Journal of Molecular Graphics and Modelling 50 (2014) 134–141

Contents lists available at ScienceDirect

Journal of Molecular Graphics and Modelling

journal homepage: www.elsevier.com/locate/JMGM

Voxel based parallel post processor for void nucleation and growth analysis of atomistic simulations of material fracture

H. Hemani*, M. Warrier, N. Sakthivel, S. Chaturvedi

Bhabha Atomic Research Centre, IDA Block B, 4th Cross Road, Visakhapatnam, Andhra Pradesh 530012, India

ARTICLE INFO

Article history:
Accepted 4 April 2014
Available online 16 April 2014

Keywords:
Voxel
Void nucleation and growth
Parallel algorithm
Molecular dynamics
Fracture

ABSTRACT

Molecular dynamics (MD) simulations are used in the study of void nucleation and growth in crystals that are subjected to tensile deformation. These simulations are run for typically several hundred thousand time steps depending on the problem. We output the atom positions at a required frequency for post processing to determine the void nucleation, growth and coalescence due to tensile deformation. The simulation volume is broken up into voxels of size equal to the unit cell size of crystal. In this paper, we present the algorithm to identify the empty unit cells (voids), their connections (void size) and dynamic changes (growth and coalescence of voids) for MD simulations of large atomic systems (multi-million atoms). We discuss the parallel algorithms that were implemented and discuss their relative applicability in terms of their speedup and scalability. We also present the results on scalability of our algorithm when it is incorporated into MD software LAMMPS.

1. Introduction

Molecular dynamics (MD) is a widely used tool to study the deformation and fracture of materials subjected to high strain rates [1–4]. MD simulations generate atom positions at various time instances as an output. Typically, for single crystals, these simulations consist of the dynamics of a few thousands to several millions of particles for several hundred pico-seconds with the atomic positions output at least every tenth of a pico-second [5]. A few million particles are necessary to avoid finite size effects in tri-axial deformation of copper [5] and the choice of 0.1 pond as the frequency of output makes sure that individual atomic oscillations are smeared out. These atom positions at various times need to be post processed to find the number of nucleated voids, their connectivity (void size) and how they coalesce. This information is used to fit a macroscopic void nucleation and growth (NAG) model [4,6] as part of a hierarchical multi-scale model [7,8].

The existing methods for void analysis of atomistic simulation output are based on Voronoi (or Delaunay) tessellation [9-17]. Voronoi tessellation algorithms are an extensively studied topic in computational geometry. A Voronoi tessellation on a set of points in three dimensional space is a graph representation called a Voronoi network. The atomic positions in a MD simulation can correspond to these points. The Voronoi network obtained after tessellation consists of Voronoi cells. Each atom in the crystal corresponds to a Voronoi cell. A Voronoi cell is constituted by points, whose distance from the corresponding atom center, is less than or equal to its distance from any other atom in the crystal [10]. The MD software, LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [18] uses the Voro++ package [19] to carry out Voronoi tessellation on the atoms. For each Voronoi cell, it outputs its volume and the number of faces. Since the atom sizes are known, one can analyze the Voronoi (or Delaunay) network to obtain the characteristics of the voids [9,19]. Void size is typically characterized by the diameter of the largest sphere that can accommodate the void [20]. At times, it is also interesting to know the size of the largest sphere that can diffuse through in a void. A number of algorithms for Voronoi tessellation can be found in [16,17]. A parallel algorithm, which is essential for performing Voronoi tessellation on large domains, typically with about a billion particles on a 557-teraop IBM BlueGene/P (BG/P) supercomputer, is described in [21].

The worst case complexity (how computation time varies with input size) of the best known Voronoi tessellation algorithm for points in a three dimensional space is $O(n^2)$ and the expected time complexity is $O(n)$ [22]. The Voronoi tessellation is followed by calculation of largest fitting sphere for calculating the void size, which further increases the computation cost. Furthermore, MD simulations may output atom positions over several thousands of time

* Corresponding author. Tel.: +91 891 2892189.
E-mail addresses: harshhemani@live.com (H. Hemani), manoj.warrier@gmail.com (M. Warrier), shakthiveln@gmail.com (N. Sakthivel), shashankvizag@gmail.com (S. Chaturvedi).

http://dx.doi.org/10.1016/j.jmgm.2014.04.004
1093-3263/

steps, depending upon the problem. Therefore, this process needs to be repeated for each such output time step in order to obtain void nucleation, growth and coalescence parameters. Such an analysis by a Voronoi tessellation based algorithm involves large number of floating point operations and are costly in terms of computational time. They however have the advantage of being accurate. For MD simulations involving domains having multimillion atoms simulated over several hundreds of time steps, techniques such as Voronoi tessellation, prove to be very expensive as discussed in Section 3.

In this paper, we propose a voxel based algorithm for void size analysis which is faster than Voronoi tessellation. It is the practical solution for analyzing void nucleation, growth and coalescence in large domains [23,24]. The size of the domain that can be analyzed limited by the memory size in the system. Since the domains can be very large (billions of atoms), and the simulation is run over typically hundred-thousands of time-steps, we must complete void analysis in a reasonable time. Therefore, we present three parallel versions of the algorithm, each suitable in some particular situations. Since domain output is obtained at multiple time steps, the first parallel algorithm deals with the problem by distributing the work among the processors from different output time-steps. The second and third parallel algorithms are based on domain decomposition. In this, the domain at each time-step is decomposed and work is distributed among different processes. The first two parallel algorithms are intended to reduce the computation time while the third one is mainly targeted to handle the restriction on the domain size imposed by the maximum memory available on a single computer.

We introduce the concept of voxelization in Section 2. We discuss the need of voxelization and the algorithm for performing it over the simulation domain. We discuss the serial algorithm for performing the analysis of void nucleation, growth and coalescence in Section 3. The three parallel algorithms for void analysis are discussed in Section 4. We present the results and discussions in Section 5. Conclusions are presented in Section 6.

2. Voxelization of domain

The output of an MD simulation, for a single time step, consists of coordinates of the atoms in three dimensional space. At any instant, a crystal consists of atoms located at fixed positions. Now, depending on how one chooses to define a void, the Voronoi graph of the system can be analyzed to calculate the void volumes. Creation and analysis of Voronoi graphs is computationally expensive as we deal with real valued 3 dimensional space. In order to reduce the computational time, we discretized the domain into fixed sized voxels. A voxel or *volume element* is a 3D analogue of a 2D pixel or *picture element*. It is a rectangular cuboid geometric element.

We then classify each voxel as empty or occupied. Each empty voxel corresponds to a void. All the connected empty voxels can then be analyzed to determine the number of voids and their size. The size of a voxel can be selected based on how one chooses to define a void. We choose the voxel size to be equal to the unit cell size in the crystal. We characterize a voxel to be empty if no atom center lies its corresponding unit cell volume. Even if a single atom center belongs inside the unit cell corresponding to a voxel, it is considered to be occupied. Therefore, in this case, the smallest void is of the size of a unit cell. The implication of using this definition for a void is that the computed void volume could be less than the void volume if one would have considered a smaller voxel size. For example, the small void formed when a single atom shifts within its unit cell by a distance lesser than the voxel size will not be accounted for in this formulation.

Assuming the voxels to be cuboidal makes it easier to check if the coordinates of an atom lie inside it. Checking this for non-cuboidal voxels could be complex. For the sake of simplicity, we restrict our discussion to the cases where voxels are rectangular cuboids. This method is extensible to non cuboidal voxels. The simulation domain consisting of atoms is to be discretized into voxels. The number of voxels is decided in such a way that the entire simulation domain is covered. The number of voxels along the three principle axes (*X*, *Y* and *Z*) are denoted by Nx, Ny and Nz respectively. Each voxel is associated with a label, which is an integer value. These labels are used to denote if the associated unit cell is occupied or empty. Empty voxels are assigned a label 0. A non-empty voxel is assigned a label 1. A three dimensional array is used to store the label values of the voxels. Let us call this array *OccFlag* (Occupancy flag). The occupancy array is defined as *Integer OccFlag[Nx][Ny][Nz]*.

The process of converting the atom domain into voxel domain by discretization is called voxelization. The output of the voxelization step is an array (OccFlag) which tells the occupancy state of each voxel in the domain. This method is described in Algorithm 1. The time complexity for the voxelization is $O(n)$. Here n refers to the number of atoms in the domain. This step needs to be repeated for each time step. Voxelization is directly followed by the void analysis.

Algorithm 1. Voxelization algorithm

Input: List of atom coordinates (x,y,z)
Output: OccFlag array: showing status of each unit cell
for $i \leftarrow 1$ **to** Nx **do**
  for $j \leftarrow 1$ **to** Ny **do**
    for $k \leftarrow 1$ **to** Nz **do**
      OccFlag[i][j][k] = 0;
    end
  end
end
for *each atom* A_i, *having coordinates* (x_i, y_i, z_i) **do**
  $i = \lfloor \frac{x}{ax} \rfloor$;
  $j = \lfloor \frac{y}{ay} \rfloor$;
  $k = \lfloor \frac{z}{az} \rfloor$;
  OccFlag[i][j][k] = 1;
end

3. Void analysis algorithm: serial approach

Void analysis is the procedure used for calculating the total number of voids, the size of the each void (void volumes), and the fraction of voids in a given domain. We are interested in computing the temporal evolution of voids. When two neighboring voxels are empty, they constitute a single void. All connected empty voxels are explored. Together, they constitute one single void. The size of the void is calculated by counting the number of connected empty voxels. A domain can have several such voids each constituted by multiple voxels. In order to distinguish them from each other, and to have a deterministic exploration algorithm, each void is to be assigned a unique label (>1), i.e. while exploring a void, the empty voxels that have been processed are assigned a label by setting their corresponding OccFlag value. The voxels with a label 0 denote that they are yet to be processed.

All the voxels that correspond to a single void need to be assigned the same label. We have initialized the OccFlag array using label 0 for empty voxel and label 1 for occupied voxel. We scan through this array searching for label 0. When we find the label to be 0 for the first time, we update its value to label 2. All its neighboring empty voxels are assigned the same label. The neighborhood of a voxel is defined being the six voxels with which shares a face with this voxel. We refer to these voxels as *front*, *back*, *top*, *bottom*, *left*, and *front*. This process is repeated with the neighboring voxels

of these voxels until all such connected empty voxels have been assigned the new label.

After this, the scan through OccFlag continues to search for label 0. The same procedure is repeated again, but now we assign them a label which is one higher than the last label. Once we have scanned through whole OccFlag array, the number of entries corresponding to a particular label number represent the size of that void. Number of voids can be calculated by counting the number of unique labels greater than 1 in the OccFlag array. This procedure is shown in Algorithm 2. Other interesting information, like void shape (morphology) and void locations can also be extracted in this process.

Global Variables:
OccFlag: A 3-dimensional integer array representing the occupancy state of each unit cells.
VoidSize: A 1-dimensional array which gives void size corresponding to each label.
Count: Global integer variable which holds the current value of number of connected voids

Algorithm 2. Main() Routine of the serial algorithm

```
Input: OccFlag[ ][ ][ ]: Integer
Output: VoidSize[ ]: Integer
label:=2;
for i ← 1 to Nx do
    for j ← 1 to Ny do
        for k ← 1 to Nz do
            if OccFlag[i][j][k]==0 then
                VoidSize[label]:=VoidAnalysis(i,j,k,label);
                label:=label+1;
            end
        end
    end
end
```

Algorithm 3. VoidAnalysis() subroutine

```
Input: i,j,k,label: Integer
Output: count: Integer
OccFlag[i][j][k]:=label;
count:=1;
CheckFront(i,j,k,label);
CheckBack(i,j,k,label);
CheckRight(i,j,k,label);
CheckLeft(i,j,k,label);
CheckTop(i,j,k,label);
CheckBottom(i,j,k,label);
return(count);
```

Algorithm 4. CheckFront() subroutine

```
Input: i,j,k,label: Integer
Output: count: Integer
if i==Nx then
    return;
end
if OccFlag[i+1][j][k]==0 then
    OccFlag[i+1][j][k]:=label;
    count:=count+1;
    CheckFront(i+1,j,k,label);
    CheckRight(i+1,j,k,label);
    CheckLeft(i+1,j,k,label);
    CheckTop(i+1,j,k,label);
    CheckBottom(i+1,j,k,label);
end
```

In this algorithm, the size of the void currently being explored is stored in a global variable count. The CheckFront() subroutine shown checks if the voxel in front of the current voxel is empty. If it finds it to be empty, it increments the count (which represents the void size of the void that is currently being explored) and explores it further by calling the other subroutines recursively. The other subroutines (CheckBack, CheckRight, CheckLeft, CheckTop and CheckBottom) have not been shown but are defined in a similar fashion. Notice that the routine *CheckFront* does not call the routine *CheckBack*. By doing so, we avoid unnecessary processing of an already processed voxel. Similar pattern of calls is maintained while defining the other routines as well.

One can observe that the process of void exploration is similar to a depth first graph search. While visiting each void voxel, its label is changed to indicate that the void is already accounted for. The number of void voxels visited is also counted. This gives us the total void volume. The size of each void is stored in a list with its corresponding label number. Since the voxels in the domain are visited exactly once, algorithmic complexity is $O(n)$, where n is the number of voxels. Fig. 1 shows how void analysis procedure works in a 2-dimensional domain.

Once all voxels are processed, the total number of voids in the domain is obtained by counting the total number of distinct labels which are greater than 1. The void size is calculated by counting the number of voxels corresponding to each label. This process is repeated for each time step and the number of voids, the corresponding void sizes and their evolution with time is obtained.

To compare this algorithm with Voro++, tests were carried out on a set of 10 million random points in a three dimensional space. To ensure that the problem size was similar for both the algorithms, the domain was discretized into 10 million voxels. The codes were compiled using GNU C compiler (version 4.6.6). The time taken by the Voro++ (using periodic boundary condition) was 42 min. It must be noted that Voronoi tessellation must be followed by a void analysis procedure, which would increase the computational time further. The time taken by our serial code (using periodic boundary condition) was 2.53 s. This clearly shows the advantage of using the voxel based approach. In case the voxel size is larger, the number of voxels would reduce and time taken will further decrease. To further illustrate the applicability of the voxel based method to large systems, a 1 billion atom configuration was analyzed for void size/shape on a single processor with 32 GB RAM. The simulation took 1454 s and 95 percent of the time was spent on input/output (I/O). The atomic positions in this case were randomly generated and therefore resulted in relatively smaller voids, which are faster to analyze. Therefore, in order to analyze large systems (>10's of billions of atoms) spanning several timesteps, a need to parallelize the algorithm exists.

4. Parallel algorithms

The serial version of the algorithm for void analysis, as discussed in Section 3 was parallelized using three different approaches. The parallel code was written using the MPI-C library which runs on a cluster computer with 60 nodes (8 processor cores per node). We assign each MPI process to exactly 1 processor core. Data from atomistic simulations is available in the form of an ASCII text file. However, as we will see later, data from the LAMMPS simulation can read directly when we integrate our algorithm as a module in LAMMPS. This data consists of coordinates of the atoms at each time step in the simulation. The atom position data corresponding to different time steps is readily exportable into separate files.

In the first parallel algorithm, we divide the data from different time steps amongst different processes. In this scheme, the rank 0 process (called the head process), is reserved for distributing work amongst the other processes, (worker processes). The data corresponding to each time step is available in a separate file. The head process is given the task of specifying the name of an unprocessed file (which corresponds to a single time step) when a worker process asks for work. Dividing equal number of files among all

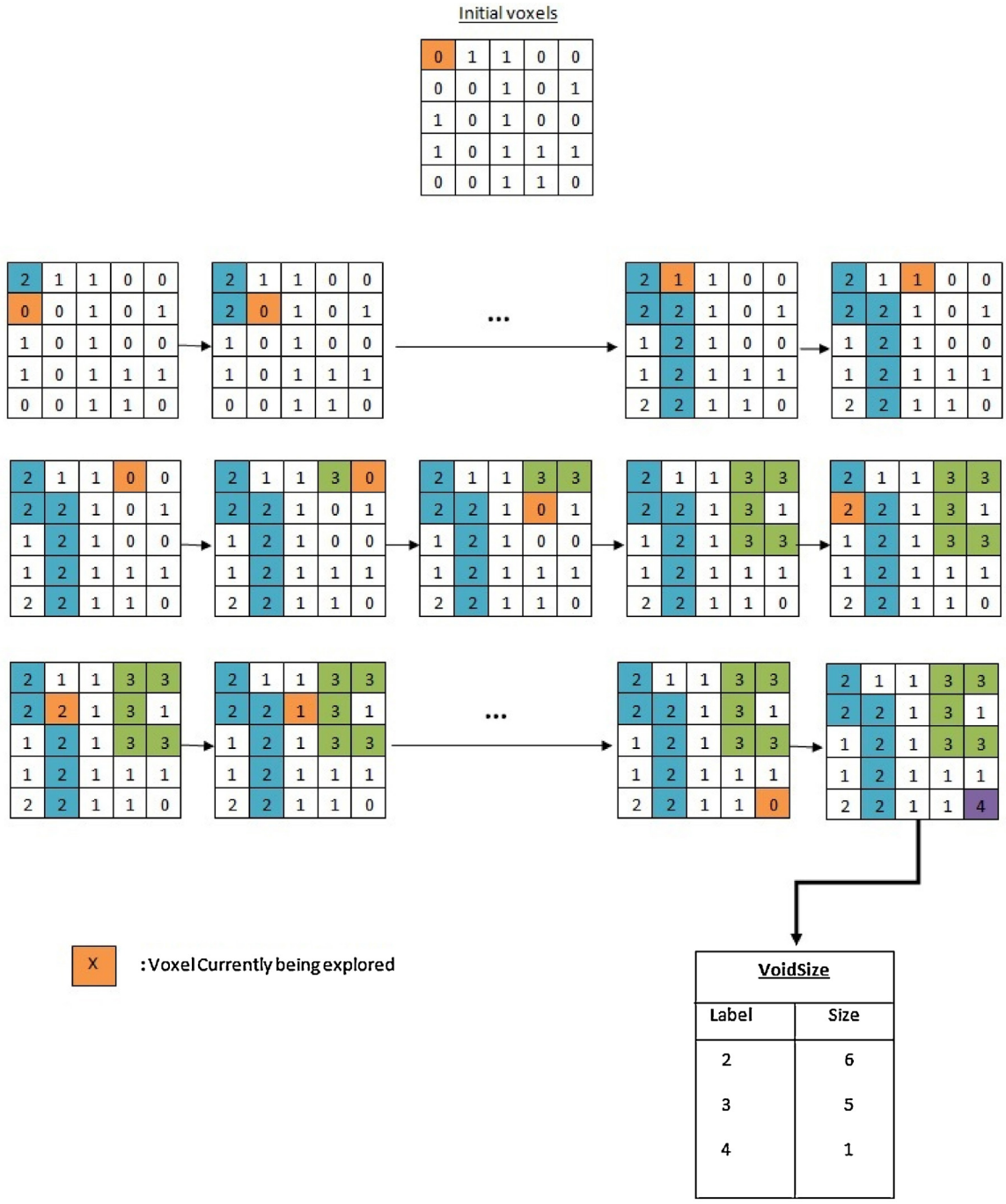


Fig. 1. Void analysis using serial algorithm.

processes seems to be a straightforward approach, but it lacks load balancing. Data from different time-steps have different distribution of voids, and hence would take different amount of times to be post processed. Simply dividing equal number of files among all processes would cause some processes to finish early while some others would still be busy. In our algorithm, load balancing was achieved as follows: each time a worker process completes post processing a domain, it asks the head process for more work. The head process keeps track of the work remaining and assigns the next available domain to be processed to it. When data corresponding to all time-steps has been post processed, the head process responds to each request from the worker processes with a signal, which asks the worker process to exit the processing loop. After all worker processes have been signaled to exit, the head process also exits.

This is a simple but effective approach. The performance improves almost linearly with the number of worker processes. The scalability of this approach is limited by the number of time steps in the simulation. The overall time in void analysis is reduced, but the time for processing the data of a single time step is not changed. This is done by the second approach.

The second parallel algorithm is based on pseudo domain-decomposition. In this, the voxelized domain at each timestep is divided into P sub-domains amongst the P processes. Each sub-domain shall have an integer number of voxels. It is also required that each sub-domain is a cuboid. Each process is assigned atleast

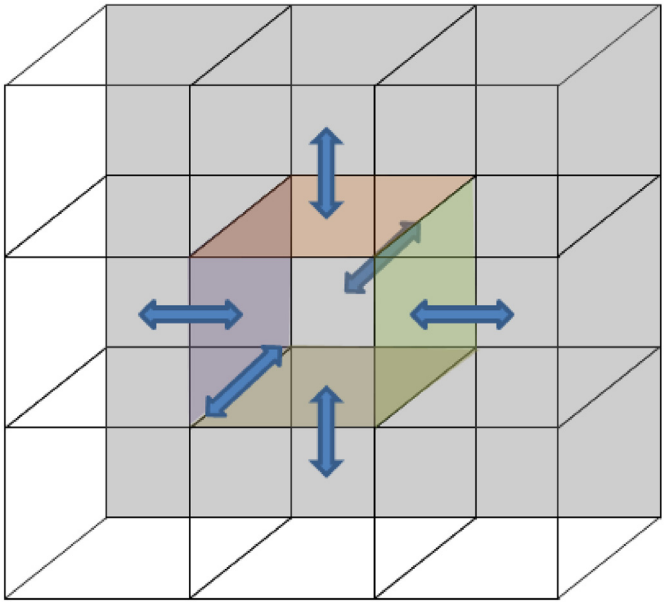


Fig. 2. Sharing data with neighbors.

one sub-domain for void analysis. A process may need to analyze the sub-domain assigned to some other process under certain conditions. Therefore, each process must have the data pertaining to the entire domain. This is why we call this decomposition pseudo. Initially, each process would perform void analysis on its own sub-domain by the serial algorithm listed in the previous section. There could be voids that are shared among the sub-domains. One way of calculating the size of the shared void is to allow the process to explore a void even if it has to access the voxels which belong to the sub-domain of some other process. In order to avoid exploration of the same void spanning different sub-domains by different processes, they are assigned priorities. Each process is associated with a unique rank. Processes with lower rank will have priority over those with higher rank when exploring a shared void. That is, a process with rank p, would be allowed to read the data assigned to the sub-domain of process with rank q, only if $p < q$. Assume that shared void is being explored by process q, and it observes that the void extends into the sub-domain of process p, given that $p < q$, it would stop exploring it and report the size of the corresponding void as 0.

This algorithm handles data pertaining to one time-step at a time and not multiple time-steps at once as in the first approach. When sub-domains do not share any voids, the performance improvement is expected to scale linearly. The worst-case scenario for this algorithm is when entire domain is empty. In this case, the process with the lowest rank (highest priority) would have to explore the entire domain to compute the void size. Hence it would take time equal to that of the serial algorithm.

The first two parallel algorithms have a limitation. The largest domain size that can be analyzed is limited by the amount of memory available on the computer. We try to eliminate this limitation with the third parallel algorithm which is based on pure domain-decomposition. Unlike the second approach, where all processes needed to know the occupancy status of the entire domain, here each process needs to have knowledge of its own sub-domain only. In this approach, each process will initially explore its own sub-domain and calculate the void sizes while assigning the labels to the voids. After this, each process will exchange the occupancy status of the voxels on each of the six faces of its sub-domain with the corresponding neighboring processes. This is shown in Fig. 2.

By exchanging the face-voxels with neighboring processes, each process computes a connectivity table. This connectivity table lists the relation between the labels corresponding to shared voids between two processes. For example, voxel at location i, j, k on one of the faces of the sub-domain has label 4 and the voxel belonging to location $i-1, j, k$, which belongs to a process whose sub-domain lies to the right of the current process, has a label say, 56. The process will have an entry in the connectivity table saying that its label 4 is same as label 56 in sub-domain of process with rank, say 5. Once this connectivity information is gathered, each process will try to find the size of the shared voids on their interfaces. This involves exchange of messages to determine the size of the shared voids. To minimize communication and computation, we restrict all processes from computing the sizes of all the shared voids they have noticed in their sub-domains. We allow a process to explore a shared void until it sees that the void is extending into the sub-domains of processes that has a lower rank than its own rank. As soon as it finds its void to be shared with a lower rank process, it stops calculating its size and marks it as invalid. In this way, for a void that is shared amongst a group of multiple processes, only the lowest ranked process in the group will report the size of the shared void. Similarly, if a void is found to loop among different sub-domains and comeback to initial sub-domain (possibly with a different label), the lowest label will finally report the void size while higher ones will discard that void.

With this simple approach, we eliminate the memory limitation as faced by the previous two algorithms. This algorithm can hence be used to perform void analysis of even larger domains typically having several billions of atoms. As with the second approach, the performance would degrade when the number of voids shared among sub-domains increase, as this would mean that more data has to be exchanged among processes. Also, as the length of a shared voids increase, more computations are needed for exploring the connectivity.

Each algorithm is better than the others in some scenarios. The first approach, as we shall see in next section, gives better performance than the other two algorithms on our test problem. The second parallel algorithm will be faster than the other two parallel approaches when the sub-domains do not share any voids. The third parallel algorithm is suitable for analyzing large domains, where the other two algorithms fail due to limited memory on a single machine.

5. Results and discussion

To test the relative performance of the algorithms (serial and parallel) a Molybdenum sample was subjected to isotropic tensile deformation at a strain rate of 10^9 s^{-1} for 20 ps. The domain consists of 20 Million atoms, with periodic boundary conditions with a time step of 1 fs. The position data was output every 500 time-steps.

The four algorithms (one serial and three parallel) were run on the same data and void analysis results are obtained and found to match precisely. The total void volume with time is shown in Fig. 3. Note that soon after initial nucleation of voids, the void volume exponentially increases and again grows linearly as expected [4,6]. The evolution of the number of voids with time is shown in Fig. 4. The number of voids initially increases, then goes through a maximum and falls off. This happens as the total void volume is continuously increasing, and is a sign that void coalescence is taking place.

The time taken by voxelization algorithm is proportional to the number of atoms in a domain. The time taken by the serial algorithm for performing the void analysis is proportional to the number of empty voxels or unit cells in the domain. In the worst case, all voxels in the domain could be empty, and the time

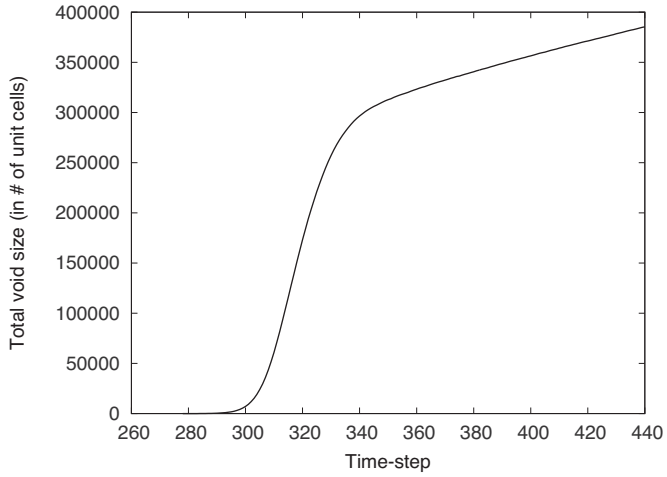


Fig. 3. Void size distribution.

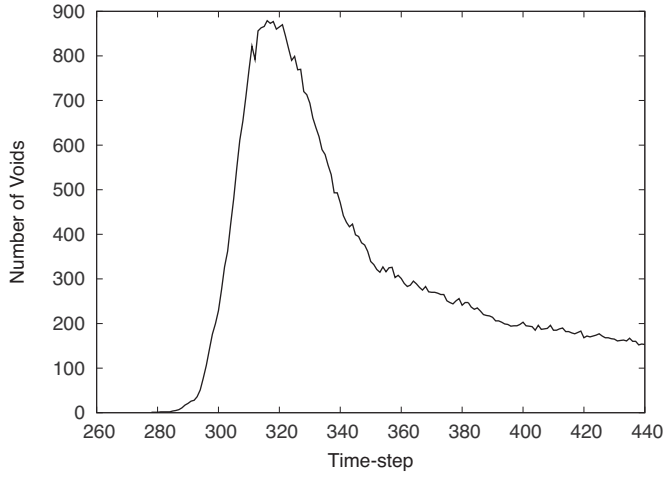


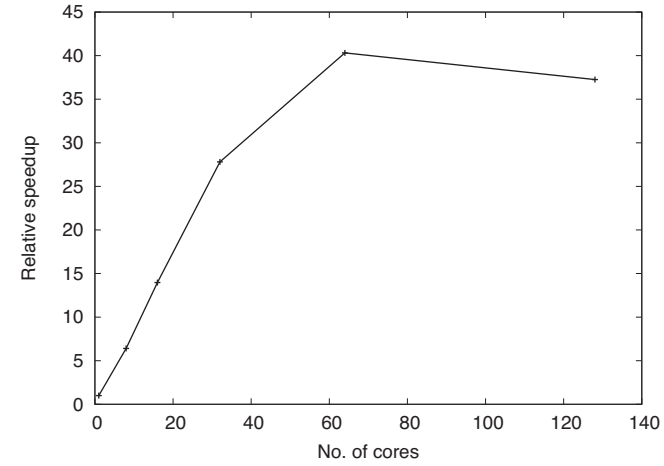
Fig. 4. Evolution of number of voids.

required for void analysis becomes proportional to the total number of voxels. This gives us an upper bound on the time required. The first parallel scheme is a straight-forward extension of the serial scheme. It is limited in its scalability only by the number of timesteps for which the simulation was done. Moreover, the size of the domain which can be analyzed is restricted by the amount of memory available in each machine in the cluster.

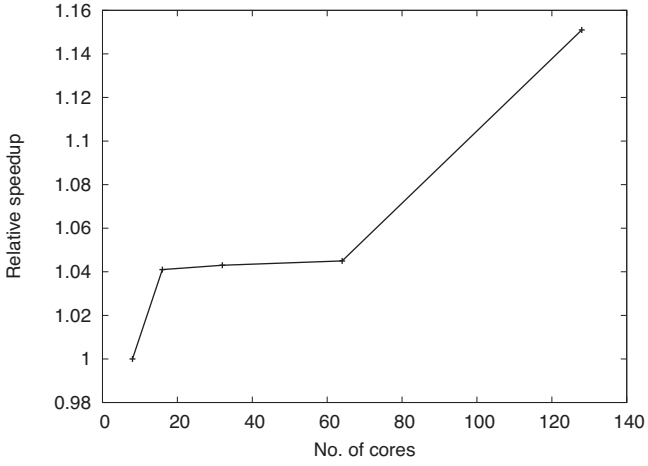
The second approach is basically to get an improvement in speed of void analysis. Here the work is divided amongst multiple processes, while storing the information about the entire domain at each process. This gives good results in the initial stages of simulation where voids are not big and do not extend over several sub-domains. But as the number of shared voids across domains become big and extends over several sub-domains, the performance degrades. This approach is also limited by the memory available for each process.

The third approach is scalable and can accommodate any number of atoms by decomposing the domain accordingly. The problem with this approach is the amount of communication needed for exchanging shared void information. The overhead is the amount of memory needed to store the connectivity information. The speedup plots for the three parallel algorithms are shown in Fig. 5.

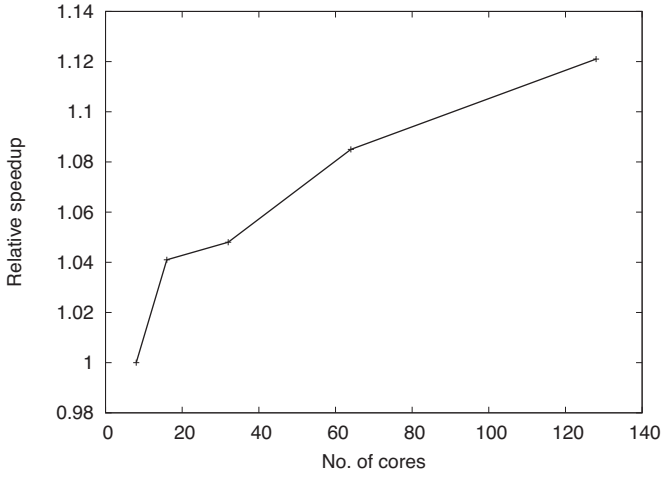
The explanation for the speedup behavior of different algorithms is as follows: The most crucial problem with all the four approaches is that the amount of I/O operations involved in reading



(a) Speedup for temporal parallelization



(b) Speedup for pseudo domain decomposition parallelization



(c) Speedup for domain decomposition parallelization

Fig. 5. Speedup plots.

the atoms positions is the most time consuming task here. One can observe from the speedup plots of the parallel approaches that there is almost no speed-up seen for the second and the third approach. It was found that more than 98 percent of the time required by these implementations was spent in reading the data file. The actual void analysis took less than 2 percent of the time. The reason for good speedup achieved by the first approach is as

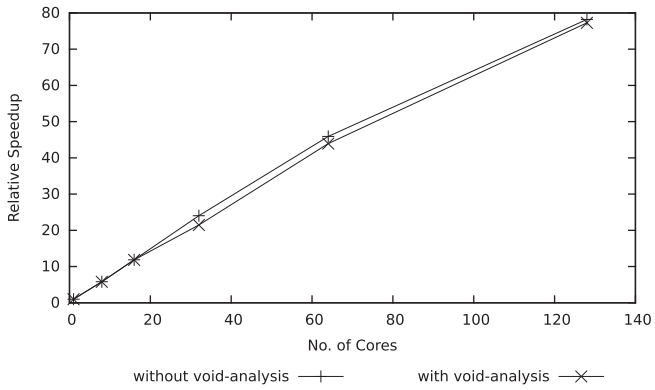


Fig. 6. Comparison of simulation speedup with and without void analysis enabled.

follows: The data in the compute cluster we use is stored on a Network File System (NFS) server. All storage space needed by the program is provided on it. In the first parallel approach, the data corresponding to different time-steps is present in separate data files. All files are stored on the NFS server. When the head process has to assign work to a worker process, it only specifies which file the worker would needs to read. When a worker tries to read the file from the NFS server, it caches the file directly on its local storage. This helps in reducing the congestion that would occur in case when all processes read from same disk.

In case of the other two approaches, the data of all time-steps is present as a single file. This is because we have performed void analysis of one time-step data at a time. This means, the time required by these two parallel algorithms would be governed by the serial read speed of a very large file. The size of the data file for this test problem is 240 GB. This problem of spending most of the time in IO operations can be eliminated, if the post processor is embedded into the MD Simulation software. In that case, it would pick up the data values directly from the memory for processing hence eliminating the need of heavy IO entirely. We implemented our third parallel strategy (domain decomposition) into LAMMPS software as a compute module. We utilized the existing data-structures of LAMMPS to obtain atom positions in each sub-domain to perform void analysis. Rest of the algorithm is similar to what is described in Section 4. The void nucleation and growth parameters were obtained in the LAMMPS log file. We compared the time taken by LAMMPS simulation with and without our void analysis module for a domain of 1,024,000 atoms using different number of processes to find out how scalable our code is. The results are presented in Fig. 6. One can observe that scaling is almost linear in both cases. It is also seen that maximum speedup achieved here is better than all the previous cases. Implementing void analysis module into LAMMPS has a clear advantage over implementing it as a post processor of large data files, because a lot of time is saved by averting disk access.

6. Conclusion

A voxel based approach to determining void nucleation and growth from MD simulation output is proposed. It has been used to calculate void sizes with reasonable accuracy. The evolution of void sizes and the number of voids conform to what is expected by the NAG model. It is seen that Voronoi tessellation is much slower in void analysis as compared to the voxel based approach. Moreover, after Voronoi tesselation, further post processing has to be carried out to obtain void sizes. In the voxel based method, a 1 billion atom simulation was analyzed for void sizes for a single time step, on a single processor, in 1454 s. It is seen that the time spent in reading the atom positions from the file took 95 percent of the time. To avoid this problem, we recommend to do the computation in-memory.

Three parallel algorithms have been explored. It is the first parallel approach (dividing work amongst processes such that different time steps can be analyzed by different processes) shows better speedup than the other two algorithms. But this is true only when void analysis is implemented as a post processor on large data files. When the third algorithm (domain decomposition based) is incorporated into LAMMPS, we see that speedup is even higher. The third algorithm has the advantage of handling systems N times bigger than the other two methods because it is no more limited by the memory available on a single machine. Here N is the number of available processors. The advantage of incorporating the void analysis algorithm into LAMMPS is that computation is done in-memory avoiding the IO time entirely and saves storage space.

References

[1] J. Belak, On the nucleation and growth of voids at high strain-rates, J. Comput. Aided Mater. Des. 5 (1998) 193–206.
[2] E.M. Bringa, J.U. Cazamias, P. Erhart, J. Stölken, N. Tanushev, B.D. Wirth, R.E. Rudd, M. Caturla, Atomistic shock Hugonoit simulation of single crystal copper, J. Appl. Phys. 96 (2004) 3793–3799.
[3] S.G. Srinivasan, M.I. Baskes, G.J. Wagner, Spallation of single crystal nickel by void nucleation at shock induced grain junctions, J. Mater. Sci. 41 (2006) 7838–7842.
[4] S. Rawat, M. Warrier, S. Chaturvedi, V.M. Chavan, Effect of material damage on the spallation threshold of single crystal copper: a molecular dynamics study, Modell. Simul. Mater. Sci. Eng. 20 (2012) 015012.
[5] S. Rawat, M. Warrier, S. Chaturvedi, V.M. Chavan, Temperature sensitivity of void nucleation and growth parameters for single crystal copper: a molecular dynamics study, Modell. Simul. Mater. Sci. Eng. 19 (2011) 025007.
[6] D.R. Curran, L. Seaman, L. Shockey, Dynamic failure of solids, Phys. Rep. 147 (1987) 253–388.
[7] S. Rawat, V.R. Ikkurthi, M. Warrier, S. Chaturvedi, V.M. Chavan, R.J. Patel, Computation of spall data from atomistic simulations with pre-existing defects, in: 9th Int. Conference on New Models and Hydrocodes for Shock Processes in Condensed Matter (NMH-2012), London, April 23–27, 2012.
[8] V. Ikkurthi, S. Rawat, R. Sugandi, M. Warrier, S. Chaturvedi, Multi-scale model for single crystal spallation at high strain rates, in: 4th Int. Congress on Computational Mechanics and Simulation (ICCMS), IIT Hyderabad, 10–12 December, 2012.
[9] M. Pinheiro, R.L. Martin, C.H. Rycroftabcd, M. Haranczyk, High accuracy geometric analysis of crystalline porous materials, CrystEngComm 15 (2013) 7531–7538.
[10] H. Baaser, D. Gross, Analysis of void growth in a ductile material in front of a crack tip, in: 11th International Workshop on Computational Mechanics of Materials, Volume 26, January, 2003, pp. 28–35.
[11] S. Bhattacharya, K.E. Gubbins, Fast method for computing pore size distributions of model materials, Langmuir 22 (2006) 7726–7731.
[12] M.F. Sanner, A.J. Olson, Reduced surface: an efficient way to compute molecular surfaces, Biopolymers 38 (3) (1996) 305–320.
[13] T.F. Willems, C.H. Rycroft, M. Kazi, J.C. Meza, M. Haranczyk, Algorithms and tools for high-throughput geometry-based analysis of crystalline porous materials, Microporous Mesoporous Mater. 149 (2012) 134–141.
[14] A.V. Anikeenko, M.G. Alinchenko, V.P. Voloshin, N.N. Medvedev, M.L. Gavrilova, P. Jedlovszky, Implementation of the Voronoi–Delaunay method for analysis of intermolecular voids, in: Computational Science and Its Applications – ICCSA 2004, Lecture Notes in Computer Science, 2004, pp. 217–226.
[15] M.G. Alinchenko, A.V. Anikeenko, et al., Morphology of voids in molecular systems. A Voronoi–Delaunay analysis of a simulated DMPC membrane, J. Phys. Chem. B 108 (2004) 19056–19067.
[16] P. Su, R.L. Scot Drysdale, A Comparison of Sequential Delaunay Triangulation Algorithms, http://www.cs.berkeley.edu/jrs/meshpapers/SuDrysdale.pdf
[17] M.J. Golin, H.-S. Na, On the average complexity of 3d-Voronoi diagrams of random points on convex polytopes, Comput. Geom. Theor. Appl. 25 (2003) 197–231, urlhttp://www.cse.ust.hk/golin/pubs/3D_Voronoi_I.pdfhttp://www.cse.ust.hk/golin/pubs/3D_Voronoi_I.pdf.
[18] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comp. Phys. 117 (1995) 1–19 http://lammps.sandia.gov/index.html
[19] C.H. Rycroft, VORO++: a three-dimensional Voronoi cell library in C++, Chaos 19 (4) (2009), http://dx.doi.org/10.1063/1.3215722 (1 p., Gallery of Nonlinear Images Chaos 19, 041111).
[20] T.F. Willems, C.H. Rycroft, M. Kazi, J.C. Meza, M. Haranczyk, Algorithms and tools for high-throughput geometry-based analysis of crystalline porous materials, Microporous Mesoporous Mater. 149 (1) (2012) 134–141, http://dx.doi.org/10.1016/j.micromeso.2011.08.020, ISSN: 1387-1811, pii:S1387181111003738.

[21] T. Peterka, J. Kwan, A. Pope, H. Finkel, K. Heitmann, S. Habib, Meshing the universe: identifying voids in cosmological simulations through in situ parallel Voronoi Tessellation, Conference Paper ANL/MCS-P 2087-0512, 2012.
[22] R.A. Dwyer, Higher-dimensional Voronoi diagrams in linear expected time, Discrete Comput. Geom. 6 (1) (1991) 343–367.
[23] R.K. Kalia, A. Nakano, A. Omeltchenko, K. Tsuruta, P. Vashishta, Role of ultrane microstructures in dynamic fracture in nanophase silicon nitride, Phys. Rev. Lett. 78 (11) (1997) 2144–2147.
[24] J. Ashburner, K.J. Friston, Voxel-based morphometry – the methods, Neuroimage 11 (6) (2000) 805–821.