WEB-BASED FRONT-END DESIGN AND SCIENTIFIC COMPUTING FOR MATERIAL STRESS SIMULATION SOFTWARE

A Thesis
Presented to
The Academic Faculty

by

Tien-Ju LIN

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
Computer Science

Georgia Institute of Technology
December 2014

WEB-BASED FRONT-END DESIGN AND SCIENTIFIC COMPUTING FOR MATERIAL STRESS SIMULATION SOFTWARE

Approved by:

Dr. Cedric Pradalier, Advisor
Advisor and Committee Chair
School of Computer Science
Georgia Institute of Technology

Dr. Laurent Capolungo
School of Mechanical Engineering
Georgia Institute of Technology

Dr. Ronald R. Hutchins
Office of Information Technology
Georgia Institute of Technology

Date Approved: December 3, 2014

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my advisor, Dr. Cedric Pradalier for giving me this opportunity to grow as a researcher and for his valuable suggestions. The completion of my thesis would not have been possible without his constant support and unwavering guidance.

I would like to thank Dr. Laurent Capolungo for giving me a chance to collaborate with his team and to develop an application with creative freedom.

I would like to thank Dr. Ronald R. Hutchins for reviewing my thesis and providing his precious advices.

I would like to thank Dr. Stéphane Vialle for helping me enlarge my knowledge in the field of High Performance Computing.

I would like to acknowledge the help of Nicolas Bertin, with whom I discussed the basics of material science and the structure of material-stress simulation software.

Finally, I appreciate very much the support of Georgia Tech Lorraine and Supelec Metz that provided me all research resources for my work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

DDD	Discrete Dislocation Dynamics
CLI	Command Line Interface
MPI	Message Passing Interface
HTTP	Hypertext Transfer Protocol
MVC	Model-View-Controller
REST	Representational State Transfer
DOM	Document Object Model
API	Application Programming Interface
JSON	JavaScript Object Notation
DFS	Depth First Search
BFS	Breadth First Search
CGAL	Computational Geometry Algorithms Library

SUMMARY

In this thesis, we will discuss the front-end design and the algorithm optimization necessary in order to build successful material-stress simulation software that can satisfy both research needs and educational needs. A precise simulation requires a large amount of input data such as geometrical descriptions of the crystal structure, the external forces and loads, and quantitative properties of the material. Although some powerful applications already exist for research purposes, they are not widely used in education due to complex structure and unintuitive operation. To cater to the generic user base, a front-end application for material simulation software is introduced. With a graphic interface, it provides a more efficient way to conduct the simulation and to educate students who want to enlarge knowledge in relevant fields. We first discuss how we explore the solution for the front-end application and how to develop it on top of the material simulation software developed by mechanical engineering lab from Georgia Tech Lorraine. The user interface design, the functionality and the whole user experience are primary factors determining the product success or failure. This material simulation software helps researchers resolve the motion and the interactions of a large ensemble of dislocations for single or multi-layered 3D materials. However, the algorithm it utilizes is not well optimized and parallelized, so its performance of speedup cannot scale when using more CPUs in the cluster. This problem leads to the second topic on scientific computing, so in this thesis we offer different approaches that attempt to improve the parallelization and optimize the scalability. These will be presented in details along with the comparison of test results.

CHAPTER 1

INTRODUCTION

With the growing attention to material science over the past years, more and more researchers have dedicated themselves to developing simulation software to study the fundamental mechanism of plasticity. NumoDIS [2] has been jointly introduced by CEA and CNRS, but it's not accessible to the public due to its limited distribution. Another popular software, ParaDIS[1,3], has been developed to enable massive dislocations simulation on a supercluster with more than 1000 CPUs. Its highly parallel and complex structure makes it difficult to extend. Moreover, these two software packages are operated through Command-Line Interface (CLI), giving a novice user with limited knowledge in mechanical engineering and of how the software generally works, a steep learning curve in creating an input file and running the simulation.

The downsides presented in these two software packages may not be significant for the researchers in the laboratories because they are generally well trained and equipped with professional hardware and software. However, for college students, the complexities discussed definitely reduce the educational effectiveness due to the inaccessibility and the finite computing resources available to run the software package on. To resolve this difficulty, we propose a web-based application based on our own material simulation software. With a simple graphical user interface, it can abstract the command line operation and thereby help users generate input files intuitively in the presence of various functions. The entire front-end development mainly involves data management, UI Design and server setup. In the following sections, the details concerning application development will be presented. The objective of this application is to offer a platform-free tool providing an effective and efficient education for college

students who study in related fields. By distributing it as free software so that everyone can easily access, download, and learn it.

Discrete Dislocation Dynamics (DDD) code developed by Intermat Lab is a parallel software package written in Fortran 90 using Message Passing Interface (MPI) [23] that aims at simulating the motion and interactions of a large ensemble of dislocations for single or multi-layered 3D materials on which loading conditions are applied [8,9]. A dislocation is a linear defect in a material. Due to its nature, it corresponds to a discontinuity in the lattice structure and hence induces strain and stress fields. When subject to stress, dislocations in a material can glide and interact with each other. The origin of the stress acting on a dislocation comes from two contributions: 1) internal stress generated by the dislocations present in the material and 2) external stress coming from the applied loading conditions. Due to the non-linear behavior of the dislocation motion and interaction processes, the simulation is performed through an iterative process. At each time step, the motion of all the dislocations is computed and the interactions are performed if applicable. Once the dislocation lines are updated, the mechanical state associated to the newly computed dislocation configuration is evaluated and the code can proceed to the next step. The software will perform as many steps as required in order to complete the simulation run.

Moreover, this program is designed to run a small or medium scale dislocation simulation, so the user can conduct it either with a personal laptop or with a professional cluster in the laboratory. Knowing that the computational complexity for dislocation dynamics is $O(n^2)$ due to the forces interaction between dislocations, the computation cost will become enormous if there are a number of dislocations in crystal structure. This is often the case in real world. Thus, parallelizing the algorithm could use up to 100 processes, thereby reduce computation time significantly reduced when analyzing a more complex scenario. However, the performance of speedup of the current DDD algorithm

doesn't scale well. The need to optimize the DDD algorithm along with the approaches of scientific computing is the second focus of this work.

Over past years, much research has been done in the fields of Algorithm Engineering and High Performance Computing. For example, cache-oblivious algorithms [4] are efficient at all levels of the memory hierarchy in theory, but so far these algorithms are widely used in practice. In [5], a new highly scalable distributed memory parallel algorithm is designed and implemented for resolving graph matching and vertex coloring problems. Shared memory parallelization is also introduced in a data mining algorithm [6] to improve the speed. Nowadays, some incredibly complex problems are hard to solve even with large resources. In this case, instead of applying an exact algorithm, a heurist approach [7] can still result in approximate solutions and have acceptable run time. To sum up, some approaches can be general applied to all kinds of problems, but others are only used in particular types of problems. In our case, we firstly studied and analyzed the existing structure of a program, MPI communication and core algorithms for the mechanical calculation. After identifying the possible causes of poor performances, we attempted to apply different parallelization schemes with new data structure, implement better algorithms to deal with graph problems and test other heuristic approaches. For each modification, we measured the execution time and compared the test results to previous outcome. This work gave us clear idea of how to redesign the code as a parallel algorithm and how to construct the program in a more structured way.

This thesis aims at developing a user friendly portal which allows users (experts, scholars, college students) to conduct material stress-strain simulations and analyze the results in an efficient way. A second objective is to provide a variety of insights for the optimization of DDD algorithm by means of scientific computing.

CHAPTER 2

FRONT-END DESIGN

Before entering the main discussion about the design and development, we need to explain what kind the advantages the front-end application will bring and how it will improve the user experiences because these features of the front-end will guide the functional specification and the architecture. Using the traditional workflow of executing the material-stress simulation as in Figure 1, the user edits a long input file composed of around 100 quantitative fields without preliminary guidance. One careless error may result in the crash of program, and the user must check those 100 lines again to carry out the modification. Once the simulation is completed, the user needs to open the output files in a particular folder, and then draw the charts and watch the animation with another software package. The whole procedure is lengthy for researchers and not efficient for learning process. Therefore, our team took these important points into the product design and expected to achieve two major goals: 1) facilitate the generation of the input file and 2) provide an overview of the simulation results. As for the deployment, we would like to distribute the laptop version to every college student who is interested in material science and also install the hosted version in the cluster for special research projects.

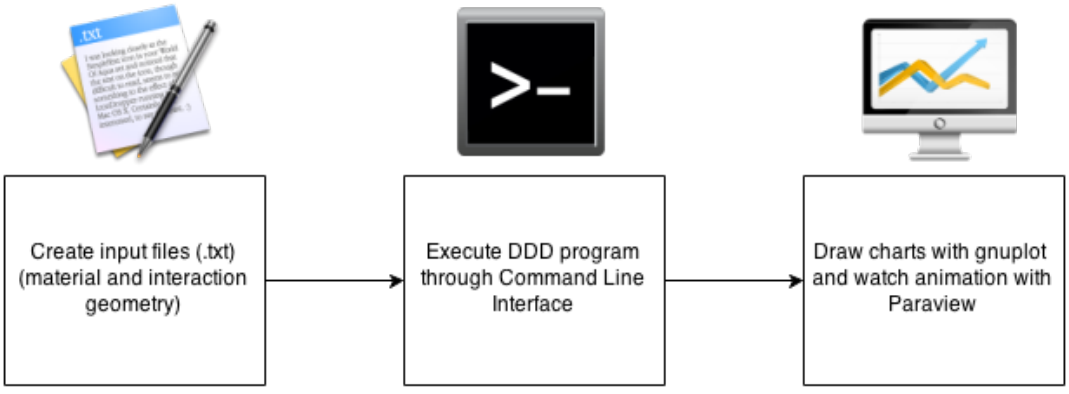


Figure 1 Workflow of executing the material-stress simulation

2.1 Architecture

2.1.1 Web based application

As mentioned previously, we need two versions of the application that satisfy the research needs and educational needs respectively: 1) a laptop version for personal use to be installed locally 2) a hosted version for professional use to be deployed in the dedicated cluster. There are a variety of solutions in the industry to use for building a front-end application. They can be simply divided into two groups, desktop software and web-based application. For desktop software, we can choose to have DDD binary executable built-in but this would require that the package is configurable for different operating systems. Even if the DDD software is excluded, the user is not able to run complex simulations without extra communication capabilities to access the centralized cluster that provides more computational resources for research projects. Moreover, our team wants to develop the first efficient prototype of application with two versions, so we chose a web-based solution because this method of application design can offer a range of benefits [11] as follows:

- *Accessible anywhere*: Users access the application from any computer connected to the Internet using a standard browser like IE or Firefox. No installation is required in advance.
- *Effective development*: While the user interaction with the application needs to be thoroughly tested on different web browsers, the application itself needs only be developed for a single operating system. There is no need to develop and test the application on all possible operating system versions and configurations.
- *Easy customization*: Web programming language such as HTML, Javascript and CSS makes it easier to update the look of the application, or to provide customized information to different user groups.

- *Security & Maintenance*: Web-based applications are typically deployed on dedicated servers. This is more effective than monitoring hundreds or even thousands of client computers, as is the case with desktop applications.

The above advantages defeat the desktop solution because the web-based application meets better the needs for users. Everyone can easily access and use it, and the administrator can also upgrade the application quickly once the new version is released.

The diagram shown in Figure 2 represents the general architecture of the DDD portal. For front-end development, there are many options such as HTML5/CSS/Javascript, Ruby on Rails, Django, etc. Due to our familiarity, we chose HTML5/CSS/Javascript to build the user interface adding some useful plugins and frameworks. To manage the data transfer and carry out the communication between the portal and the database or the file system, a light weight HTTP server (or DDD server) is needed. Here we implemented it in Python with which we had more experience, but Java, C#, node.JS or other programming languages are also available for the server setup. In addition, an Apache server is in between the DDD server and the DDD portal. This is because the first prototype of application also aims for a centralized version which an authorized user can access to conduct complex simulations. Apache server only takes responsibility for monitoring network security and relays the HTTP request and HTTP response to the client. More details concerning each module will be presented in the subsequent sections.

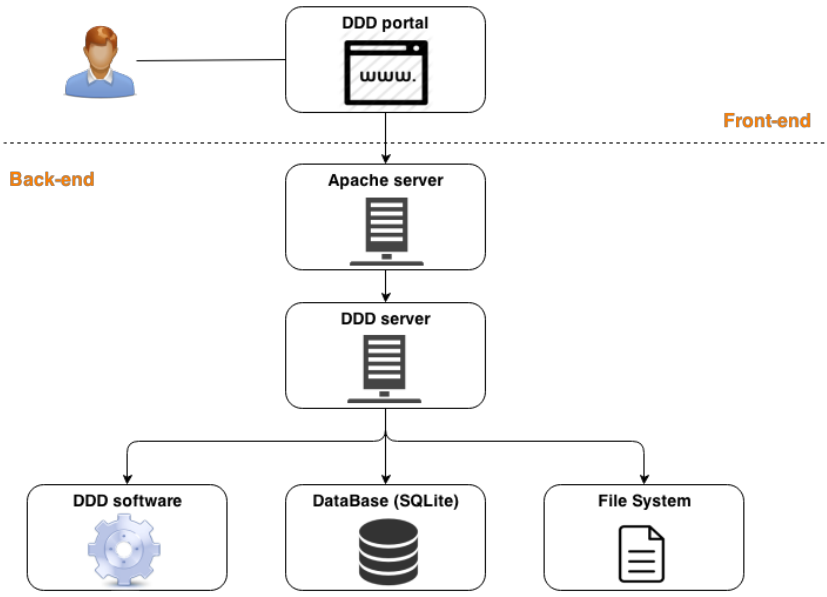


Figure 2 The architecture of DDD portal

2.1.2 Applied Frameworks & Libraries

To develop the DDD portal more efficiently, a variety of Javascript APIs providing different basic functionalities is used in this project. On top of that, we leveraged them to accomplish more advanced goals or tasks required for the user interface.

Backbone.js [12] gives a fundamental basis to the DDD portal due to its Model-View-Controller (MVC) structure shown in Figure 3. It provides models with key-value binding and custom events, collections with a rich Application Programming Interface (API) of enumerable functions, views with declarative event handling, and connects it all to the existing API over a Representational State Transfer (RESTful) interface. With its structure, the template of each page can be quickly created and the contents can be easily rendered after different user actions.

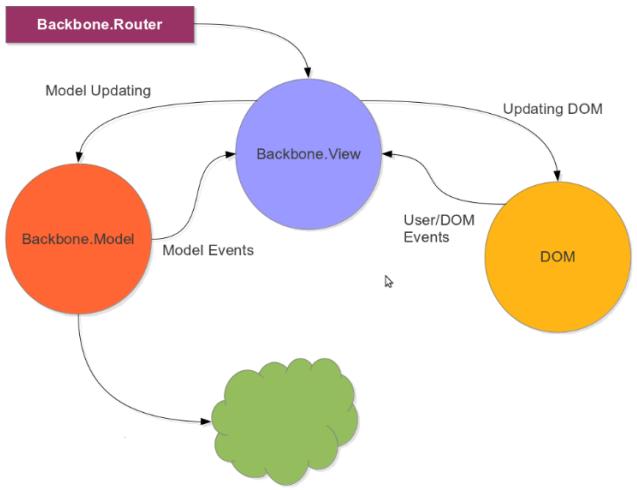


Figure 3 The relationship between Document Object Model (DOM) and MVC structure of *Backbon.js* [12]

Knowing that the configurations of material simulations are the main information to be displayed, two useful plugins, *jQuery Steps* and *DataTable*, are included to organize the input fields and arrange the tabular data. The user interface is also expected to have the data visualization, so *jqplot* [13] is used to build the charts and render 3D graphics. Apart from those small plugins, a general front-end framework is required. *Bootstrap* [14] provides plenty of interface components such as navigation, forms, buttons and even design templates, which help develop a responsive web application and customize the layout for each page. Table 1 shows the list of plugins used in the front-end development.

Table 1 The list of JS Plugins

JS Library	Purpose	Page
Backbone.js	MVC structure	All
JQuery Steps	Wizard building	Configuration
DataTable	Tabular data arrangement	Simulation
Three.js	3D Rendering	Configuration
OrbitControl.js	3D Orbit control	Configuration
jqplot.js	Chart plot	Report
Bootstrap	Interface Components	All

2.1.3 Database

Database management systems (DBMSs) can be divided into two categories, desktop databases and server databases. Desktop databases reside on standard personal computers mostly for single-user applications and server databases contain mechanisms to ensure the reliability and consistency of data and are geared toward multi-user applications. In order to build the first prototype of the DDD portal as fast as possible, our team chose SQLite as the database due to its efficiency, independence and simplicity. Moreover, SQLite also provides good portability, allowing data to be easily shared, duplicated or even removed. Nevertheless, for future version of the DDD portal, a server database such as MySQL or PostgreSQL may be better for scalability, concurrency and centralization.

Knowing that a long input file is composed of around 100 different fields, we divided them into certain high-level groups and created the corresponding tables. Although most of them share a one-to-one relation with the main table *Configuration*, separating them still provides the convenience in managing data more efficiently. Besides, matrices of different dimensions are frequently used in the input file, so we also established the additional tables for data reuse. Below Figure 4 and Figure 5 are two examples of entity-relation diagrams. A simulation (or experiment) has only one configuration that connects to different groups of quantitative fields such as volume information, numerical procedure, simulation control, etc. However, a simulation doesn't necessarily have one report unless its configuration is completed and is executed. The second diagram illustrates the surrounding relation of table *Material.*

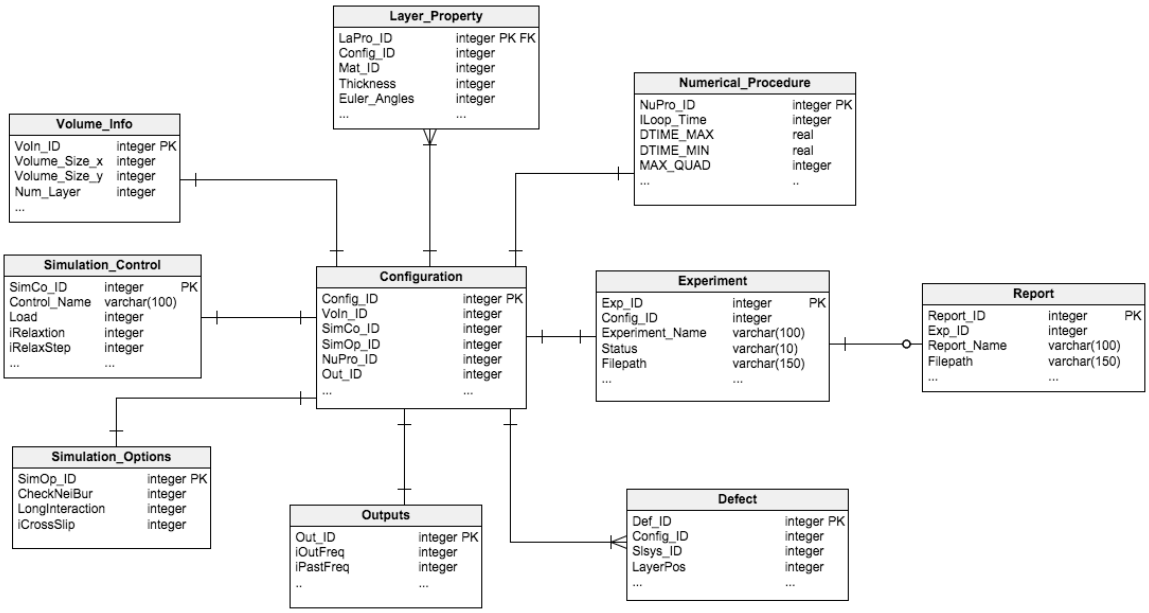


Figure 4 Entity-Relationship diagram for table *Configuration*

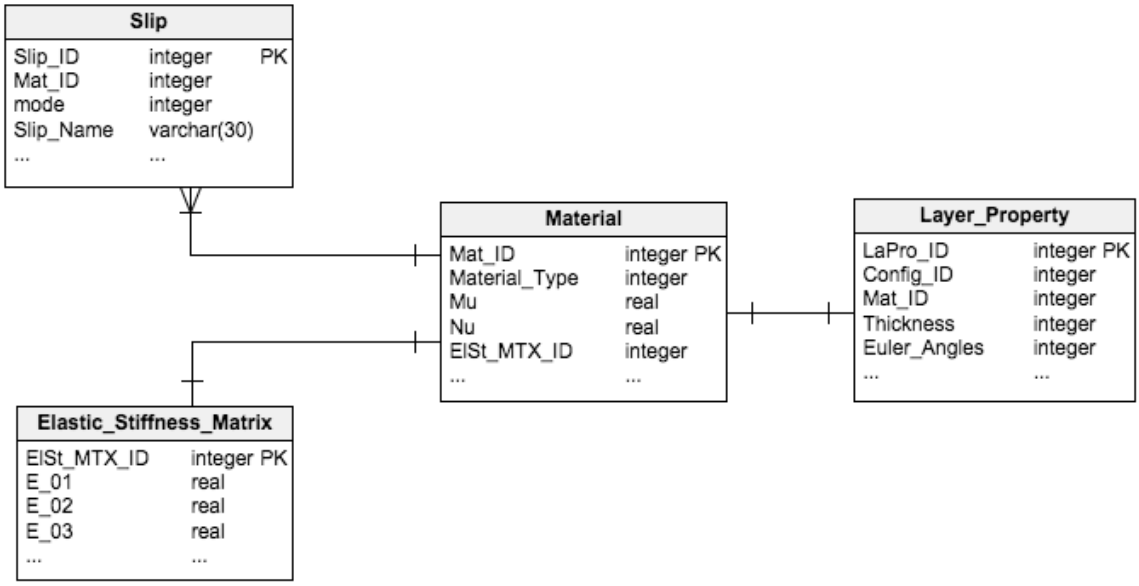


Figure 5 Entity-Relationship diagram for table *Material*

2.1.4 DDD server & Apache Server

The role of the Apache server is to provide the authentication and access control. For the hosted version of the DDD portal, it can protect the cluster from potential security issues. A Python HTTP server, called here the DDD server, is implemented to tackle the HTTP requests and responses, thus the communication is conducted in an indirect fashion because the Apache server relays the data in the middle.

Considering the example below in Figure 6 of extraction of data, users first press the button “Load” and the client portal will immediately send an HTTP request with name-value array. The Apache server relays the request as transformed data in XML format and the DDD server replies correspondingly by connecting to the database and packaging data in JSON format to the portal. The other functionalities are all realized in a similar way with the same communication path.

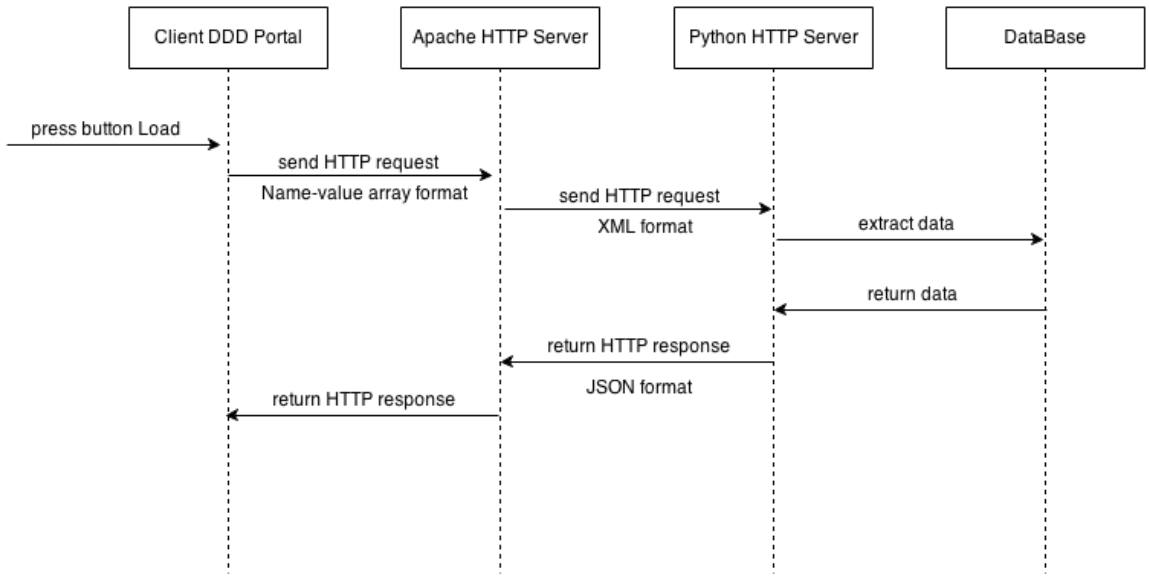


Figure 6 Communication sequence diagram for extraction of data

2.2 UI Design

2.2.1 Principle of design

The importance of good user interface design can be the difference between acceptance and rejection of a product in the market. If users feel it is not easy to learn nor intuitive to use, otherwise an excellent product could fail. According to Larry Constantine and Lucy Lockwood in their usage-centered design [15], the main principles of design are:

- *The structure principle*: Design should organize the user interface purposefully, in meaningful and useful ways based on clear, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.
- *The simplicity principle*: The design should make simple, common tasks easy, communicating clearly and simply in the user's own language, and providing good shortcuts that are meaningfully related to longer procedures.
- *The visibility principle*: The design should make all needed options and materials for a given task visible without distracting the user with extraneous or redundant information. Good designs don't overwhelm users with alternatives or confuse with unneeded information.
- *The feedback principle*: The design should keep users informed of actions or interpretations, changes of state or condition, and errors or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.
- *The tolerance principle*: The design should be flexible and tolerant, reducing the cost of mistakes and misuse by allowing undoing and redoing, while also

preventing errors wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.

- *The reuse principle*: The design should reuse internal and external components and behaviors, maintaining consistency with purpose rather than merely arbitrary consistency, thus reducing the need for users to rethink and remember.

2.2.2 DDD Portal – Graphic Interface

Based on the above guidelines, the DDD portal is seeking simplicity, conciseness and clearness. The site map in Figure 7 indicates that different pages (or templates) have been created for the purposes of generating input files and visualizing the simulation results. In this section, the layout of each page will be presented in detail with an explanation why it is adopted particularly.

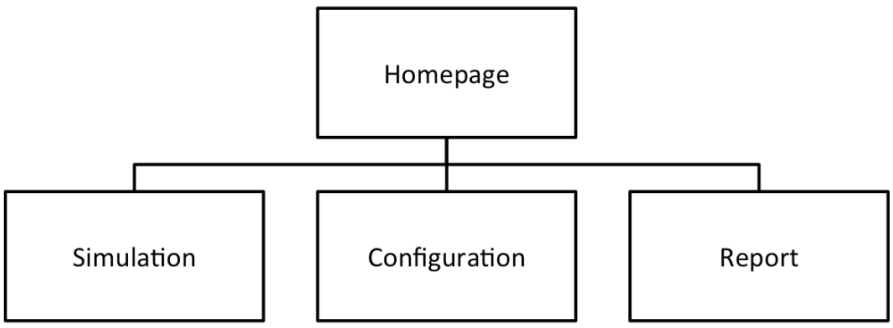


Figure 7 Site map of DDD portal

- Homepage: It is a page composed of one navigation bar and five primary buttons in the middle. The user can set up his/her own account, or modify the default existing materials and simulation controls. Clicking either navigation bar or button can access the page simulation. A button for resetting the database is also available if the user wants to remove the simulations created before.

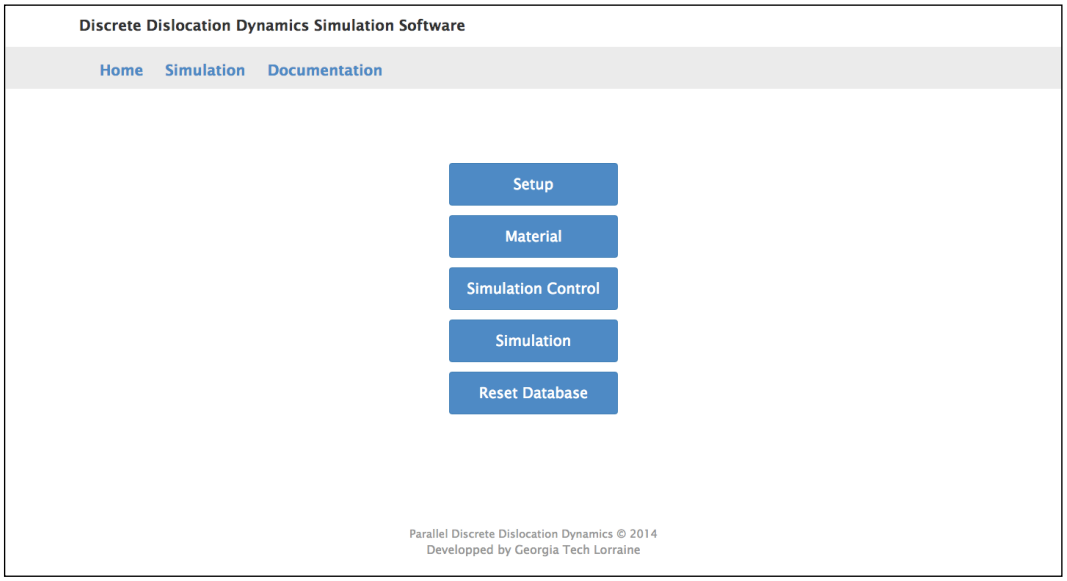


Figure 8 The screenshot of homepage

- Simulation page: The main components here are two tables of simulations and reports, and the user can switch the view between them by clicking the tab. With the plugin *DataTable*, the user can sort data by different columns and navigate across different items with the provided pagination. For both of the tables, there are different sets of buttons at the top of the page for the basic operations.

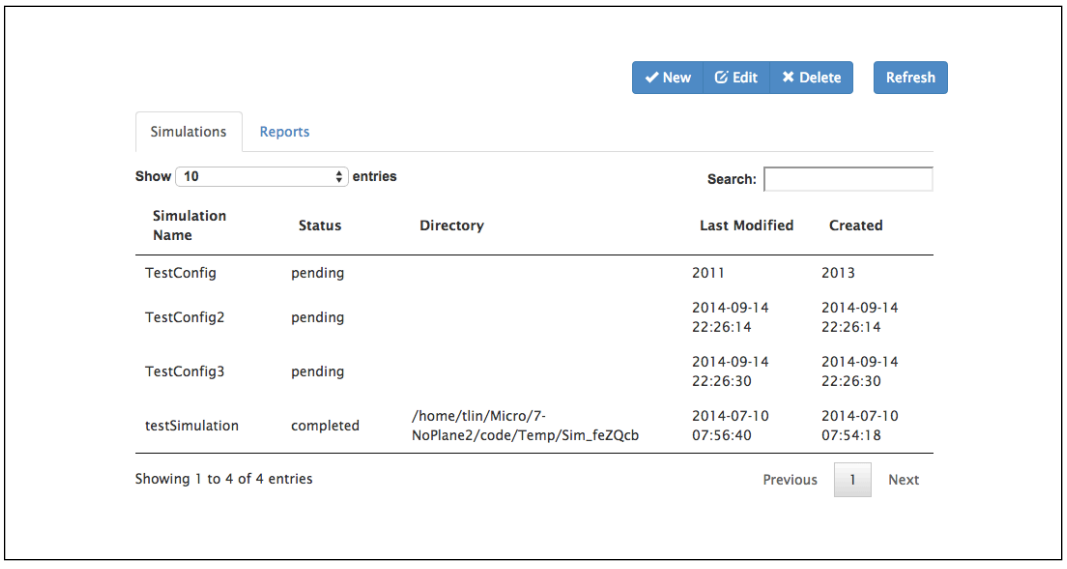


Figure 9 Table of simulations in simulation page

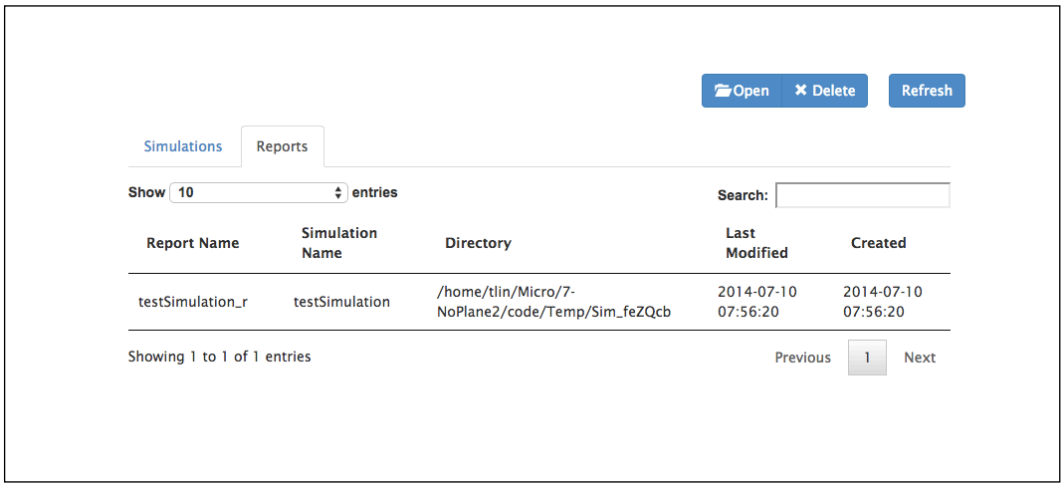


Figure 10 Table of reports in simulation page

- Configuration page: The role of this page is to provide a form where the user configures the simulation. The input fields on this form (or wizard) are divided into several input areas such as Volume & Layer Property, Dislocation, Simulation Control, etc. There are two types of buttons, general buttons and local ones. The general buttons in orange are always available when navigating through different steps, but the local ones in blue only show up in particular steps. They support a variety of functionalities that will be covered in the section 2.3.3. One configuration can possess as many layers as the user requires, so we use another set of buttons (e.g. Layer 1) to toggle the content of each layer. In this way, the view is more organized for the modification of field values in a particular layer.

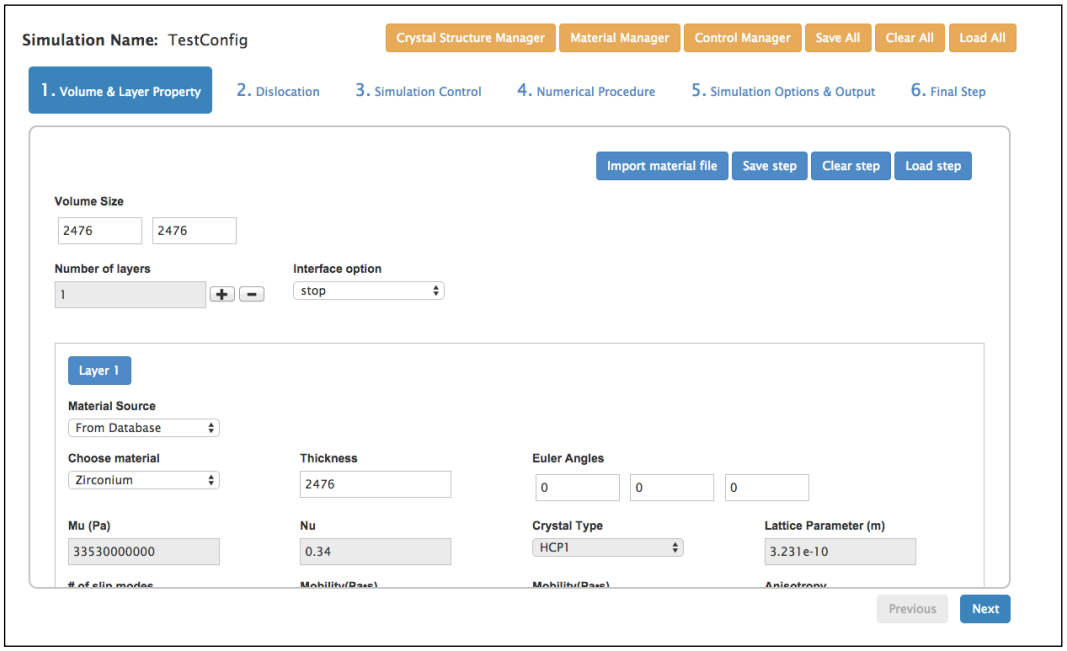


Figure 11 The wizard of configuration - Step 1 Volume & Layer Property

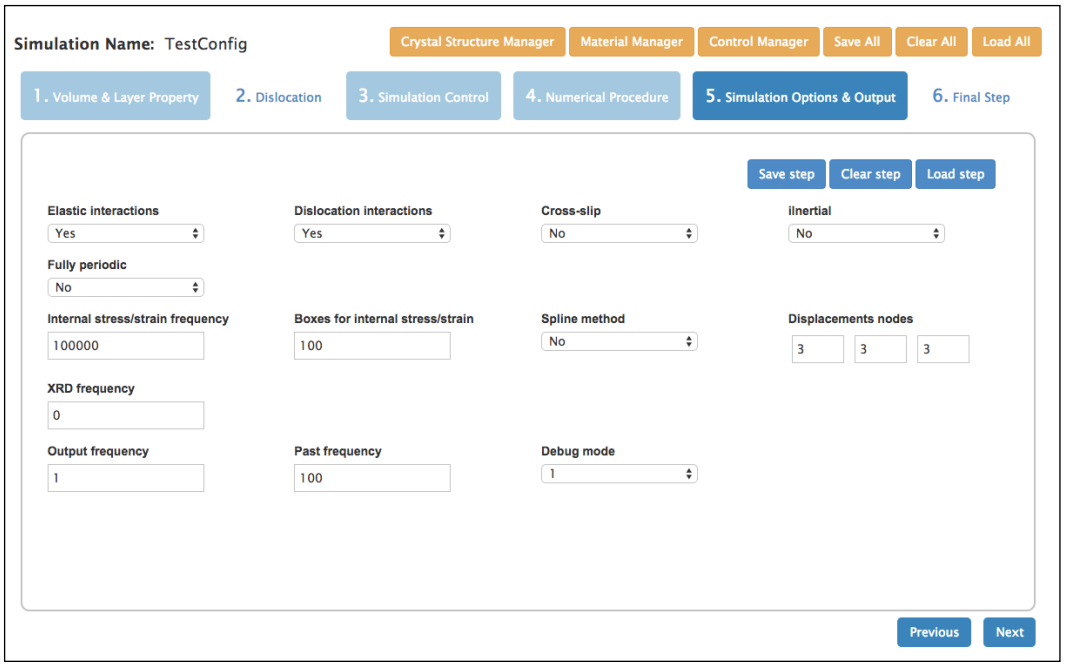


Figure 12 The wizard of configuration – Step 5 Simulation Options & Output

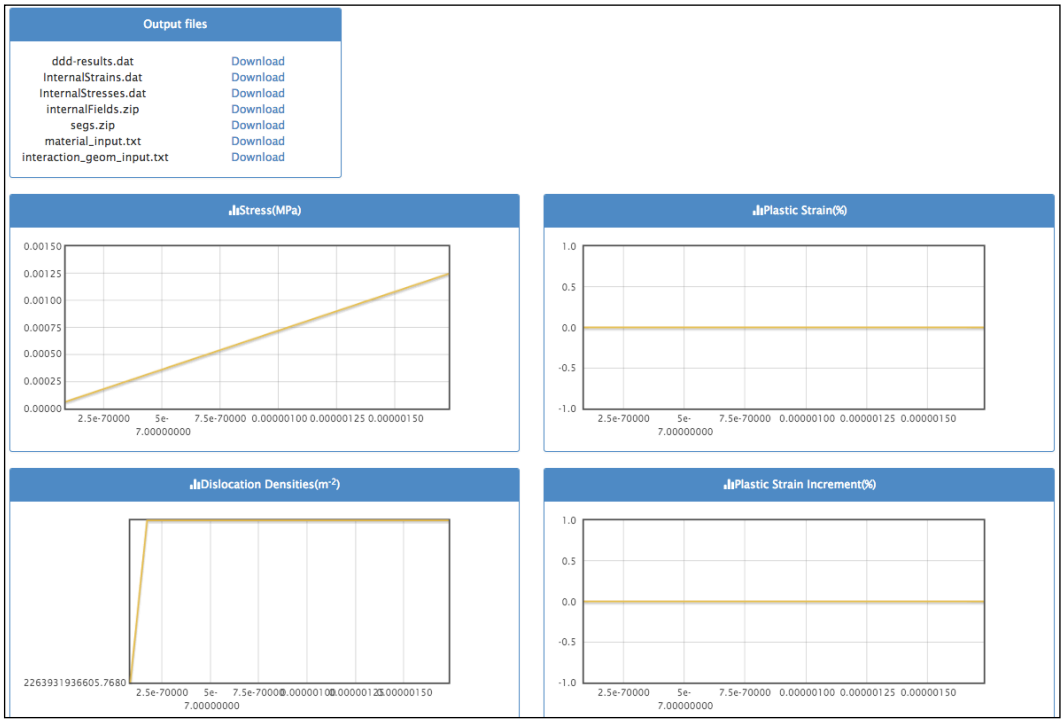


Figure 13 Report page with download links and charts

- Reports page: A report is created and shows up only after the completion of simulation. The download links and the charts regarding the simulation results are the main information in this page. The dashboard gives the user a quick overview about how the simulation went.

2.2.3 DDD Portal – Functions

Behind the visual design, a variety of functions in the DDD portal have been developed to support different user actions. They are all listed as follows in order to explain when and how they will be used.

- *Create/Delete a simulation*: The user can create a new simulation through the popup and delete the existing ones from the table.
- *Save/Edit configuration*: Once a new simulation is created, the user can edit the configuration anytime by modifying input values on the form and saving them.

The “global save” is for the whole configuration including all steps in the wizard, and “local save” is only for a specific step.

- *Load configuration*: If the input values are similar to the ones of another simulation created previously, the user can choose to load the whole configuration or one specific step without entering the same information again. All the previously created configurations are always available, so giving similar input files which have only little differences will be much more efficient.

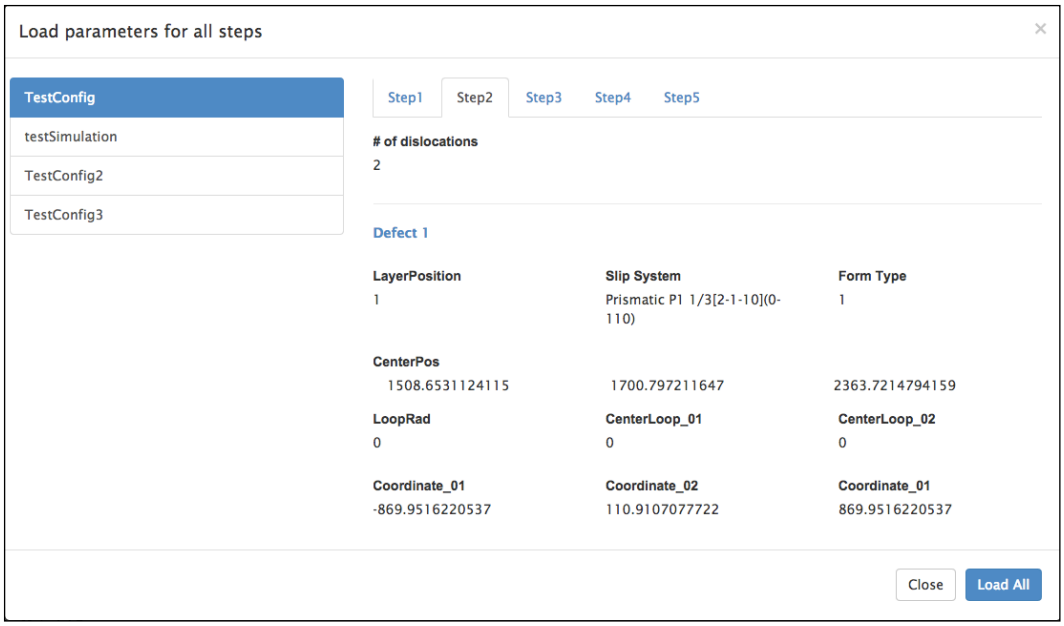


Figure 14 The popup for loading the data from existing simulations

- *Import material/dislocation input files*: The user who are already familiar with input text files of DDD software can also import those existing files into DDD portal to configure a simulations.
- *Visualize 3D microstructure*: The user can visualize the distribution of dislocations inside 3D microstructure defined in configuration. And the dislocations are represented in different colors based on their individual slip system.

- *Generate random dislocation*: The user can choose to generate a set of random dislocations which meet the certain conditions defined in advance.
- *Display informative definition of fields*: When the user moves the cursor to a particular input field, the information associated with such field will pop up.

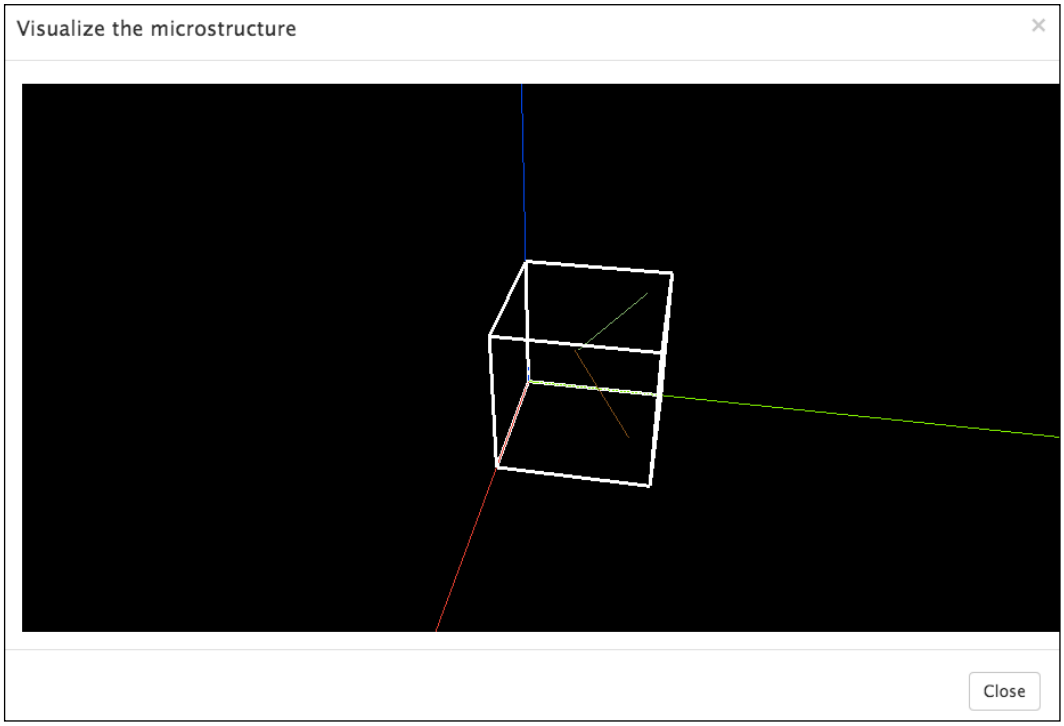


Figure 15 Visualization of dislocations in 3D microstructure

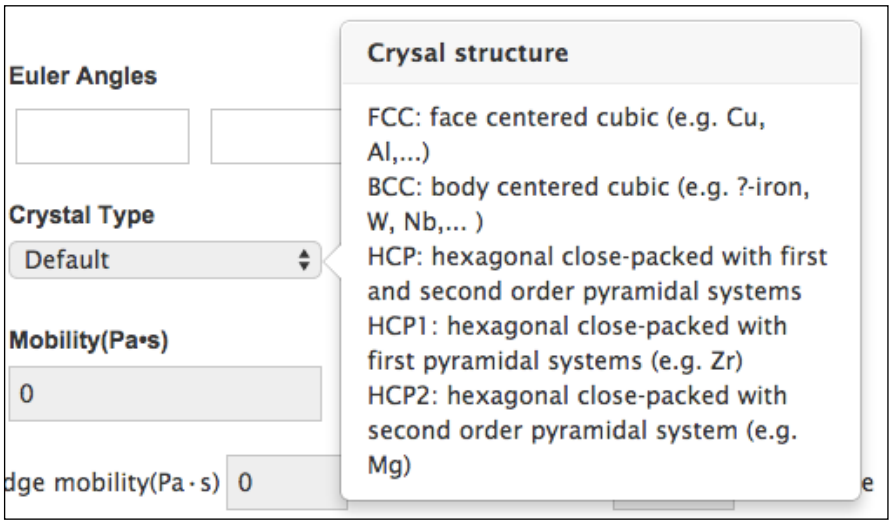


Figure 16 Display the information for particular field

- *Crystal managers:* This independent manager helps the user configure existing standard crystals or create new ones. Once the new crystal is created, the user can add a new slip system on top of it.
- *Material mangers:* The standard materials are provided in Material managers, so the user can select one of them directly to configure the layer. In addition, the user is able to modify them or create a new standard material when needed.
- *Simulation Control managers*: The standard simulation controls are provided in Simulation Control managers, so the user can select one of them directly in the configuration page (step 3). In addition, the user is able to modify them or create new standard controls when needed.
- *Run simulation*: After configuring the input fields, the user can launch the simulation directly from the DDD portal by selecting the number of processors to use. If it is the laptop version, the reasonable number is between 2 and 8. If it is a hosted version, the value can be up to 64 or even more.
- *Download simulation results*: Once the simulation is completed, the user can access the report and download the simulation results including the text files and animation files.

Figure 17 Material manager - modify/delete existing material

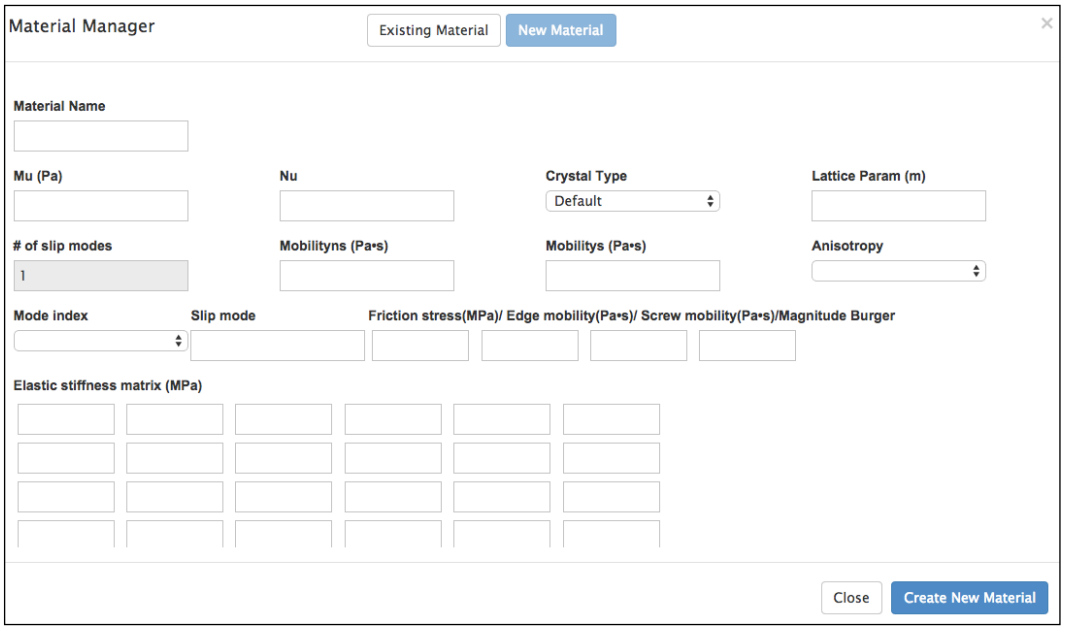


Figure 18 Material manager - create new material

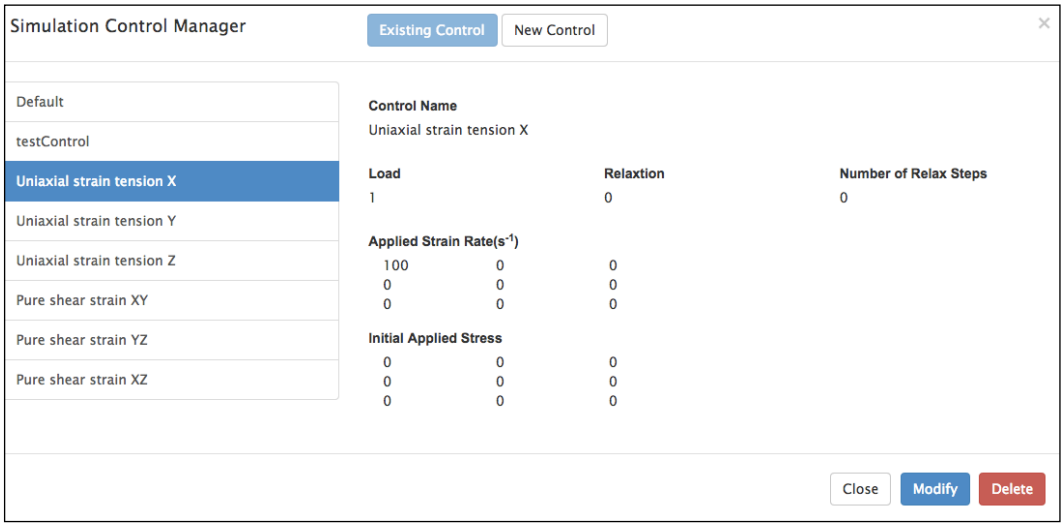


Figure 19 Simulation control manager - modify/delete existing control

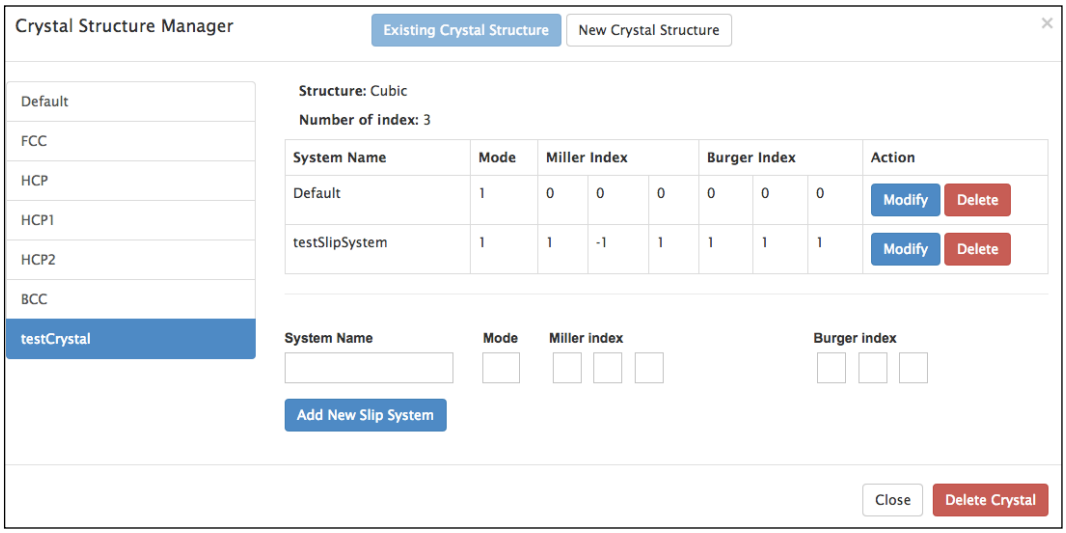


Figure 20 Crystal manager – add a new slip system or delete existing crystal

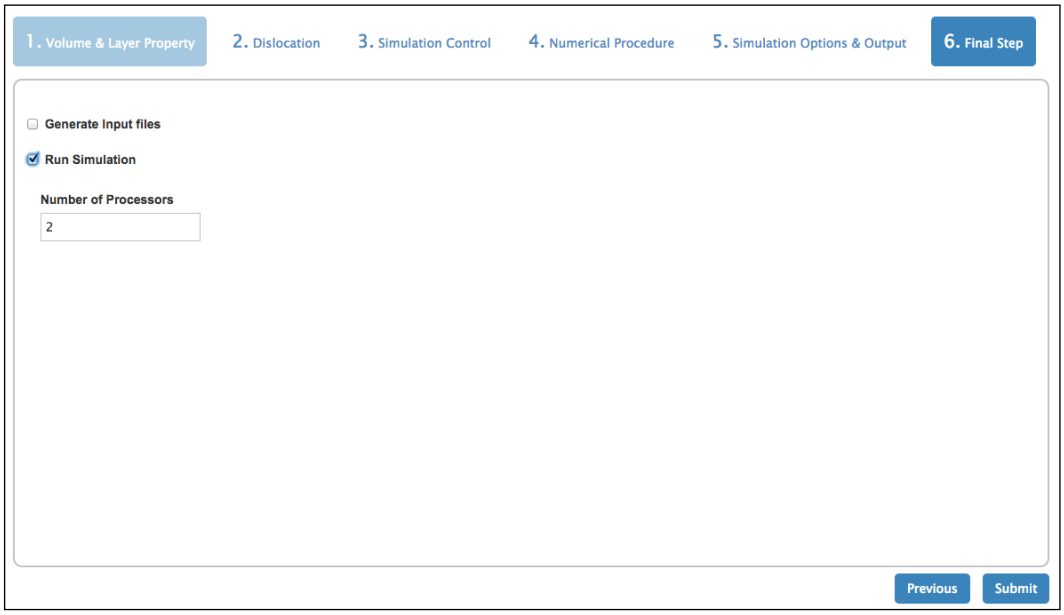


Figure 21 Run a simulation by assigning the number of processors

2.3 Deployment

The DDD portal is a package that comprises of a web application, database, apache server, DDD server and simulation software. The deployment across diverse platforms (or operating systems) is no easy task since the approaches of installation are different. In the short-term, our approach uses a virtual appliance as a carrier where all modules are pre-installed in the distribution as in Figure 22. From the user's side, a virtual machine such as virtual box is required. Once the appliance is powered on, the server will be launched and start to listen for requests. Then the user can access the DDD portal with a pre-assigned IP address.

The virtual appliance solution is generally intended for personal use so that every college student can install it and test different configurations with a small set of dislocations. For the professional projects requiring the use of the cluster, another DDD portal is deployed as a hosted solution, so only authorized users are able to access it and run the simulation with the computational resources of the cluster.

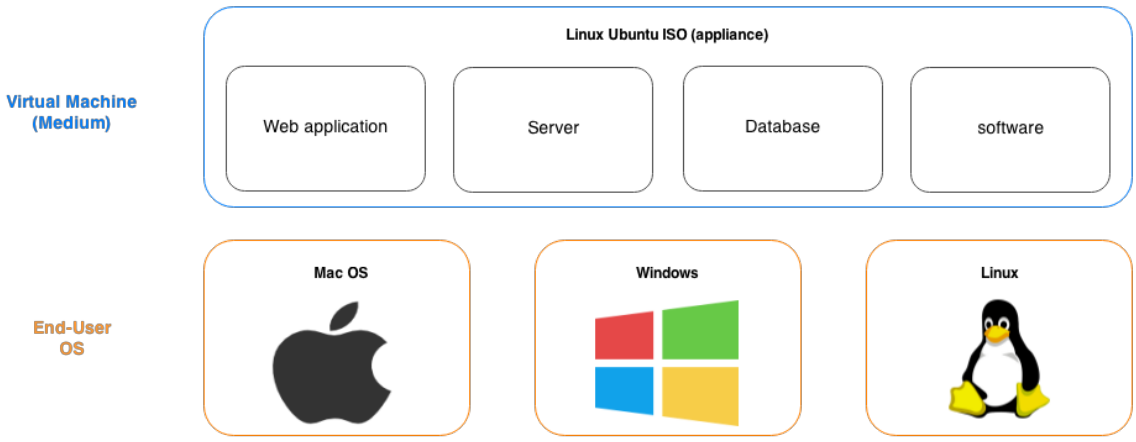


Figure 22 Virtual appliance solution for DDD portal package

2.4 User Experience Evaluation

ISO 9241-210 defines user experience (UX) as "a person's perceptions and responses that result from the use or anticipated use of a product, system or service". Hence, the evaluation needs to consider all the users' emotions, beliefs, preferences, physical and psychological responses that occur before, during and after using the product. As in Figure 23, a user experience is also affected by external factors [16] such as social factors, cultural factors and context of use. We produce an overall score or degree of satisfaction for the product through certain quantitative methods.

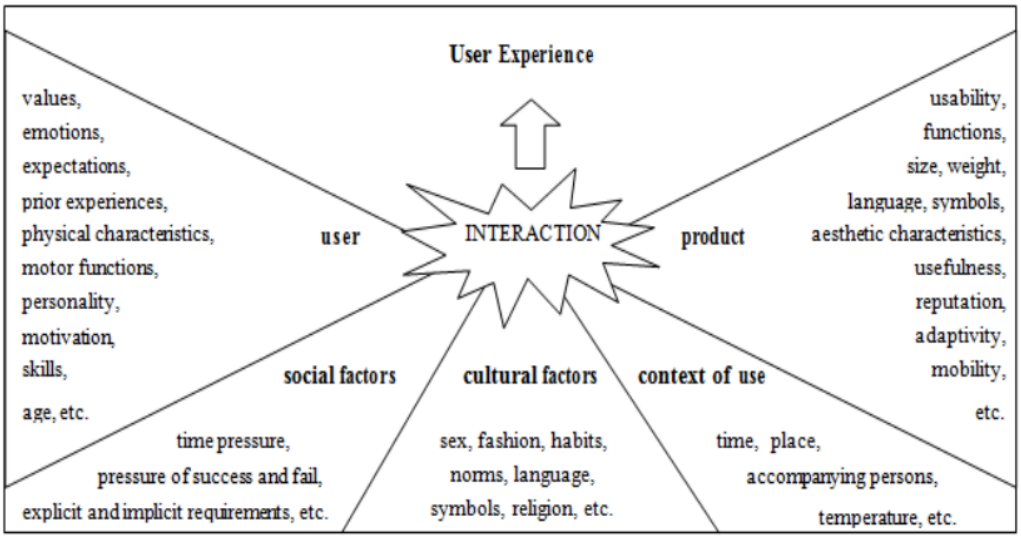


Figure 23 User experience forms in interaction with user and product in the particular context including social and cultural factors [16]

The QSA-GQM questionnaire is a technique based on the Goal-Question-Metric paradigm, used in Software Engineering to assess software quality. The Repertory Grid Technique (RGT) elicits and evaluates people's subjective experiences of interacting with technology through the individual way they construe the meanings of those experiences under investigations. The semi-structured experience interview is to make arrangements for a meeting in which the interviewer asks questions, listens and records the answers. In general, a diversity of evaluation methods exists in research and in industry, but the specific purpose for each must be determined.

To date, user experience studies have mostly focused on short-term assessment of the initial adoption of new products. The UX curve method [17], a long-term evaluation, has also been introduced because the relationship between products and users evolves over a long period. According to [18], the actual experience of usage doesn't cover all relevant UX concerns. Instead, people can have indirect experiences before their first encounter through expectations formed by existing experiences of related technology,

presentation and demonstrations and extend these expectations similarly after usage. Figure 24 explains this relationship.

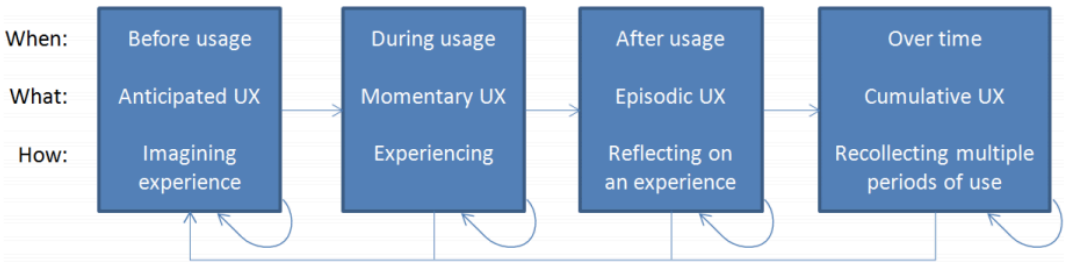


Figure 24 Time spans of user experience, the terms to describe the kind of user experience related to the spans, and the internal process taking place in the different time spans [18]

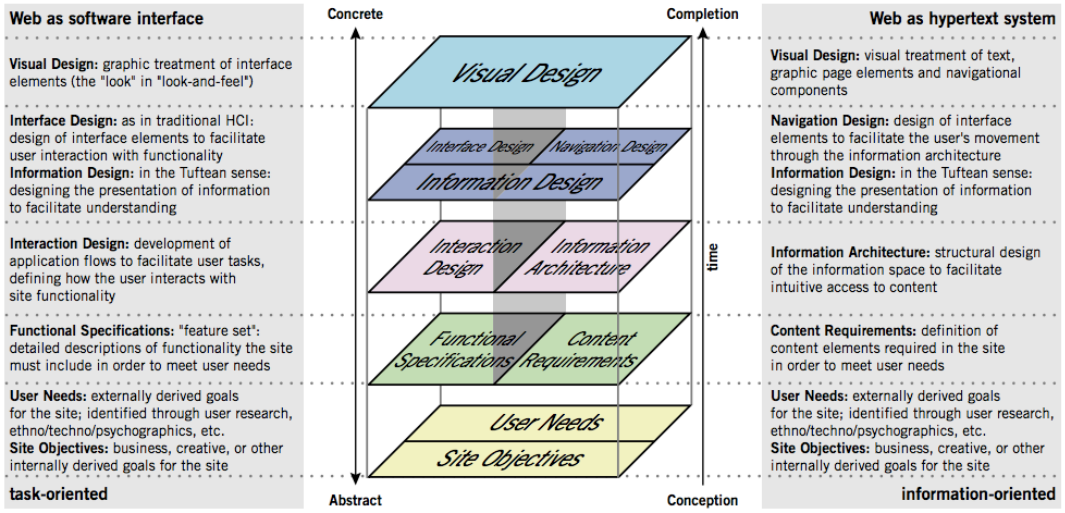


Figure 25 The elements of user interface [19]

Since our product is a web-based application, user experience can be studied in five consecutive levels [19] which include visual design, information design, interaction design, functional specifications, and site objectives as in Figure 25. As a concrete objective, the focus is on how users feel about the look of interface and how the presentation of information facilitates the understanding. More abstract topics include how users interact with the functions of the application, whether the functionalities meet

the original specification, and whether the objective of application is fulfilled according to user needs identified through relevant research.

The first version of the DDD portal has been developed and will be deployed for academic use in the near future. A structured UX evaluation plays a significant role in the improvement of functions and the evolution of the whole application, because it can help the developer team discover what users like and dislike and what they expect from the product. In the short-term, our team will collect the feedback and the remarks from college students or researchers by adopting the questionnaire approach. Based on the prior discussion, the questionnaire in Appendix A covers a range of subjects from accessibility, usability, quality, and user expectation. The long-term goal is to keep track of user responses to the new features of the application, and eventually to reflect the potential future needs and evolve the product.

CHAPTER 3

SCIENTIFIC COMPUTING

Scientific computing (or computational science) is an interdisciplinary field in which mathematical models and quantitative analysis techniques are applied to solve real-world scientific problems. It often requires the availability of a massive number of computers for performing large-scale experiments. Researchers use high-performance computing solutions and installed facilities such as clusters and super computers to analyze the complexity of problems and attempt to resolve them.

3.1 Purpose

The DDD algorithm aims to solve the dislocation dynamics and provide the material stress metrics for the study of plasticity. The main computation involves an iterative process of forces interaction between dislocations within the crystal structure. Although the current version of the algorithm is already parallelized, the resulting speedup of the program doesn't increase proportionally with the computational resources and even stagnates when using too many processors. This limit makes the software unable to tackle complex simulations in which there are a massive number of dislocations configured.

This problem leads us to the discussion on high performance computing. The poor scalability may result from unstructured parallelization, communication between processors, inefficient data structures, the algorithms used in mechanical computation, etc. In the second part of this thesis, we analyze certain parts of the DDD algorithm that take major portions of computation cost (or execution time), and identify the possible causes for poor performance. A variety of solutions are proposed regarding the nature of the problem and the best fit into the original structure of the program. Last but not least,

some ideas worth trying but not yet tested will be presented as well for future extensions of the product.

3.2 Introduction of DDD algorithm

3.2.1 Program Flow

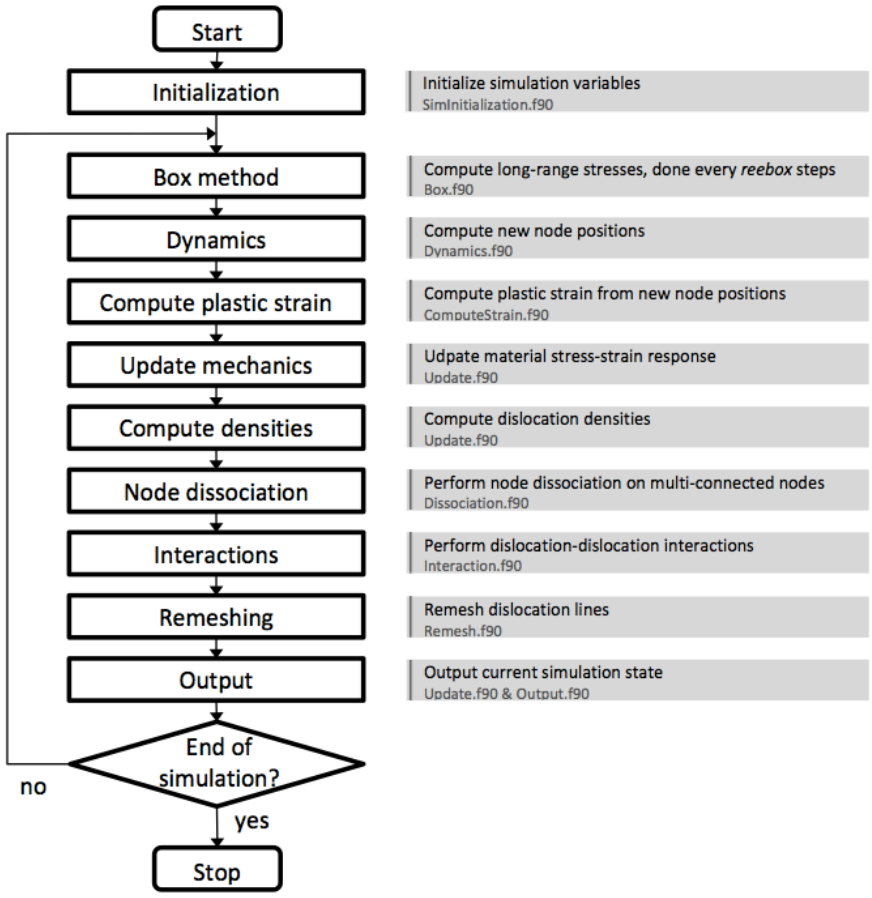


Figure 26 The flow chart of DDD algorithm

The major operations performed by DDD are shown in Figure 26. The program starts with initializing the MPI environment and reading the input files, and then distributes the data to all the assigned processors. The Box method [28] computes far-field elastic stresses for each box that contains an ensemble of dislocations. The interactive forces between dislocations are calculated during the dynamic process. As

soon as the overall force applied on each dislocation node is calculated, its new position will be determined correspondingly.

Based on those new changes, the program computes plastic strain and updates material stress-strain responses and dislocation densities. The next step is to perform node dissociations, interactions between dislocations and the remeshing process if certain mechanical conditions are met. To finalize the simulation, the program writes output files and provides the desired information concerning the current simulation state.

3.2.2 Variables

Dislocation node and dislocation segment are two major variable types used in the program to represent the graph relation of dislocations. The dislocation node variable stores information about the forces, the connections, the position, and the slip system. And the dislocation segment variable shows the coordinate, and the number of neighbor segments and two nodes of the segment. Table 2 and Table 3 present the detail of these two variable types, and Figure 27 shows an example of dislocation configuration with the associated data.

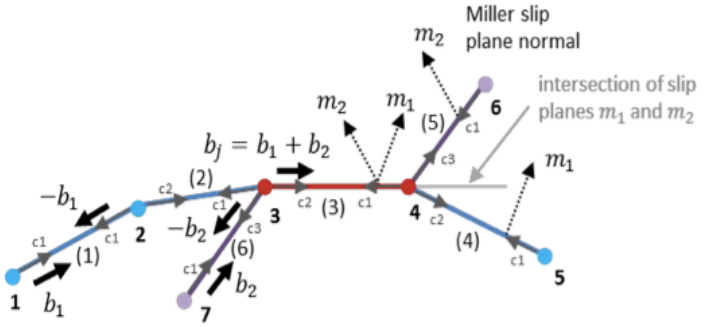
Table 2 Variable type - dislocation node

PROPERTY	**TYPE**	**DESCRIPTION**
iID	integer	Node identifier
iLoopID	integer	Loop in which the node belongs
iGroupID	integer	Group in which the node belongs
iLayerID	integer	Layer in which the node lies
iType	integer	Node type: iDPTypeFree: free moving node, iDPTypeFixed: fix node (generally end node), iDPTypeInter: interaction (junction) node
exited	integer	Set to 1 if the node has exited the volume for no PBCs simulation (finiteBox=1), 0 otherwise
tvPG	type(vector)	Global coordinates of the node (x,y,z)
tvTG	type(vector)	Tangent direction at the node, used for spline method (iSplineFe=1)

tvPreV	type(vector)	Global node velocity
tvPreVT	type(vector)	Global node tangential velocity (not used anymore)
tvAcce	type(vector)	Node acceleration, used for inertial computation (not implemented)
force	type(vector)	Nodal force
dpMass	double	Nodal mass, used for inertial computation (not implemented)
iNumConn	integer	Number of node connections. Upper limit set by MAX_CONN.
iConnP	MAX_CONN*integer	IDs of connected node.
iConnBurgers	MAX_CONN*integer	IDs of the Burgers vector for each connection
iMiller	integer	Reference plane of the node, corresponds to the first plane of the iListMiller plane list.
iNumMiller	integer	Number of planes the node belongs to (max=3)
iListMiller	3*integer	List of the planes the node belongs to.
lStat	boolean	Node flag, used to check if node has been already visited.
lMovingNode	boolean	Used in dynamics to know if the node should be moving.

Table 3 Variable type - dislocation segment

PROPERTY	**TYPE**	**DESCRIPTION**
iID	integer	Segment identifier
iLoopID	integer	Loop in which the segment belongs
iGroupID	integer	Group in which the segment belongs
iLayerID	integer	Layer in which the segment lies
iBox	integer	Box in which the segment lies
Node1	integer	ID of the first node of the segment
Node2	integer	ID of the second node of the segment
tvPG	type(vector)	Global coordinates of the middle of the segment (x,y,z)
iNumNei	integer	Number of segment neighbors
iNeiID	*integer	IDs of segment neighbors
lStat	boolean	Segment flag, used to check if segment has been already visited



Data associated with Node 3:

```
DPoints(3)%iID=3
DPoints(3)%iLoopID=-1
DPoints(3)%iGroupID=1
DPoints(3)%iType=iDPTypeInter
DPoints(3)%iNumConn=3
DPoints(3)%iConnP(1)=2
DPoints(3)%iConnP(2)=4
DPoints(3)%iConnP(3)=7
DPoints(3)%iConnBurgers(1)=2
DPoints(3)%iConnBurgers(2)=5
DPoints(3)%iConnBurgers(3)=4
DPoints(3)%iConnNumMiller(1)=1
DPoints(3)%iConnMiller(1,1)=1
DPoints(3)%iConnNumMiller(2)=2
DPoints(3)%iConnMiller(2,1)=1
DPoints(3)%iConnMiller(2,2)=2
DPoints(3)%iConnNumMiller(3)=1
DPoints(3)%iConnMiller(3,1)=2
```

Figure 27 Example of dislocation configuration and the associated data

3.2.3 Existing MPI framework

In this section, the existing communication flow of DDD is presented. The master processor distributes global variables and boxes in the initialization process and then executes the main body of the program where there are two main computation,

- *DynamicsSolver*: A parallel function composed of 4 main steps: 1) distribute the dynamic dislocation segments 2) compute the nodal force 3) derive the overall force through *MPI_AllReduce* 4) resolve the new position of dislocation nodes.
- *Update_Output_Statistics*: With parallelized sub-functions, it computes the important mechanical metrics for the changes, and performs short-range interaction calculations such as node dissociation and the remeshing process.

The parallelization is already implemented in these two high-level functions because they account for the major fraction of execution time at each iteration. Many MPI communications are also contained in the program as shown in Figure 28 because of the need for data transfer and work distribution. DDD is a master-slave program which heavily relies on the master process to distribute tasks and update changes. In general, the program uses *MPI_Bcast, MPI_Reduce* and *MPI_AllReduce* to broadcast and collect the

data respectively. For point-to-point data transfer, *MPI_Isend* (non-blocking communication) is also used with the advantage that the function call can return immediately without waiting for an acknowledgement from the receiver. To ensure the reception of data and the completion of calculation at each step, *MPI_Barriers* are inserted in many places for the purpose of synchronization.

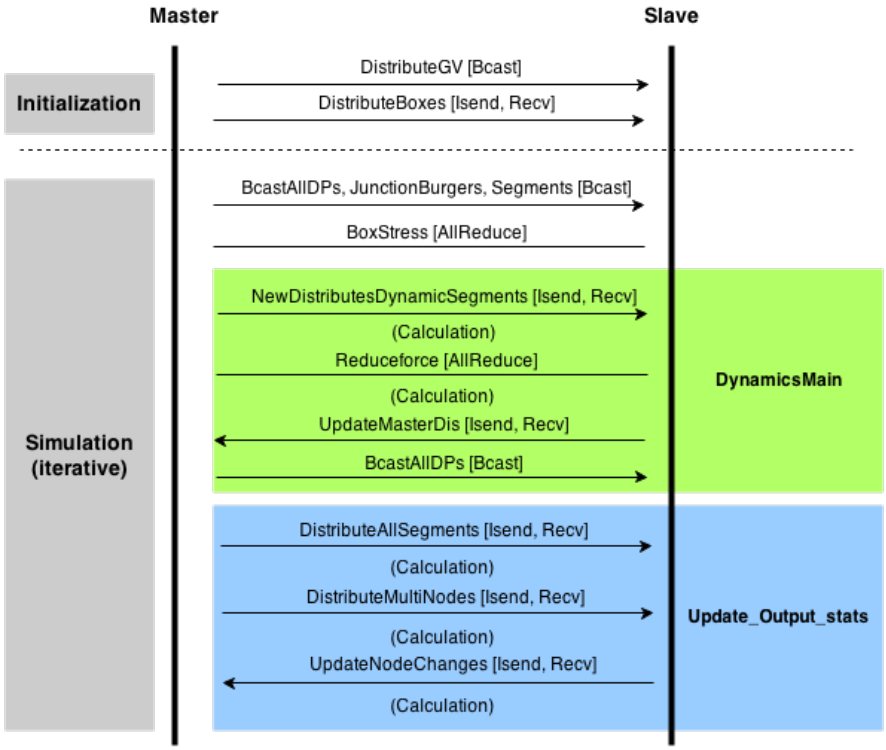


Figure 28 The communication flow of parallel DDD program

3.3 Performance Metrics

Execution time, speedup are two metrics commonly used to measure the performance of MPI programs [25]. These metrics are affected by several factors such as the sequential part's fraction of the program, the complexity of the problem, the number of processors, and inter-processor communications. For the subsequent sections, we generally use speedup as the main indicator to reveal the performance of the DDD parallel program.

Execution time

Execution time is defined as the time elapsed from the start of the first processor in the execution of the program to the completion of the last processor. The execution time *T* is given by:

$$T = T_{comp} + T_{comm} + T_{idle}$$

where T_{comp} is the computation time, T_{comm} is the communication time consumed by processors to send and/or receive messages, and T_{idle} is the time a process spends waiting for the other processors.

Speed up

Speedup is another indicator that takes processors count *p*, and problem size *n*, into account. The total parallel execution time of a program is given by:

$$T_{parallel} = \sigma(n) + \frac{\varphi(n)}{p} + \kappa(n,p)$$

where $\sigma(n)$ is the execution time of the serial part of program, $\varphi(n)$ is the execution time of the parallel part of program, and $\kappa(n,p)$is the communication time. Generally, speed up is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with multiple processors. The speedup metric for solving an *n*-size problem using *p* processors is expressed by:

$$\psi(n,p) \leq \frac{T_{serial}}{T_{parallel}}$$

Amdahl's Law [26] is used to predict the maximum achievable speedup for a given program. The law assumes that a fraction *f* of a program's execution time was infinitely parallelizable with no overhead, while the remaining fraction, 1-*f*, was totally serial. According to this law, the speedup of n-size problem on p processors is governed by:

$$\psi(n,p) = \frac{1}{f + (1-f)/p}\ , 0 \leq f \leq 1$$

Amdahl's law considers problem size as a constant and hence the execution time decreases as the number of processors increases. Gustafson's law [27] gives another formula for predicting maximum achievable speedup which is described by:

$$\psi(n,p) = p + (1-p)s$$

where *s* is the fraction of total execution time spent in serial code. Both of these two laws ignore the communication cost, so the maximum speedup will be overestimated.

3.4 Experimental Setup

We ran the benchmarks of the DDD program on the research cluster Cameron of SUPELEC, Metz Campus with 16 nodes (or machines) that are interconnected across a 10-Gbit/s Ethernet switch (an OmniSwitch Alcatel 6900-X20-F) with up to twenty 10Gbit/s ports. Each node has an Intel Xeon E5-1650 processor clocked at 3.2GHz composed of 6 physical hyperthreaded CPU cores (12 logic cores), and equipped with 8 GBytes of global DDD3 RAM on a 1600MHz memory bus. This cluster utilizes the Linux 64 bits, fedora core 16 operating system.

There are several important terms which will be used in subsequent sections to explain the performance of scalability. A node is a machine which can contain multiple processors (or CPU cores). The total number of processes possible is the product of the number of nodes and the number of processes available per node. For example, 16 processes may be created using two nodes with 8 processes per node, or 16 nodes with 1 process per node.

3.5 Problems in DDD algorithm

In advance of changing the parallelization and refactoring the code, it is important to identify existing problems in the DDD algorithm. These problems can provide the hints as to improve and how to carry out the modifications. For this work, we used Vampir [20], a popular profiling tools to analyze the MPI communications between

processors with the aid of statistical charts such as function summary, message summary, communication matrix, call tree, etc. A screenshot of Vampir interface is shown in Figure 30. Moreover, we manually measured the execution time and the volume of message transfer to discover more possible causes and support the findings from Vampir.

3.5.1 Analysis by Vampir

Vampir requires a working monitoring system with built-in support for the performance data file format. We use Score-P [21] as the code instrumentation and run-time measurement framework because it supports the generation of trace log files with the Open Trace Format Version 2 (OTF2). Figure 39 illustrates the overview of Score-P measurement system architecture.

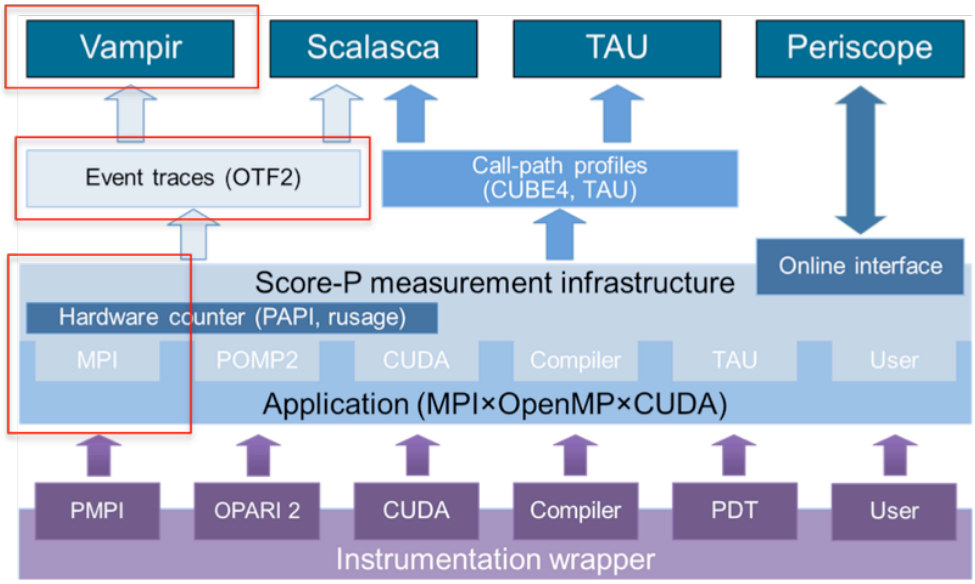


Figure 29 Overview of the Score-P measurement system architecture and the tools interface [21]

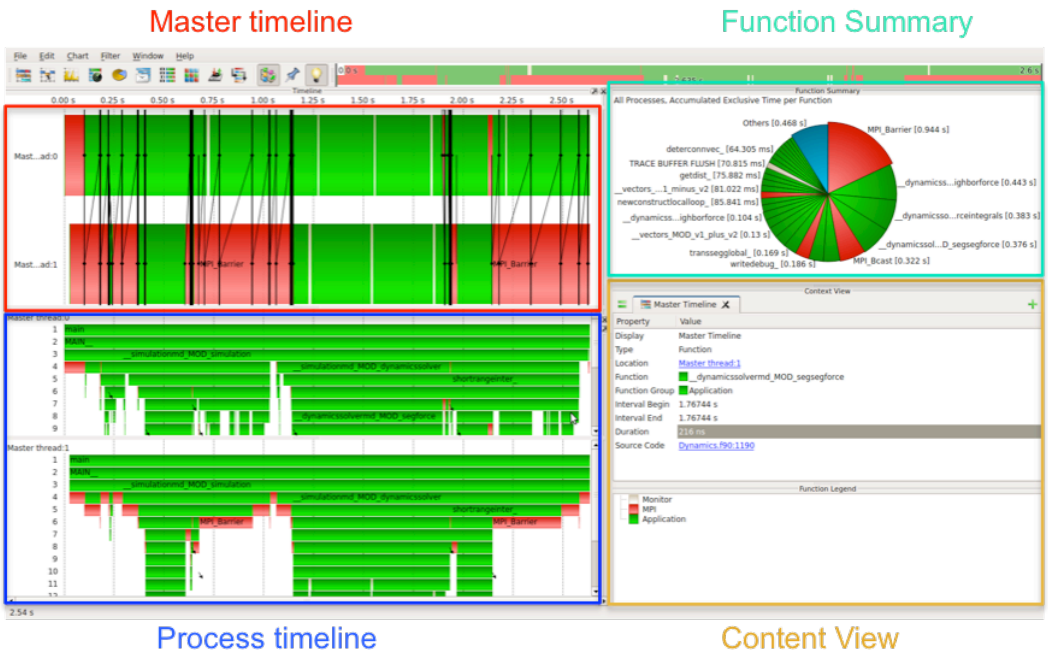


Figure 30 The interface of Vampir with a number of statistic charts.

Even though the timeline of events, the message transfer between processors and the execution time of subroutine can be easily traced with the aid of the visualization tools, there are still some limitations in the trial version: 1) Inability to display all the events once the number of iterations of the program exceeds a certain limit, 2) Inaccuracy in the execution time of functions due to extra code instrumentation. Those constraints only allow the parallel program to be executed with 2 to 16 processes. Moreover, the number of iterations in the simulation is set to 2 in order to avoid an excess of event information that Vampir cannot handle. In spite of all the inconveniences, it can still help us to identify the following anomalies that may explain why the performance doesn't scale up well in the presence of distributed-memory parallelization.

Non-negligible MPI barriers

In the DDD parallel program, *MPI Barriers* account for a non-negligible fraction of the total execution time. From Figures 31 and 32, we noticed that with 2 processes, the fraction is less than 25%. However, when the number of processes increases, it becomes

a more and more important component of delay and even reaches more than 50% with 8 processes. This tendency seems reasonable because the execution time of a parallel part of algorithm is reduced with more processes while the time consumed by *MPI_Barrier* is roughly the same. The purpose of *MPI_Barrier* is to synchronize processes to prevent some processors from running faster than the others, but this call are overused in our program as the check points to ensure that every processor reach the same step of calculation.

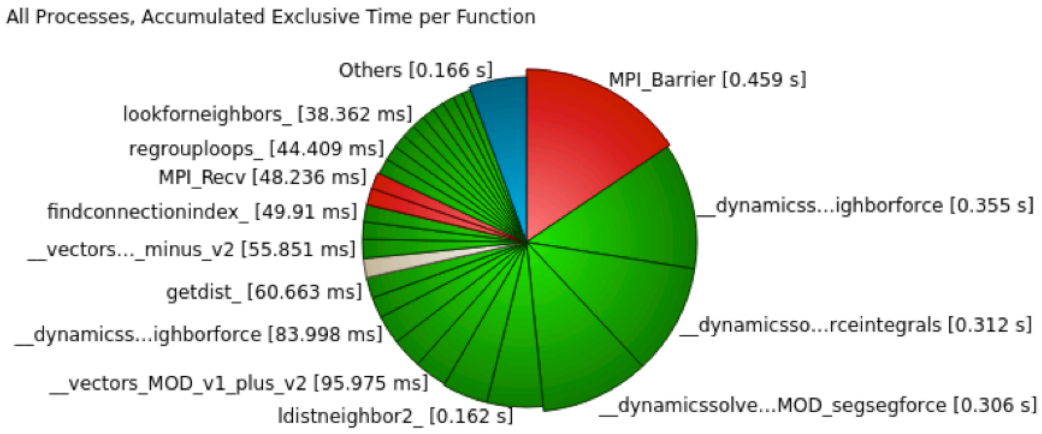


Figure 31 Execution time per functions with 2 processes, 1 process per node (~1000 dislocations)

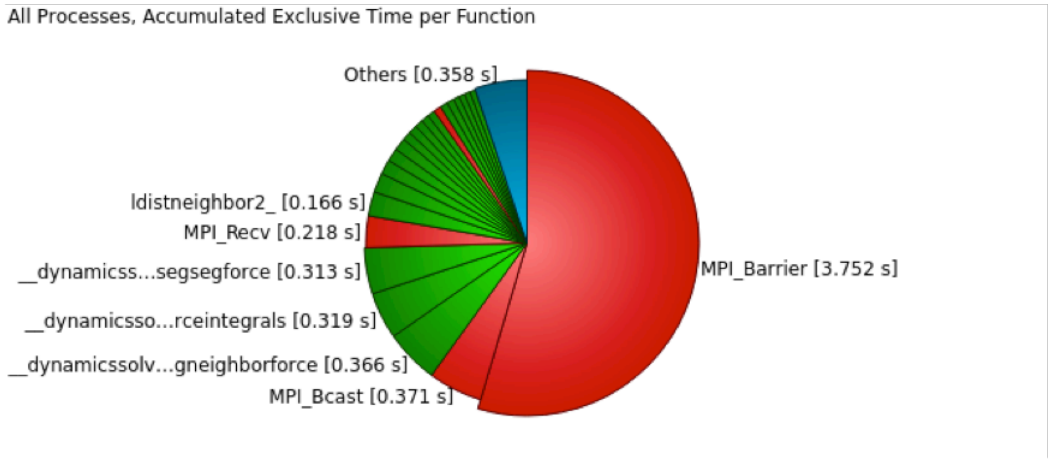


Figure 32 Execution time per function with 8 processes, 1 process per node (~1000 dislocations)

Non-parallel functions:

The master timeline in Figure 33 indicates that the function *shortrangeinte*r is not yet parallelized, so the master processor is the only worker that carries out the computation. There may be more non-parallelized parts in the program which will hinder the performance of speedup by introducing extra *MPI Barrier* calls in our program.

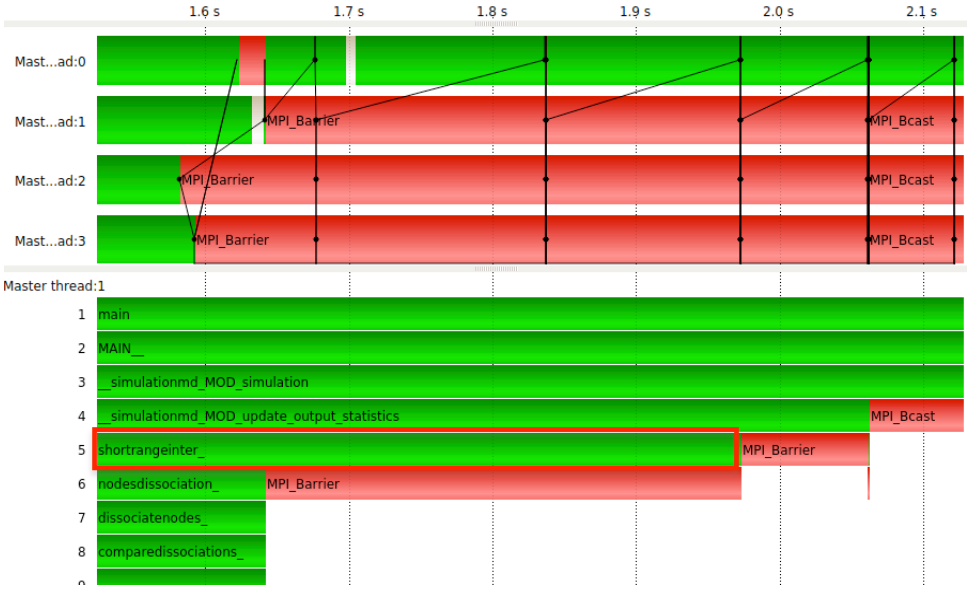


Figure 33 The function *shortrangeinter* is only executed by the master processor (~1000 dislocations)

Bad Load Balancing:

The function *dynamicMain* is a core parallel part of the DDD algorithm that consumes the largest fraction of execution time. Its speedup does scale up but is not as good as theory would predict. We noticed that the computational load for the dislocation dynamics may not be evenly distributed, so some processors complete their work earlier than the others and then begin to idle. Figure 34 below is an example that contains approximately 1000 dislocations. The master processor has more computation to do, so the other processors wait for significant time to begin the next step.

Figure 34 Imbalance of the work distribution between processors (~1000 dislocations)

3.5.2 Analysis by manual measurement

Along with the analysis from the profiling software, we also conducted manual measurement of execution time and message volume. The results provided further information regarding the computational complexity of dislocation dynamics, the overall speedup of the DDD algorithm and message volume transferred among processors.

Computational complexity

The computational complexity of dislocation dynamics is $O(n^2)$ because every force induced by each pair of segments needs to be calculated. In the DDD algorithm, the box method is applied to focus on short-range interaction and thereby divide the crystal volume into a certain number of boxes. Every segment belongs to a particular box and only the segments contained in 26 neighbor boxes will be considered as neighbor segments for the ones in central box. In this way, using more boxes to partition the crystal volume will result in less dislocation segments in each box, so the computation can be significantly reduced. Figure 35 shows the 2D schematic of the Box method [28]. The

central red box is surrounded by the neighbor boxes marked in green. For the red dislocation, the elastic stress field induced by green dislocation segments in the neighbor boxes will be accurately computed.

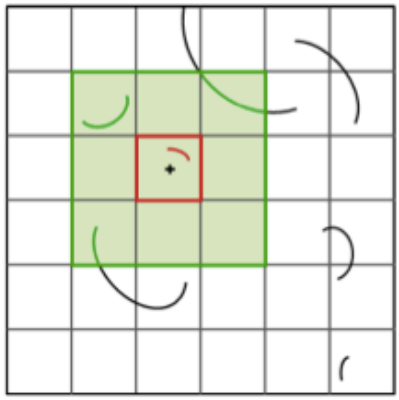


Figure 35 2D schematic of the Box method. For the red dislocation, the elastic stress field induced by green dislocation segments in the neighbor boxes will be accurately computed.

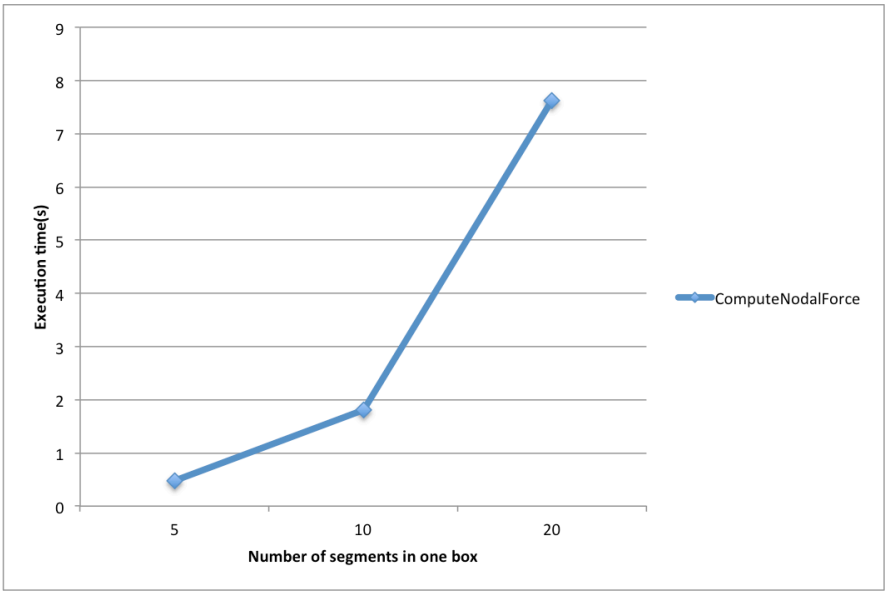


Figure 36 The computational complexity of dislocation dynamics. Using 1000 boxes (10x10x10) to partition the crystal volume and varying the total number of dislocation segments from 5000 to 20000 leads to distinct number of dislocation segments in one box.

Based on the box method, the computational complexity still remains the same, however the problem size n becomes the average amount of dislocation segments in one box instead of total amount of dislocation segments in crystal. The curve in Figure 36 hints at the quadratic relationship $O(n^2)$ between the execution time and the problem size.

Overall speedup

In order to measure overall speedup, we inserted timers at the beginning of initialization and in the last iteration of simulations to measure the total execution time. The performance with different number of processes is depicted in Figure 37. We conducted the measurement using 2 to 8 processes. With this configuration set, the speedup closely follows the theoretical line, whereas it starts to deviate when using more than 16 processes. This trend suggests that a previously unknown serial part of algorithm exists or that the increasing cost of MPI communication negatively impacts the scalability.

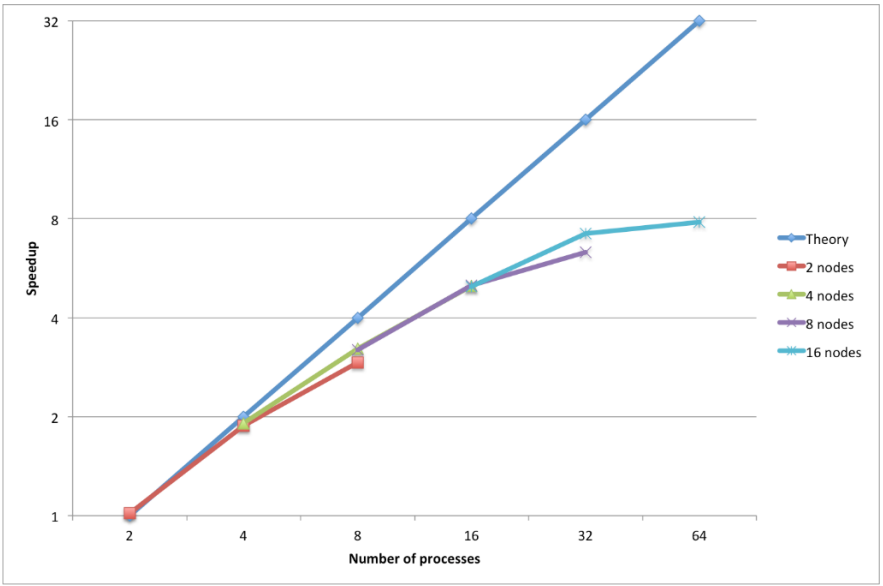


Figure 37 The overall speedup of DDD algorithm (~20000 dislocations, 30 iterations)

Parallel computation – Dynamic solver

Since the parallel function *DynamicSolver* accounts for the major part of simulation time, the overall speedup will be significantly impacted by its performance. Figure 38 shows that in the case of 20000 dislocations, the time spent in *DynamicSolver* is more than 90% when using 2 processes. We made use of two configurations, 2000 dislocations and 20000 dislocations, to verify its scalability. Figures 39 and 40 are the results of the measurement. For 2000 dislocations, the performance is close to theoretical predictions when using fewer processes. However, in the case of 20000 dislocations, the speedup doesn't increase proportionally at all with the number of processors. The possible reason for the performance with 20000 dislocations being worse than the one with 2000 dislocations is that the serial code for distributing dynamic segments consumes more of the total execution time and it limits the maximum achievable speedup.

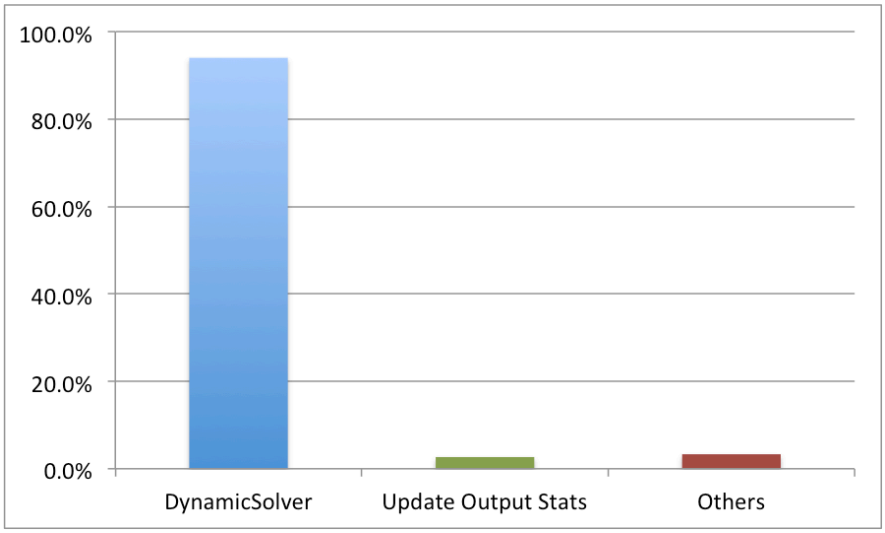


Figure 38 *DynamicSolver* accounts for more than 90% of execution time for each step (~20000 dislocations, 2 processes)

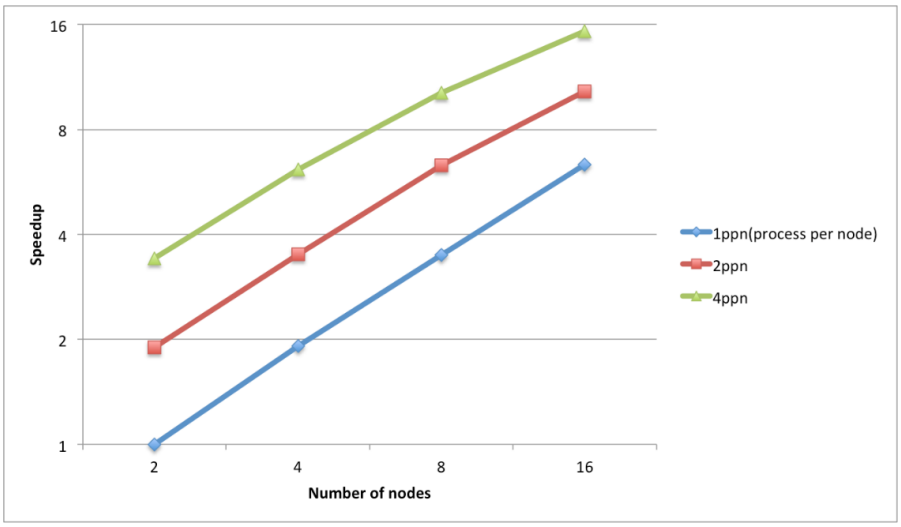


Figure 39 The speedup of function *DynamicSolver* (~2000 dislocations, 1000 iterations)

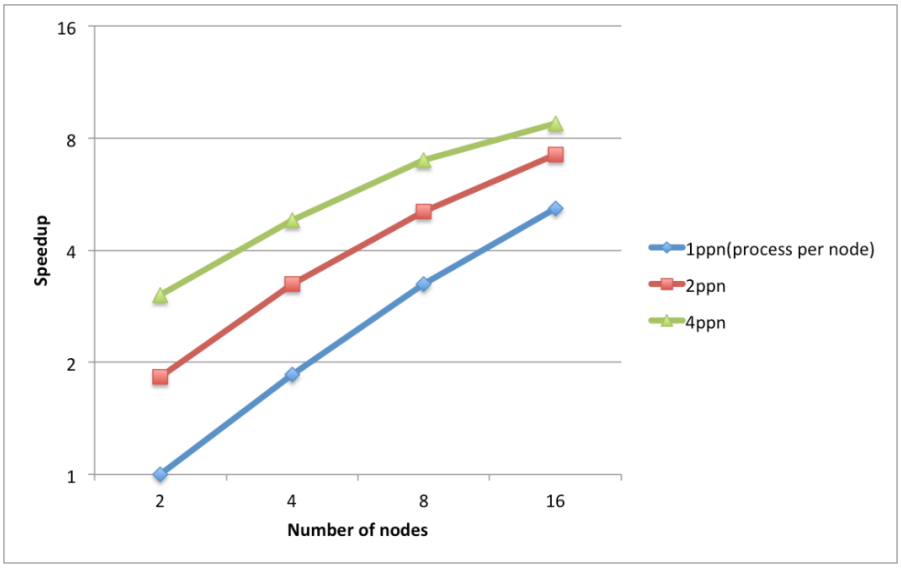


Figure 40 The speedup of function *DynamicSolver* (~20000 dislocations, 30 iterations)

Message Volume

MPI processes constantly exchange messages (or data) over program execution. As message exchange can comprise a large part of communication delay, so we also tracked the message volume in different important steps in order to verify if bottlenecks exist over the network in the cluster. We identify several important points associated with message passing in the program as indicated by Figures 41 and 42:

- The message volume in the initialization phase is less significant compared to the message passing that takes place during the iterative loops.
- At each iteration, the MPI Broadcast accounts for the largest fraction of message transfer because it sends the complete dataset of dislocation nodes.
- If more dislocations are added in the simulation scenario and the total message volume will certainly increase. However, the maximum capacity of a typical switch is at least 100Mbit/s (our cluster is 10Gbit/s), so the message volume should not be the cause of poor scalability.

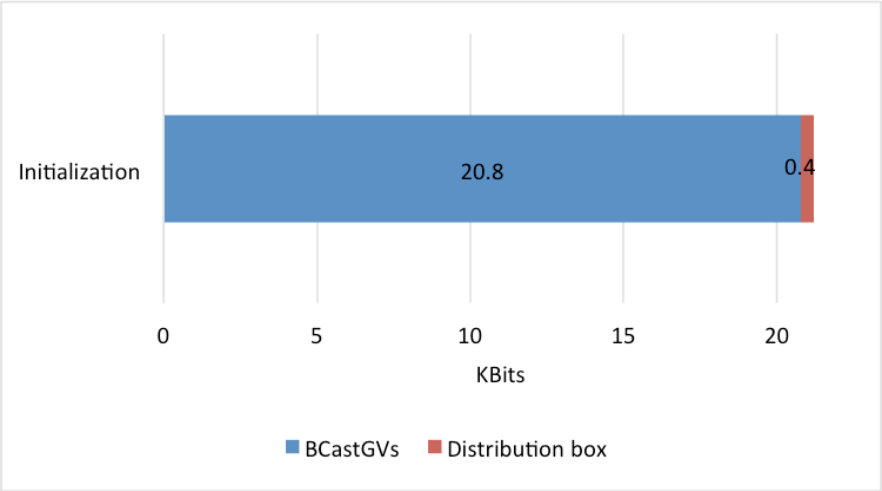


Figure 41 Message volume in the initialization phase (~1000 dislocations)

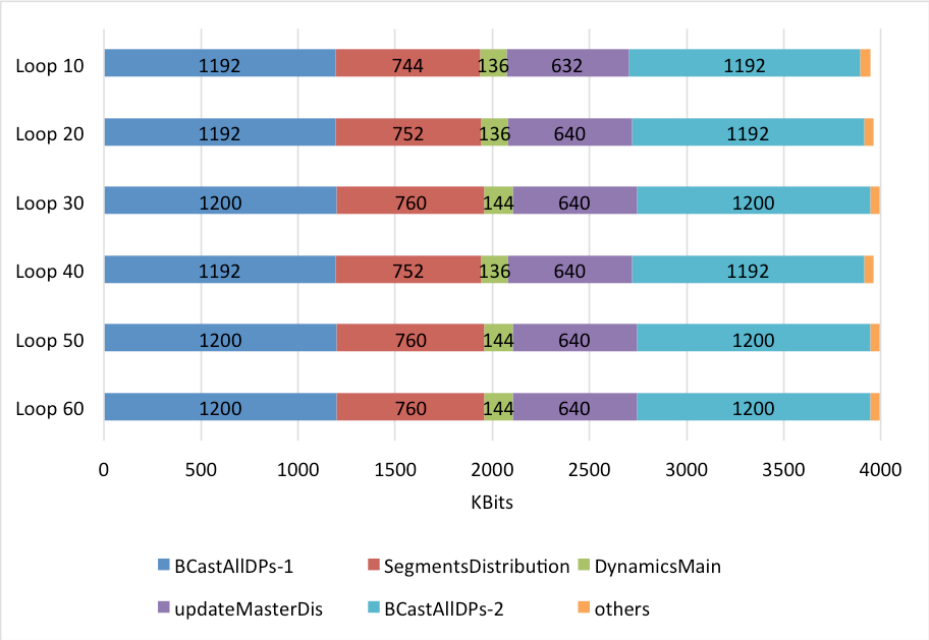


Figure 42 Message passing during the iterative loops (~1000 dislocations)

3.6 Proposed Solutions

The findings in previous section provide significant information of where refactoring of the algorithm will be possible. Considering the nature of these types of problems and the best fit into the structure of existing program, we have tried a variety of solutions and compared the performances after each modification. Some of the indicated solutions make sense for performance improvement and the algorithm structure, but some are not suitable because they increase the execution time or need to trade massive memory for the speed of numerical calculation. The details of these solutions will be explained along with the statistical charts showing results of each.

3.6.1 Dynamic Load Balancing

The first challenge in creating MPI programs is to determine how to divide the main problem into several smaller problems. The goal to data partition is to divide the data into pieces of roughly the same size and distributed these smaller data sets to

different MPI processors. Each of these MPI processors only operates on the assigned data. Also, because the data required for the problem solution may be dynamic during different iterations of the algorithm, tracking these changes dynamically and rebalancing them efficiently is very challenging. For the existing strategy in the DDD program, we consider the load as the computation of short-range interactive forces induced by the neighbor segments, so we count the total number of them for each dislocation segment based on the definition in the Box method. After sorting them in descending order by means of an insertion-sort algorithm, the dislocation segments are assigned to different MPI processors in round-robin fashion. Each processor only computes the interactive force for a portion of dislocation segments.

Table 4 Example of the load of dislocation segments. The dislocation segments are distributed to different MPI processors in round-robin fashion

Segment ID	No. of neighbor segments	Load	Processor ID
408	1000	1000	1
407	1000	1000	2
99	1000	1000	3
100	1000	1000	4
101	980	980	1
102	980	980	2
33	770	770	3
34	770	770	4

This approach has two drawbacks. First the insertion-sort algorithm is not efficient enough due to its complexity of $O(n^2)$. The sorting is the serial part of *DynamicSolver,* so it will affect the best achievable speedup when the number of dislocations is massive. Therefore, we replaced it with a heap-sort algorithm which has a better complexity of $O(nlogn)$. Next the round-robin model cannot reach the best construction. The first processor always takes the largest data set and the last one takes the smallest data set during the cycle of distribution, so the disparity between these two

processors is most likely very large. In order to achieve an even distribution, we created an array called *processor load* to track the amount of data already assigned to a particular processor. The processor that has the least load will take the next data set in the cycle of distribution, so the possible difference of load between processors will be minimized.

Furthermore, the original scheme of load balancing in the DDD program is static. It means that the load for each dislocation segment needs to be re-estimated and re-assigned at each iteration by repeating the steps, counting the number of neighbor segments, sorting them, assigning them, etc. To avoid redundant calculations, we have attempted to apply a dynamic balancing scheme for which the partition of load is based on dislocation nodes instead of dislocation segments due to the constraint of existing data structure of the DDD program. However, the definition of load remains unchanged, so the number of neighbor segments is calculated indirectly. Each dislocation node may have one or more connections (or segments) and we add up the number of neighbor segments for connections to derive the load. Table 5 is an example of this idea. Inspired by the strategy and the concept in [29], we built the scheme as shown in Figure 43:

Table 5 Example of the load of dislocation nodes. The load is calculated indirectly by adding up the number of neighbor segments of connections

Node ID	Connection ID	No. of neighbor segments	Load	Processor ID
1	407	1000	2000	1
	408	1000		
2	99	1000	2000	2
	100	1000		
10	33	770	770	3
11	34	770	770	4
12	123	130	130	3
97	125	130	130	4

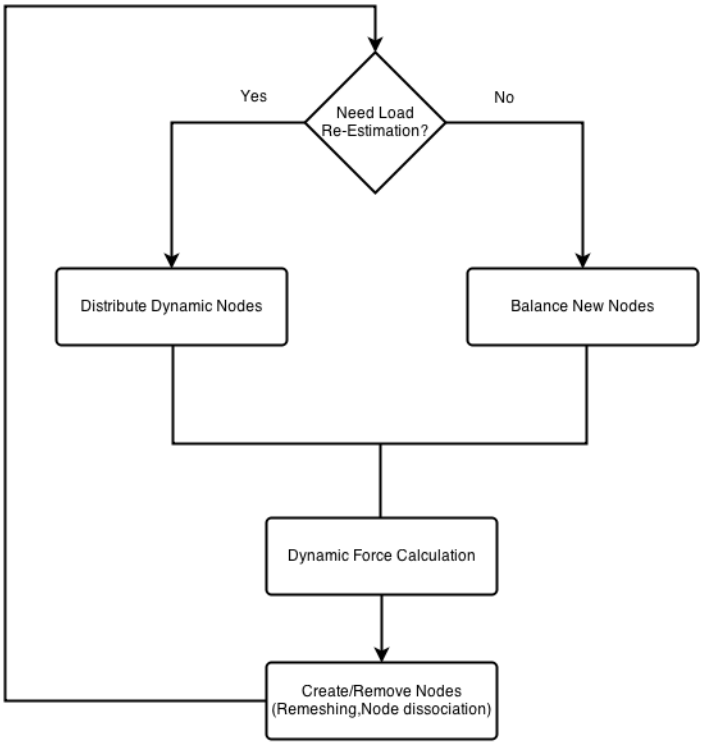


Figure 43 The scheme of dynamic load balancing on dislocation nodes

At the beginning of each iteration, we decide if there is a need to re-estimate the load. If yes, the calculation of neighbor segments for all the dislocation nodes will be carried out. Then these nodes will be assigned to different MPI processors for the next step. If not, we only take into consideration the new dislocation nodes created in the prior iteration, derive the load for them, and make the additional assignments. With this scheme, we can dynamically track the new incoming load and rebalance it. However, the difference in the loads between dislocation nodes can be immense because of the way we calculate it. Suppose that there are two dislocation nodes, one has many connections and each connection has many neighbor segments, but another only has one connection and this connection has few neighbor segments. In this case, an uneven distribution of the load to the processors will likely occur. As a result, more time is consumed in *DynamicSolver* and therefore the scalability is worse (as previously noted) when using more processes. In Figure 44, the speedup with the heap-sorting algorithm is significantly

improved as opposed to Figure 37 because the time spent in serial code is reduced. Meanwhile, Figure 45 indicates that the performance remains roughly the same because the extra execution time is introduced by the uneven distribution of the load whereas the scheme of dynamic load balancing can save time for assignment.

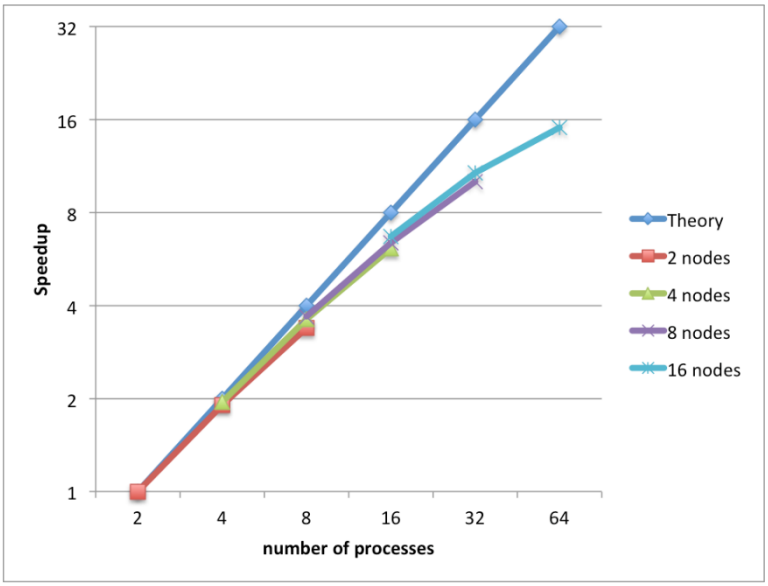


Figure 44 The overall speedup with heap-sorting and static load balancing on dislocation segments (~20000 dislocations, 30 iterations)

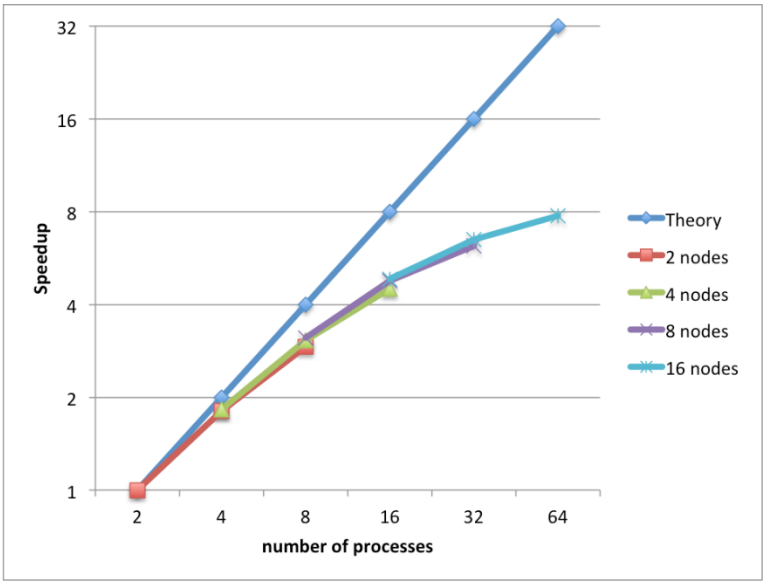


Figure 45 The overall speedup with heap-sorting and dynamic load balancing on dislocation nodes (~20000 dislocations, 30 iterations)

To sum up, utilizing dynamic load balancing is generally better than static load balancing because it prevents the repetitive re-estimation of the load and then the necessary re-assignment of this load to different processors. However, a new and more organized data structure for dislocation segments is needed to allow us to dynamically track the change of load based on dislocation segments.

3.6.2 Tabularization for the mathematical functions

Knowing that there are a massive number of numerical computations in the function *DynamicSolver*, we also used another profiling tool, *Valgrind*, to identify which parts of the calculation in this program consume the most computation time. The tree map from *Kachegrind* in Figure 46 reveals that the mathematical functions *log* and *atan* are called approximately 7 million and 3 million times, respectively, in the case of 1000 dislocations and 20 iterations. Those amounts of calls represent respectively 10% and 4% of the total computation time in calculating the interactive force.

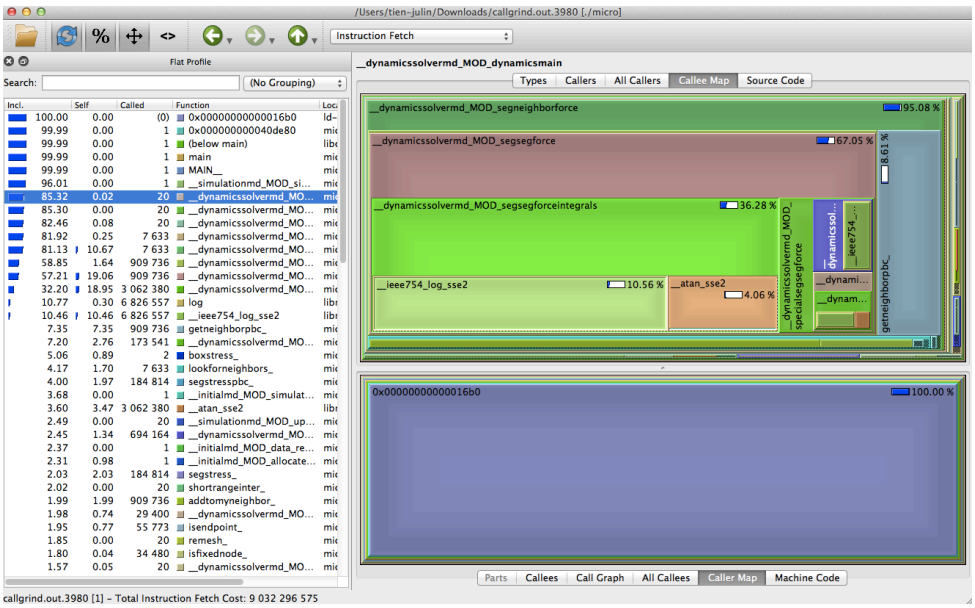


Figure 46 The functions *log* and *atan* consume the significant time for solving the dislocation dynamics (Tree map from *Kachegrind*)

Because of this significant time of computation, we generated a model based tabularization. This model attempts to create the arrays beforehand to store the output values given the possible inputs for the function *log* and *atan*. Since the domain of input is continuous without boundaries, we only target a certain range and evenly partition it into several intervals. Each interval is represented by the value in its center, and we pre-compute the output values based on those representatives. Tabularization provides the convenience of direct memory access so that the output value can be determined simply by which interval the input value falls in. In this way, we expect that the total time spent in solving the dislocation dynamics could be significantly reduced. Figures 47 and 48 illustrate how we define the representative and the precision for the tabularization.

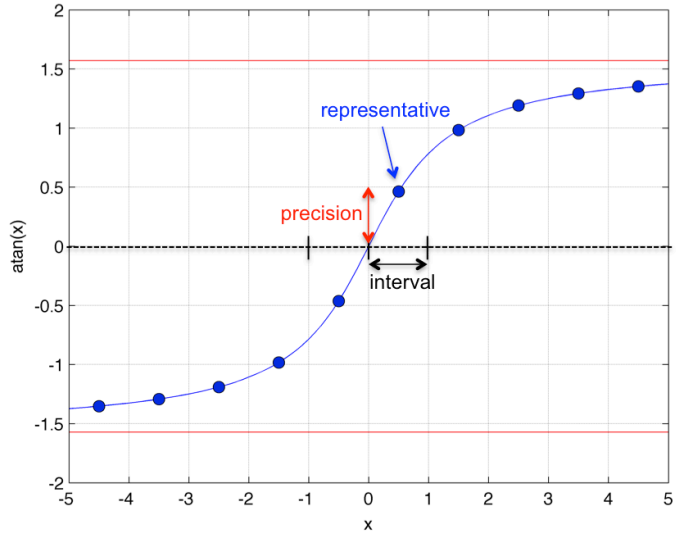


Figure 47 Tabularization for *atan*. Here the range is between -5 and 5 and the number of intervals is 10. Each blue point is the representative of its interval, and the precision is defined as the maximum possible difference between the exact mathematical calculation and the approximation

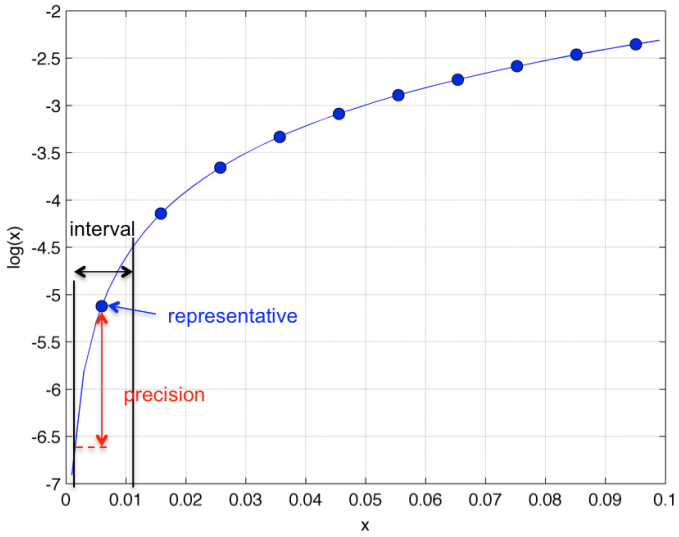


Figure 48 Tabularization for *log*. Here the range is between 0.001 and 0.1 and the number of intervals in 10.

The tabularization can only be implemented for a certain range of values, so it is first necessary to analyze the distribution of possible input values for these two mathematical functions during the program. With the aid of the histograms in Figures 49 and 50, we noticed that the possible inputs for *atan* are centered between -100 and 100 and the ones for *log* are less than 0.1. Although the tabularization has the performance in direct memory access, it also introduces a precision problem that may affect the whole simulation. In the general sense, the larger size the array has, the more precision that can be achieved for a fixed range of inputs.

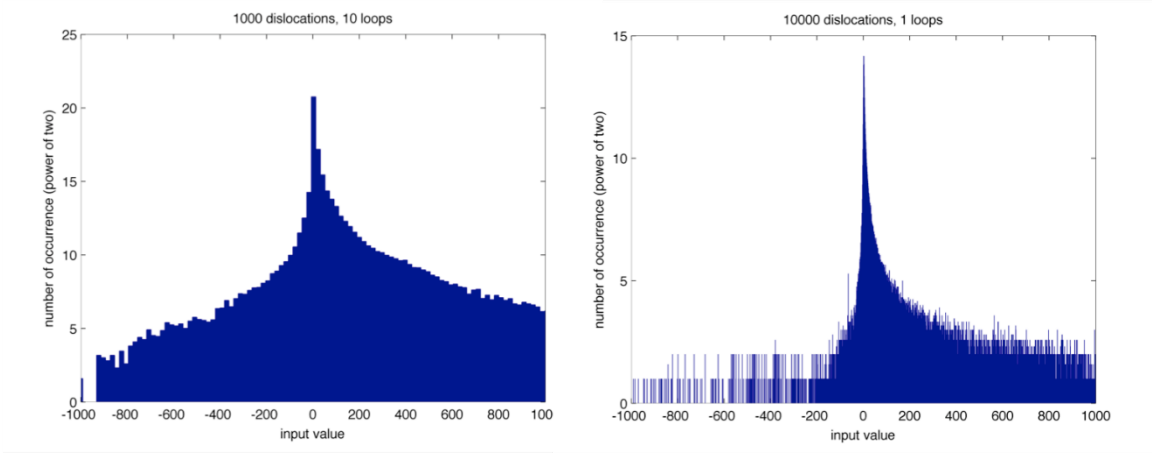


Figure 49 The distribution of input values for *atan*

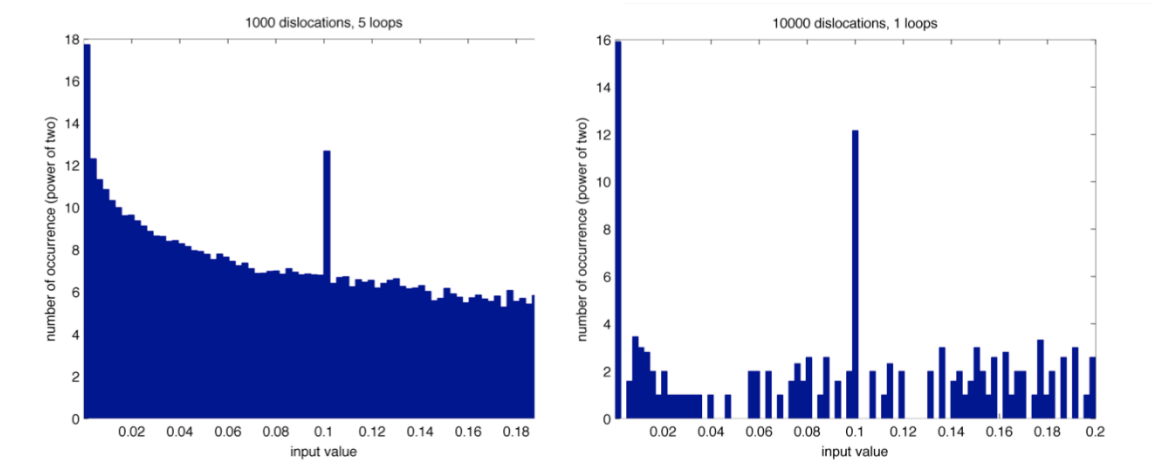


Figure 50 The distribution of input values for *log*

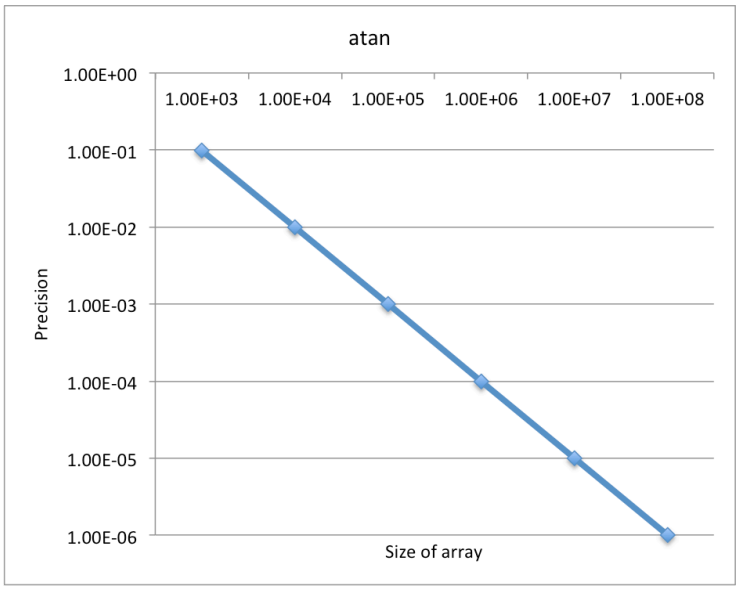


Figure 51 The relationship between the array size and the precision for *atan*. The range of tabularization is between -100 and 100.

Figure 52 The relationship between the array size and the precision for *log*. The range of tabularization is between 10^{-7} and 0.1

Figures 51 and 52 reveal the relationship between the array size and the precision. To verify if the precision of tabularization provides a significant impact on the material-stress simulation, we used two specific configurations requiring very high precision to compute the dynamic interaction among the dislocations. The results showed us that the

function *DynamicSolver* requires a precision of 10^{-7} for the function *atan* and 10^{-4} for the function *log*. A bad approximation could lead to wrong timing of the evolution of dislocations or unstable force interactions. Figures 53 and Figure 54 are the screenshots of animations of these two different test simulations.

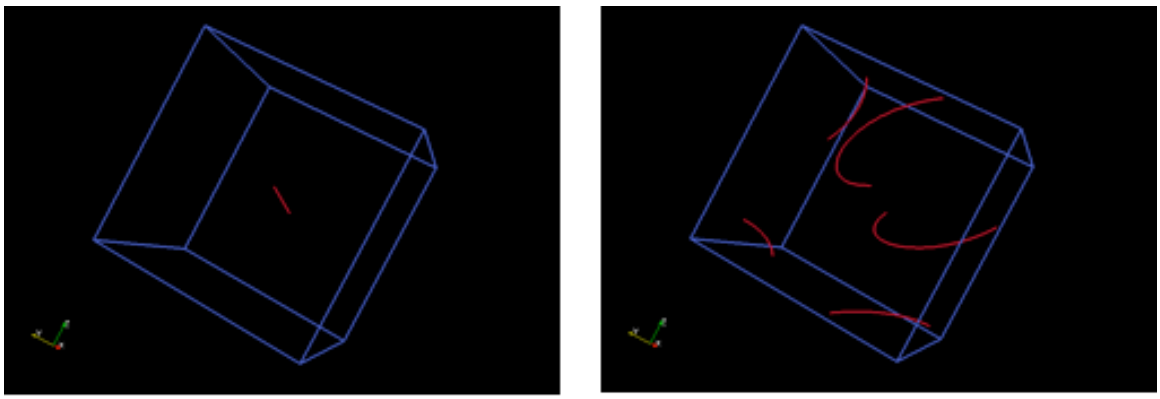


Figure 53 Precision test – Evolution of dislocations (Activation of a Frank Read source)

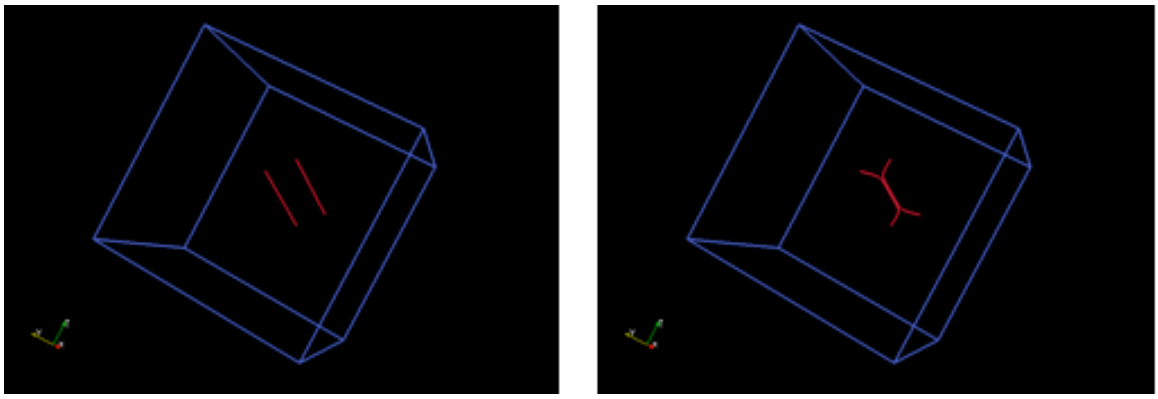


Figure 54 Precision test – Stable force interaction (Dislocation dipole)

In order to achieve such high precision, we can either reduce the range of inputs or increase the array size in order to have more possible inputs. For the former solution, the improvement on speed is limited because inputs outside the range still need to be calculated through mathematical functions. And for the latter one, a very large memory space, more than 1GB, must be consumed in order for the improvement to realized, so it is not practical for most of machines. Apart from the tabularization, the article [30] offers

a means for fast and convenient calculation of the logarithmic function to essentially four significant digits. In this way, no extra memory is demanded during any phase of pre-computation, but the precision is still not sufficient for the numerical computation in dislocation dynamics.

3.6.3 Algorithm tuning - regrouploop

The function *regrouploop* is used for the segmentation of dislocations during the process of remeshing and node dissociation. This function finds the total number of groups and loops of dislocations and assigns the corresponding loop ID and group ID to each dislocation node. Figures 55 and 56 show the examples of groups and loops. Since the relationship among dislocation segments and dislocation nodes is a graph, it makes more sense to resolve this problem by applying particular graph algorithms.

A group of dislocations is considered as an independent connected component in which any two nodes are connected to each other by a path and which is connected to no additional nodes in the graph. A graph that is itself connected has exactly one connected component consisting of the whole graph. To find groups, we can either use depth first search (DFS) or breath first search (BFS). Here, we implemented DFS because it can be reused to tackle the problem of finding loops with its searching priority. A search that begins at some particular node *v* will find the entire group containing *v* before returning. By looping through all the nodes and marking the ones that have been visited to avoid re-traversal, we can identify all groups in the dislocation graph.

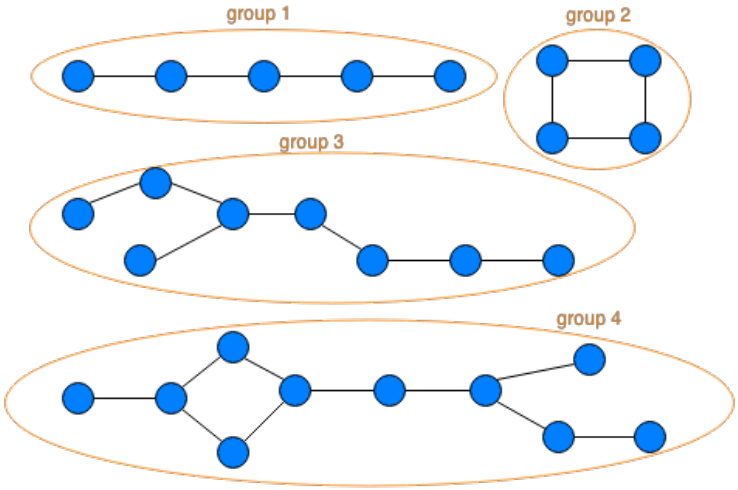


Figure 55 Example of groups in dislocation graph

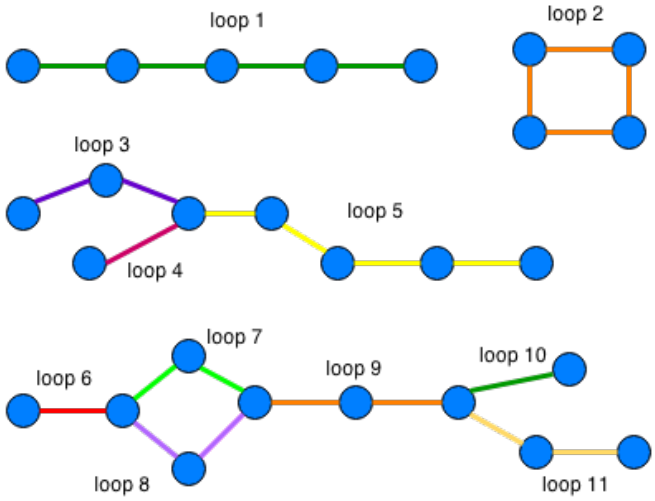


Figure 56 Example of loops in dislocation graph

The solution for finding loops is not as straightforward as the one for finding groups. A loop can be seen as a branch in the graph in which any node has two connections except the end nodes that may have only one connection or more than two. So there are more possible cases like the examples in the previous figure. Basically, the shape of the loop is either a path or a circuit. In order to find all of them, we initially

decompose each group into several sub-groups by duplicating the nodes with more than two connections and decoupling them from each other. This method of decomposition is shown in Figure 57. With the new graph composed of several sub-connected components, we can do the same traversal to find the total number of loops. Furthermore, DFS also brings one additional convenience, namely the prevention of wrongly counting the circuits since it always visits the child nodes before the neighbor nodes.

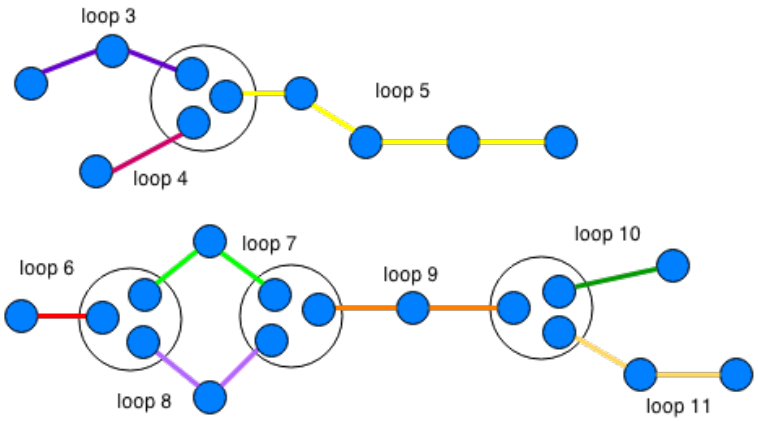


Figure 57 Decompose the group by duplicating the nodes with more than two connections and decouple them from each other

To sum up, the two main objectives of the function *regrouploop* can be accomplished by DFS. Its complexity is $O(|V|+|E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges, so the cost of computation is linear which is much better than the old algorithm. Figure 58 shows the execution time of new *regrouploop* and old one. Due to the fact that the old algorithm is designed informally, we cannot compare them directly in terms of complexity.

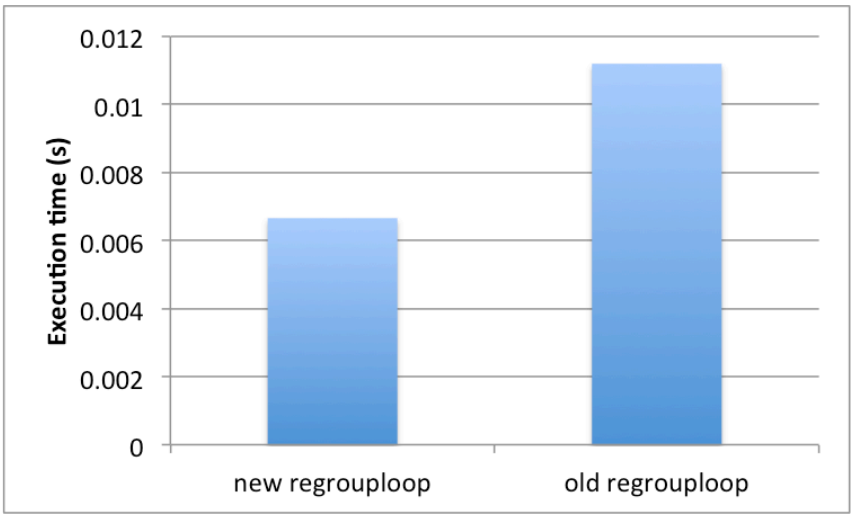


Figure 58 The average execution time of new *regrouploop* and old *regrouploop* (~20000 dislocations, 10 iterations)

3.6.4 Operating points

In this section, we focus on the relation between the overall speedup and the problem size of dislocation dynamics. In the Box method, two important parameters, the total number of dislocation segments and the box size, determine the problem size *n* (the number of dislocation segments in one box). For instance, with around 1000 dislocations in crystal volume, a box size of 3 is the special case where all the dislocation segments except the ones in central box are considered as neighbor segments. If the box size is 4, we can derive *n* roughly as 16 ($\sim 1000/4^3$). And if the box size is 5, *n* is 8. Nevertheless, if the box is 6, *n* is less than 5, leading to over-approximation. In general, the box size is always greater or equal to 3 and less than the number that results in the problem size less than 5.

As Gustafson's law says, with the increasing problem size and the roughly fixed fraction of execution time for serial code such as data partition and I/O operations, the best achievable speedup will increase as well. In our case, we still need to consider the MPI communication cost, but the performance will not change drastically based on these cost since the bandwidth of network in the cluster is sufficient. Knowing that most of the

material-stress simulations for research projects are lengthy and time-consuming, it is better to know the suitable operating points where we can launch the simulation with the proper number of processors according to different problem size. Figure 59 is an example of 10000 dislocations for which we varied the number of processes and the box size to obtain the best achievable speedups respectively. The contour chart gives a clear view of different levels of speedup marked in corresponding colors. Other than the configuration of 1000 dislocations, there are three more practical ones (5000, 10000, and 20000 dislocations) and their contour charts are shown in Figures 60, 61 and 62.

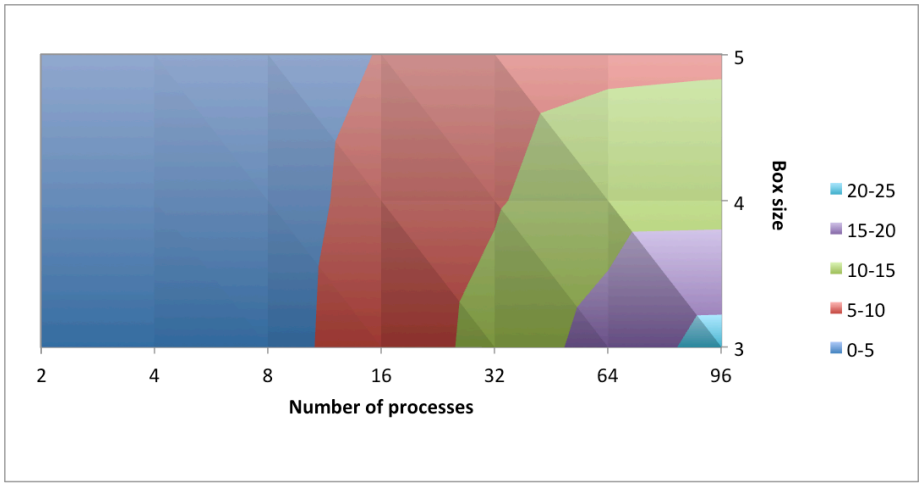


Figure 59 The overall speedup with different combinations of the number of processes and the box size (~1000 dislocations)

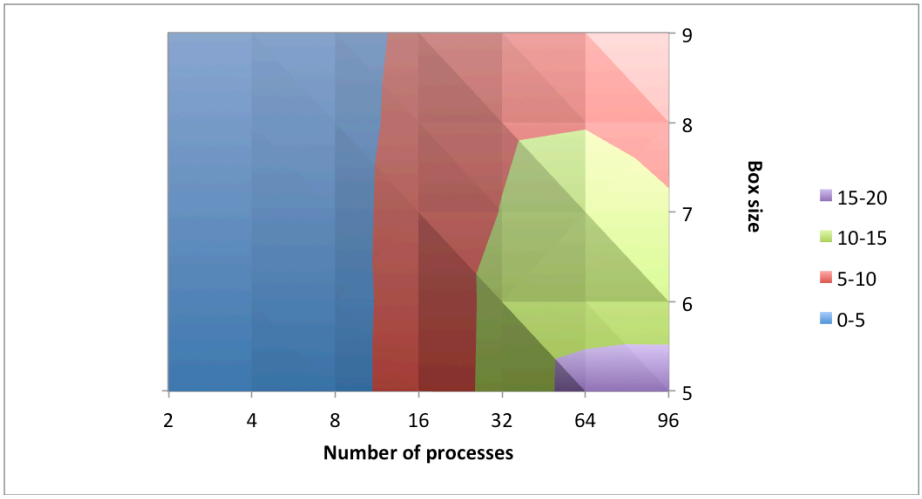


Figure 60 The overall speedup with different combinations of the number of processes and the box size (~5000 dislocations)

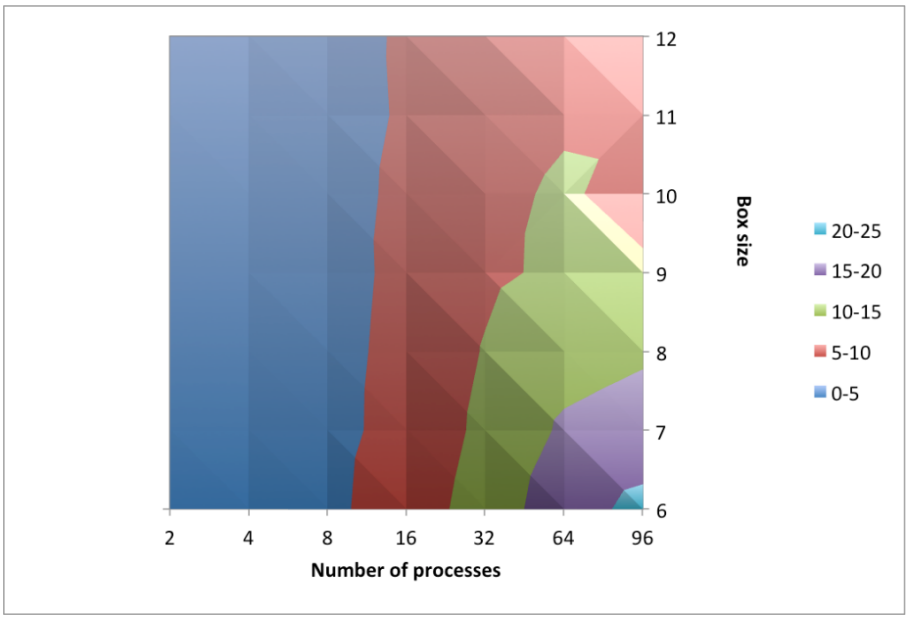


Figure 61 The overall speedup with different combinations of the number of processes and the box size (~10000 dislocations)

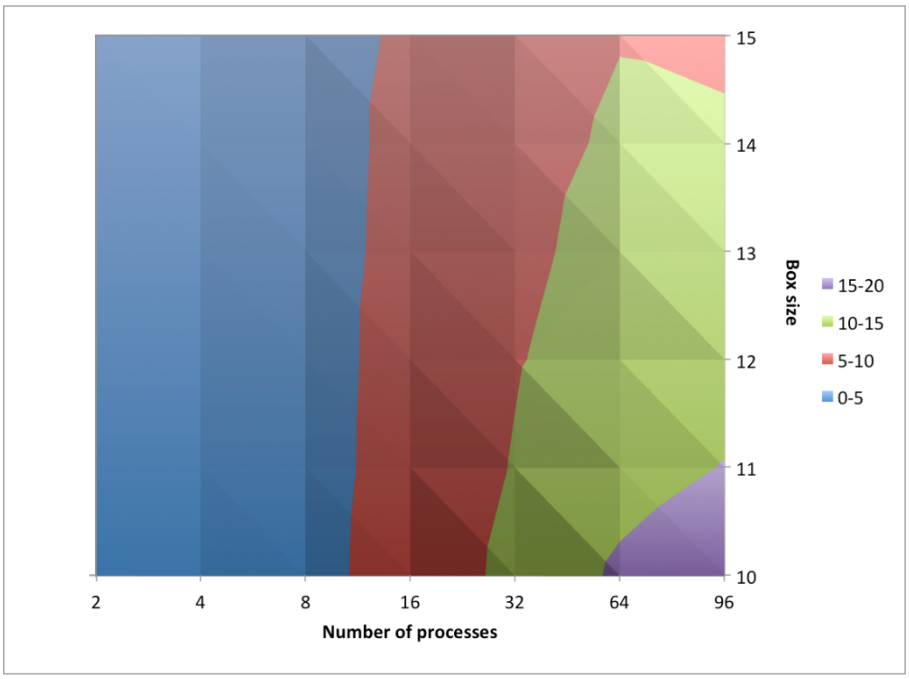


Figure 62 The overall speedup with different combinations of the number of processes and the box size (~20000 dislocations)

3.6.5 Efficient use of memory

From the experimental results, we see that some of the serial code consume more time when using more processes per node. For those functions, the memory allocation and deallocation occur very frequently, so these operations can be very expensive under the structure of MPI which does the parallelization in distributed memory fashion. When the separate processors in the same node attempt to demand memory space, it takes a fixed amount of time for the node to carry out the arrangement. Therefore, with more than one process per node, the extra time for the organization of memory space is required.

CHAPTER 4

CONCLUSION

In this thesis, we made the following contributions to 1) the development of front-end application for material-stress simulation software and 2) the optimization of the parallel DDD algorithm. Compared to the command line interface based software such as ParaDIS and NumoDIS, the DDD portal with a graphic interface provides more intuitive and efficient operations. A person who is either an expert or a novice user in relevant fields can easily generate the long input file for the material-stress simulation and analyze the simulation results with the aid of a rich set of functions available in the software. As for the deployment, we distribute two versions of application. The laptop version is intended for any user who has the interest in material science and the hosted version which is intended to be installed in a computer cluster with limited access for the professional use. Apart from the front-end development, the optimization of the DDD algorithm must also deal with the complicated simulation scenarios with appropriate performance characteristics. The analysis performed using both manual measurement and profiling identified the potential root causes of poor performance and scalability. Based on these important findings, we proposed various solutions such as dynamic load balancing, tabularization and algorithm tuning. The best achievable speedup was found to depend on the problem size, so we ran a number of simulations for different configurations to derive the proper operating points. These valuable insights can help us understand how to improve the parallel DDD algorithm.

CHAPTER 5

FUTURE WORKS

The DDD portal is the first prototype of a simulation tool using a front-end application, so room for the improvement still exits regarding the architecture, the choice of programming languages, the functions and the deployment. For instance, the visualization of simulation results might in the future be implemented in a real-time fashion, allowing the user to see the instant change in the curves reflecting the important indicators for the plasticity. Also, using an appliance to package a front-end application is only intended as a temporary solution, so future work would customize the package for different unique operating systems. An innovative application can be only created through having a good understanding of the needs of the user and strong in-depth knowledge about the new technologies of front-end development. For future work of optimizing the DDD algorithm, a focus can be placed on the data structures and the computation of dislocation dynamics since these two elements determine how the parallelization scheme is constructed and thereby impacts the performance and speedup. More organized data structures will help enforce the dynamic load balancing on dislocation segments to avoid the repetitive re-estimation of load and re-assignment of work to different processors. Because the problem of dislocation dynamics involves a great quantity of geometric computation, we can make use of some existing libraries such as CGAL (Computational Geometry Algorithms Library) to deal with this computation. Although CGAL is a C++ library, we can combine it with Fortran code and integrate these together into a single executable that knows how to interface the function calls. In addition, a hybrid solution of OpenMP [24] and MPI should also be investigated because it includes the benefits of distributed memory system in a high level and shared-memory system in each local machine.

APPENDIX A

SOFTWARE EVALUATION QUESTIONNAIRE

DDD portal Evaluation Form					
Name:			Department:		
Trainer:			Class Name:		
Role: professor researcher student					
Accessibility	**5**	**4**	**3**	**2**	**1**
Easy to access the documentation					
Easy to download the software					
Easy to install the software					
Usability:	**5**	**4**	**3**	**2**	**1**
Navigation					
Does the site provide the clear indication of current location					
Are all major parts of the site accessible from homepage					
Is the site simple without unnecessary levels					
Function					
Are all necessary functions available and clearly labeled					
Do all functions perform their intended tasks					
Quality:	**5**	**4**	**3**	**2**	**1**
How user-friendly is our software's interface					

How easily do you find particular information in our documentation					
How successful is our software in performing its intended task					
How often do you find our software freeze or crash					
How helpful is the support service of our team					
Expectation:					
How can we improve the software?					
What other functions (capabilities) we should add?					
What other sections or information we should add in our documentation?					
5: Excellent, Extremely often 3: Good, Moderately often 1: Poor, Moderately often					

REFERENCES

[1] "ParaDIS" http://paradis.stanford.edu/

[2] "NumoDIS" http://www.numodis.fr/

[3] V. Bulatov and W. Cai, "Computer simulations of dislocations", *Oxford University Press*, chapter 10, 2006

[4] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms", *In Proc. 40th Ann. Symp. Foundations of Computer Science (FOCS-99), IEEE Press*, pp 285–297, New York, NY, 1999

[5] Umit V. Catalyurek, F. Dobrian, A. Gebremedhin, M. Halappanavar and A. Pothen, "Distributed-memory parallel algorithms for matching and coloring"

[6] Ruoming Jin, Ge Yang, and Gagan Agrawa , "Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance", *IEEE Transactions on knowledge and data engineering*, vol. 16, no. 10, October 2004

[7] Natallia Kokash, "An introduction to heuristic algorithms"

[8] L.P. Kubin, G. Canova, M. Condat, B. Devincre, V. Pontikis and Y. Brechet, "Dislocation microstructures and plastic flow: a 3D simulation", *Solid State Phenomena*, vol. 23&24, pp 455–72, 1992

[9] A. Arsenlis, W. Cai, M. Tang, M. Rhee, T. Oppelstrup, G. Hommes, T. Pierce and V. Bulatov, "Enabling strain hardening simulations with dislocation dynamics", *Modelling Simul. Mater. Sci. Eng. 15*, pp 553-595, 2007

[10] N. Bertin, C.N. Tomé, I.J. Beyerlein, M.R. Barnett, and L. Capolungo, "On the strength of dislocation interactions and their effect on latent hardening in pure Magnesium", *International Journal of Plasticity 62*, pp 72-92, 2014

[11] "Magic Web Solution" http://www.magicwebsolutions.co.uk

[12] “Backbone.js” http://backbonejs.org/

[13] “jqplot” http://www.jqplot.com/

[14] “Bootstrap” http://getbootstrap.com/

[15] Larry L. Constantine and Lucy A.D. Lockwood, “Software for use: a practical guide to the essential models and methods of usage-centered design”, 1999

[16] V. Balasubramoniam and N. Tungatkar, “Study of user experience (UX) and UX evaluation methods”, *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, vol 2, issue 3, March 2013

[17] V. Roto, E. Law, A. Vermeeren and J. Hoonhout, “User Experience white paper”, 2011

[18] S. Kujala, V. Roto, K. Väänänen-Vainio-Mattila, E. Karapanos and A. Sinnelä, “UX Curve: A method for evaluating long-term user experience”, *Interacting with Computers*, 2011

[19] Jesse James Garrett, “The elements of user experience: user-centered design for the web”, 2002

[20] “Vampir” https://www.vampir.eu/

[21] “Score-P” http://www.vi-hps.org/projects/score-p/

[22] David A. Bader, Bernard M.E. Moret, and Peter Sanders, “Algorithm engineering for parallel computation”

[23] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard”, University of Tennessee, Knoxville, TN, Version 3.0, 2012

[24] OpenMP Architecture Review Board, “OpenMP Application Program Interface”, Version 4.0, July 2013

[25] Alaa Ismail El-Nashar, “To parallelize or not to parallelize, speed up issue”, *International Journal of Distributed and Parallel Systems (IJDPS)*, Vol.2, No.2, March 2011

[26] Mark D. Hill and Michael R. Marty, “Amdahl’s Law in the Multicore Era”

[27] J. Gustafson, “Reevaluating Amdahl's Law”, *Communications of the ACM*, vol 31, no. 5, pp 532-533, 1988

[28] M. Verdier, M. Fivel, I. Groma, “Mesoscopic scale simulation of dislocation dynamics in FCC metals: principles and applications”, *Model. Simul. Mater. Sci. Eng., 6 (1998)*, pp. 755–770

[29] Marc H. Willebeek-LeMair and Anthony P. Reeves, “Strategies for dynamic Load balancing on highly parallel computers”, *IEEE transactions and parallel and distributed system*, vol. 4, no. 9, September 1993

[30] Ron Doerfler, “Fast approximation of the tangent, hyperbolic tangent, exponential and logarithmic Functions”, June 2007