CAVS REPORT
MSU.CAVS.CMD.2010-R0002

Early experiences and results on parallelizing discrete dislocation dynamics simulations on multi-core architectures

Florina M. Ciorba, Sebastien Groh and Mark F. Horstemeyer

Prepared by: *Florina M. Ciorba*
Center for Advanced Vehicular Systems
Mississippi State University
Mississippi State, MS 39759

Contents

1 Introduction **1**
1.1 Related Work 4

2 Parallelization of Micromegas **6**
2.1 Discovering the parallelism 7
2.2 Expressing the parallelism 9
2.3 Debugging the parallel application code 10
2.4 Analyzing the parallel application code 11
2.5 Tuning the parallel application code 13
2.6 Verifying the correctness of the parallel application code 14

3 Evaluation **16**

4 Conclusions and Future Work **18**

Abstract

Materials science simulations are among the leading applications for scientific supercomputing. Discrete dislocation dynamics (DDD) is a numerical tool used to model the plastic behavior of crystalline materials using the elastic theory of dislocations. DDD simulations require very long running times to produce meaningful scientific results. This work presents early experiences and results on improving the running time of Micromegas, an application code for three-dimensional DDD simulations. We used open source profiling and tracing tools to analyze the behavior and performance, as well as to identify the performance bottlenecks of Micromegas. The major performance bottleneck of Micromegas, amounts to $\sim$73% of the total sequential run time and is parallelized using OpenMP. Evaluation and validation tests conducted on a Nehalem quad-core processor show $\sim$50% improvement in the simulation time for 3-D DDD over 30,000 time steps. The correctness and accuracy of the scientific data produced by the parallel Micromegas are successfully validated against those of the original version.

Chapter 1

Introduction

Computational simulations are a valuable approach in enabling scientists to examine the scientific phenomenon by providing more information than is available from experimental testing. Research or production scientific simulation codes are usually ‘legacy’ codes developed over many years by multi-member teams. Advances in computing systems, however, occur at a much faster rate. Therefore, there is an ever widening gap between ‘legacy’ scientific codes and the computing systems they are executed on. This gap can be closed via a ‘performance lift’ approach comprising various performance optimizations (compiler-based optimization, multi-threaded parallelizations, etc.). In this work we ‘lift’ the computational performance of Micromegas to the computing levels of multi-core systems.

Description of the problem

The application code is at the heart of any predictive computational simulations. Micromegas is a legacy application code used to study the mechanical properties (e.g., plastic deformation) of systems (e.g., crystalline materials) with a dislocation. A typical stress-strain curve for the study of the plastic deformation occurring in various crystalline materials under certain amounts of applied stress is shown in Figure 1. In crystalline materials, plastic deformation results from the collective interactions, motion and reaction of a high density of dislocations.The complex nature of scientific phenomena captured by Micromegas, in conjunction with its serial implementation result in very long running simulations, which, depending on the input parameters, can take up to a month until they produce the desired information. The desired information is usually in the form of a higher strain rate such that the simulation exceeds the “linear elastic range” and offers as much insight as possible about the behavior of the material in the “plastic range”, as illustrated in Figure 1. Using smaller time-steps may provide more accurate simulations, but it also requires more computations, taking even more time.

The computational overhead in Micromegas is due to the long-range character of the dislocation stress field. This overhead is distributed between part (1) *calculating the interaction force between the dislocation segments* and part (2) *handling the reactions between the dislocation cores*. In part (1), called FORCE, the driving force on the dislocation segment is the combination of the following: the interaction force between dislocations, the self-force [9] and the projection on the slip system of the applied force. The interaction force between dislocations is calculated using the analytical form of [5] transformed for numerical use by Devincre et al [6]. The computation of the interaction force between dislocations represents the most expensive calculations in the simulation.The computational complexity of calculating the interaction force is $O(N^2)$, where the number of dislocation segments, N, increases during the simulation. Contrary to the molecular dynamics methodology where the number of atoms is constant, in discrete dislocation dynamics the number of segments increases with the plastic deformation. The computational efficiency of calculating the dislocation elastic field can be improved using a multipolar expansion method, also called the Greengard algorithm [13][30][28]. Using this method, the complexity is reduced from $O(N^2)$ to O(N), with an error of 0.1%. Even in this case, however, the simulation was still limited to less than 0.5% of the plastic deformation [32][7].

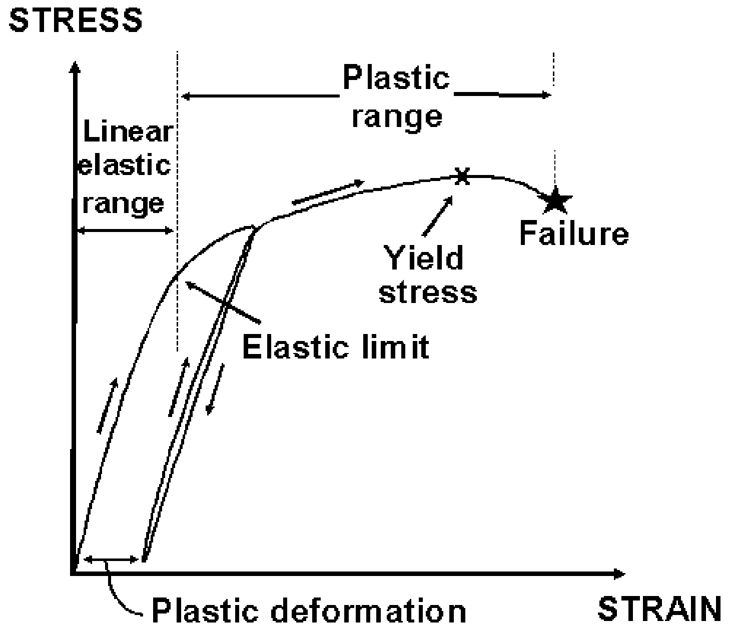


Figure 1.1: Typical stress-strain curve for the study of plastic deformation. Magnitude of stress at which appreciable deformation takes place without any appreciable increase in the applied stress (*http://www.srs.org/professionals/glossary/biomechanics.php*)

In part (2), called UPDATE, handling the reactions between dislocations requires powerful computational resources for operations of a different type than the ones used in calculating of the interaction forces between dislocations. This is due to the fact that the handling is not based on analytical formulae. A specific algorithm is used to perform an exhaustive search for dislocation segments present in the area swept by the moving segment. Whenever segments are detected in that area, a series of different possible reactions involving the moving segment and the obstacle segment are tested, and the one with the lowest energy will be formed. The characteristics of the dislocation segment are then updated and the resulting plastic strain rate is calculated accordingly. The computational complexity of this part of the code is $O(N^2)$.

Motivation

The performance of Micromegas is influenced by the parameters of the problem, which, unfortunately evolve during the course of the simulation as illustrated in Figure 2. The evolving nature of the application is due to the fact that N (total number of segments), and ND (the number of moving segments with length greater than zero) take different values after every simulation time step, following an increasing trend as illustrated in Figure 12 (explained later in the document). Therefore, the problem size and required computational resources become known only at the beginning of a new time step. The number of time steps in a typical simulation run ranges from 10^4 to 10^9 steps. Additionally, small source code changes can lead to significant performance changes, including performance degradation. Thus, *analyzing and improving the performance of 3-D DDD simulations across a variety of existing computing systems constitutes a topic of research.* New algorithms and methods are needed to efficiently simulate the longtime behavior of dislocations, an area that has proven to be less amenable to parallelization than large system size problems. This work is the first to attempt the parallelization of Micromegas, in general, and on multi-core architectures, in particular.

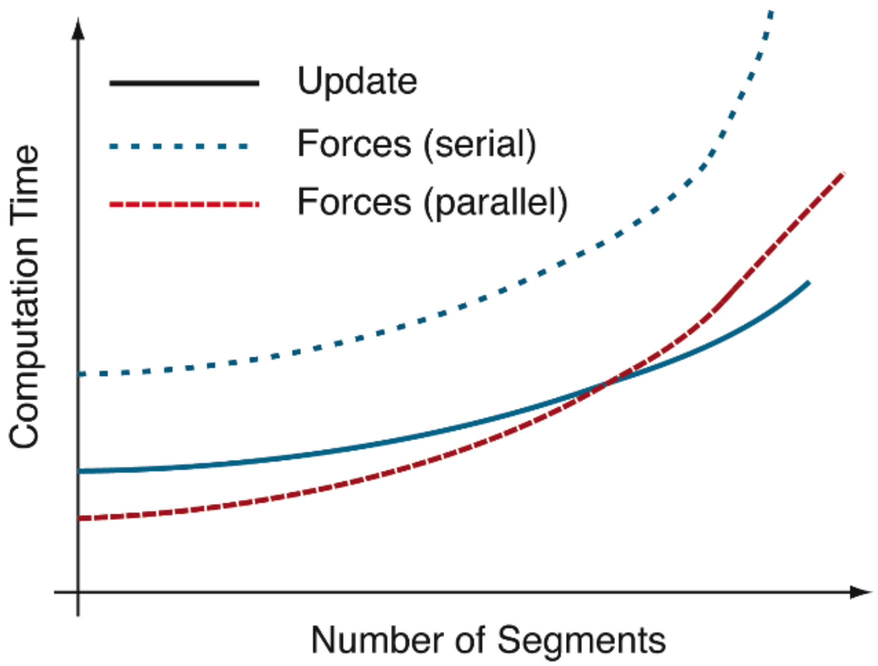


Figure 1.2: Evolution of the computational time spent in UPDATE and FORCE (serial and parallel) with increasing number of segments.

Contribution

The behavior and performance of Micromegas is analyzed using TAU [36], an open source performance measurement system. The use of TAU allowed for identification of the major performance bottlenecks. The main contribution of this work is parallelizing the most important bottleneck

(FORCE) using a multi-threaded fork and join approach. The parallelization significantly reduced the overhead of FORCE, while the total simulation time was reduced by $\sim$50% on 4 threads. The scientific simulation data (e.g., stress, strain and dislocation density) of the parallel simulation are successfully validated for correctness and accuracy against those of the sequential simulation.

The rest of this document first discusses related work in Section 1.1, and then describes our parallelization and implementation in Section 2. Section 3 describes how we evaluated our parallelization and implementation and presents the results obtained. Section 4 presents our conclusions and outlines directions for future work.

1.1 Related Work

In the mid-1960's, Brown [4], Bacon [3] and Foreman [9] initiated the development of discrete dislocations simulations in proposing a framework to characterize the curvature of a line of dislocation under an applied stress. It was not until the early 1990's that the first 2-dimensional and 3-dimensional discrete dislocation simulations were proposed by Ghoniem and Amadeo [1] and Kubin et al. [18], because of the high performance computational resource required by the original framework to model interactions between a large population of dislocations.

The methodologies developed for the three-dimensional dislocation dynamics can be categorized into two groups based on the topology of a general curved dislocation line. The first method is based on an edge-screw (Kubin et al. [18]) or edge-mixed-screw (Madec et al. [19], Devincre et al. [7]) discretization of the dislocation lines. The basic idea of this approach is that the discrete segments move on a discrete lattice superimposed onto the crystallographic lattice, but on the orders of a larger magnitude. The second category of methods simulates dislocations as smooth flexible lines discretized either by linear splines (Zbib et al. [30][31], Arsenlis et al. [2], Kristian and Kratochvil [16][17]), cubic splines (Ghoniem and Sun [10], Ghoniem et al. [11]) or circular arcs (Schwarz [24][25]). Although the description of a curved dislocation differs, these two categories converge to each other when the number of degree of freedom is increased in the edge-mixed-screw model.

Over the last decade, the discrete dislocations framework has been applied to model different crystal structures. Moulin et al. [22] and Tang et al. [27] extended the edge-screw model to diamond cubic and BCC (body-centered cubic) crystal structure, respectively. Madec and Kubin [20] implemented the BCC crystal structure in the edge-screw-mixed model and investigated the strength of the junction in FCC (face-centered cubic) and BCC crystal structures. Monnet et al. [21] extended the edge-mixed-screw model to the HCP (hexagonal close-packed) crystal structure with only basal dislocations. Durinck et al. [8] implemented the orthorhombic symmetry for olivine in a 3-D discrete dislocation dynamics. They demonstrated that no junction formation results from the interaction between [100] and [001] dislocations in this crystal structure. The linear splines models were developed for FCC [31][16], and for BCC crystal structures [2]. The FCC crystal structure was implemented in the cubic splines models [10][24][25].

It has been shown that the numerical limitations associated with 3-D dislocation dynamics may be overcome by the use of parallel computing. Rhee et al. [23] developed a parallel version of the DDD code called *micro3d*. They showed that the parallel code yields a significant speedup, but does not scale with the number of processors. Parallelism saturation is mainly due mainly to the short-range reactions between the dislocation segments, which necessitates *frequent* communication among processors. Nevertheless, using this parallel code, large-scale dislocations problems and dislocations-defects problems were analyzed (see, e.g., Khraishi et al. [14][15]). Recently, Shin et al. [26] proposed a parallel algorithm to speed up the edge-screw model, while Wang et al. [29] and Arsenlis et al. [2] proposed a parallel algorithm to speed up the cubic and linear splines models, respectively. Using the parallel version of the DD simulation code, Arsenlis et al. [2] were able to reach 1.7% of the plastic deformation during the tensile test of a specimen of molybdenum at an elevated temperature. Senger et al. [34] used a parallel DDD code to study stress distributions in polycrystalline materials with FCC crystal structure. The interested reader is referred to a recent review by Groh and Zbib [12] of the discrete dislocation methodology and its implication to multiscale modeling of the mechanical behavior of crystalline materials.

Until today, the development of Micromegas was focused on the physics involved in the plastic deformation of various crystal structures (FCC, BCC, HCP, CC), while all the above parallel codes are dedicated to a single crystal structure.

Chapter 2

Parallelization of Micromegas

Micromegas is written in a mix of Fortran 90 and Fortran 95, consisting of 16 source modules and containing roughly 25,000 lines of code. Figure 3 gives the pseudocode of the `MAIN` module in Micromegas. A typical simulation run in Micromegas requires somewhere between 10^4 to 10^9 time steps, which are imposed by the desire to gain more insight about the plastic deformation range. Simulations with a smaller number of steps are very likely not to capture the plastic range of deformation, which constitutes the region of interest for the materials scientists studying plastic deformation. For 2x10^4 time steps, each time step of 10^{-9} seconds, the serial version of Micromegas takes on average 52 hours to reach 0.035% of the plastic deformation on a 4-way Intel Xeon W3570 processor, with 6GB of triple channel 133MHz DDR-3 RAM. As explained in Section 1 and Section 1, one needs to run simulations of about 10^9 time steps in order to achieve the desired percentage of deformation, that is, as high over 1% as possible.

```
! Module MAIN: simulation time loop
TIME: do = 1, STEPS
...
  call SOLLI  ! Apply load
  call DISCRETI ! Discretize the simulation volume into dislocation lines/segments
  call FORCE ! Calculate interaction forces
       !FORCE calls SIGMA_INT_CP to calculate short range interaction forces
       !FORCE calls SIGMA_INT_LP to calculate long range interaction forces
  call DEPPREDIC ! Predict moving segments
  call UPDATE ! Search for obstacles, determine & make contact reactions, update positions of segments
  call CORRIGER_CONFIG ! Check the connections between all segments
   ...
enddo TIME
```

Figure 2.1: Serial Micromegas: Pseudocode of the MAIN Micromegas module illustrating the most important subroutines called during each simulation time step.

In this work, we are interested in parallelizing the part of Micromegas expected to give the highest “performance lift”, i.e., the part calculating the interaction forces. The pseudocode in Figure 3 indicates that Micromegas belongs to the class of time-stepping scientific applications. Time-stepping applications are not easily amenable to parallelization. This indicates that our efforts need to be concentrated on speeding up the execution of a single time step. We target multi-core processors by employing the fork-and-join model of parallelism. The parallelization approach consists of the following five steps: (1) discovering the parallelism, (2) expressing the parallelism,

(3) performing the thread-safety analysis, (4) analyzing the parallel application code (5) tuning the parallel code and (6) verifying the correctness of the parallel code. Each of these steps are a significant part of the parallelization process and are elaborated individually below.

2.1 Discovering the parallelism

To achieve our goal, we begin by extensive measures of Micromegas' performance using TAU [36], a state-of-the-art portable profiling and tracing package. TAU allowed easy and customizable instrumentation of Micromegas. Given that a typical Micromegas simulation run requires 10^4 to 10^9 time steps and takes over 50 hours or longer, we decided to execute the instrumented Micromegas version over 1,000 time steps. The instrumented version of the original code was executed on a single core of the machine. TAU enabled us to collect profiling information at various levels: outer loops, routine level and memory leaks. Using TAU, we were also able to collect traces of the serial execution of the instrumented code. Being able to obtain these measurements was crucial to understanding the behavior of the simulation code and to identifying the performance bottlenecks of the application.

We collected profiling information from a serial run of the application over 1,000 time steps. Profiling yields timing information summed over all invocations of a function. The execution callgraph of the original application code is illustrated in Figure 4. The callgraph was illustrated with Paraprof, TAU's 3-D profiling data visualization tool. The red box indicates the most time consuming part of the application, i.e., the 'hot spot', namely subroutine SIGMA_INT_CP of module ELASTI. The green box indicates the second most time consuming part of the application, i.e., the 'warm spot', namely subroutine UPDATE in module CONTACT. Blue color indicates subroutines that take less time than those in the green or red color boxes, and are called 'cold spots'.

On the basis of the profiling information obtained with TAU, we were able to sort the modules of Micromegas in decreasing order of importance, with respect to the percentage of the total serial time attributed to each module. The profiling data for the simulations over 1,000 time steps indicated that the most important module was ELASTI, and the most important subroutine of this module was SIGMA_INT_CP, called by subroutine FORCE. Similarly, the second most important module was CONTACT with the most important subroutine of this module being UPDATE.

The profiling information collected allowed us to understand the callpath relations among the most important subroutines (see Figure 6 - *left*). Figure 6 - *right* gives the percentile distribution of the serial time per subroutine. One can see that the ELASTI::SIGMA_INT_CP subroutine accounts for 72.675% of the serial run time, while subroutine CONTACT::UPDATE is responsible for 26.885% of the serial run time, respectively.

Tracing yields a time line of events occurring throughout the execution of an application. The traces produced by TAU can be converted to various formats, that can be interpreted by other trace visualization tools. Due to the fact that tracing is a lower level type of application performance analysis, the volume of trace data is much higher than that of profiling data. Using TAU, we collected traces of the serial execution of the application only for 10 time steps. These traces were converted to the SLOG2 format and visualized with Jumpshot [37], a Java-based visualization tool for doing postmortem performance analysis. Figure 5 illustrates the serial execution trace visualized with Jumpshot. The legends highlight the most time consuming subroutines, i.e., FORCE, SIGMA_INT_CP and UPDATE.

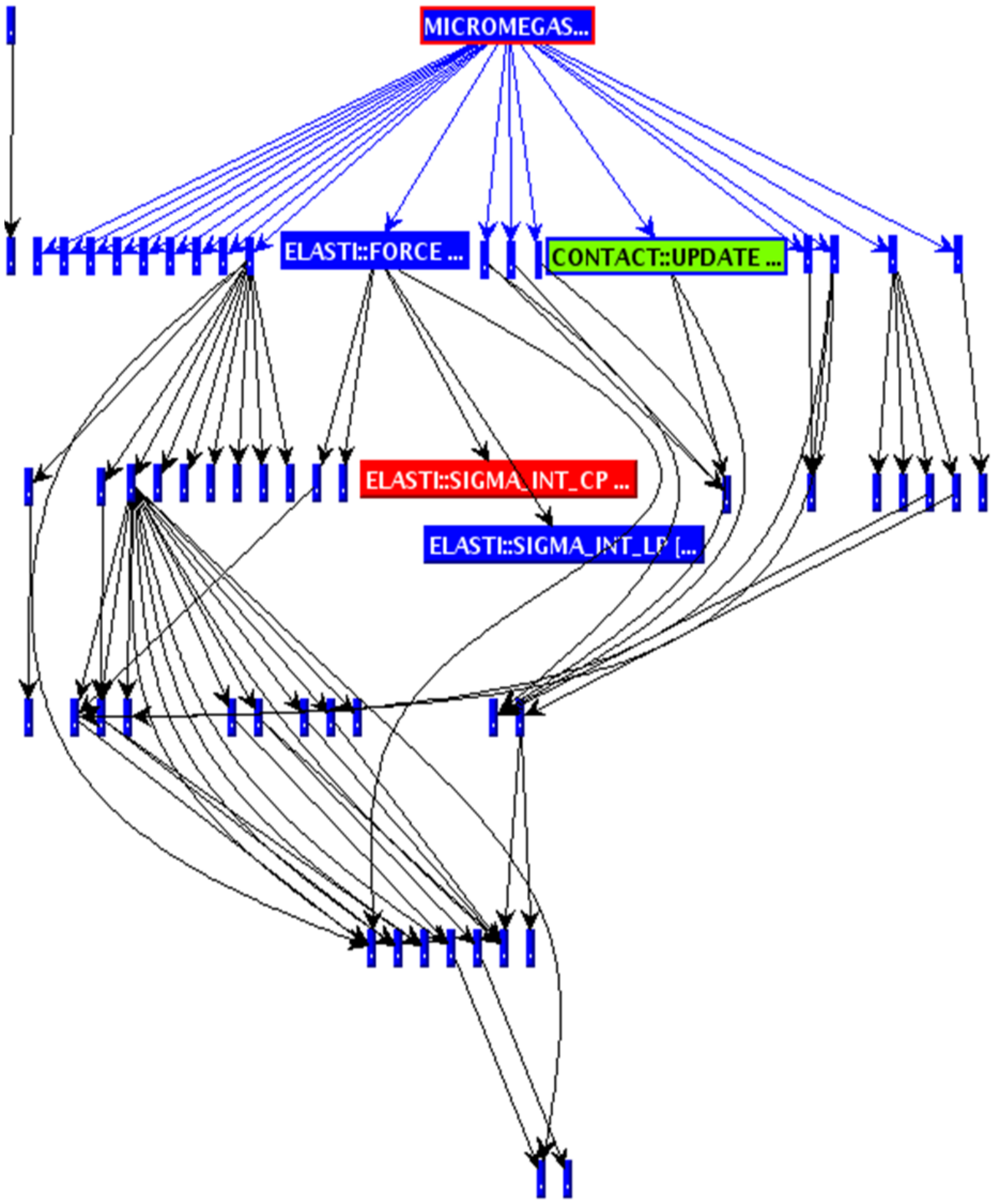


Figure 2.2: Serial Micromegas: Visualization of serial execution callgraph for 1,000 time steps. Important subroutines are highlighted in colors, where red denotes a 'hot spot', green a 'warm spot' and blue a 'cold spot'.

Upon analyzing both the profiling and tracing data, in Figures 4 and 5, respectively, and given the structure of the application code (see Figure 3), we identified the fact that speeding up the cal-

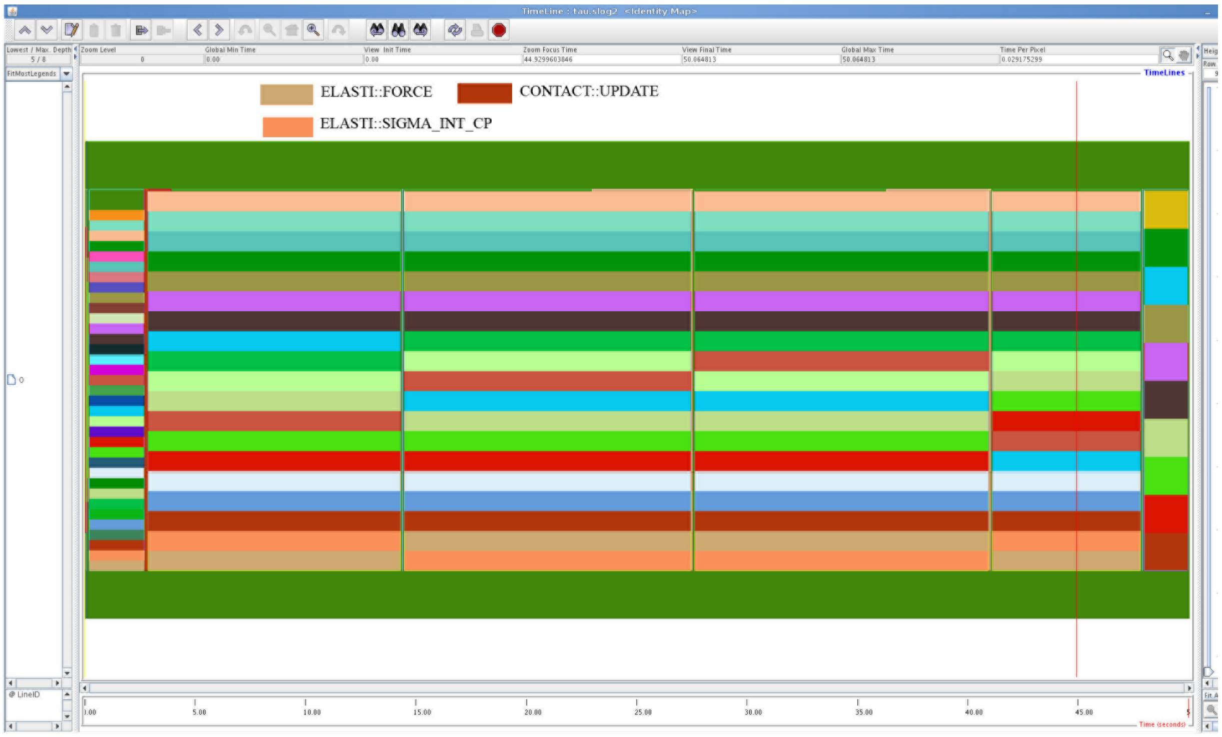


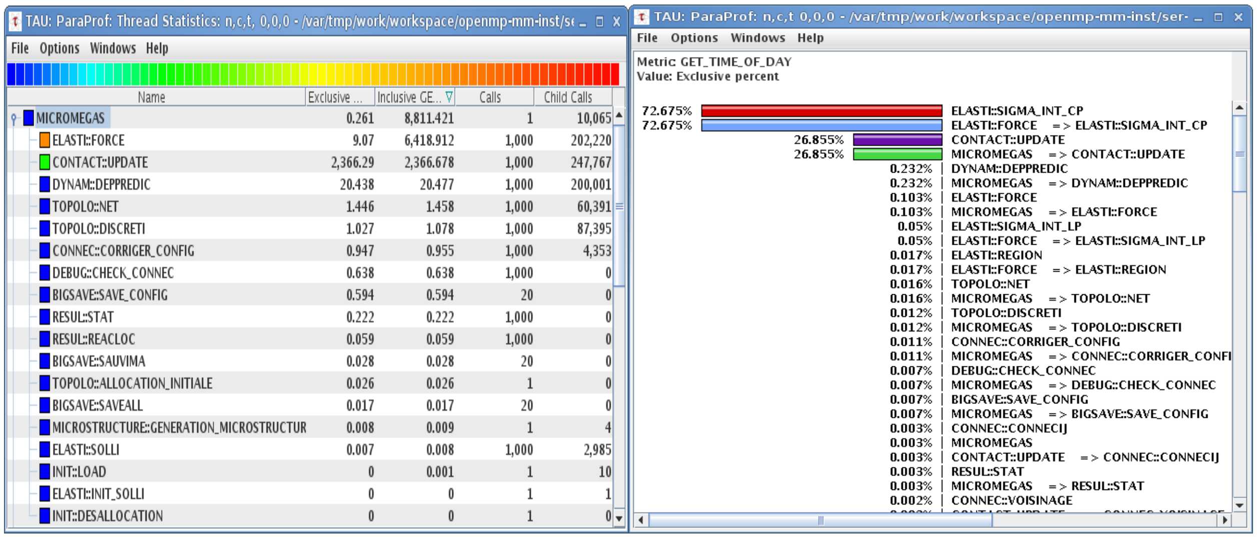
Figure 2.3: Serial Micromegas: Visualization of serial trace for 10 time steps. The legend highlights FORCE, SIGMA_INT_CP and UPDATE.

culations of a single time step will require that our efforts be focused on parallelizing the subroutine SIGMA_INT_CP, which calculates the short range interaction forces between the dislocation segments. Senger et al. [34] used a similar idea to parallelize the interaction forces calculations in their DDD code, which based on the dynamics of nodes, whereas Micromegas is based on the dynamics of segments.

2.2 Expressing the parallelism

In this step, we consider parallelizing the Micromegas modules in decreasing order of importance, based on the serial time distribution among subroutines. The most important module is ELASTI, while the most important subroutine of this module is SIGMA_INT_CP, called by subroutine FORCE. We parallelized this module using the fork-and-join model in which the code consists of alternating serial and parallel regions. We chose the fork-and-join model because of its ease of implementation and non-invasive modifications to the original serial code. The fork-and-join model is implemented using OpenMP [35] directives.

Subroutine SIGMA_INT_CP contains a mix of nested DO loops and a series of consecutive DO loops that iterate over *the number of boxes*, *the number of segments* in each 3-D box and over *each segment* in a box. These nested DO loops calculate the short-range interaction forces between the segments, and, therefore, the iterations of these loops are *independent* of each other. We parallelized the outermost loop, which iterates over the number of boxes of the simulation domain, by distributing its iterations among the processing cores in a master-worker fashion.


TAU: ParaProf: Thread Statistics: n,c,t, 0,0,0 - /var/tmp/work/workspace/openmp-mm-inst/se
File Options Windows Help
TAU: ParaProf: n,c,t 0,0,0 - /var/tmp/work/workspace/openmp-mm-inst/ser-
File Options Windows Help

Name	Exclusive ...	Inclusive GE...	Calls	Child Calls
MICROMEGAS	0.261	8,811.421	1	10,065
ELASTI::FORCE	9.07	6,418.912	1,000	202,220
CONTACT::UPDATE	2,366.29	2,366.678	1,000	247,767
DYNAM::DEPPREDIC	20.438	20.477	1,000	200,001
TOPOLO::NET	1.446	1.458	1,000	60,391
TOPOLO::DISCRETI	1.027	1.078	1,000	87,395
CONNEC::CORRIGER_CONFIG	0.947	0.955	1,000	4,353
DEBUG::CHECK_CONNEC	0.638	0.638	1,000	0
BIGSAVE::SAVE_CONFIG	0.594	0.594	20	0
RESUL::STAT	0.222	0.222	1,000	0
RESUL::REACLOC	0.059	0.059	1,000	0
BIGSAVE::SAUVIMA	0.028	0.028	20	0
TOPOLO::ALLOCATION_INITIALE	0.026	0.026	1	0
BIGSAVE::SAVEALL	0.017	0.017	20	0
MICROSTRUCTURE::GENERATION_MICROSTRUCTUR	0.008	0.009	1	4
ELASTI::SOLLI	0.007	0.008	1,000	2,985
INIT::LOAD	0	0.001	1	10
ELASTI::INIT_SOLLI	0	0	1	1
INIT::DESALLOCATION	0	0	1	0

```
Metric: GET_TIME_OF_DAY
Value: Exclusive percent

72.675%   ELASTI::SIGMA_INT_CP
72.675%   ELASTI::FORCE      => ELASTI::SIGMA_INT_CP
26.855%   CONTACT::UPDATE
26.855%   MICROMEGAS      => CONTACT::UPDATE
 0.232%   DYNAM::DEPPREDIC
 0.232%   MICROMEGAS      => DYNAM::DEPPREDIC
 0.103%   ELASTI::FORCE
 0.103%   MICROMEGAS      => ELASTI::FORCE
  0.05%   ELASTI::SIGMA_INT_LP
  0.05%   ELASTI::FORCE      => ELASTI::SIGMA_INT_LP
 0.017%   ELASTI::REGION
 0.017%   ELASTI::FORCE      => ELASTI::REGION
 0.016%   TOPOLO::NET
 0.016%   MICROMEGAS      => TOPOLO::NET
 0.012%   TOPOLO::DISCRETI
 0.012%   MICROMEGAS      => TOPOLO::DISCRETI
 0.011%   CONNEC::CORRIGER_CONFIG
 0.011%   MICROMEGAS      => CONNEC::CORRIGER_CONFI
 0.007%   DEBUG::CHECK_CONNEC
 0.007%   MICROMEGAS      => DEBUG::CHECK_CONNEC
 0.007%   BIGSAVE::SAVE_CONFIG
 0.007%   MICROMEGAS      => BIGSAVE::SAVE_CONFIG
 0.003%   CONNEC::CONNECIJ
 0.003%   MICROMEGAS
 0.003%   CONTACT::UPDATE      => CONNEC::CONNECIJ
 0.003%   RESUL::STAT
 0.003%   MICROMEGAS      => RESUL::STAT
 0.002%   CONNEC::VOISINAGE
```

Figure 2.4: Serial Micromegas: Callpath relations (*left*) showing the inclusive and exclusive time together with the total number of calls and child calls per each subroutine. FORCE is the subroutine of the module ELASTI that calls SIGMA_INT_CP. The bar chart (*right*) shows the percentage distribution for every subroutine. Due to the large number of subroutines in Micromegas, we show only the most important subroutines, with respect to their percentage of the total serial time. ELASTI::SIGMA_INT_CP is the most important performance bottleneck accounting for 72.675% of the serial run time over 1,000 time steps. CONTACT::UPDATE is the second performance bottleneck accounting for 26.885% of the serial run time over 1,000 time steps.

2.3 Debugging the parallel application code

Following parallelization, the next important step was to ensure the thread safety of the parallelized application code. This requires a good knowledge of the application code and of its behavior. Gaining sufficiently thorough knowledge of the application code can be time consuming, especially for applications with a very large number of lines of code and complex modules dependencies, such as Micromegas. However, failing to ensure the thread safety of the parallelized code has a direct impact on both the performance and the accuracy of the parallelized application code.

Ensuring thread safety required identification of all global data dependencies and their effective privatization. To ensure the tread safety of the parallelized SIGMA_INT_CP subroutine, we started by identifying the global data that are *defined outside the parallel loop* and only read inside the loop. Thread safety was achieved by declaring this data in the list of shared variables. Next, we identified global data with dependence relations, that is data that is *defined outside the loop but computed/updated inside the loop*. In these instances, multiple iterations update the same data structure, e.g., the array of forces. Thread safety was achieved via fine grained locking, which ensures that the updates on the shared data occur in a sequential order, since the final result is independent of the ordering of the iterations. Finally, we identified local data with dependence relations, which refers to data *defined inside the loop and computed/update inside the loop*. In this case thread safety was achieved by privatizing and initializing the data to the values of the first executing thread.

2.4 Analyzing the parallel application code

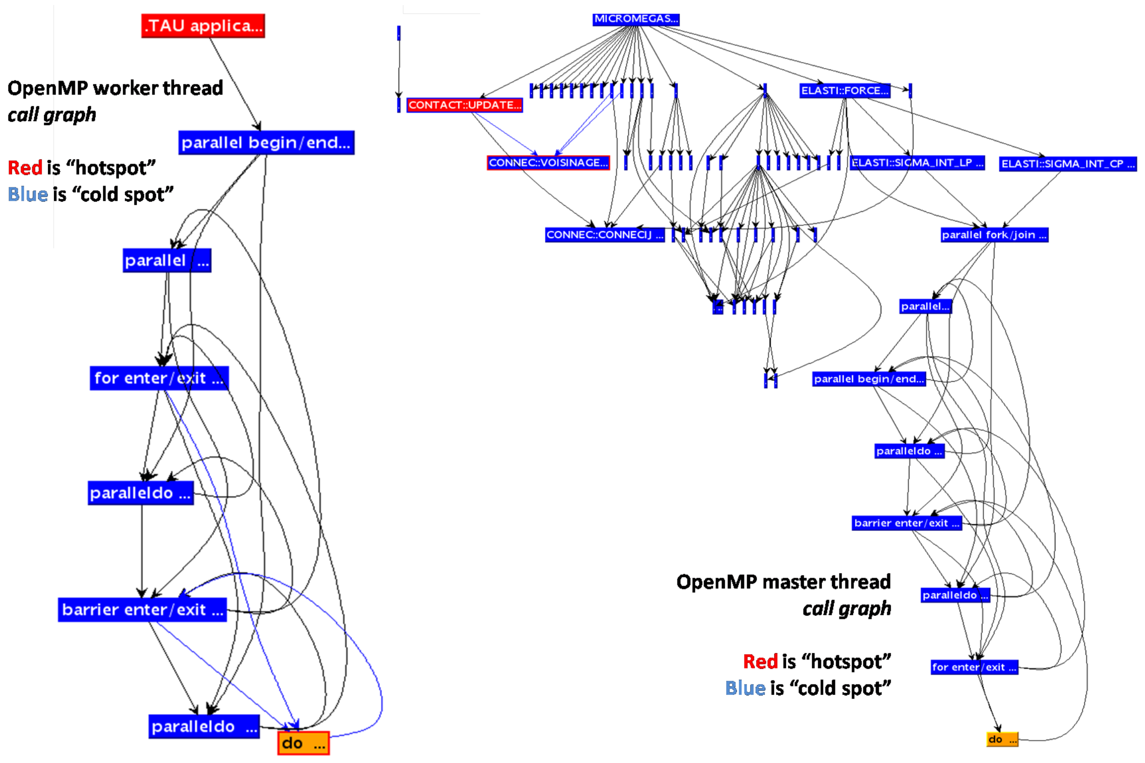


Figure 2.5: Parallel Micromegas: Callgraphs visualization for parallel execution on 4 cores - master thread callgraph (*left*) and the worker threads callgraph (*right*) for 1,000 time steps.

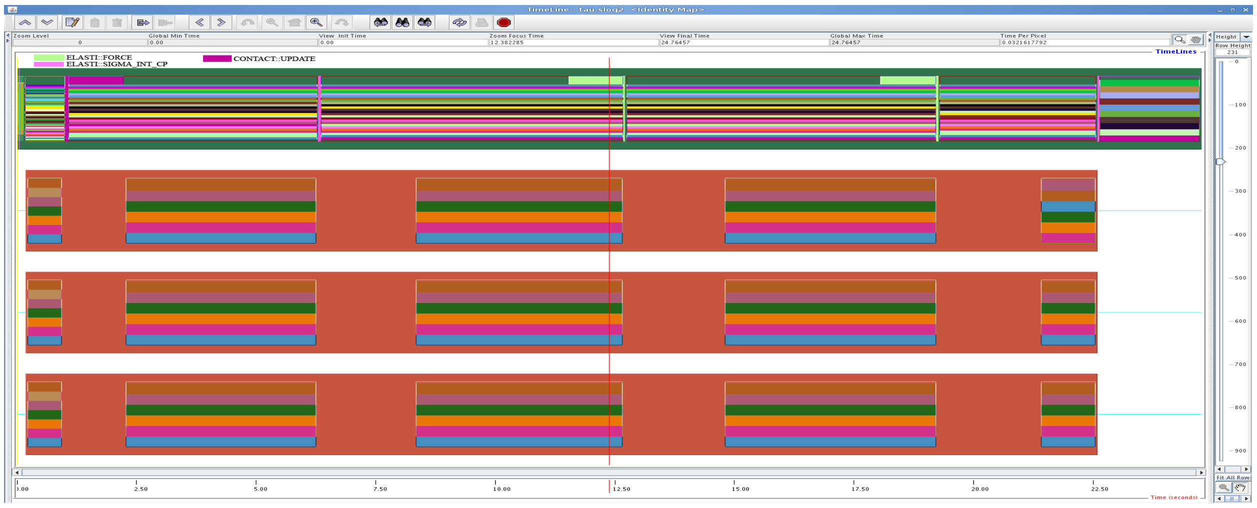
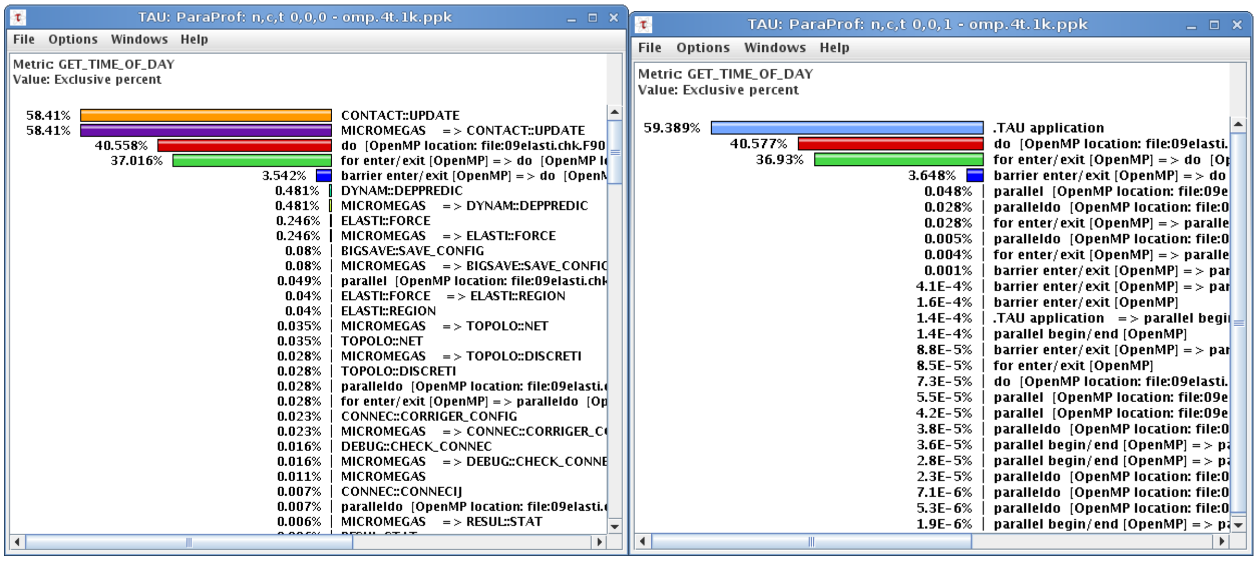


Figure 2.6: Parallel Micromegas: Visualization of parallel trace for 10 time steps. The legend highlights FORCE, SIGMA_INT_CP and UPDATE.



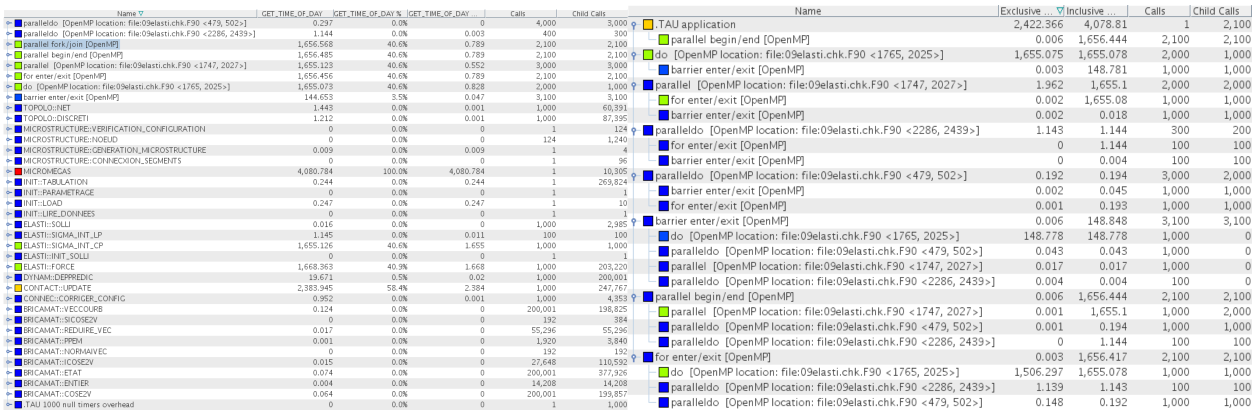
```
TAU: ParaProf: n,c,t 0,0,0 - omp.4t.1k.ppk
File  Options  Windows  Help
Metric: GET_TIME_OF_DAY
Value: Exclusive percent

 58.41%   CONTACT::UPDATE
 58.41%   MICROMEGAS   => CONTACT::UPDATE
40.558%   do  [OpenMP location: file:09elasti.chk.F90
37.016%   for enter/exit [OpenMP] => do  [OpenMP lo
 3.542%   barrier enter/exit [OpenMP] => do  [OpenM
 0.481%   DYNAM::DEPPREDIC
 0.481%   MICROMEGAS   => DYNAM::DEPPREDIC
 0.246%   ELASTI::FORCE
 0.246%   MICROMEGAS   => ELASTI::FORCE
  0.08%   BIGSAVE::SAVE_CONFIG
  0.08%   MICROMEGAS   => BIGSAVE::SAVE_CONFIC
 0.049%   parallel  [OpenMP location: file:09elasti.chk
  0.04%   ELASTI::FORCE   => ELASTI::REGION
  0.04%   ELASTI::REGION
 0.035%   MICROMEGAS   => TOPOLO::NET
 0.035%   TOPOLO::NET
 0.028%   MICROMEGAS   => TOPOLO::DISCRETI
 0.028%   TOPOLO::DISCRETI
 0.028%   paralleldo  [OpenMP location: file:09elasti.
 0.028%   for enter/exit [OpenMP] => paralleldo  [Op
 0.023%   CONNEC::CORRIGER_CONFIG
 0.023%   MICROMEGAS   => CONNEC::CORRIGER_C
 0.016%   DEBUG::CHECK_CONNEC
 0.016%   MICROMEGAS   => DEBUG::CHECK_CONNE
 0.011%   MICROMEGAS
 0.007%   CONNEC::CONNECIJ
 0.007%   paralleldo  [OpenMP location: file:09elasti.
 0.006%   MICROMEGAS   => RESUL::STAT
```

```
TAU: ParaProf: n,c,t 0,0,1 - omp.4t.1k.ppk
File  Options  Windows  Help
Metric: GET_TIME_OF_DAY
Value: Exclusive percent

59.389%   .TAU application
40.577%   do  [OpenMP location: file:09elasti.
 36.93%   for enter/exit [OpenMP] => do  [Op
 3.648%   barrier enter/exit [OpenMP] => do
 0.048%   parallel  [OpenMP location: file:09e
 0.028%   paralleldo  [OpenMP location: file:0
 0.028%   for enter/exit [OpenMP] => paralle
 0.005%   paralleldo  [OpenMP location: file:0
 0.004%   for enter/exit [OpenMP] => paralle
 0.001%   barrier enter/exit [OpenMP] => par
4.1E-4%   barrier enter/exit [OpenMP] => par
1.6E-4%   barrier enter/exit [OpenMP]
1.4E-4%   .TAU application   => parallel begi
1.4E-4%   parallel begin/end [OpenMP]
8.8E-5%   barrier enter/exit [OpenMP] => par
8.5E-5%   for enter/exit [OpenMP]
7.3E-5%   do  [OpenMP location: file:09elasti.
5.5E-5%   parallel  [OpenMP location: file:09e
4.2E-5%   parallel  [OpenMP location: file:09e
3.8E-5%   paralleldo  [OpenMP location: file:0
3.6E-5%   parallel begin/end [OpenMP] => pa
2.8E-5%   parallel begin/end [OpenMP] => pa
2.3E-5%   paralleldo  [OpenMP location: file:0
7.1E-6%   paralleldo  [OpenMP location: file:0
5.3E-6%   paralleldo  [OpenMP location: file:0
1.9E-6%   parallel begin/end [OpenMP] => pa
```

Figure 2.7: Parallel Micromegas: Master thread functions bar chart (*left*) and woker thread functions bar chart (*right*) showing the percentile distribution for each subroutine. In the master thread, the parallelized ELASTI::SIGMA_INT_CP is no longer the primary performance bottleneck and accounts for only 40.558% of the parallel run time over 1,000 time steps on 4 threads. CONTACT::UPDATE is now the most time consuming subroutine and accounts for 58.41% of the parallel run time over 1,000 time steps on 4 threads.



Name	GET_TIME_OF_DAY	GET_TIME_OF_DAY %	GET_TIME_OF_DAY ...	Calls	Child Calls
paralleldo [OpenMP location: file:09elasti.chk.F90 <479, 502>]	0.297	0.0%	0	4,000	3,000
paralleldo [OpenMP location: file:09elasti.chk.F90 <2286, 2439>]	1.144	0.0%	0.003	400	300
parallel fork/join [OpenMP]	1,656.568	40.6%	0.789	2,100	2,100
parallel begin/end [OpenMP]	1,656.485	40.6%	0.789	2,100	2,100
parallel [OpenMP location: file:09elasti.chk.F90 <1747, 2027>]	1,655.123	40.6%	0.552	3,000	3,000
for enter/exit [OpenMP]	1,656.456	40.6%	0.789	2,100	2,100
do [OpenMP location: file:09elasti.chk.F90 <1765, 2025>]	1,655.073	40.6%	0.828	2,000	1,000
barrier enter/exit [OpenMP]	144.653	3.5%	0.047	3,100	3,100
TOPOLO::NET	1.443	0.0%	0.001	1,000	60,391
TOPOLO::DISCRETI	1.212	0.0%	0.001	1,000	87,395
MICROSTRUCTURE::VERIFICATION_CONFIGURATION	0	0.0%	0	1	124
MICROSTRUCTURE::NOEUD	0	0.0%	0	124	1,240
MICROSTRUCTURE::GENERATION_MICROSTRUCTURE	0.009	0.0%	0.009	1	4
MICROSTRUCTURE::CONNECXION_SEGMENTS	0	0.0%	0	1	96
MICROMEGAS	4,080.784	100.0%	4,080.784	1	10,305
INIT::TABULATION	0.244	0.0%	0.244	1	269,824
INIT::PARAMETRAGE	0	0.0%	0	1	1
INIT::LOAD	0.247	0.0%	0.247	1	10
INIT::LIRE_DONNEES	0	0.0%	0	1	1
ELASTI::SOLLI	0.016	0.0%	0	1,000	2,985
ELASTI::SIGMA_INT_LP	1.145	0.0%	0.011	100	100
ELASTI::SIGMA_INT_CP	1,655.126	40.6%	1.655	1,000	1,000
ELASTI::INIT_SOLLI	0	0.0%	0	1	1
ELASTI::FORCE	1,668.363	40.9%	1.668	1,000	203,220
DYNAM::DEPPREDIC	19.671	0.5%	0.02	1,000	200,001
CONTACT::UPDATE	2,383.945	58.4%	2.384	1,000	247,767
CONNEC::CORRIGER_CONFIG	0.952	0.0%	0.001	1,000	4,353
BRICAMAT::VECCOURB	0.124	0.0%	0	200,001	198,825
BRICAMAT::SICOSE2V	0	0.0%	0	192	384
BRICAMAT::REDUIRE_VEC	0.017	0.0%	0	55,296	55,296
BRICAMAT::PPEM	0.001	0.0%	0	1,920	3,840
BRICAMAT::NORMAIVEC	0	0.0%	0	192	192
BRICAMAT::ICOSE2V	0.015	0.0%	0	27,648	110,592
BRICAMAT::ETAT	0.074	0.0%	0	200,001	377,926
BRICAMAT::ENTIER	0.004	0.0%	0	14,208	14,208
BRICAMAT::COSE2V	0.064	0.0%	0	200,001	199,857
.TAU 1000 null timers overhead	0	0.0%	0	1	1,000

Name	Exclusive ...	Inclusive ...	Calls	Child Calls
.TAU application	2,422.366	4,078.81	1	2,100
└ parallel begin/end [OpenMP]	0.006	1,656.444	2,100	2,100
do [OpenMP location: file:09elasti.chk.F90 <1765, 2025>]	1,655.075	1,655.078	2,000	1,000
└ barrier enter/exit [OpenMP]	0.003	148.781	1,000	1,000
parallel [OpenMP location: file:09elasti.chk.F90 <1747, 2027>]	1.962	1,655.1	2,000	2,000
└ for enter/exit [OpenMP]	0.002	1,655.08	1,000	1,000
└ barrier enter/exit [OpenMP]	0.002	0.018	1,000	1,000
paralleldo [OpenMP location: file:09elasti.chk.F90 <2286, 2439>]	1.143	1.144	300	200
└ for enter/exit [OpenMP]	0	1.144	100	100
└ barrier enter/exit [OpenMP]	0	0.004	100	100
paralleldo [OpenMP location: file:09elasti.chk.F90 <479, 502>]	0.192	0.194	3,000	2,000
└ barrier enter/exit [OpenMP]	0.002	0.045	1,000	1,000
└ for enter/exit [OpenMP]	0.001	0.193	1,000	1,000
barrier enter/exit [OpenMP]	0.006	148.848	3,100	3,100
└ do [OpenMP location: file:09elasti.chk.F90 <1765, 2025>]	148.778	148.778	1,000	0
└ paralleldo [OpenMP location: file:09elasti.chk.F90 <479, 502>]	0.043	0.043	1,000	0
└ parallel [OpenMP location: file:09elasti.chk.F90 <1747, 2027>]	0.017	0.017	1,000	0
└ paralleldo [OpenMP location: file:09elasti.chk.F90 <2286, 2439>]	0.004	0.004	100	0
parallel begin/end [OpenMP]	0.006	1,656.444	2,100	2,100
└ parallel [OpenMP location: file:09elasti.chk.F90 <1747, 2027>]	0.001	1,655.1	1,000	2,000
└ paralleldo [OpenMP location: file:09elasti.chk.F90 <479, 502>]	0.001	0.194	1,000	1,000
└ paralleldo [OpenMP location: file:09elasti.chk.F90 <2286, 2439>]	0	1.144	100	100
for enter/exit [OpenMP]	0.003	1,656.417	2,100	2,100
└ do [OpenMP location: file:09elasti.chk.F90 <1765, 2025>]	1,506.297	1,655.078	1,000	1,000
└ paralleldo [OpenMP location: file:09elasti.chk.F90 <2286, 2439>]	1.139	1.143	100	100
└ paralleldo [OpenMP location: file:09elasti.chk.F90 <479, 502>]	0.148	0.192	1,000	1,000

Figure 2.8: Parallel Micromegas: Callpath relations in the master thread (*left*) and the worker thread (*right*) showing the inclusive and exclusive time together with the total number of calls and child calls per each routine.

Following parallelization and thread safety, we collected profiling data for the execution of the parallelized Micromegas. Figure 7 shows the execution callgraphs of the master (left) and worker threads (right). In the fork-and-join model, only the master thread "out-lives" the execution of a parallelized code region, while the worker threads "are alive" only inside a parallelized code region. The callgraph of the master thread subsumes the worker thread callgraph. This is due to the fact that inside a parallel code region the master thread behaves just like a worker thread, as dictated by the fork-and-join model. This explains the similarity between the callgraph of the master thread and the callgraph of the main thread of the serial execution from Figure 4. The callgraph of a worker thread is the same for all worker threads.

Similarly to Figure 4, the red box in Figure 7 (right) indicates the most time consuming subroutine of the application. In this case, however, the red box indicates the subroutine UPDATE, which in the serial code was indicated in a green box. Also, subroutine SIGMA_INT_CP is no longer indicated in a red box but in a blue box, which signifies that it is no longer the most time consuming part of the application.

Figure 9 shows the ordering with respect to the percentile distribution of the total parallel time per module/subroutine executed by the master thread and by the worker threads. In this figure, the ordering for the subroutines executed by the master thread is different from the ordering of the subroutines executed by the single thread in Figure 3. According to Figure 5, subroutine CONTACT::UPDATE became the most important performance bottleneck in the parallel execution, accounting for 58.41% of the parallel run time over 1,000 time steps. The second most time consuming part corresponds to the parallel region of ELASTI::SIGMA_INT_CP and accounts for 40.558% of the parallel run time over 1,000 time steps. Figure 9 illustrates the callpath relations among the most important subroutines in the master thread and worker threads, respectively.

In addition to profiling, we also collected traces of the parallel application for 10 time steps. These traces were again produced by TAU and visualized with Jumpshot. Figure 8 illustrates the parallel execution trace on 4 cores visualized with Jumpshot. The legends highlight the most time consuming subroutines, i.e., FORCE, SIGMA_INT_CP and UPDATE.

2.5 Tuning the parallel application code

The choice of the thread scheduling method among the three methods supported in OpenMP, i.e., static, dynamic or guided, is very important and can have a high impact on the performance of the parallelized application code. Dynamic self-scheduling has been proven to provide the best parallel performance in general. However, the overhead associated with this choice of scheduling can be a considerable performance degradation factor; dynamic self-scheduling should be chosen when the size of the loop compensates for this overhead.

Due to the longtime nature of the simulations, so far we were able to investigate only the static scheduling methodologies. We chose static scheduling for distributing the iterations of the outermost loop in SIGMA_INT_CP, which iterates over the number of boxes, where the chunk size was equal to the number of boxes divided by the number of threads. Then, we used selected a different chunk size as half of the original chunk size, that is the number of boxes divides by twice the number of threads. The results presented in this work were obtained with static scheduling and the second choice of chunk size. Due to the fact that all cores are homogeneous and due to the dedicated test system, the master thread executes SIGMA_INT_CP in a perfectly load balanced fashion simultaneously with the worker threads.

Figure 11 gives a 3-D view of the time distribution per subroutines over 1,000 time steps for each thread. The master thread (0.0) together with the worker threads (0.1, 0.2 and 0.3) execute ELASTI::SIGMA_INT_CP simultaneously in a load balanced fashion. However, only the master thread (0.0) executes CONTACT::UPDATE, which is the second most time consuming part of the original application code.

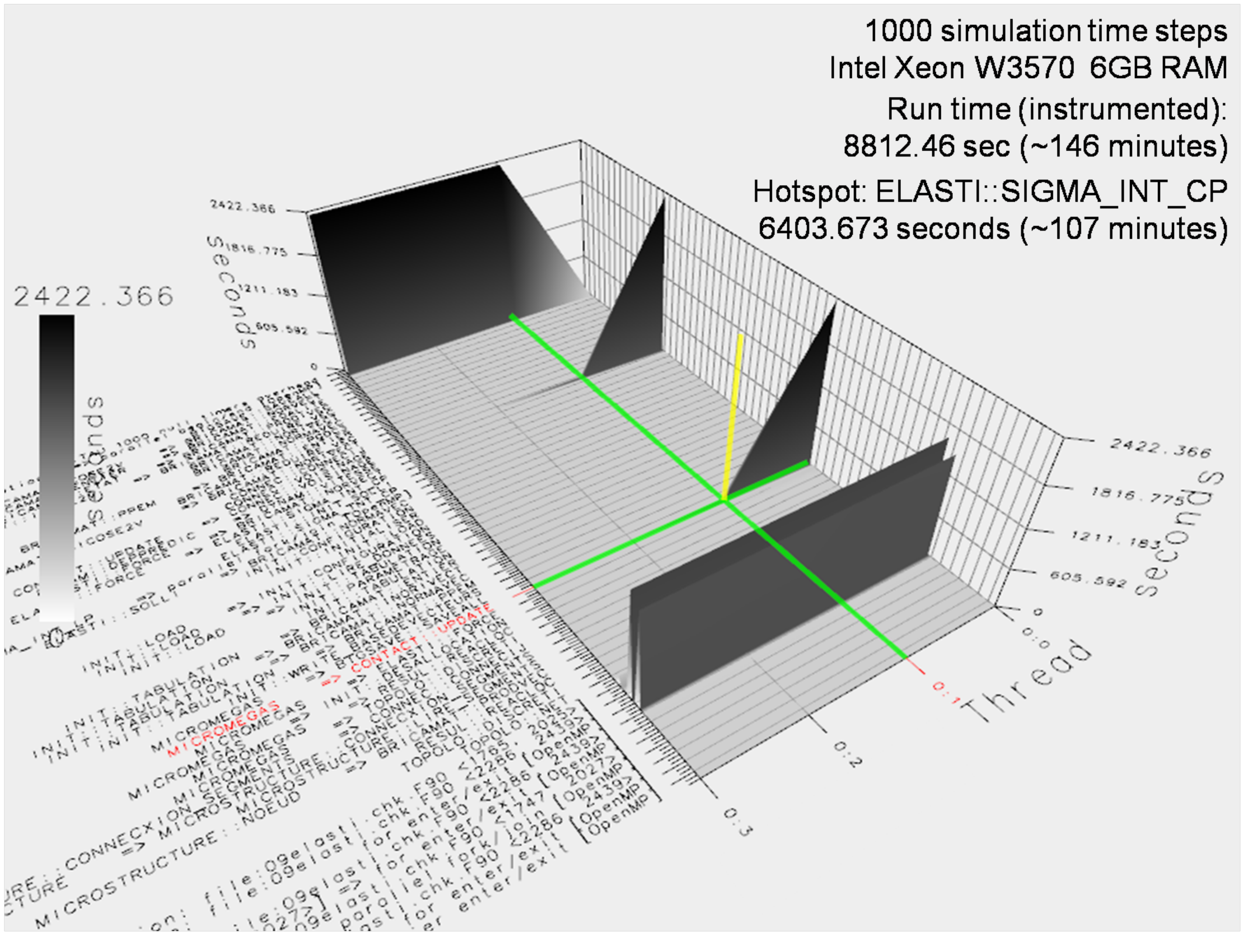


Figure 2.9: Parallel Micromegas: 3D view of the time distribution per subroutines for each thread over 1,000 time steps. Master thread (0.0) is responsible for executing CONTACT::UPDATE, the second most time consuming part of the original application code. Master 0.0 together with worker threads 0.1, 0.2 and 0.3 execute ELASTI::SIGMA_INT_CP in parallel in a load balanced fashion.)

2.6 Verifying the correctness of the parallel application code

The final step in the parallelization methodology deals with the correctness of the results obtained with the parallel application code. To achieve this we conducted two sets of serial and parallel tests. The goal of the first set was to verify the total number of segments, N, and the number of moving segments with length greater than zero, ND, produced by the serial and parallel application codes. The tests indeed verify that evolution of N and ND over 30,000 time steps is the same for both serial and parallel application and is depicted in Figure 12.

Encouraged by the first set of verification tests, we compared the mechanical properties (stress-strain curve, resolved sheer stress response and dislocation density) produced by the serial and parallel application codes. The verification test set shown in Figure 13, confirms that the mechanical properties produced by the serial and the parallel version of Micromegas are the same.

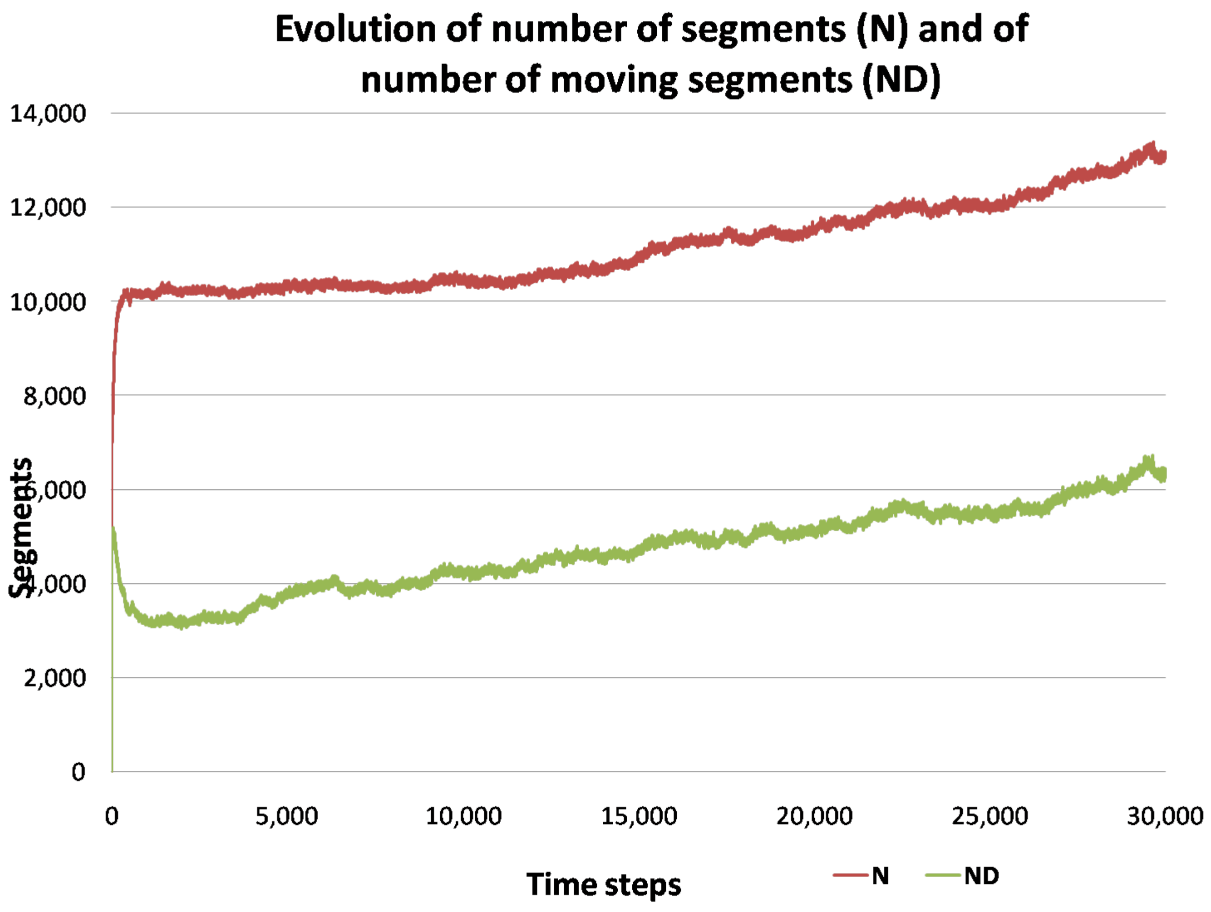


Figure 2.10: Evolution of the total number of segments, N, and the number of dislocated segments, ND, over 30,000 time steps. N and ND from the parallel simulation run are equal to those from the serial simulation run.

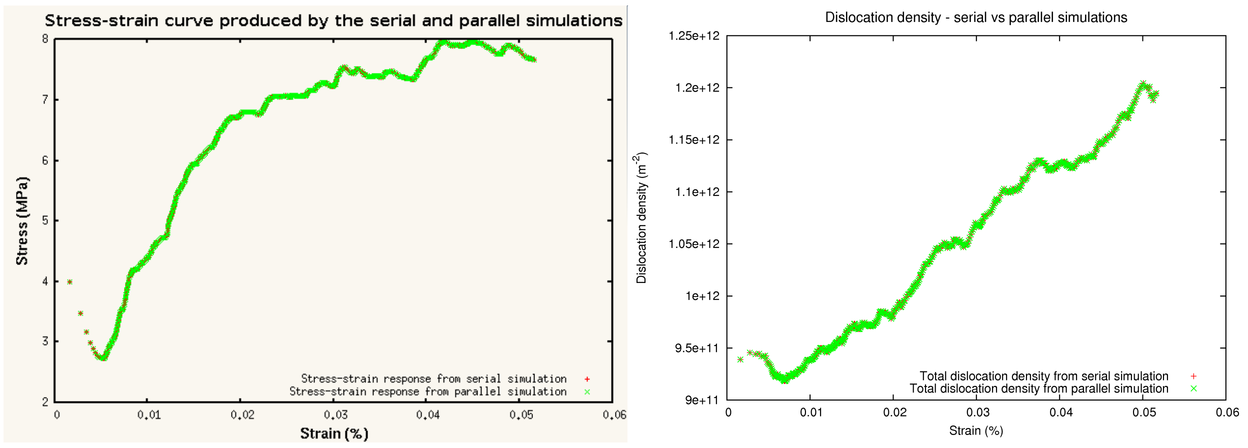


Figure 2.11: Validation of scientific data obtained from the parallel simulation against scientific data obtained from the serial simulation over 30,000 time steps.

Chapter 3

Evaluation

Improving the performance of Micromegas required the analysis of the original application code, its parallelization and its evaluation under conditions similar to those of a representative DDD simulation. The simulation parameters of a representative Micromegas simulation are: 0.5$\%$ of plastic deformation, in a box of dimension 10x10x10 μm^3 with an initial density of 10^12 m^{-2} and a strain rate of 10 s^{-1} in multislip conditions. Multislip calculations were performed to demonstrate the efficiency of the parallel version of the application code. Representative volume elements of Al (FCC crystal structure with Burgers vector of magnitude b = 2.86) of dimensions 9x10x12 μm^3 were loaded along the [001] direction with a strain rate of 20 s^{-1} at a temperature of 300 K. Screw dislocations were not allowed to cross-slip at any time, while a time step of 10^{-9} seconds was considered.

The evaluation tests were conducted on a commodity 4-way Intel Xeon W3570 processor, running at 3.2GHz with 6GB DDR-3 RAM, SLES 10 OS, 2.6.16.60 Linux kernel. We are interested in the speedup and efficiency of the parallel application code relative to the serial application code.

In the first step of the parallelization methodology it was determined that SIGMA_INT_CP, accounts for $\sim$73$\%$ of the serial execution time. With infinite parallelism, there is a theoretical speedup of 4.54 as dictated by Amdahl's law. Attributing the parallelizable part of Micromegas to SIGMA_INT_CP, we can say that S=27$\%$ is the serial part of Micromegas. Assuming P=4 available processing elements, the speedup through parallelism on these processing elements is $1/(S + ((1 - S)/P) = 2.20$, while the theoretical speedup limit assuming $P- > \infty$ is then $1/S = 4.54$.

Figure 14 shows the speedup of the parallel application code relative to the serial application code for simulations over 30,000 time steps on 4 threads. The efficiency (E) of the parallel application code is shown for each parallel test case. Both performance metrics (speedup and efficiency) demonstrate the scalability of parallel Micromegas with the number of time steps increasing from 5,000 to 30,000.

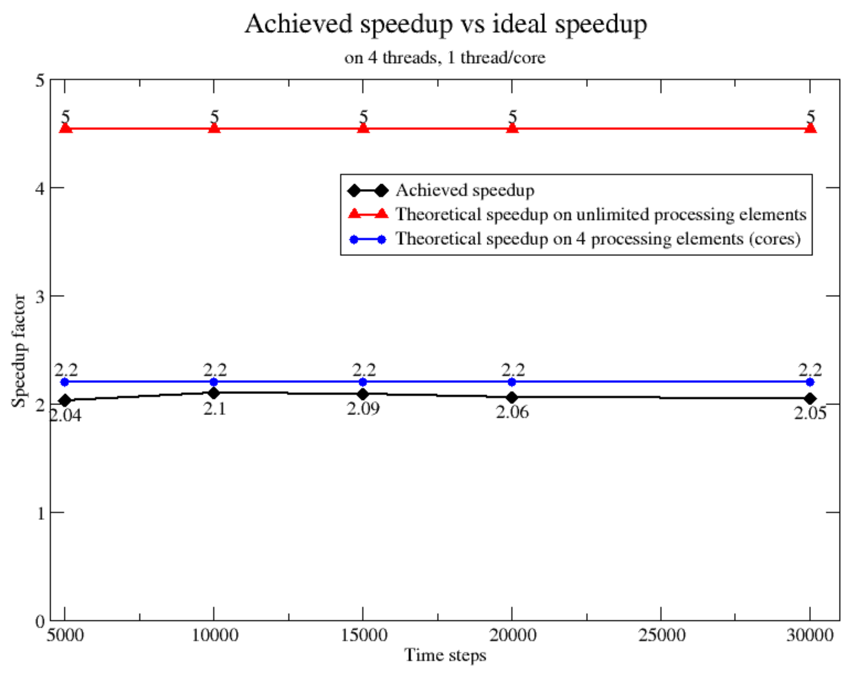


Figure 3.1: Speedup of the parallel application code relative to the serial application code over 30,000 time steps on 4 threads. The efficiency (E) of the parallel application code is shown for each parallel test case. Both performance metrics (speedup and efficiency) show the scalability of parallel Micromegas with the number of time steps increasing from 5,000 to 30,000.

Chapter 4

Conclusions and Future Work

Advancements in application analysis tools have made parallelization of complex application codes an easier task than before. Good implementation practices together with a fair knowledge of the application code, however, are imperative for making parallelization a less complicated and faster task.

In this work we narrowed the gap between Micromegas and multi-core computing systems. We achieved this via a ‘performance lift’ based on the multi-threaded parallelization of the major performance bottleneck of the original application code. The outcome of this work is an improved simulation code for research in discrete dislocation dynamics for materials science. The significance of this work lies in that the improvements in the simulation code permit the scientist to perform longer simulations of higher temporal fidelity, using more processors, which were not feasible previously.

The results presented in this work are the first efforts to parallelize Micromegas. There is certainly room for improving the current results that could be addressed via several directions, which may include different scheduling methods for the parallelized subroutines, scaling the parallel code beyond 4 cores, or even using graphics-based accelerators (GPUs). Such directions will be addressed in future work.

Acknowledgments

The authors would like to thank the developers of TAU for their valuable support in installing and configuring the program and performance analysis tool framework. The funding provided for this study by the US Department of Energy under Grant No. 008860-013 and by the Center for Advanced Vehicular Systems of the Mississippi State University is gratefully acknowledged.

Bibliography

[1] Amadeo, R., Ghoniem, N., 1990. "Dislocation dynamics. i. a proposed methodology for deformation micromechanics,"*Phys. Rev. B 41*, pp. 69586967.

[2] Arsenlis, A., Cai, W., Tang, M., et al., 2007. "Enabling strain hardening simulations with dislocation dynamics," *Modelling Simul. Mater. Sci. Eng. 15*, pp. 553-95.

[3] Bacon, D., 1967. "A method for describing a flexible dislocation,"*Phys. Stat. Sol. 23*, pp. 527538.

[4] Brown, L., 1964." The self-stress of dislocations and the shape of extended nodes." Phil. Mag. 10, 44166.

[5] de Wit, R., 1967, "Some Relations for Straight Dislocations." Phys. Stat. Sol. 20, 567-73.

[6] Devincre, B., 1995, "Three dimensional stress fields expressions for straight dislocation segments." Solid State Communication 93, pp. 875-8.

[7] Devincre, B., Kubin, L., Lemarchand, C., Madec, R., 2001. "Mesoscopic simulations of plastic deformation." Mat. Sci. and Eng. A 309-310, pp. 2119.

[8] Durinck, J., Devincre, B, Kubin, L.P. and Cordier, P., 2007. "Modeling the plastic deformation of olivine by dislocation dynamics simulations." American Mineralogist 92, pp. 1246-57.

[9] Foreman, A., 1967. "The bowing of a dislocation segment." Phil. Mag. 15, pp. 101121.

[10] Ghoniem, N.M. and Sun L.Z., 1999. "Fast-sum method for the elastic field of three-dimensional dislocation ensembles." Phys. Rev. B 60, pp. 128-40.

[11] Ghoniem, N.M., Huang, J., and Wang, Z., 2002. "Affine covariant-contravariant vector forms for the elastic field of parametric dislocations in isotropic crystals," Phil. Mag. Lett. 82, pp. 5563.

[12] Groh, S. and Zbib, H.M., 2009. "Advances in discrete dislocations dynamics and multiscale modeling," J. Eng. Mat. Tech. 31, pp. 041209-1.

[13] Hirth, J.P., Zbib, H.M. and Lothe J., 1998. "Forces on High Velocity Dislocations." Modelling Simul. Mater. Sci. Eng. 6, pp. 165-69.

[14] Khraishi, T, Zbib, H.M., Diaz de la Rubia T. and Victoria, M., 2001. “Modeling of Irradiation in Metals Using Dislocation Dynamics.” Philos. Mag. Letters 81, pp. 583-93.

[15] Khraishi, T, Zbib, H.M., T. Diaz de la Rubia and M. Victoria, 2002, “Localized Deformation and Hardening In Irradiated Metals: Three-Dimensional Discrete Dislocation Dynamics Simulations,” Metallurgical and Materials Transactions B, 33B, pp. 285-96.

[16] Kristan J. and Kratochvil, J., 2007. “Interactions of glide dislocations in a channel of a persistent slip band.” Philos. Mag. 87, pp. 4593-613.

[17] Kristan J. and Kratochvil, J., 2008. “Estimates of Stress in the Channel of Persistent Slip Bands Based on Dislocation Dynamics,” A special issue of Mater. Sci. Forum, pp. 4058.

[18] Kubin, L., Canova, G., Condat, M., Devincre, B., Pontikis, V., Brechet, Y., 1992. Solid State Phenomena 23-24, pp. 45572.

[19] Madec, R., Devincre, B., Kubin, L., 2001. “New line model for optimized dislocation dynamics simulations.” In L.P. Kubin, J.L. Bassani, K. Cho and R.L.B. Selinger (M. materials modeling, ed.), Mat. Res. Soc. Symp. Proc. 653 Z1.8.16.

[20] Madec, R. and Kubin, L.P., 2008. “Second-order junctions and strain hardening in BCC and FCC crystals.” Scripta Mater. 58, pp. 767-70.

[21] Monnet, G., Devincre, B. and Kubin, L.P., 2004. “Dislocation study of prismatic slip systems and their interactions in hexagonal close packed metals: application to zirconium,” 52, pp. 4217-28.

[22] Moulin, A., Condat, M. and Kubin. L.P. 1997. “Simulation of Frank-Read sources in Silicon.” Acta mater., 45, pp. 2339.

[23] Rhee, M., Zbib, H.M., Hirth, J.P., Huang, H. and de La Rubia T. D., 1998. “Models for Long/Short Range Interactions in 3D Dislocation Simulation.” Modelling Simul. Mater. Sci. Eng., 6, pp. 467-92.

[24] Schwarz, K.W., 1999. “Simulation of dislocations on the mesoscopic scale. I. Methods and examples.” Journal Appl. Phys. 85, pp. 108-119.

[25] Schwarz, K.W., 1999. “Simulation of dislocations on the mesoscopic scale. II. Application to strained-layer relaxation.” Journal Appl. Phys. 85, pp. 120-9.

[26] Shin, C.S., Fivel, M.C., Verdier, M. and Kwon, S.C., 2006. “Numerical methods to improve the computing efficiency of discrete dislocation dynamics simulations.” J. Comp. Physics, 215, pp. 417 - 429.

[27] Tang, M., Kubin, L.P. and Canova G.R, 1998. “Dislocation mobility and the mechanical response of b.c.c. single crystals: a mesoscopic approach.” Acta Mater. 46, pp. 3221-35.

[28] Verdier, M., Fivel, M. and Groma, I., 1998. “Mesoscopic scale simulation of dislocation dynamics in FCC metals: principles and applications.” Modelling Simulation Mater. Sci. Eng. 6, pp. 755770.

[29] Wang Z.Q., Beyerlein I.J. and LeSar R., 2007. “The importance of cross-slip in high rate deformation.” Modelling Simul. Mater. Sci. Eng. 15, pp. 675-690.

[30] Zbib, H.M., Rhee, M. and Hirth, J.P., 1996. “3D Simulation of Curved Dislocations: Discretization and Long Range Interactions,” in: Advances in Engineering Plasticity and its Applications, eds. T. Abe and T. Tsuta. Pergamon, NY, pp. 15-20.

[31] Zbib, H.M., Rhee, M and Hirth, J.P., 1998. “On plastic deformation and the dynamics of 3d dislocation.” J. Mech. Sci. 40, pp. 11327.

[32] Zbib, H. M., de La Rubia, T. D., Rhee, M., Hirth, J. P., “3D Dislocation Dynamics: Stress-Strain behavior and Hardening Mechanisms in FCC and BCC Metals”, J. Nuc. Maters., 276, pp. 154-165, 2000.

[33] Zbib, H.M. and Diaz de la Rubia, T., 2001. “A Multiscale Model of Plasticity: Patterning and Localization.” Material Science For the 21st Century, Ed. Vol A, The Society of Materials Science, Japan, pp. 341-7.

[34] Senger, J., Augustin, W., Weygand, D. M., Kraft, O., Heuveline, V., Gumbsch, P., 2006. “Parallel Discrete Dislocation Dynamics: Stress Distribution in Polycrystalline Metallic Film.” Multiscale Modeling of Materials, Materials Research Society Fall 2006 Meeting.

[35] The OpenMP API specification for parallel programming, [http://openmp.org/wp/openmp-specifications/]

[36] TAU Performance System, [http://www.cs.uoregon.edu/research/tau/home.php]

[37] Performance Visualization for Parallel Programs, online, [http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm#Jumpshot]