Available online at www.sciencedirect.com
SCIENCE DIRECT
Journal of Computational Physics 215 (2006) 417–429
JOURNAL OF COMPUTATIONAL PHYSICS
www.elsevier.com/locate/jcp

Numerical methods to improve the computing efficiency of discrete dislocation dynamics simulations

C.S. Shin [a,*], M.C. Fivel [b], M. Verdier [c], S.C. Kwon [a]

[a] *Nuclear Material Technology Division, Korea Atomic Energy Research Institute, 150 Dukjin-dong, 305-353, Yuseong-gu, Daejeon, Republic of Korea*
[b] *Génie Physique et Mécanique des Matériaux, Institut National Polytechnique de Grenoble, CNRS, BP 46, 38402, Grenoble, France*
[c] *Laboratoire de Thermodynamique et Physico-chemie Metallurgiques, Institut National Polytechnique de Grenoble, CNRS, BP 75, 38402 Grenoble, France*

Received 3 February 2005; received in revised form 30 October 2005; accepted 31 October 2005
Available online 9 December 2005

Abstract

Dislocation dynamics (DD) is a method to simulate the collective dynamic behavior of dislocations and the plasticity of metals on a mesoscopic scale. A DD simulation is computationally demanding due to the fact that the stress field of a dislocation segment is long-ranged and it needs to examine a possible intersection between dislocation segments during their motion. The computing efficiency of a serial DD code is enhanced by using the so-called 'box method'. The box method employing 21^3 boxes achieves 30-fold speed ups in the case involving 20,000 segments. The modified serial DD code has then been parallelized by using the standard message passing interface (MPI). Both the stress computation and handling segment intersection have been parallelized by using the domain decomposition method. Performance test on IBM p690 architecture shows that the parallel scheme adds up 20-fold speed ups when using 36 processors. Thus the parallel DD code presented here is about 600 times faster than the previous code. We present a parallel algorithm for highly complex dependencies in handling segment intersections and the performance test results in detail.

MSC: 65Y99, 74S30

Keywords: Dislocation dynamics; Plasticity; Parallelization; Domain decomposition; Message passing interfaces; Algorithm

1. Introduction

Dislocation dynamics (DD) simulations deal with physical phenomena resulting from a collective motion of many dislocations. A DD is a very attractive tool for the plasticity of metals from the view that dislocation lines, which are microscopic carriers of the metallic plasticity, are represented explicitly and its length and time scale fills the gap between atomistic simulations and continuum mechanics. A DD method, thus, is

* Corresponding author. Tel.: +82 42 868 4953; fax: +82 42 868 8549.
E-mail address: cshin@kaeri.re.kr (C.S. Shin).

0021-9991/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2005.10.025

expected to play a significant role in multiscale materials modeling and provides a physics-based description of plasticity.

The DD approach has been applied to a wide range of dislocation-related problems, e.g., classical issues such as dislocation patterning [1], strain hardening [2], elementary properties such as a junction formation [3], interaction with second phase obstacles [4]. The behavior of dislocation structures under a complex loading condition has also been extensively studied, e.g., fatigue simulations [5] and dislocation motion in a thin foil [6].

There exist several DD methods which keep tracks of complex dislocation ensemble in three dimensions [1,7–11]. The methods can be classified mainly by the description of dislocation lines: a curved dislocation line is represented by: (i) a succession of finite segments [1,7], (ii) nodal points [8,10,11], and (iii) parametric description [9]. The computational sequence of each DD method is however quite universal. Dislocation lines are introduced in the simulation volume, forces on each line are computed comprising applied loads and inter-dislocation stresses, and then dislocation configuration is updated according to a given mobility law. This motion of dislocations results in a plastic deformation of the simulation volume. Noteworthy is the fact that dislocation lines may intersect (or collide) and annihilate during its motion.

There is a certain limit on the computational size with which DD methods can handle. The main computational barrier comes from the fact that inter-dislocation stresses need $O(N^2)$ computations with N being a number of segments or nodal points. Moreover, dislocation density normally increases during plastic deformation, and so does a number of dislocation segments. For example, initial four segments multiply, becoming 70,000 segments in a fatigue simulation [12].

Numerous are the needs for large simulations, e.g., spontaneous formation of dislocation cell structures, deformation of a volume containing myriad heterogeneities. To overcome the computational hurdle, manifold numerical algorithms have been invented and adopted for inter-dislocation stress computation, which is the most time-consuming part in DD methods. One of them is to partition the dislocation segments around a segment into near and remote segments, and compute the stresses due to the remote segments less frequently [13]. Replacing remote segments by a small number of equivalent superdislocations is another [7]. Development of a multipole expansion method is plausible in that it may provide $O(N)$ computations [14]. Despite all these efforts, a maximum plastic strain currently reached is lower than 0.1%. It is noted that all the attempts mentioned above are concerned with a serial code, i.e., running on a single processor.

Another viable way which can boost the computing speed of DD is to employ a parallel computing scheme. Parallel computing reduces a computation time by distributing the computational loads to several processors and execute them simultaneously. Indeed, parallel computing has become popular in computation sciences, which is the fruit of the development of parallel computing architectures and communication interfaces. Parallel computing has shown a success in many research fields and makes large-scale simulations, which are not attainable with a single processor despite of the fast progression in technology, feasible [15,16].

To realize the high computing power of parallel machines, it is indispensable to adopt an appropriate parallel algorithm or to develop one if necessary. Candidate algorithms also depend on a parallel computing architecture to be used. In the case of distributed memory architectures, a parallel algorithm needs to decompose the data flow dependencies and set a sequence of message passings between processors to send and receive a data set which belongs to a remote memory but vital for the local computation.

Recently in the field of DD, parallelization of a serial code has been attempted and newly developed parallel codes enlarge the expectation to have a physics-based plasticity model. Objects in DD (i.e., segments or nodal points) actually represent a part of dislocation lines. So each object has an information about its connectivity or topological configuration. Upon collision, this connectivity may change. Difficulties then arise how to set a sequence of gathering objects information to detect collision and redistributing modified line topologies.

An algorithm developed at Lawrence Livermore National Laboratory [11] partitions the simulation space into domains by hierarchical recursive bisection. Each domain is then distributed to each processor. Regular cubic cells are overlayed for stress computation. Data of nodal points needed to detect line collisions are communicated between processors which posses neighbor cells. An algorithm developed at UCLA is based on hierarchical N-body methods [17] and builds a tree to split points representing dislocation lines. The structure of this tree is then used to examine possible collisions and compute inter-dislocation stresses.

In this work, we present our efforts to improve the computing efficiency of the discrete dislocation dynamics (DDD) code, which is based on segment representation and originated from Kubin et al. [1], and further developed by Verdier et al. [13]. We partitioned the simulation space into regular boxes and used spatial domain decomposition method. We overcame the complex data dependencies by paying attention to similarities between the method we decompose the simulation space and the algorithms developed for finite difference method.

We arranged this paper as follows. Section 2 describes the DDD method. In Section 3, the details of domain decomposition and an algorithm for parallelizing the DDD code are discussed. Section 4 presents the results of the performance test, and Section 5 summarizes this work.

2. Description of the DDD method

A dislocation line is generally curved in shape and moves on a specific plane which is called 'slip plane'. Each dislocation line holds the dislocation–displacement vector, i.e., the Burgers vector **b**. This vector determines elastic properties of a dislocation (e.g., self energy, stress field, etc.), and dislocation reactions. Possible sets of the Burgers vector and the slip plane depend on the crystal structure involved. In the face-centered cubic (FCC) crystal structure, for example, the slip planes and the Burgers vectors are $\{1\,1\,1\}$ and $\langle 1\,1\,0\rangle$ type, respectively.

In the DDD code used in this work, a curved dislocation line is represented by a succession of two orthogonal dislocation segment sets. The line vector of one segment set (edge segment) is perpendicular to **b** and that of the other set (screw segment) is parallel to **b**. Thus the method is called as the edge-screw model. A glide direction of a segment is always perpendicular to its line direction. Maximum length of a segment is preset, and any segment with a length longer than the maximum length is further subdivided.

Each segment is represented numerically by a set of integers, which comprises the coordinates of the starting point, the length and the indexes of the line and the glide direction vector. The connection of a line is built through a pointer of segments index.

Stresses are computed at the mid-point of each segment, and resolved on the slip plane along the glide direction. The velocity of each segment along glide direction is then given explicitly by a mobility law, and dislocations subsequently propagate on the simulation crystal lattice by updating the positions of segments through an integration of velocities.

The stresses on each segment include following four contributions: (i) the inter-dislocation stresses produced by all the other segments in the simulation volume, (ii) the externally applied stresses on the simulation volume, (iii) the line tension of the segment, and (iv) the Peierls stress or the lattice resistance.

The stress field due to a dislocation segment is obtained using the stress field solution of a semi-infinite dislocation, originally formulated in [18] and modified by Devincre [19] in the compact forms. External stresses are applied either homogeneously to all the segments in the simulation volume or heterogeneously by coupling with a finite element method. The line tension is accounted for by calculating the local curvature of each segment. The Peierls stress is simply implemented as a frictional force which exerts a back stress to motion of a segment.

In FCC metals, a linear form of a mobility law is known to predict well the velocity of a dislocation segment as shown in Eq. (1) with the resolved shear stress τ on the segment and the phonon drag coefficient B

$$v = \frac{\tau|\mathbf{b}|}{B} \tag{1}$$

The next position of a segment is obtained by explicit integration of Eq. (1) using the time step Δt. In practice, the use of a constant value of Δt in the range from 0.5×10^{-9} to 1×10^{-9} s has been verified successful. The maximum velocity $v_{\max}$ is imposed so as to avoid the singularity of the elastic stress solution of a dislocation segment.

A glide of a segment may change the length of the neighbor segments or create new segments if the neighbor segment has the same line direction. A segment can collide with other segments during its glide, which produce dislocation reactions. The type of reactions is determined by the relation of the Burgers vectors and the slip planes involved. The overlapped portion of two colliding segments is annihilated if two segments have the

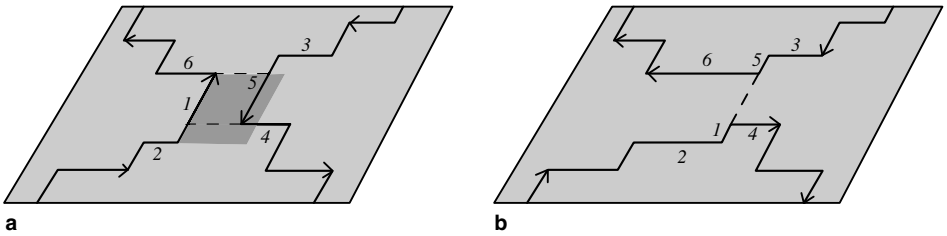


Fig. 1. (a) Before annihilation: the virtual glide area of segment 1 is represented by shade, and segment 1 collides with segment 5. (b) After annihilation: the neighbor segments of segment 1 are now changed to segments 2 and 4.

same Burgers vector and opposite line directions on the same slip plane. Then the links of the rest segments are rebuilt as shown in Fig. 1. In the case of involving different slip planes, two segments of the Burgers vector $\mathbf{b}_1$ and $\mathbf{b}_2$ are considered to form a junction if the norm of $\mathbf{b}_1 + \mathbf{b}_2$ is lower than the addition of the norm of each Burgers vector. Once a junction has formed, segments are switched to junctions and the index of the associated segment is registered.

Typical outputs of the DDD simulations are dislocation configuration, dislocation density, applied stress and resulting plastic strain for each time step. Plastic strain is the resultant of dislocation motion. The slip $\gamma^{(s)}$ of a slip system 's' is computed as

$$\gamma^{(s)} = \frac{|\mathbf{b}|A^{(s)}}{V} \tag{2}$$

with $\mathbf{b}$ being the Burgers vector, V the volume of the simulation box and $A^{(s)}$ the area swept by all the mobile dislocations of the slip system s over a time step ($A^{(s)} = \sum_i L_i v_i \Delta t$). The components of the plastic strain tensor are given by

$$\epsilon_{ij} = \sum_{s=1}^{12} \frac{1}{2}\left(n_i^{(s)} b_j^{(s)} + n_j^{(s)} b_i^{(s)}\right)\gamma^{(s)} \tag{3}$$

with $n_i^{(s)}$ and $b_i^{(s)}$ being the component of the slip plane normal and the Burgers vector of the slip system s, respectively. The interested readers are referred to [13] for further details of the DDD method.

3. Numerical methods to accelerate the DDD code

3.1. Characteristics of the DDD method

Before developing a parallel algorithm suitable for the DDD method, a few characteristics of the method are summarized.

Inter-dislocation stress computation is the most computationally intensive part in the DDD method. This is due to the fact that the stress field at a distance r from a dislocation line is proportional to $1/r$. The stress field of a dislocation line is thus long-ranged. The use of a cutoff distance, beyond which the stress is neglected, may cause a spurious formation of cells [20] and therefore it is requisite to compute the stresses of all the dislocations in the volume. The stress computation has no flow dependence. The elapsed time for the stress computation will thus decrease by a factor of $1/P$, if the computation load is distributed ideally over P processors.

Another time consuming part in the DDD method is handling the dislocation segments intersections, which involves an examination of the possible collisions between segments or between a dislocation segment and internal obstacles. This part has a highly complex flow dependency in that a movement of a segment modifies not only its own position and connection, but also the links of the surrounding dislocation segments. This is due to the fact that dislocation lines are represented as connected sets of segments and their connections are often changed by cutting the dislocation lines. Dislocation segments are created or annihilated with time, so the total number of dislocation segments is not constant.

3.2. Domain decomposition into boxes

It is often observed that the majority of dislocation segments are idle or do not change abruptly within a few time steps during the DDD simulations. The so-called 'box method' proposed in [13] is based on this experience, and decomposes the inter-dislocation stresses into a rapidly changing part and a slowly varying part. In this work, the box method is revised by using a linked-list of segments and presented below.

The simulation volume is first decomposed into boxes. For the sake of the simplicity of the computation scheme, each side of the simulation volume is divided into M boxes. Hence the simulation volume comprises of M^3 boxes. To facilitate the identification of the segments in the box '*ib*', linked-lists of segments are constructed. Using the linked-lists, it is now easy to differentiate near and remote segments, i.e., all the segments in the Lth neighboring boxes plus the segments in the same box are taken to be the near segments of a specific segment. Short-distance stresses ($\boldsymbol{\sigma}^{SR}$) stem from the near segments, and are computed explicitly at every simulation step. The stresses due to the remote segments which are the rest segments except the near segments are taken to be long-distance stresses ($\boldsymbol{\sigma}^{LR}$). $\boldsymbol{\sigma}^{LR}$ is computed at the center point of the box '*ib*' and shared by all the segments in the same box. The computation of $\boldsymbol{\sigma}^{LR}$ is updated every f step and the previous value is used between updates. Hence $\boldsymbol{\sigma}^{LR}$ is approximated both spatially and temporally.

The parameters which control the spatial error are box size, or inversely the number of boxes M. The parameter responsible for the temporal error is f. The number of neighbor layers L for grouping near and remote segments affects both the spatial and the temporal error. The parameters of the method should be chosen so that they guarantee both the optimum speed-up and accuracy. The effects of each parameter on computation efficiency and error are studied. The results are presented in Section 4.1.

The same boxes are used for detecting segments collisions. The minimum size of the boxes is chosen so that the size of boxes is at least bigger than the maximum free-flight distance of a segments, i.e., $v_{\max} \cdot \Delta t$. This criterion for the minimum size of the boxes reduces computing cost in handling segment intersections, because only the collisions with segments within the first neighboring boxes need to be considered instead of inspecting all the segments in the simulation volume. This thus reduce the number of segments to be inspected without any approximation. Fig. 2(a) shows a simulation volume decomposed into 10^3 boxes.

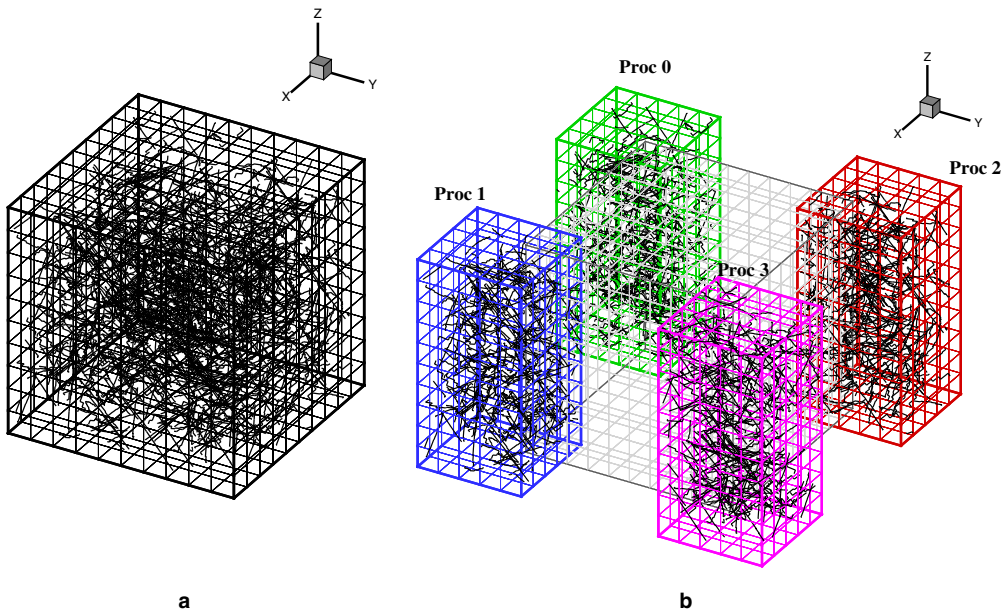


Fig. 2. (a) Decomposition of a simulation volume into $10 \times 10 \times 10$ boxes. (b) Parallel-piped subsystems allocated to four processors.

3.3. Parallelization algorithm

The boxes which decompose a simulation volume (Fig. 2(a)) are partitioned into parallel-piped subsystems (Fig. 2(b)). The processors in a parallel computer are then logically arranged according to the topology of the physical subsystems, and assigned to each subsystem. For each processor, the six face-sharing neighbor processors are identified by a sequential array, *nni*(1:6). Periodic boundary conditions can be imposed by constructing a torus connection of the processors. Indexes of the boundary boxes of each subsystem are stored in the array *ibs*(:) for each processor: *ibs*(1), *ibs*(3) and *ibs*(5) save the first box index along the x, y and z dimensions, respectively, and *ibs*(2), *ibs*(4) and *ibs*(6) represent the last box index along each dimension. *ibs*(:) are varied regularly to balance computing load between processors. The load balance scheme will be explained in Section 3.4.

Each processor computes the long-distance stresses of the boxes only in its subsystem at every f step. Segments in each processor are identified by looking up the linked-lists of the segments in its own subsystem. The short-distance stresses of the segments in each processor are then computed.

Handling segment intersections has a highly complex flow dependency as mentioned in Section 3.1. This dependency can be shown as follows. Let $a(i,j,k)$ represent the topology of the objects (e.g. the connectivity of segments, the number of segments, etc.), in a box (i,j,k) indexed along the x, y and z dimensions. In order to update $a\ (i,j,k)$, all the information of the first neighboring boxes are needed because all the segments in the first neighboring boxes are responsible for the segments collisions. In addition, the topology of the segments in the first neighboring boxes are susceptible to a modification by the motion of the segments in the (i,j,k) box. This dependency is represented in Fig. 3 in a simple 2D configuration. Thus special attention should be paid to handling segment intersections so that no neighboring boxes are overlapped between the processors when treating the segments' motion.

The key idea of handling dislocation intersections is to avoid any overlap of the neighboring boxes of the concurrently updated boxes. This is managed by first dividing the boxes inside a processor p into three groups according to the topology of the neighboring boxes: inner boxes (IB), boundary boxes (BB) and corner boxes (CB). Categorization of the boxes in a subsystem designated to processor '6' in $4 \times 4 \times 1$ parallel-piped subsystems (Fig. 4(a)) is represented in Fig. 4(b). It should be noted that at least three boxes are required along each dimension in each subsystem to categorize the boxes into these three groups.

IB has all of its neighboring boxes in the same processor, thus the motion of the segments in IB modifies the segments located in the same processor only. The segments' positions in IB of each processor can be updated simultaneously and independently without involving any message passings because all the information needed to handle the dislocation intersections are stored in a local memory and also there are no overlaps of the neighboring boxes between the adjacent processors. BB and CB, on the other hand, have their neighboring boxes

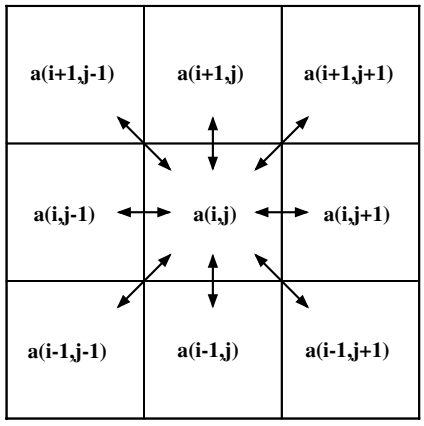
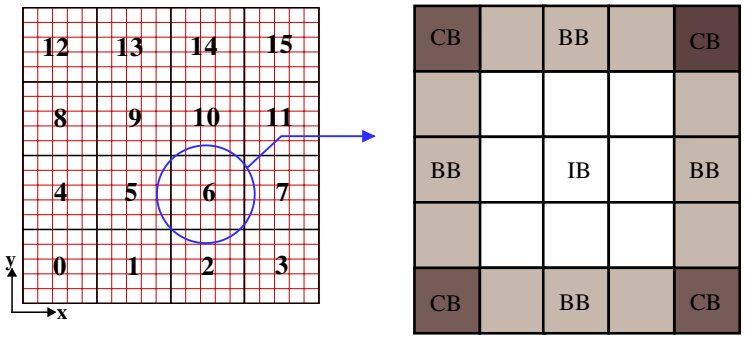


Fig. 3. Dependency on the neighbors: the center element $a(i,j)$ is being computed. All of the surrounding elements are used in the computation and they may be also modified after computing the center element.



(a) Parallel-piped subsystems (b) Categorization of the boxes

Fig. 4. (a) Top view of the $20 \times 20 \times 20$ boxes being assigned to the $4 \times 4 \times 1$ processors: Number represents processor ID. Processor 6 is bounded by $ibs(1) = 11$, $ibs(2) = 15$, $ibs(3) = 6$, $ibs(4) = 10$ and neighbor processors are $nni(1) = 5$, $nni(2) = 7$, $nni(3) = 2$, $nni(4) = 10$. (b) Three category of boxes in a processor p: inner boxes (IB), boundary boxes (BB) and corner boxes (CB).

extending into one or several processors. Thus they need message passings between the processors to obtain the segments' information of the corresponding neighboring boxes and to send back the information modified by the segment collisions. In the case of BB, all the missing neighboring boxes are situated in one neighbor processor, therefore message passings only with an adjacent processor is sufficient to provide the missing information. CB, however, has neighboring boxes scattered in four different processors including itself (in a 2D configuration), and thus are bound to involve complex message passings. Thus a segment motion in CB is handled by one designated processor, or Master processor.

Updating the positions of the segments is performed by the following three steps. In the first step, all the segments in IB of each processor are updated independently and simultaneously. In the second step, the segments in BB are updated through message passings with the corresponding neighbor processor. The order of computation is set from right to left in the x, y and z dimension order. In the final step, all the information of the segments are collected into the Master processor, and the segments' motion in CB is treated by the Master processor only. This procedure at least avoids complex message passings among several processors.

The overall flow chart of the new parallel DDD code is shown in Fig. 5. The 'Motion of the segments' step is composed of three parts corresponding to IB, BB and CB. The involved message passings are also indicated. It should be noted that all the processors begin each time step with the same segment information (marked as '(1)' in Fig. 5). After the segment discretization, each processor computes and thus alters its local segments' data independently up to 'Inner boxes' step ('(2)'). While updating information in BB, two adjacent processors

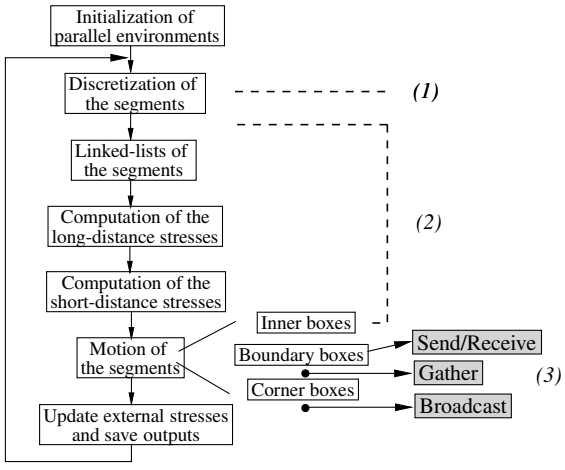


Fig. 5. The overall flow chart of the new parallel DDD code.

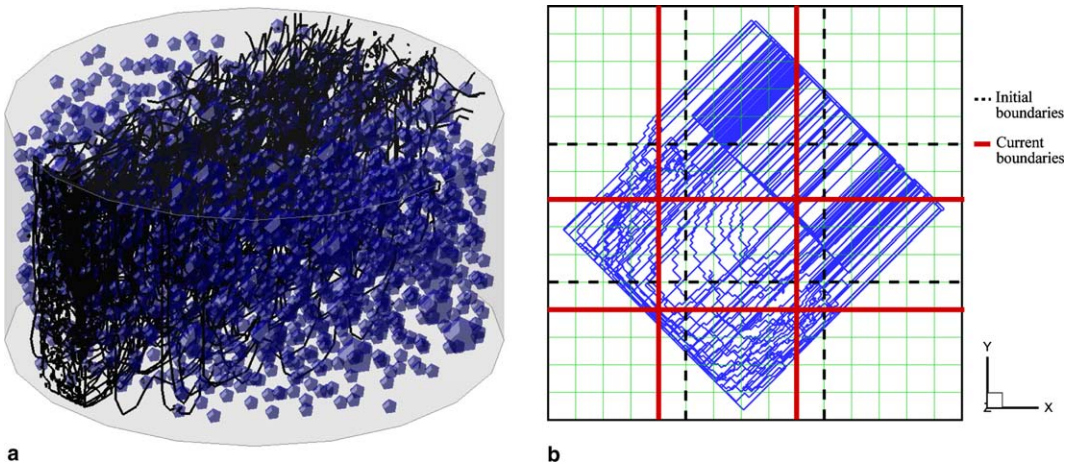


Fig. 6. An example taken from the fatigue tests of a cylindrical simulation volume containing particles of a bimodal size distribution and a load balance scheme. (a) Intense slip bands of the fatigue tested volume containing bimodal-sized particles. (b) Decomposition of the simulation volume by $3 \times 3 \times 1$ processors.

mutually send and receive data and then send the local data to one processor (as indicated 'Gather' in Fig. 5). The master processor then updates all the information in CB and broadcasts data to the other processors (as indicated 'Broadcast' in Fig. 5). Hence all the processors share the same segment information.

3.4. Load balancing

In many cases, DDD simulations involve highly heterogeneous dislocation structures. An example is the formation of intense slip bands in fatigue simulations as shown in Fig. 6 [12]. One obvious way to better balance the loads is to shift the boundaries of each subsystem, or the array *ibs*, so that each processor has approximately the same number of segments, since the computation time is normally proportional to the number of segments. Elapsed times are measured column-wise and the size of a column increases or decreases in the x, y and z dimension order. The boundary can move until the number of boxes has reached the minimum number of boxes (i.e., 3 boxes) of a subsystem in any dimension. An example of a boundary adjustment procedure during a fatigue test is shown in Fig. 6(b).

The load balancing scheme for the parallel-piped subsystem used in this study has the following limitations: (i) different subsystems on the same column should have the same width, thus the computing load is balanced among the columns, not among the processors, (ii) there should be at least three boxes along each axis of each subsystem, thus a load concentration can not be balanced adopting smaller than three boxes in any dimension.

4. Results

4.1. The computing efficiency and error of the domain decomposition method

The computing efficiency of the domain decomposition method is analyzed. The gain in stress computation can be written as Eq. (4), where n_s^{orig} is the number of computations of the original computation method and n_s^{box} is that of the decomposition method. It is assumed that N_{segm} segments are homogeneously distributed over the simulation volume decomposed by M^3 boxes and that periodic boundary conditions are applied along each axis.

$$\text{Speed-up} = \frac{n_s^{\text{orig}}}{n_s^{\text{box}}} = \frac{N_{\text{segm}}^2}{(2L+1)^3 \frac{N_{\text{segm}}^2}{M^3} + \frac{(M^3-(2L+1)^3)N_{\text{segm}}}{f}} \tag{4}$$

Solid lines in Fig. 7 represent Eq. (4) as a function of M for N_{segm} being 10,000, 20,000 and 90,000. The number of layers L is set to 1 and the $\boldsymbol{\sigma}^{LR}$ update frequency f is 20. There exists a maximum in the speed-up curve,

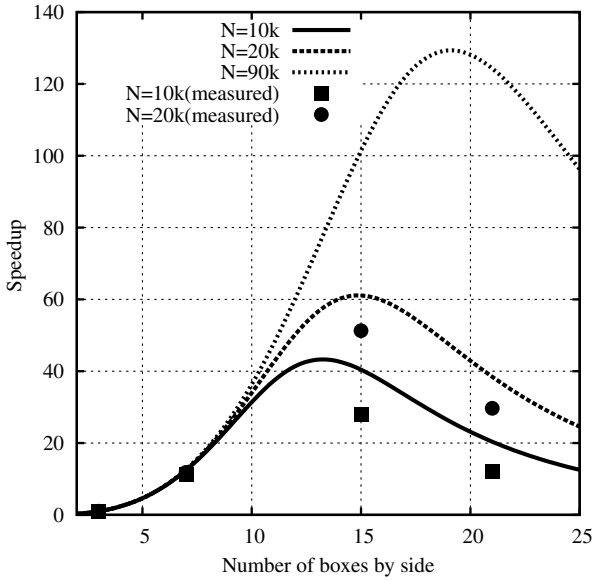


Fig. 7. Evolution of the speedup of the domain decomposition method as a function of the number of boxes and the number of segments.

and the optimum number of boxes is dependent on the number of segments. The actual gains in stress computation time are plotted by solid dots in Fig. 7, which are measured with a 3.0-GHz Intel Pentinum 4 processor and 1 GB memory. The measured data reflect well the characteristic of Eq. (4). It is found that the optimum number of boxes increases with L and f. As for handling a segment intersection, there is always a gain (M^3-fold speed ups) by increasing the number of boxes.

There exist two sources of errors in the internal stress computation, i.e., a spatial and a temporal error. A spatial error occurs because $\boldsymbol{\sigma}^{LR}$ is computed at the center point of a box and assigned to all the segments in that box. A large box size (small M) is bound to have a large spatial error at the end of the diagonal due to a large deviation from the $\boldsymbol{\sigma}^{LR}$ computation position (the center of a box). A small box size (large M) also has a large spatial error because $\boldsymbol{\sigma}^{LR}$ is now no longer slowly varying. Fig. 8 shows the

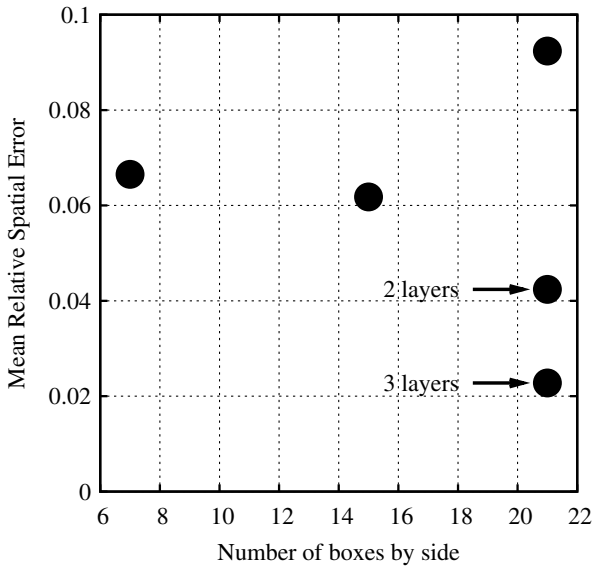


Fig. 8. Mean relative spatial error of the stress computation as a function of the number of boxes.

variation of the mean relative spatial error as a function of the box number M with $L = 1$. The relative errors are measured along a diagonal of a box. The total number of segments is around 20,000 ($\rho \simeq 2.5 \times 10^{12}\ \mathrm{m}^{-2}$, which is taken from a tensile simulation along [0 0 1]) and distributed homogeneously in the volume. It is clear that there is an optimum number of boxes to minimize the spatial error. L is increased up to 3 in the case of $M = 21$. The mean relative spatial error decreases down to 2% by increasing L. The most effective way to minimize the spatial error is thus using the smallest box size with a certain number of layers L.

A temporal error is induced by updating $\boldsymbol{\sigma}^{LR}$ with a frequency f. It is difficult to evaluate a temporal error since the error is strongly dependent on the type of mechanical test simulated and the time step. Thus it is hard to set an optimum f a priori. A test with 20,000 segments and $\dot{\epsilon} = 10^3\ s^{-1}$ showed as much as a 5% mean relative error in stress with $f = 30$ and a 2% error with $f = 10$.

In conclusion, the computing efficiency increases 17–30-fold in the stress computation with the computing error less than 4% by the optimal choice of parameters.

4.2. Performance of the new parallel code

A simple speed-up model of the new parallel algorithm is made and compared to the actual performance results. It is assumed that the simulation volume is decomposed into $M \times M \times M$ boxes and the total number of processors used is P. Processors divide the boxes into a 2D array of $P^{1/2} \times P^{1/2} \times 1$ or into a 3D array of $P^{1/3} \times P^{1/3} \times P^{1/3}$. The elapsed time in a single processor is approximately the sum of the time needed for the stress computation ($t^{\mathrm{s}}_{\mathrm{stress}}$) and the time used to update segments' positions ($t^{\mathrm{s}}_{\mathrm{update}}$). Assuming that $t^{\mathrm{s}}_{\mathrm{update}}$ is a fraction of $t^{\mathrm{s}}_{\mathrm{stress}}$ ($t^{\mathrm{s}}_{\mathrm{update}} = \alpha t^{\mathrm{s}}_{\mathrm{stress}}$), the total elapsed time t^{s} can be written as Eq. (5).

$$t^{\mathrm{s}} = t^{\mathrm{s}}_{\mathrm{stress}} + t^{\mathrm{s}}_{\mathrm{update}} = (1 + \alpha) t^{\mathrm{s}}_{\mathrm{stress}}. \tag{5}$$

The number of IB (B_{I}), BB (B_{B}) in each processor and the total number of CB (B_{C}) can be expressed using M and P as Eq. (6) in the case of a 2D array and as Eq. (7) in the case of a 3D array of processors. It is assumed that all the processors have the same subsystem size.

$$B_{\mathrm{I}} = M\left(\frac{M}{P^{1/2}} - 2\right)^2; \quad B_{\mathrm{B}} = 4M\left(\frac{M}{P^{1/2}} - 2\right); \quad B_{\mathrm{C}} = 4MP, \tag{6}$$

$$B_{\mathrm{I}} = \left(\frac{M}{P^{1/3}} - 2\right)^3; \quad B_{\mathrm{B}} = 6\left(\frac{M}{P^{1/3}} - 2\right)^2; \quad B_{\mathrm{C}} = P\left(12\frac{M}{P^{1/3}} - 16\right). \tag{7}$$

Assuming that dislocation segments are homogeneously distributed over the processors, the elapsed time for the stress computation of each processor ($t^{\mathrm{p}}_{\mathrm{stress}}$) equals $t^{\mathrm{s}}_{\mathrm{stress}}/P$. Considering that the elapsed time for updating segments' positions in a box is $t^{\mathrm{s}}_{\mathrm{update}}/M^3$, the elapsed time of a processor (t^{p}) in a parallel architecture can be expressed as:

$$t^{\mathrm{p}} = t^{\mathrm{p}}_{\mathrm{stress}} + t^{\mathrm{p}}_{\mathrm{update}} = \frac{t^{\mathrm{s}}_{\mathrm{stress}}}{P} + \frac{t^{\mathrm{s}}_{\mathrm{qupdate}}}{M^3}(B_{\mathrm{I}} + B_{\mathrm{B}} + B_{\mathrm{C}}) + t_{\mathrm{c}} \tag{8}$$

The elapsed time for updating B_{C} is included at all the processors because each processor waits until the update of B_{C} by the Master processor is finished. t_{c} represents the time needed for the message passings.

A speed-up ($t^{\mathrm{s}}/t^{\mathrm{p}}$) is plotted in Fig. 9 in the case of a 2D and a 3D array of processors. The parameters used are $M = 21$ and $\alpha = 0.02$. Two cases are compared, i.e., $t_{\mathrm{c}}/t^{\mathrm{s}}_{\mathrm{stress}} = 0$ and $t_{\mathrm{c}}/t^{\mathrm{s}}_{\mathrm{stress}} = 0.02$. The solid line represents the case of a perfectly ideal case. The curve is drawn up to $P = 49$ in the case of a 2D array of processors. Note that a maximum of 49 processors can be used with $M = 21$ in the 2D array of the processors, since there should be at least three boxes along any dimension.

It is found that the efficiency of the algorithm is strongly dependent on the network speed (i.e., t_{c}). If the network is fast enough ($t_{\mathrm{c}}/t^{\mathrm{s}}_{\mathrm{stress}} \simeq 0$), the algorithm speed-up can be as high as 23 using 25 processors with a 2D array of the processors. It seems that the 3D array of the processors has an advantage over the 2D array if the same number of processors is used. In reality it is controversial, however, since a 3D array of processors involves more message passings than a 2D array but the size of each message is smaller in a 3D array.

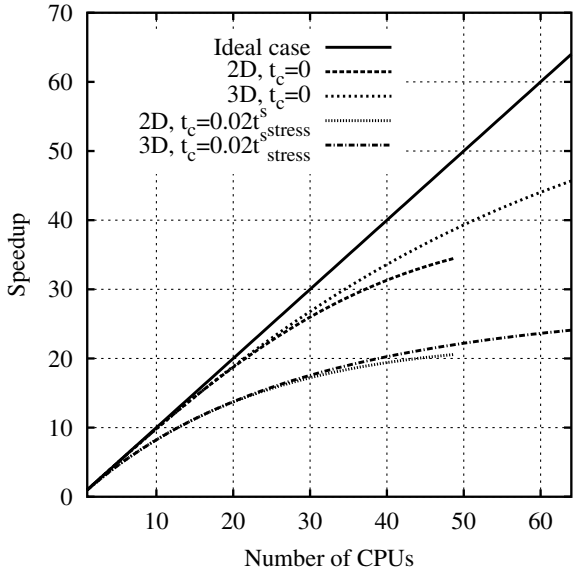


Fig. 9. Speed-up model of the algorithm (Eq. (8)) with $M = 21$, $t^s_{update} = 0.02t^s_{stress}$ for 2D array of processors (2D) and 3D array of processors (3D).

Dislocation structures with 13,185, 37,182, 57,605 and 77,198 segments are extracted from a simple tensile test of a single crystal with $M = 20$. Then the execution time for 100 steps with zero applied stress is measured and the average elapsed time per step is obtained on the IBM p690 architecture with 1.7 GHz POWER4 processors. Fig. 10 shows the measured speed-ups for each number of segments, and they are compared with the speed-up model. Measured data correspond well with the model except the 13,185 segments case. A decrease in the speed-up of the 13,185 segments case with the number of processors is due to the fact that the proportion of the computation time to the communication time decreases.

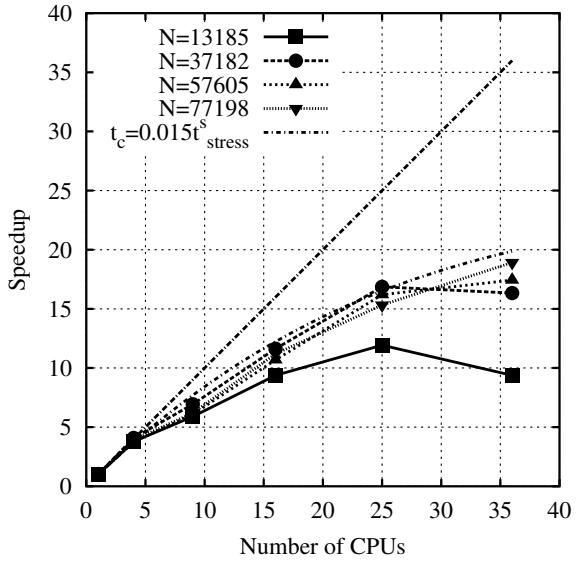


Fig. 10. Speed-up by using P processors in a 2D array for 13,185, 37,182, 57,605 and 77,198 segments (on IBM p690 architecture).

5. Summary

The use of a doma in decomposition of a simulation volume and the linked-lists of segments enables an easy disintegration of the stress fields into short and long-distance stresses and a fast handling of possible dislocation intersections. Minimum box size is chosen such that a dislocation segment collides only with segments in the first neighboring boxes. Long-distance stresses are computed every f simulation time steps at the center of each box.

Boxes are divided into a 2D or 3D array of processors for a parallelization. Parallelization of the complex flow dependency of handling segment intersection has been accomplished by categorizing the boxes into inner, boundary and corner boxes and carefully controlling the sequence of the message-passings. The results of the performance tests show that we achieved an increased speed-up with the number of processors despite the complex flow dependency.

During the simulations, the number of segments can change dramatically. It is not unusual that a single initial Frank–Read source produces millions of segments. At the beginning of a simulation, it is desirable to use a small number of processors, because the speed-up can be reversed by increasing the number of processors in the case of involving small number of segments (Fig. 10). One way to guarantee a maximum efficiency with varying number of segments would be to change the number of processors dynamically based on the current number of segments. This can be done, for example, by creating a new communicator of n [1:N] processors in the initial communicator of N processors.

There is no gain from a memory aspect of the program by using several processors in the present parallel version. The parallel code can further be improved by decomposing the data space, i.e., by making each processor use only a necessary and sufficient amount of memory. This would save a memory space for a parallel computation, and would also decrease the communication overheads and eventually increase the performance of the code.

Acknowledgments

The authors acknowledge the support from the KISTI (Korea Institute of Science and Technology Information) under 'The Sixth Strategic Supercomputing Support Program'. The use of the computing system of the Supercomputing Center is also greatly appreciated.

References

[1] L.P. Kubin, G. Canova, M. Condat, B. Devincre, V. Pontikis, Y. Bréchet, Dislocation microstructures and plastic flow: a 3D simulation, Solid State Phenom. 23&24 (1992) 455–472.
[2] M. Fivel, L. Tabourot, E. Rauch, G. Canova, Identification through mesoscopic simulations of macroscopic parameters of physically based constitutive equations for the plastic behaviour of fcc single crystals, J. de Phys. IV 8 (Pr8) (1998) 151–158.
[3] R. Madec, B. Devincre, L.P. Kubin, T. Hoc, D. Rodney, The role of collinear interaction in dislocation-induced hardening, Science 301 (2003) 1879–1882.
[4] C.S. Shin, M.C. Fivel, M. Verdier, K.H. Oh, Dislocation-impenetrable precipitate interaction: a discrete dislocation simulation analysis, Philos. Mag. 83 (31–34) (2003) 3691–3704.
[5] C. Déprés, C.F. Robertson, M.C. Fivel, Low-strain fatigue in 316L steel surface grains: a three dimensional discrete dislocation dynamics modelling of the early cycles. Part-1: dislocation microstructures and mechanical behaviour, Philos. Mag. 84 (2004) 2257–2275.
[6] Z. Wang, R.J. McCabe, N.M. Ghoniem, R. LeSar, A. Misra, T.E. Mitchell, Dislocation motion in thin Cu foils: a comparison between computer simulations and experiment, Acta Mater. 52 (2004) 1535–1542.
[7] H.M. Zbib, M. Rhee, J.P. Hirth, On plastic deformation and the dynamics of 3D dislocations, Int. J. Mech. Sci. 2–3 (1998) 113–127.
[8] K.W. Schwartz, Simulation of dislocations on the mesoscale: I. Methods and examples, J. Appl. Phys. 85 (1999) 108–119.
[9] N.M. Ghoniem, L.Z. Sun, Fast-sum method for the elastic field of three-dimensional dislocation ensembles, Phys. Rev. B 60 (1999) 128–140.
[10] D. Weygand, L.H. Friedman, E. van der Giessen, A. Needleman, Discrete dislocation modeling in three-dimensional confined volumes, Mat. Sci. Eng. A 309–310 (2001) 420–424.
[11] V.V. Bulatov, W. Cai, J. Fier, M. Hiratani, T. Pierce, M. Tang, M. Rhee, K. Yates, A. Arsenis, Scalable line dynamics of ParaDiS, in: Proceedings of the ACM/IEEE Super Computing 2004 Conference (SC'04), p. 19.
[12] C. Shin, M.C. Fivel, M. Verdier, C. Robertson, Dislocation dynamics simulations of fatigue of precipitation-hardened materials, Mat. Sci. Eng. A 400–401 (2005) 166–169.

[13] M. Verdier, M. Fivel, I. Groma, Mesoscopic scale simulation of dislocation dynamic in fcc metals: principle and applications, Model. Simul. Mater. Sci. Eng. 6 (6) (1998) 755–770.
[14] Z. Wang, N.M. Ghoniem, R. LeSar, Multipole representation of the elastic field of dislocation ensembles, Phys. Rev. B 69 (2004) 174102-1–174102-7.
[15] D. Brown, J.H.R. Clarke, M. Okuda, T. Yamazaki, A domain decomposition parallelization strategy for molecular dynamics simulations on distributed memory machines, Comput. Phys. Commun. 74 (1993) 67–80.
[16] W.L. George, J.A. Warren, A parallel 3D dendritic growth simulator using the phase-field method, J. Comput. Phys. 177 (2002) 264–283.
[17] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, J. Hennessy, Load balancing and data locality in adaptive hierarchical n-body methods: Barnes–Hut, fast multipole, and radiosity, J. Parallel Distr. Com. 27 (1995) 118–141.
[18] R. de Wit, Some relations for straight dislocations, Phys. Stat. Sol. 20 (1967) 567–573.
[19] B. Devincre, Three dimensional stress field expressions for straight dislocation segments, Solid State Commun. 93 (1995) 875–878.
[20] A.N. Gullouglu, D.J. Srolovitz, R. Lesar, P.S. Lomdahl, Dislocation distributions in two dimensions, Scripta Metall. 23 (1989) 1347–1352.