$ are investigated: *atomic operations* and *fine-grained locks*.

3.1 Atomic operations

Atomic operations, which combine several semantic instructions in one non-preemptible function, can be implemented in hardware or in software. Hardware-supported atomic operations are generally more efficient than operations implemented in software as the thread synchronisation of the software implementation is very complex.

Use of atomic operations Let the function `atomic_add(`*double* a, double b*`)` be a function which adds the value of b to the value which a points to in a non-preemptible way. Algorithm 2 uses this function for an alternative implementation of Alg. 1: Instead of reducing the local solution vectors $\underline{y}^\circ$ to the overall solution vector $\underline{y}$ globally at the end of the parallel execution (Lines 8 to 10 in Alg. 1), each vector entry of $\underline{y}^\circ$ is now added individually to the corresponding entry of $\underline{y}$. Each update of an entry is synchronised by using an atomic addition operation (Line 8 in Alg. 2).

Atomic operations using compare & swap For many operations, such as addition, subtraction or logical operations, there exist atomic hardware instructions on most common platforms. If, however, no such atomic hardware instruction exists for the operation required, it has to be emulated in software. For this emulation, the atomic *compare & swap instruction* (`CAS`), which is available on most platforms, can be used.

```
1 bool CAS(T *location, T oldVal, T
  newVal)
2 begin
3 |  begin atomic
4 |  |  if *location == oldVal then
5 |  |  |  *location = newVal;
6 |  |  return (*location ==
  |  |  oldVal);
7 |  end atomic
8 end
```

Algorithm 3: Compare & swap according to [6]

```
1 void atomic_add(double *sum,
  double a)
2 begin
3 |  repeat
4 |  |  double oldSum = *sum;
5 |  |  double newSum = oldSum +
  |  |  a;
6 |  until CAS(sum, oldSum,
  |  newSum) ;
7 end
```

Algorithm 4: Emulation of an atomic addition using compare & swap

In this article, the compare & swap instruction as defined in Alg. 3 is used: First, the value of the variable *oldVal* is compared to the value of **location*. If these values are equal, the value of *newVal* is written to the memory location which *location* points to. The return value of `CAS` is the result of the comparison.

Algorithm 4 shows how any binary operation can be emulated using the atomic `CAS` operation taking the addition as an example [4]: The old value of the result, **sum*, is stored in the variable *oldSum*. This variable is used to calculate the new value *newSum*. If the memory location which *sum* points to has not been altered by another processor intermediately, *newSum* is written to this location. Otherwise, the operation is repeated with the current value of **sum*.

```
1  int* locks[N_LOCKS]; // initialise with 0
2  void lock(int i)
3  begin
4      while (true) do
5          int lock_status = atomic_add(
6          &locks[i % N_LOCKS], 1);
7          if (lock_status == 0) then
8              break;
9          unlock(i);
10         usleep(1);
11 end
12 void unlock(int i)
13 begin
14     atomic_add(
15     &locks[i % N_LOCKS], -1);
16 end
```

Algorithm 5: Implementation of the functions `lock()` and `unlock()`

3.2 Fine-grained locks

Optimising the granularity of locks for given conditions has been investigated for a long time [10]. In this subsection, a fine-grained locking technique is presented which uses a separate locking variable for a small number of entries of the solution vector $\underline{y}$ instead of always locking the whole vector $\underline{y}$ as in Alg. 1. Before each access to an entry i of $\underline{y}$, the corresponding lock is acquired by calling `lock(`i`)`; after the access it is released by calling `unlock(`i`)`. Line 8 in Alg. 2 is replaced by the instruction $\underline{y}[i] := \underline{y}[i] + \underline{y}^{\circ}[i]$, surrounded by the lock and unlock statements.

The implementation of the functions `lock()` and `unlock()` is shown in Alg. 5. N_LOCKS lock variables are stored in the array *locks*. The parameter of `lock()` and `unlock()` is the index of the data array entry to be accessed. The current thread attempts to acquire the lock corresponding to the given index by incrementing the lock variable. If this is successful, *lock_status* is equal to zero. Otherwise, the attempt is undone by calling `unlock()` and another attempt to acquire the lock is made. The function `unlock()` releases the lock by atomically decrementing the value of the lock variable by 1. Section 5.2 investigates which values should be chosen for N_LOCKS. The instruction `usleep(1)` in Line 9 avoids livelocks by suspending the current thread for one microsecond if acquiring the lock fails.

4 Implementation

Solving the linear system of equations (3) is implemented in the function *ppcgm* in the FEM investigated. The parallel section of the OpenMP implementation is shown in Listings 1 and 2. In Listing 1, the reduction of the private array `Y` is performed by OpenMP when leaving the parallel section. In contrast, in Listing 2 each access to the shared array `Y` is performed in an atomic way so that the reduction operation at the end of the parallel section can be avoided.

The loop in Line 6 of the source code of the Listings 1 and 2 runs over all elements which have been assigned to the current processor. The calculation of

```
1  real*8 Y(N),U(N),El(N,Nnod*Ndof)
2  real*8 Uel(Nnod*Ndof),Yel(Nnod*Ndof)
3  integer L(N),i,j,k,Kn,N,Ndof,Nel,Nnod
4  Y = 0d0
5  !$omp parallel reduction(+:Y)
6    do k=1,Nel
7      call UtoUel(Ne0,Ndof,Uel,U)
8      call DSPMV('l',Nnod*Ndof,1.0d+0,
           El(k),Uel,1,0D+0,Yel,1)
9      do i=1,Nnod
10       Kn = L(i)*Ndof
11       do j=1,Ndof
12
13         Y(Kn+j) = Y(Kn+j) + Yel(Ndof
               *(i-1)+j)
14       end do
15     end do
16   end do
17 !$omp end parallel
```

Listing 1: Implementation of *ppcgm* in OpenMP using coarse-grained reduction

```
1  real*8 Y(N),U(N),El(N,Nnod*Ndof)
2  real*8 Uel(Nnod*Ndof),Yel(Nnod*Ndof)
3  integer L(N),i,j,k,Kn,N,Ndof,Nel,Nnod
4  Y = 0d0
5  !$omp parallel
6    do k=1,Nel
7      call UtoUel(Ne0,Ndof,Uel,U)
8      call DSPMV('l',Nnod*Ndof,1.0d+0,
           El(K),Uel,1,0D+0,Yel,1)
9      do i=1,Nnod
10       Kn = L(i)*Ndof
11       do j=1,Ndof
12 !$omp atomic
13         Y(Kn+j) = Y(Kn+j) + Yel(Ndof
               *(i-1)+j)
14       end do
15     end do
16   end do
17 !$omp end parallel
```

Listing 2: Implementation of *ppcgm* in OpenMP using fine-grained reduction with atomic addition

$\underline{x}_{el}$ (Line 5 in Alg. 1) is shown in Line 7 in the source code. The following matrix-vector multiplication is performed by the BLAS routine `DSPMV`. The addition of the $\underline{y}_{el}$ to $\underline{y}^{\circ}$ (Line 7 in Alg. 1) is performed in lines 9 to 15 of the source code. The variable `Kn` (Line 10) contains the index of `Y` to which the current node of `Yel` is added. This assignment, which is defined by L_{el} in Alg. 1, is stored in the array `L` in the source code. The private arrays `Y` of all threads, that contain the intermediate results, are added to the shared array by OpenMP at the end of the parallel section in Line 17. This section corresponds to the lines 8 to 10 in Alg. 1.

In contrast, `Y` is a shared variable in Listing 2. In Line 13, each thread writes its results directly to the shared array `Y` without using an intermediate array. The write operation is synchronised by the `atomic` OpenMP statement in Line 12 which corresponds to Line 8 in Alg. 2.

5 Experiments

In synthetic tests, the execution time of the different implementation variants of the reduction operation have been investigated. Also, the actual implementation of the FEM was investigated using an example object as input to explore which speedup can be achieved if the reduction performed by the different implementation variants and in order to find a suitable number of locks for the implementation variant presented in Sect. 3.2.

For the experiments a 24-core shared-memory *Intel machine* with 4 × Intel Xeon X5650 CPUs @ 2.67 GHz and 12 GB of RAM and a 24-core *AMD machine* with 4 × AMD Opteron 8425 HE CPUs @ 2.1 GHz and 32 GB of RAM have been used. For the tests with the actual FEM implementation, the example object

bohrung, which represents a cuboid with a drill hole, has been used, see Fig. 2. The object initially consist of 8 elements and hence of 32 nodes.

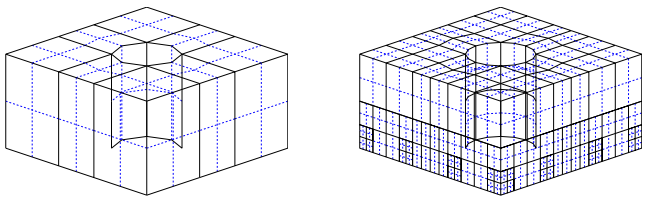


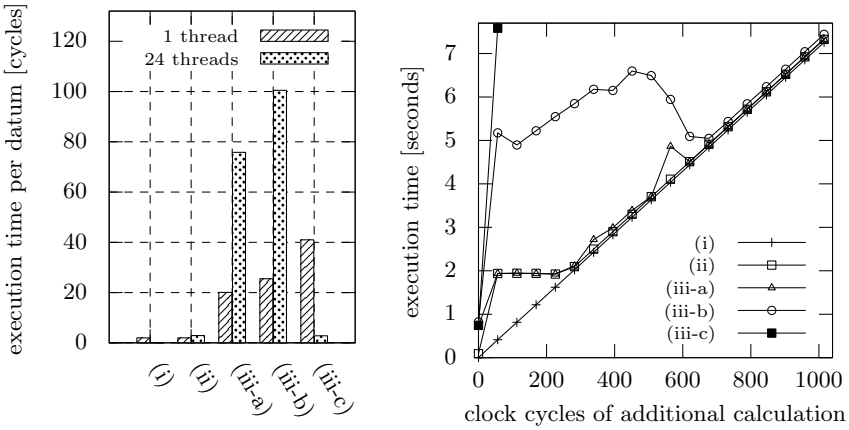
Figure 2: Example object *bohrung* in initial state and after 3 refinement steps

5.1 Synthetic tests

A first test investigated how many CPU clock cycles the AMD machine needs for performing an integer addition $a := a + b$ in the following scenarios: (i) a is a private variable for each thread, (ii) a is a shared variable updated without synchronisation, possibly leading to a wrong result, (iii) a is a shared variable with updates synchronised by (iii-a) an atomic hardware operation, (iii-b) an atomic operation emulated using compare & swap, or (iii-c) explicit locks. A loop performing 100 000 additions was used for the measurement. This loop was executed once by a single thread and once by 24 threads in parallel. The execution time of the loop was measured using hardware performance counters accessed via the PAPI library [3].

The results of this test are shown in Fig. 3a. The value for using a single thread shows how many clock cycles are needed in any case for performing the addition and, if applicable, the synchronisation operation. The value for using 24 threads shows the behaviour of the execution time when there are concurring accesses. If private memory is used, the execution time decreases to $\frac{1}{24}$ of the original value as the addition can be performed in parallel without any conflicts. If shared memory is used without synchronisation, the execution time increases as the updated value of the variable has to be propagated to the caches of all processors, i.e. they have to be kept coherent. If the atomic add hardware instruction is used, the execution time increases as all writes to the result have to be serialised. The increase of the execution time for the emulated atomic instruction using the compare & swap operation is even larger as in the case of conflicts, one thread has to wait using busy waiting for the other threads to finish their write operations. If explicit locks are used, only short waiting times occur, resulting in an execution time decrease to approximately $\frac{1}{15}$.

In a second test on the same AMD machine, a certain amount of computation was performed between two addition operations. The duration of this additional computation was varied. The results for the scenarios (i)–(iii-c) as defined above are shown in Fig. 3b. The curves show the execution time needed by 24 threads



(a) Number of clock cycles needed by the addition for different implementation variants

(b) Execution time for addition operations with additional computation between two operations

Figure 3: Execution times for addition operations on a single memory location: (i) using private memory, (ii) with unsynchronised access, (iii) with synchronised access using (iii-a) an atomic add machine instruction, (iii-b) compare & swap, (iii-c) locks

for 15 million addition operations on one variable including the time for the additional computation. The time for the additional computations is shown in the abscissa. The use of private memory, i.e. without synchronisation, results in a straight line. Compared to that, the synchronised or unsynchronised access to a shared variable causes extra costs. The results indicate that no substantial extra costs are introduced if the atomic hardware operation is used with an additional computation of at least 300 clock cycles or, if the atomic operation using compare & swap is used, with an additional computation of at least 700 clock cycles between two operations. In contrast, the execution time increases rapidly when using the explicit lock.

5.2 Fine-grained explicit locks

The method of using fine-grained explicit locks presented in Sect. 3.2 has been investigated concerning the execution time of the function *ppcgm* when varying the number of lock variables with the example object *bohrung*. Figure 4 shows the execution times for three refinement levels of the FEM consisting of 28 152, 58 736 and 84 720 nodes with data array sizes of 84 456, 176 208 and 254 160, respectively: For all array sizes, the execution time decreases rapidly with an increasing number of lock variables until a value of approximately 240 is reached, whereas it remains nearly constant afterwards. In order to minimise memory consumption, it seems reasonable not to provide more than 240 lock variables.

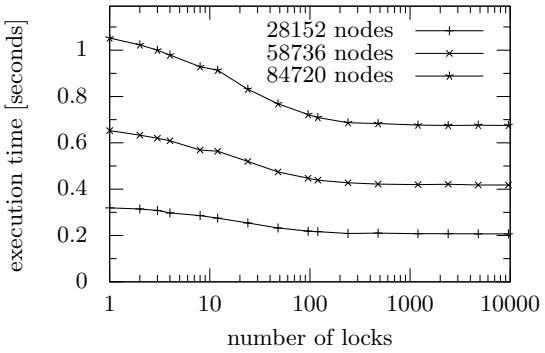


Figure 4: Execution time of the function *ppcgm* depending on the number of lock variables used for different array sizes

5.3 Reduction operations in the FEM implementation

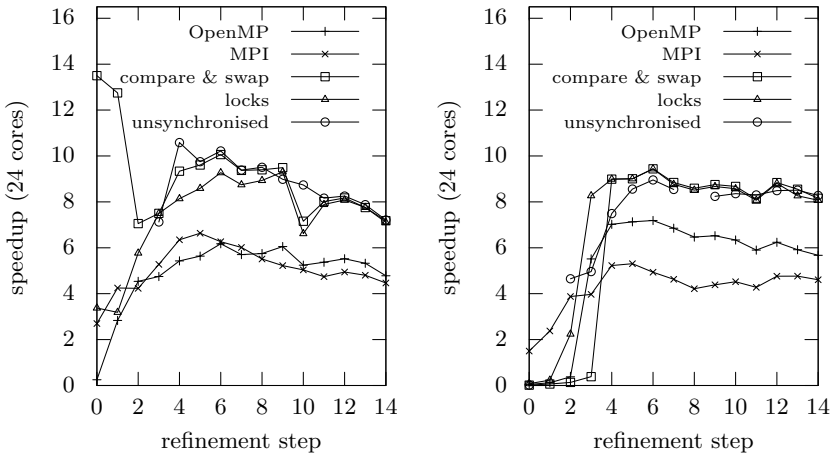
The reduction operation for solving the linear system of equations (3) has been implemented in the function *ppcgm* FEM investigated in the following variants:

(a) coarse-grained reduction according to Alg. 1
  - using a distributed memory model in MPI (MVAPICH2 1.5.1),
  - using a shared memory model in OpenMP (GNU Fortran 4.4.7),

(b) fine-grained reduction
  - with atomic addition operation using compare & swap according to Alg. 2,
  - with explicit locks,
  - without synchronisation (potentially producing wrong results).

As there is no hardware instruction for adding double precision floating point numbers atomically on the available hardware, that variant could not be used. The results obtained with the example object *bohrung* for the Intel and the AMD machines are shown in Fig. 5. The execution times of the OpenMP variant obtained with GNU Fortran did not differ significantly from results obtained using the commercial Intel Fortran compiler.

For the parallel implementation of the function *ppcgm*, which does contain both, sequential and parallel parts, a speedup between 4 and 6 is achieved for the variants with coarse-grained reduction, independent of the memory model used. For the variants with the fine-grained implementation of the reduction using compare & swap or explicit locks, a speedup of approximately 8 is achieved. This speedup is equal to the speedup of the variant with an unsynchronised addition of the results. The results for the variants with fine-grained reduction show that no additional execution time is required for avoiding memory access conflicts. As it appears, the time intervals between two write accesses to an element are large enough to hide the reduction operation between calculations as shown in Sect. 5.1. The waiting time which occurs when using the coarse-grained reduction can be eliminated nearly completely by using fine-grained reduction operations.



(a) AMD machine (b) Intel machine

Figure 5: Speedup of the function *ppcgm* when using the different implementation variants of the reduction operation

6 Related work

Recent works which deal with reductions investigate the reduction of scalars in most cases. They often achieve a runtime benefit by combining operations for implementing the barrier needed by the reduction with the calculation of the reduction result [11, 12]. [12] performs a tree-like reduction which synchronises sibling nodes in the tree using busy waiting. Yet, it does not need atomic read-modify-write or compare & swap instructions. [11] introduces a novel concept of *phaser accumulators* which achieves a runtime benefit by separating phases of the reduction in order to enable overlap between communication and computation.

A molecular dynamics particle simulation, which calculates forces between a number of atoms, is investigated in [8]. The forces are stored in an array and each processor calculates a part of each force acting on a particle. The partial forces are added to the total force acting on the respective particle using a reduction. Among the investigated implementation variants, the variant utilising the OpenMP statement *atomic* performs worst. The variant using the OpenMP statement *reduction* performs better, but still worse than two other variants, one using a private array and one using the BLAS routine `DGEMV`. The results of [8] are contrary to the results of this article, where the application benefits from atomic add instructions to shared arrays. One can assume that this is due to the memory access pattern of the particle simulation which has more frequent write operations to the same memory location than the reduction in the FEM. Similar results where array privatisation yields a performance benefit are, e.g., presented in [5] and [7].

7 Conclusion

This article investigated several implementation variants of a reduction of arrays on shared-memory machines. A fine-grained reduction has been compared to existing implementations of coarse-grained reductions in OpenMP and MPI.

A result of this work is that the operations needed for synchronising write accesses to a shared vector can be hidden between computations if there is enough time between the write accesses. In the parallel routine investigated in detail, the vector being reduced has non-null values for each entry only on a small number of processors and the write operations to the result vector can be interleaved with computations. Thus, the condition mentioned above is fulfilled, and using fine-grained reduction improves the runtime of the adaptive FEM. The results of Sect. 5.3 show that, in contrast to other works which commonly suggest array privatisation, also writing directly to shared arrays can be efficient if fine-grained reduction operations are used.

Acknowledgement: This work is supported by the cluster of excellence *Energy-Efficient Product and Process Innovation in Production Engineering (eniPROD)* funded by the European Union (European Regional Development Fund) and the Free State of Saxony. This work is also part of a project cooperation granted by the German Research Foundation DFG-PAK 97 (ME1224/6-2 and RU591/10-2).

References

1. Beuchler, S., Meyer, A., Pester, M.: SPC-Pm3AdH v1.0 – Programmer's manual. Preprint SFB393 01-08, TU Chemnitz (2001, revised 2003)
2. Basic linear algebra subprograms technical (BLAST) forum standard (2001)
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. 14(3), 189–204 (2000)
4. Case, R., Padegs, A.: Architecture of the IBM System/370. Commun. ACM 21(1), 73–96 (1987)
5. Gao, D., Schwartzentruber, T.: Optimizations and OpenMP implementation for the direct simulation monte carlo method. Comput. Fluids 42(1), 73–81 (2011)
6. Greenwald, M.: Non-blocking synchronization and system design. Ph.D. thesis, Stanford University, Stanford, CA, USA (1999)
7. Liu, Z., Chapman, B., Wen, Y., Huang, L., Weng, T., Hernandez, O.: Analyses for the translation of OpenMP codes into SPMD style with array privatization. LNCS, vol. 2716, p. 26–41. Springer (2003)
8. Meloni, S., Federico, A., Rosati, M.: Reduction on arrays: comparison of performances between different algorithms. In: Proc. EWOMP'03 (2003)
9. Meyer, A.: A parallel preconditioned conjugate gradient method using domain decomposition and inexact solvers on each subdomain. Comput. 45, 217–234 (1990)
10. Ries, D., Stonebraker, M.: Effects of locking granularity in a database management system. ACM Trans. Database Syst. 2(3), 233–246 (1977)
11. Shirako, J., Peixotto, D., Sarkar, V., Scherer, W.: Phaser accumulators: A new reduction construct for dynamic parallelism. In: Proc. IPDPS (2009)
12. Speziale, E., di Biagio, A., Agosta, G.: An optimized reduction design to minimize atomic operations in shared memory multiprocessors. In: Proc. IPDPS, Workshops and PhD Forum (2011)