

PARALLEL ALGORITHMS APPLIED TO PROBLEMS IN TWO
DIMENSIONAL DETONATION SHOCK DYNAMICS

BY

ALBERTO M. HERNANDEZ

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Mechanical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor Donald Scott Stewart

Abstract

This design and applications project consists in the development of a parallel extension for a two-dimensional Detonation Shock Dynamics code, and to demonstrate how it can be applied for solving engineering problems in detonation physics. Detonation Shock Dynamics (DSD) is an asymptotic theory that describes the evolution of a multidimensional curve detonation shock in terms of an intrinsic evolution equation for the shock surface. Full-LS-DSD2D is a full level set Detonation Shock Dynamics code in Fortran 77 written by Dr. John Bdzil specifically for this project. A level set function numerical algorithm which embeds the two-dimensional detonation front in a three-dimensional field function, $\phi(x, y, t)$, is used to solve for the location of the detonation front, which is given by $\phi(x, y, t) = 0$. The code solves a modified Level Set PDE which maintains $\phi(x, y, t)$ as a distance function and uses a fully explicit scheme. A parallel extension of the code was designed, IPC-DSD2D (Illinois Parallel Cluster DSD2D), as a Message Passing model using an MPI interface. IPC-DSD2D was benchmarked for scalability, accuracy and overall performance. Benchmarking was performed on a vertical rate stick problem that had ideal load balancing properties. The test problem was run on three different computer architectures: the Turing Cluster at the University of Illinois Urbana-Champaign, an eight core Macintosh Mac Pro, and NCSA's SGI Altix (Cobalt). The benchmarking of the code showed very good performance metrics; the speedup and efficiency were high, and behaved in a stable and predictable pattern.

After the code was verified and tested for performance and efficiency, it was used in a shape optimization study. A multicomponent nonlinear optimization system was built to generate optimal, shaped charge geometries using Detonation Shock Dynamics. The idea was to use IPC-DSD2D to estimate the shock pressure along a shaped charge liner and the normal shock velocity at the apex of the liner. These flow variables are then to be used as inputs for a Lagrangian finite element code to determine the shape of the jet that is formed by the detonation shock pressure crushing the liner. Through a set of constrained objective functions, a nonlinear optimizer, a shape can be found that has optimal jet properties. By running a DSD simulation of a simplified shaped charge, it was successfully shown how DSD could be used in the design of

shaped charges. This thesis only describes the optimization system, and did not simulate the design loop.

This thesis is divided into ten chapters. Chapters 1 and 2 briefly describe the theory of DSD and some necessary concepts in parallel computing design. Chapters 3 through 5 talk about the mathematical and numerical model used in DSD2D, and the parallel implementation of the code. Chapter 6 shows numerical results using IPC-DSD2D and Chapter 7 shows the parallel benchmarking of the code using the three computer architectures mentioned earlier. Chapter 8 describes the optimization system using DSD to find optimal shape charge geometries. Chapter 9 shows how to extend IPC-DSD2D for a three-dimensional DSD code [5]. Chapter 10 has the conclusions and final thoughts about the parallel implementation of Full-LS-DSD2D and the optimization system for designing shape charges using DSD.

Acknowledgments

I want to thank my advisor, Professor Donald Scott Stewart, for his guidance and support throughout my graduate studies here at the University of Illinois. I would also like to thank Dr. John Bdzil for all his help and advice that led to the completion of this thesis. I extend my gratitude to my lab colleagues Brian Taylor and Juan Saenz for their advice on my research projects. Finally I want to thank my family for all their encouragement and support. The work in this thesis has been generously supported by grant and contract resources from the U.S. Army Armament Research, Development and Engineering Center through UTRS, subcontract UTRS 09-292 and the U.S. Air Force Research Laboratory, Munitions Directorate, Eglin AFB, FA865-05-1-0003.

Table of Contents

Chapter 1: Introduction	1
1.1 Detonation Shock Dynamics.....	2
1.2 The Level Set Method.....	3
Chapter 2: Parallel Computing.....	5
2.1 Parallel Computing Terminology	5
2.2 Memory Architectures	6
2.2.1 Shared Memory.....	6
2.2.2 Distributed Memory	6
2.3 Parallel Memory Models.....	7
2.3.1 Message Passing Model.....	7
2.3.2 Threaded Model	7
2.4 Performance Analysis	8
2.4.1 Speedup.....	10
2.4.2 Efficiency	11
2.4.3 Amdahl's Law	11
Chapter 3: Mathematical Model	13
3.1 DSD Boundary Conditions	15
3.2 Node Sorting	17
Chapter 4: Numerical Model	20
Chapter 5: Parallel Model	23
5.1 Node Sorting Model.....	24
5.2 Domain Decomposition Model.....	24
5.3 Communications Model.....	26
5.4 Internal Boundary Model.....	28
5.5 High Explosive Model	28
5.6 I/O Model.....	28
5.7 Design Model.....	29
Chapter 6: Numerical Results	32

Chapter 7: Benchmarking	36
7.1 Performance Analysis Tables and Plots.....	37
Chapter 8: Shaped Charge Optimization Using DSD.....	42
8.1 Shaped Charges.....	42
8.2 DSD Shaped Charge Simulation.....	43
8.3 Shape Charge Optimization	49
Chapter 9: Parallel extension for a 3D Version of IPC-DSD2D	53
Chapter 10: Conclusions	56
References	57

Chapter 1: Introduction

Basic detonation models consist in solving a set of Partial Differential Equations (PDEs). These equations are the compressible Euler equations, an equation of state with reactive rate laws and its accompanying progress variables. Equation (1) shows the two-dimensional compressible Euler equations in conservative form and in Cartesian coordinates, and where equation (1a) is the continuity equation, equation (1b) is the x-momentum equation, equation (1c) is the y-momentum equation and equation (1d) is the energy equation. Equation 2 is the reactive equation, where r is the reactive law as a function of pressure p , density ρ , and the reaction progress variable λ . To complete the system an equation of state, $e(p, \rho)$, also needs to be specified. On Eq.(1d) E is total energy.

$$\rho_t + (\rho u)_x + (\rho v)_y = 0 \quad (1a)$$

$$(\rho u)_t + (\rho u^2 + p)_x + (\rho uv)_y = 0 \quad (1b)$$

$$(\rho v)_t + (\rho uv)_x + (\rho v^2 + p)_y = 0 \quad (1c)$$

$$(E)_t + (uE + up)_x + (vE + vp)_y = 0 \quad (1d)$$

$$(\rho \lambda)_t + (\rho u \lambda)_x + (\rho v \lambda)_y = \rho r(p, \rho, \lambda) \quad (2)$$

Here λ satisfies the inequality $0 \leq \lambda \leq 1$ where $\lambda = 1$ is a completely reacted material and $\lambda = 0$ is fresh material. Due to the nature of the flow, discontinuities or shocks can be formed. To solve Eqs.(1-2) with the presence of discontinuities the Rankine-Hugoniot relations are needed. Under very specific conditions the Euler equations have analytical solutions, but in most engineering applications problems are complex and no analytical solution can be derived. In these cases numerical methods are used to solve the mathematical system of equations, these methods are called Direct Numerical Simulation (DNS) methods. Detonation problems are multiscale in nature; for example the length of a typical reaction zone is in the order of millimeters, and this is about two to four orders of magnitude less than other representative explosive dimensions. This means that to accurately resolve reactive zones, high numerical resolutions are needed. This requires substantial computational resources and work. It was shown in [4] that in the order of 50 to 100 computational cells are needed in the stream-wise direction to get accurate solution for the

multidimensional reaction zone. Computational work also increases as the problem gains complexity, especially if three-dimensional problems need to be solved.

1.1 Detonation Shock Dynamics

Detonation Shock Dynamics (DSD) is an asymptotic theory that describes the motion of a detonation shock in terms of the intrinsic geometry of the shocks' surface. It assumes small curvature and slow time variation of the detonation front. Unlike DNS, DSD does not solve the reactive Euler Equations, but solves a PDE (defined in Chapter 3) that relates the normal detonation shock velocity to the detonation front shock curvature. DSD simulations are computationally cheaper to run compared to DNS simulations [12]. A result of DSD theory is equation (3) which describes a simple model for the detonation shock propagation.

$$D_n = D_{CJ} - \alpha(\kappa) \quad (3)$$

Here D_{CJ} is the 1D steady Chapman-Jouguet velocity for the explosive, D_n is the normal velocity of the shock surface, and $\alpha(\kappa)$ is a function of curvature, usually found through experiment. From DSD theory the detonation shock front is known at every point in the computational domain, as well as the detonation normal velocity and curvature. The results of DSD can also be coupled with information from a particular equation of state to determine shock pressures, sound speeds and other flow properties. Figure 1.1 was taken from [4] and shows a comparison of DNS results with those obtained using DSD for a detonation wave passing over a lead disk. The white lines are results from DSD and the black lines from DNS.

Since DSD theory is based on curvature it is important to properly treat boundary conditions. The angle made between a detonation shock edge with an inert confinement, and the type of flow (sonic, supersonic and subsonic) will determine how the shock front is evolved. In DSD the angle made between the local shock normal, n_s , and the outward pointing normal vector to the boundary, n_b , is referred as the DSD edge angle, ω . Figure 1.2 shows a typical reactant/inert boundary. If the flow is locally supersonic at the DSD edge the shock reflected into the explosive does not influence the detonation shock, hence an outflow boundary condition is applied. As, ω , increases the flow eventually becomes locally sonic and two cases can occur: i) the pressure

induced in the inert is below that immediately behind the detonation shock and the confinement has no influence on the detonation. Here a sonic boundary condition is applied; ii) the pressure induced in the inert is greater than that behind the detonation shock. In this case, ω , increases until the pressure at the inert and explosive reach equilibrium. More details on DSD boundary conditions are explained in section 3.1 and in reference [12].

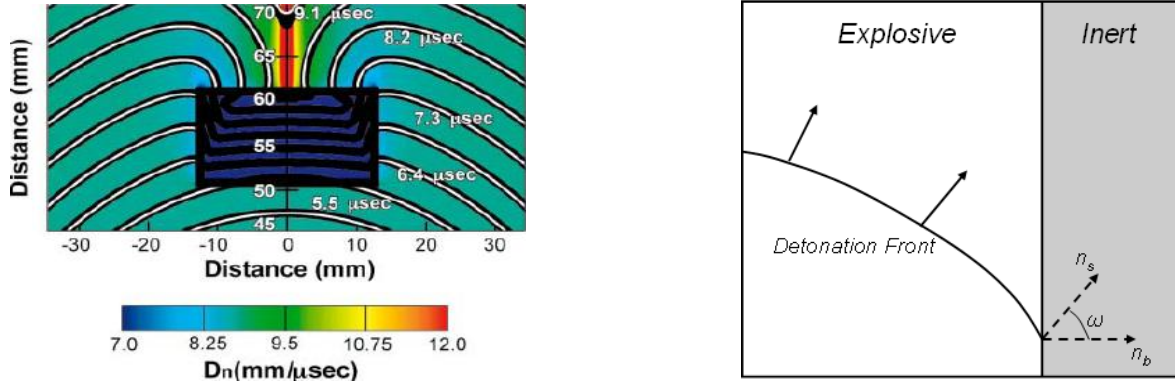


Figure 1.1 (Left): Shows a comparison between a DSD and DNS for a detonation wave passing over a lead disk. The white lines are results form DSD and the black lines from DNS.

Figure 1.2 (right): Schematic representation of an explosive / inert boundary for DSD.

1.2 The Level Set Method

A level set is a curve or surface embedded in a field of one higher dimension in the physical space of interest. For example in two-dimensional space, the level set are curves (level curves) of a function $\phi(x,y,t)$. The level curve is the curve of points where $\phi(x,y,t)$ is some constant value. Level curves are cross sections of a graph $z = \phi(x,y,t)$ taken at a constant value, $\phi = \text{constant}$. In three-dimensional space level sets are surfaces (level surfaces) of a $\phi(x,y,z,t)$ function. Figure 1.3 shows a schematic of an arbitrary surface and the projection of its level curves in a two-dimensional plane.

The level set method (LSM) is a numerical algorithm used for approximating the evolution of propagating curves and surfaces. In applying the level set method for DSD, the surface propagated is the detonation shock front which is influenced by the D_n - κ relation of Eq.(3). The

level curve of interest is the one that evolves from some initial configuration that describes a physical problem, in DSD the level curve of interest is $\phi = 0$ in the full $\phi(x,y,z,t)$ field.

The level set method is also used in DNS models, see [2]. In multi-material simulations level sets can be used to represent material interfaces that propagate at some velocity. As an example, for solving one-dimensional detonation problems the level set is a point. Equation (4) is the level set equation used to propagate the interface. Given $\phi(x, y, z, t) = \text{const}$ then,

$$\phi_t + \left(\frac{dx}{dt} \hat{i} + \frac{dy}{dt} \hat{j} + \frac{dz}{dt} \hat{k} \right) \cdot \vec{\nabla} \phi = 0 \quad (4)$$

Level set theory has many diverse applications in fields such as reconstructive geometry, fluid dynamics, image enhancement, and grid generation. The same algorithms employed in this DSD model are also used in visual recognition physics and ray tracing.

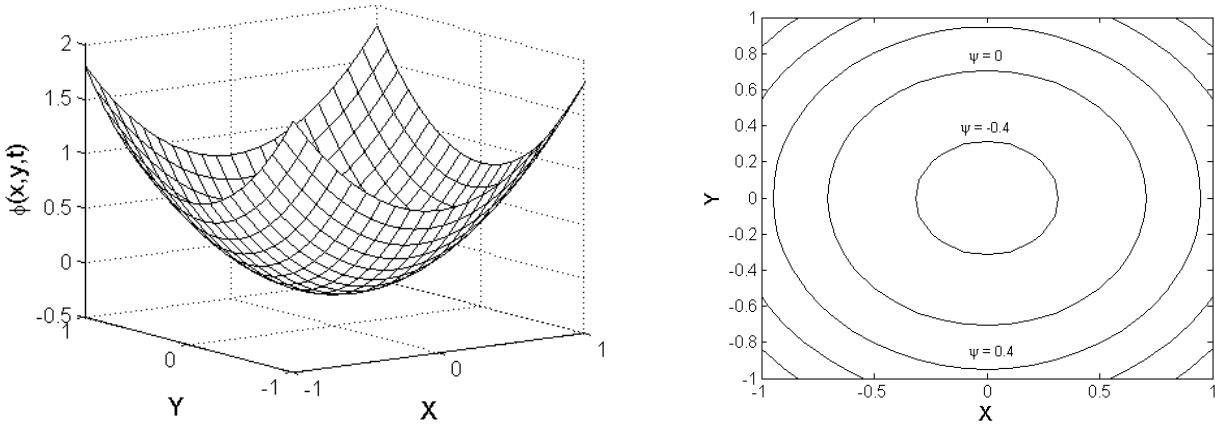


Figure 1.3: Schematic of an arbitrary surfaces and the projection of its level curves in a two-dimensional plane.

Chapter 2: Parallel Computing

Parallel computing is the use of multiple resources to solve computational problems. Resources could be a multi-core computer or a cluster of computers connected by a network. Some of the reasons why parallel computing is used might be to solve large and/or complex problems, save time and money, use resources via a network or internet, etc. This chapter will describe some key concepts and aspects necessary to understand and design a parallel algorithm.

2.1 Parallel Computing Terminology

The section will define some of the terminology associated with parallel computing that will be used throughout this, and subsequent chapters of the thesis.

- Serial / Sequential Program: a program that runs on one processor only.
- Parallel Program: a program that runs on multiple processors.
- Task: refers to a discrete operation of computational work.
- Thread: a series of instruction executed consecutively. This term is mostly used when describing OpenMP parallel implementation.
- Communications: is the means by which parallel tasks are communicated between processors. This can be a network, bus, or some interface like MPI or OpenMP.
- Parallel Overhead: is the time spent on communications. In other words the time spent coordinating parallel tasks. More details of parallel overhead are described in section 2.4 of this chapter.
- Synchronization: is a logical point in a parallel program that all tasks must reach before continuing. It is analogous to a check point in a race, where all participants must stop and rest before continuing with the rest of the race. Synchronization points are associated with increasing parallel overhead, since it involves waiting for other processors to reach the same point.
- Scalability: a parallel program is said to be scalable if when adding more processors the programs run time decreases.
- Granularity: Is the ratio between computational work and communications. More details on granularity are described in section 2.4 of this chapter.

- Multi-core: is a processing system composed of multiple independent processors / cores placed on a single chip.
- Computer Cluster: is a group of linked computers, working synchronically through the use of a network.
- Partitioning / Domain Decomposition: is the dividing or decomposition of a large problem into smaller discrete sub-problems. Most commonly this terminology refers to dividing a large computational domain into smaller sub-domains.

2.2 Memory Architectures

The memory architectures explained in this section are pertinent to multiprocessors, or multiple CPU's with a shared memory. How the memory is distributed can be divided in two main groups: Shared and Distributed Memory.

2.2.1 Shared Memory

In shared memory architecture, all processors have direct access to all memory inside a machine. This means that if some memory address is changed by one processor then the rest is affected by the change. Figure 2.1 shows a schematic representation of a shared memory machine. In these types of machines data sharing is fast because of the easy access of CPUs to memory. They are not very scalable, since adding more processors increases traffic associated with memory management.

2.2.2 Distributed Memory

Unlike shared memory machines, in distributed memory machines each processor has its own local memory. Accessing memory from one processor to another is done by means of a network. In these types of machines, memory is very scalable, and as more processors are added the memory size increases correspondently. The main difficulty with these systems is the need to map data structures associated with global memory to this type of memory system. Figure 2.2 shows a schematic representation of a distributed memory machine.

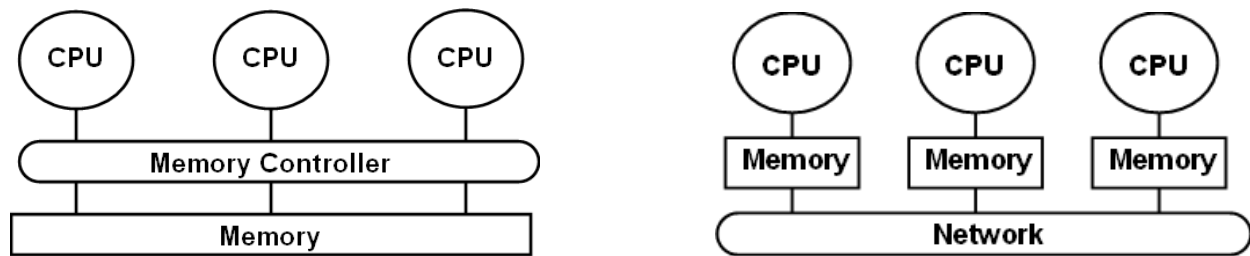


Figure 2.1 (Left): Schematic representation of a shared memory computer architecture.

Figure 2.2 (right): Schematic representation of distributed memory computer architecture.

2.3 Parallel Memory Models

There are a wide variety of parallel models, for example: Shared Memory, Message Passing, Treaded, Hybrid, etc. Which model to choose depends on the problem to be solved, available resources and computer architecture. Here we will only be concerned with the Message Passing and Thread models.

2.3.1 Message Passing Model

This model uses a distributed memory architecture and consists in exchanging information through communications between processors by sending and receiving messages. This exchange has to be coordinated in such a way that if a send is posted it must have a matching receive. Normally each task, or operation, has its own local memory. Figure 2.3 shows a two-dimensional domain of size n_x by n_y . Here the domain is partitioned into two tasks that operate on separate sets of nodes (shown as colored blocks).

The standard communication interface library is MPI, which stands for Message Passing Interface. It consists in a set of library subroutines that are imbedded/ hard coded into source code by the user. MPI can be used in almost any distributed memory parallel programming model, and is the industry standard for writing message passing programs.

2.3.2 Threaded Model

This model is designed to run using a shared memory architecture. As previously explained in this architecture, all processor have direct access to all the memory in a machine. In this model

a master thread executes the sequential portions of an algorithm; then on parts of the code where parallelism is needed, it spawns sub-threads and decommisions the threads when parallelism is not needed. For example if only a do loop needs to be run in parallel, when the master thread reaches this loop is creates sub-threads that work on certain portions of the loop; after each thread completes it's workload, all information is gathered and stored in global memory. This type of parallelization is known as dynamic or incremental, since parallelism can be executed on selected parts or blocks of a sequential algorithm. Figure 2.4 shows an example of a threaded model. Here the red represents the master thread and the green are the sub-threads. The most common library used for threaded models is OpenMP which has C++ and Fortran bindings.

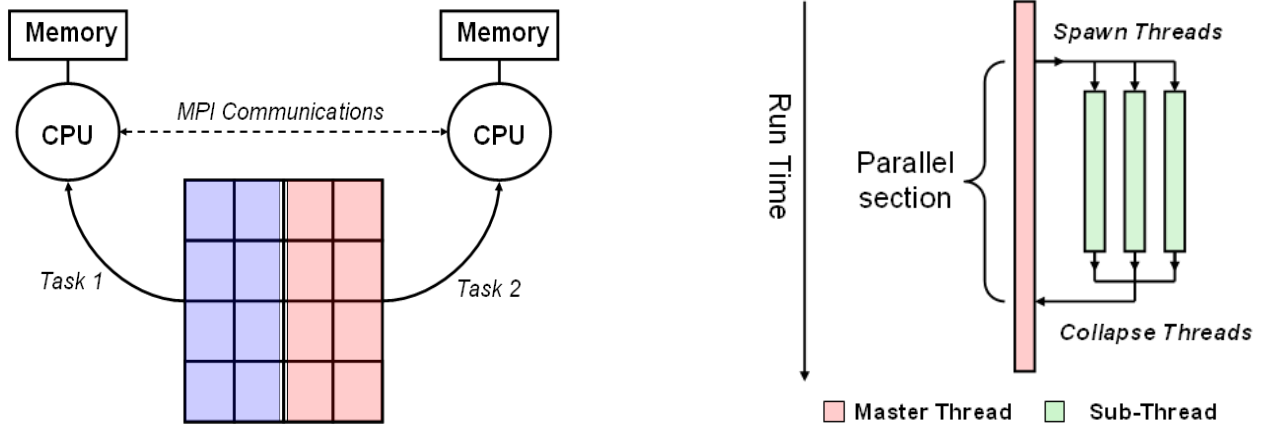


Figure 2.3 (Left): shows a two-dimensional domain of size nx nodes in the horizontal direction by ny nodes in the vertical direction. The domain is partitioned into two tasks that operate on separate sets of nodes (shown as blue and red blocks).

Figure 2.4 (right): shows an example of a threaded model. The red represents the master thread and the green are the sub-threads.

2.4 Performance Analysis

When analyzing the performance of a parallel code there are many factors that need to be considered. For simplicity, this thesis divides them into three groups [9][11]:

□ Computations that can be made to run in parallel

The more sections of a sequential code that could be made to run in parallel result in greater gains from parallelism. For example, if it is determined that fifty percent of a sequential code can be made parallel one would expect the parallel code to run faster. But how much faster will it run? It is important before deciding to write a parallel code to determine if it is feasible to do so.

□ Overhead associated with parallel communication operations

Complexity of a sequential code also plays a significant factor in determining the feasibility of a parallel design. If the parallel overhead required for a parallel design is greater than the computational work that each processor will need to execute, then the gains in parallelism are lost in communicating data; this is why granularity is an important factor to consider. Processor idle time is also significant for parallel overhead, if the computational work to communications ratio is low then some processors will have less work to do than others, meaning that they might need to wait and do nothing until other processors finish their tasks. One also needs to consider the size of problems that need to be run, and the appropriate number of processors to use in a parallel design. For example if the problem size is small and the number of processors is large then processors resources are under utilized and parallel overhead will be significant. On the other hand if the problem size is large then work to communications ratio is large and the problem would be more scalable.

□ Available resources and architectures

The performance of a parallel program is greatly affected by the type of resources to be used. Memory architecture, number of available processors, properties of the processors, available cache and main memory, bus and network speed, and other factor influence the performance of a parallel program.

The main goal of a parallel design is that it runs faster than a sequential program. How fast will it run theoretically and in practice, and how efficiently computational resources are used, are

important variables for benchmarking the performance of a parallel implementation. The following sections describe common analysis parameters that determine the performance of a parallel program.

2.4.1 Speedup

Speedup is the ratio between the time it takes to run a sequential code (serial code), by the time it takes to execute the same code in parallel, equation (5).

$$\text{Speedup} = \frac{\text{Execution time of sequential algorithm}}{\text{Execution time of parallel algorithm}} \quad (5)$$

Assuming that at least two processors are needed to run a parallel program, speedup can be divided into three categories:

- Superlinear Speedup: Speedup that is greater than the number of processors, p . Theoretically it is not achievable, but in practice it can be obtained. Some of the causes for superlinear speedup are [7]:
 - Additional memory in the parallel systems. Distributing a problem among many processors increases the effective total size of cache.
 - Sequential program is not optimal
 - Computer architecture
- Linear Speedup: Speedup is equal to p . This type of speedup is theoretically the maximum that can be achieved.
- Sublinear Speedup: Speedup that is less than p . This is normally the case for most parallel programs.

Figure 2.5 shows a speedup versus processor plot that displays typical behavior observed in practice for a fixed problem size. Typically as more processors are used, speedup increases. However, a diminishing returns like behavior is obtained as more processors are added, and eventually there is no gain in speedup as processors are added.

A parallel program's performance is limited by many factors, for example: computer architecture, communications overhead, memory-cpu bus bandwidth, available resources, problem size, problem complexity and load balance. All these and other factors will affect the performance of a parallel program.

2.4.2 Efficiency

Efficiency is a measure of how well processors are utilized. It is defined as the speedup divided by the number of processors used for a parallel run

$$Efficiency = \frac{Speedup}{Number\ of\ processors\ used} \quad (6)$$

If the efficiency is equal to one the program is exhibiting linear speedup, if efficiency is less than $1/p$, then the program is slowing down. Normally efficiency decreases as more processors are used. Typical efficiency behavior for a parallel program is shown on figure 2.6

2.4.3 Amdahl's Law

Provides an upper bound on the maximum obtainable speedup that can be achieved by estimating the fraction of a program that can be written in parallel. Amdahl's Law is based on the assumption that the problem size is fixed and it ignores the communication overhead introduced from parallelism. In equation (7) Sp is the ideal speedup, and f is the fraction of computational operations that must be performed sequentially, where $0 \leq f \leq 1$. Speedup usually increases as the problem size increases; this is called the Amdahl Effect. This will be the case if the complexity of communications increases at a slower rate than the complexity of computational work that needs to be executed.

$$Sp = \frac{1}{f + (1 - f)/P} \quad (7)$$

Let us assume that one-hundred percent of a given program can be made parallel. Then from Eq.(7), by adding more processors the speedup should increase linearly. In the limit as the processor number tends to infinity speedup tends to $1/f$, but in reality this is not what happens for reasons explained in section 2.4.

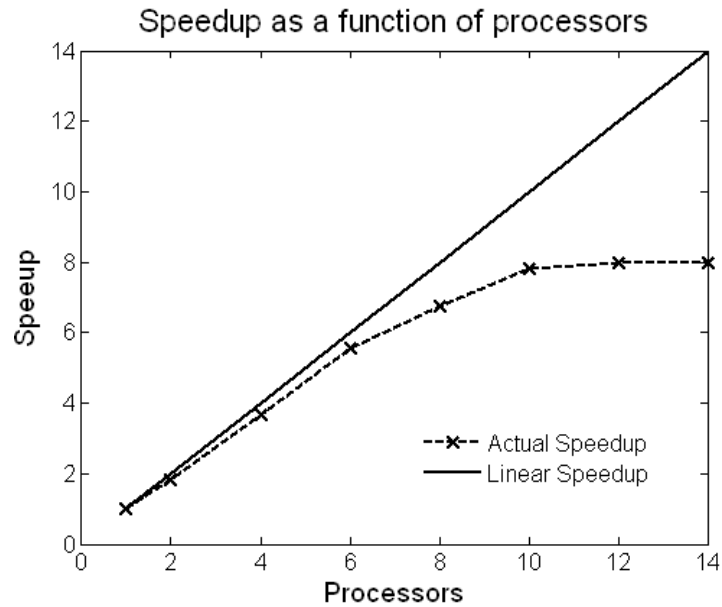


Figure 2.5: Speedup versus processor plot showing typical behavior for a fixed problem size. Typically as more processors are used, speedup increases but then starts to exhibit diminishing returns like behavior, and eventually there is no gain in speedup as processors are added.

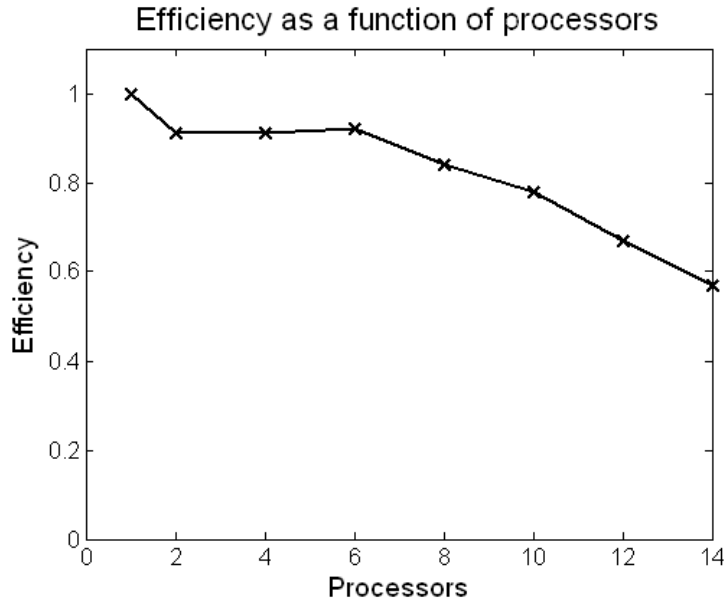


Figure 2.6: Efficiency versus processors plot for a fixed problem size. Normally efficiency decreases as more processors are used.

Chapter 3: Mathematical Model

This chapter will describe how to apply the level set method to solve DSD problems. It will define the level set PDE mentioned in Chapter 1, and then define the modified form of this PDE used in the Full-LS-DSD2D algorithm. Important aspects of the model will also be explained, such as DSD boundary condition implementation, extrapolation for the DSD boundary conditions and the internal boundary node sorting scheme.

Equation (8) is the level set equation used to solve DSD problems. As mentioned before the location of the shock is the zero level surface. Combining Eq.(3) and Eq.(8) the level set equation can be written as equation (9)

$$\frac{\partial \phi}{\partial t} + D_n(\kappa) |\vec{\nabla} \phi| = 0 \quad (8)$$

$$\frac{\partial \phi}{\partial t} - D_{CJ} |\vec{\nabla} \phi| - \alpha(\kappa) |\vec{\nabla} \phi| = 0 \quad (9)$$

Normally to numerically solve the level set equation (advance in time) operator splitting is used. The reason for the splitting is because the $\phi_t + D_{CJ} |\vec{\nabla} \phi|$ term is hyperbolic and the $\phi_t + \alpha(\kappa) |\vec{\nabla} \phi|$ term is parabolic. These operators are solved using different time steps.

Initializing a level set field is done through a distance function. This consists in setting the level curves equal to the signed minimum distance from the zero level curve. In this DSD model two level set fields are used:

- $\phi(x,y,t)$ is the field that represents the detonation shock front. The zero level set curve is the shock / explosive interface. To distinguish between a detonated and an undetonated explosive the following convention is used:
 - $\phi = 0$, is the shock front
 - $\phi > 0$, are undetonated points
 - $\phi < 0$, are detonated points

- $\psi(x,y)$ is the field that represent the detonation shock confinement. The zero level set curve is the inert / explosive interface. To distinguish between inert and explosive materials the following convention is used:
 - $\psi = 0$, is the inert / explosive interface
 - $\psi > 0$, are inert points
 - $\psi < 0$, are explosive points

Figure 3.1 shows a schematic that represents how the level set fields are used to represent boundary interfaces. Figure 3.2 shows the initialization of the $\psi(x,y)$ level set field with the respective level curves indicating the minimum signed distance from the inert / explosive interface, $\psi(x,y) = 0$.

To accurately detect the zero level curve it is important to reduce large gradients, plateaus and numerical noise as much as possible from the level set field when propagating the shock front. This noise will distort the location of the zero level curve resulting in inaccurate propagation of the detonation shock front. To reduce this noise the level set field is re-initialized to smooth it out. Equation (10) is the re-initialization equation, this is a of Hamilton-Jacobi type equation that behaves similar to parabolic PDEs. This equation drives the gradient of the level set function to one.

$$\varphi_t + S(\varphi) \left\{ |\vec{\nabla} \varphi| - 1 \right\} = 0 \quad (10)$$

Curvature, κ , is defined in terms of the level set function $\varphi(x,y,t)$ by using the expression $\kappa = \vec{\nabla} \cdot \vec{n}_b$. Here \vec{n}_b is the normal to the shock surface where $\vec{n}_b = \nabla \varphi / |\nabla \varphi|$. Expanding the above expression for curvature in two-dimensional Cartesian coordinates, equation (11) is obtained.

$$\kappa = \frac{\varphi_{xx}\varphi_y^2 - 2\varphi_{xy}\varphi_x\varphi_y + \varphi_{yy}\varphi_x^2}{(\varphi_x^2 + \varphi_y^2)^{3/2}} \quad (11)$$

The algorithm in Full-LS-DSD2D solves a modified level set PDE that joins Eq.(9) and Eq.(10); this has the same operator splitting described earlier, but uses a single time step that reflects the stability condition for the mixed problem, making the algorithm fully explicit. All redistancing is added to the hyperbolic part. Equation (12) is the new modified PDE.

$$\frac{\partial \phi}{\partial t} + Dn|\vec{\nabla} \phi| = \frac{S(\phi)}{\varepsilon} [1 - |\vec{\nabla} \phi|] \quad (12)$$

Here $S(\phi)$ is the signed function of ϕ , and ε is a small parameter. The zero level curve of the $\phi(x,y,t)$ field is the location of the detonation shock front. The $\phi(x,y,t)$ field is initialized as a signed distance function with $|\vec{\nabla} \phi| = 1$ set initially.

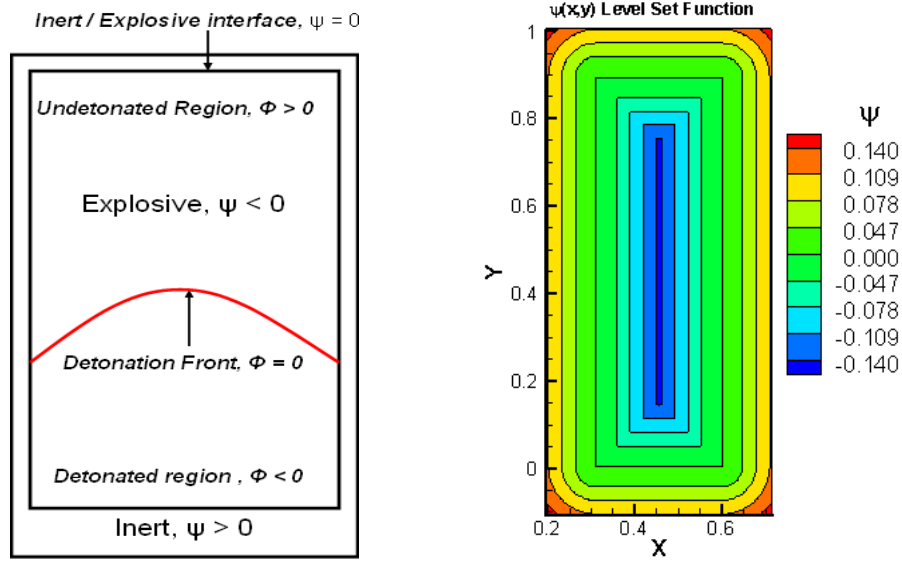


Figure 3.1 (Left): Shows the level set conventions used to define inert and explosive materials, inert / explosive interfaces, detonated and undetonated explosive points and detonation front location.

Figure 3.2 (right): initialization of the $\psi(x,y)$ level set field with the respective level curves indicating the minimum signed distance from the inert / explosive interface, $\psi(x,y) = 0$. The shape corresponds to the vertical rate stick type problem.

3.1 DSD Boundary Conditions

The solution of Eq.(12) is subject to DSD boundary conditions. As depicted in Fig.(1.2), the boundary condition depends on the DSD edge angle, ω , that is formed between the normal n_s and n_b at the explosive / inert interface. In terms of the level set functions, these values are defined as follows:

$$\hat{n}_b = \frac{\vec{\nabla} \psi}{|\vec{\nabla} \psi|} \quad (13)$$

$$\hat{n}_s = \frac{\vec{\nabla} \phi}{|\vec{\nabla} \phi|} \quad (14)$$

$$\hat{n}_s \cdot \hat{n}_b = \cos(\omega) \quad (15)$$

DSD boundary conditions depend on the type of flow as seen from an observer moving with the point of intersection of the shock and the inert edge. To apply the appropriate boundary conditions a local sonic parameter, f is used.

$$f = C_2 - U_n^2 - D_n^2 \cot^2(\omega) \quad (15.b)$$

Where C is the speed of sound and U_n is the explosive particle velocity in the shock-normal direction. If $f < 0$ the flow is supersonic and no boundary conditions are applied regardless of the degree of confinement that the inert provides to the explosive. As the DSD edge angle increases the sonic parameter will eventually reach $f = 0$ (flow is sonic). This angle is referred to ω_s , it is the smallest angle beyond which the explosive can feel the presence of the inert interface. This angle is a constant given by the explosive equation of state. For further details on the DSD boundary condition refer to [12].

Combining Eq.(13-14) and using the fact that the level set function ψ represents the explosive-inert interface meaning it is stationary, and is initialized to $|\vec{\nabla} \psi| = 1$, the boundary condition can be written as equation (16).

$$\frac{\partial \psi}{\partial x} \frac{\partial \phi}{\partial x} + \frac{\partial \psi}{\partial y} \frac{\partial \phi}{\partial y} = \cos(\omega_c) \sqrt{\left(\frac{\partial \phi}{\partial x}\right)^2 + \left(\frac{\partial \phi}{\partial y}\right)^2} \quad (16)$$

Here ω_c corresponds to the angle when the pressures at the inert and reaction zone reach equilibrium; it is found using shock polar analysis and depends on the inert/explosive pair used. The partial derivatives of Eq.(16) are found using the central difference approximations shown in equations (17-18). In these equations the Greek letter μ is used to denote either ψ or ϕ .

$$\frac{\partial \mu}{\partial x} = \frac{\mu_{i+1,j} - \mu_{i-1,j}}{2dx} \quad (17)$$

$$\frac{\partial \mu}{\partial y} = \frac{\mu_{i,j+1} - \mu_{i,j-1}}{2dy} \quad (18)$$

To apply boundary condition Full-LS-DSD2D uses ghost nodes extending into the inert region out to a depth $dx\sqrt{2}$ from the explosive / inert boundary. These ghost nodes are called Internal Boundary nodes or IB nodes. Ghost nodes are used to store extrapolated HE (explosive nodes) values in order to apply the DSD boundary conditions surrounding the explosive / inert interface. The DSD boundary condition is formally applied at the explosive node nearest the explosive/inert interface. The application node corresponds to the central difference part of the central-difference stencil used in the discrete representation of the DSD boundary condition. The extrapolation step for the DSD boundary conditions consists in extrapolating the DSD edge angle, ω , from the explosive interior to the explosive / inert boundary. This condition involves setting to zero the gradient of the surface normal component of the level set function, equation (19).

$$\hat{n}_b \cdot \vec{\nabla} \omega = 0, \quad (19)$$

this is approximated by setting $\frac{\partial^2 \varphi}{\partial \eta^2} = 0$ at the boundary, where η is the extrapolation direction coordinate. The details on how the Full-LS-DSD2D algorithm solves the roots of Eq.(19) can be found in reference [5].

3.2 Node Sorting

Ghost nodes are selected from the four first-nearest neighbors (a distance dx away), and the four second-nearest neighbors (a distance $dx\sqrt{2}$ away). The reason why second-nearest neighbors are needed is because to evaluate curvature in the Full-LS-DSD2D algorithm a nine point stencil is used. In this algorithm all the first-nearest neighbors are resolved first, and then the second-nearest neighbors. Extrapolation of ghost nodes in the Full-LS-DSD2D algorithm depends on whether a node has first or second nearest neighbors. For first-nearest neighbors,

extrapolation is based on mesh directions (y-direction or x-direction) and extrapolation along a 45 degree line ($x = y$). At leading order, the extrapolation along x , y or $x=y$ is equivalent to extrapolation along the direction of the interface normal. This follows from the assumption that $\phi(x,y,t)$ is locally planar near the explosive/inert interface. For second-nearest neighbors extrapolation is along the $x = y$ direction. Figure 3.3 shows a schematic of the two extrapolation stencils used in Full-LS-DSD2D, along the y direction and $x=y$ (45 degrees) direction. Here the red nodes represent the ghost nodes and the blue line is the explosive / inert interface.

First-nearest neighbors IB ghost nodes are further sorted by the connectivity of their defining interior explosive points. The interior explosive point is the central point of the central-difference used to apply DSD boundary conditions. IB ghost nodes can fall into four categories, priority 1, 2, 3 or 4. These priorities represent how many interior points an IB node is connected to and by how many other IB ghost nodes the interior point that defines the IB ghost node is itself connected to. Priority 1 nodes only involve one adjacent IB ghost node in the stencil of their defining interior explosive point. The extrapolation direction for such IB ghost nodes is picked as the coordinate direction corresponding to the direction between the ghost node and the defining interior point. Priority 2 nodes involve two adjacent interior points. For these the extrapolation is performed along the $x = y$ line, whose direction is defined by the IB node and the nearest neighbor interior nodes. If first nearest neighbors do not satisfy any of the above priority condition then these are assigned priority values of 3 and 4, where an increasing priority number denotes that the defining interior HE point has an increasing number of IB points that have IB points as nearest neighbors. Figure 3.4 shows internal IB nodes sorted by priority order, nodes inside the circle are explosive points.

By assigning the ghost node values for the angle boundary condition to IB ghost nodes in priority order, the interdependency is broken and only one ghost node is required to be set in applying the angle boundary condition to each near boundary interior point. For further details refer to [5].

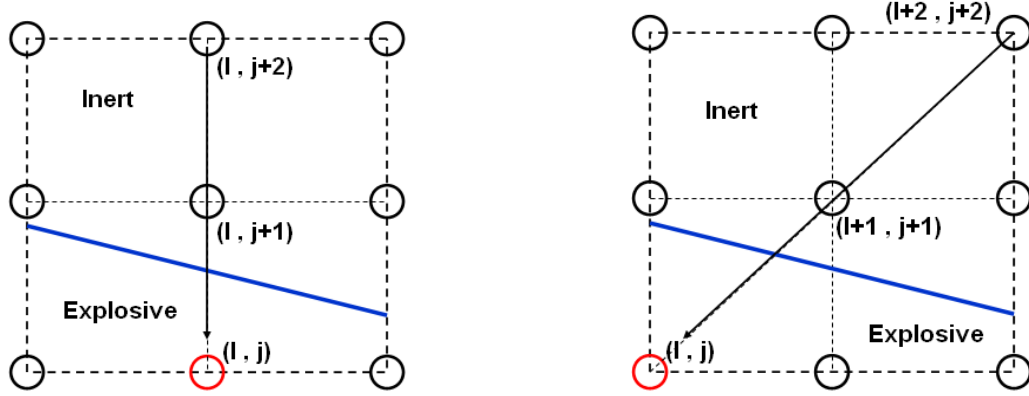


Figure 3.3: shows the two extrapolation stencils used in Full-LS-DSD2D: along the y direction (left) and $x=y$, 45° direction (right). The red nodes represent the ghost nodes, and the blue line is the explosive / inert interface. The arrow indicates the interpolation direction.

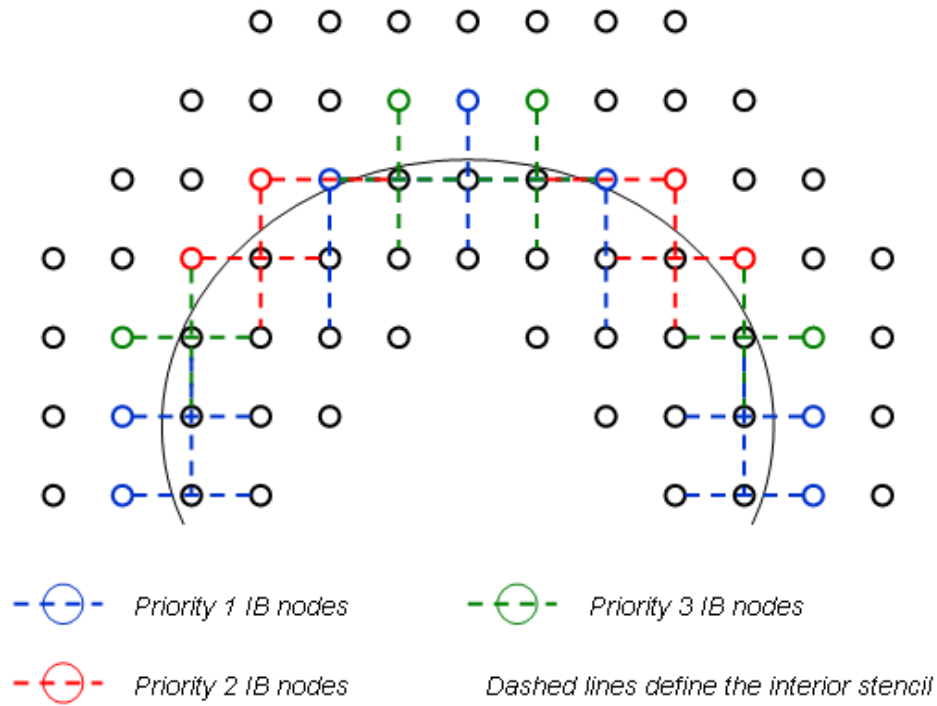


Figure 3.4: Shows Internal Boundary sorted by priority order. The big circle is the inert/explosive interface; nodes inside this interface are explosive points. The blue nodes are priority 1, red are priority 2 and green are priority 3 IB nodes. The Dashed lines define the interior stencil.

Chapter 4: Numerical Model

The level set PDE of Eq.(9) is solved on a uniform two-dimensional Cartesian grid where the vertical and horizontal grid spacing are equal, $dx = dy$. Time integration is performed via a second order Runge Kutta solver. As mentioned in Chapter 2 the Full-LS-DSD2D model solves the modified level set PDE via an explicit algorithm. This causes the lower bound on the time step needed to maintain numerical stability to be more restrictive. The time step in Full-LS-DSD2D is found by equation (20) for the case $D_n = D_{CJ} - \alpha\kappa$ where α is a constant. The time step is kept constant throughout a simulation.

$$dt = \frac{CFL}{\frac{\sqrt{2}(D_{CJ} + 1/\varepsilon)}{dx} + \frac{2\alpha}{dx^2}} \quad (20)$$

The smallness of epsilon relative to D_{CJ} determines the closeness of the approach of $|\vec{\nabla}\phi|$ to 1 and the smallness of the hyperbolic part of the Runge Kutta integration step. To apply boundary conditions, ghost nodes are used surrounding the HE region in layers that extend out to a depth $dx\sqrt{2}$ from the HE boundary. There are two rows of ghost nodes on either side of the computational domain as seen in figure 4.1. To solve for curvature at the interior high explosive region a nine point stencil is used along with an up-winding routine to determine the appropriate finite difference approximation for the hyperbolic part of the operator. Figure 4.2 shows the curvature stencil plus the four extra nodes needed for up-winding.

Full-LS-DSD2D uses a simple linear $D_n - \kappa$ law given by equation (21). Here α is a constant specified by the user. In most cases $D_n - \kappa$ is a nonlinear relationship that is usually obtained from experimental tests or computational simulations.

$$D_n = D_{CJ} - \alpha\kappa \quad (21)$$

The code first defines the explosive and inert interfaces by setting the distance function for the $\psi(x,y)$ level set function. This part is provided by the user, where the interface is described mathematically. Then it scans through all nodes in the computational domain and sorts them as

internal boundary or explosive points; and identifies first and second nearest neighbors. Next internal boundary points are further divided in priority order as explained in section 3.2. The node sorting algorithm is performed at the beginning of the program and is only executed once. After the above is complete the initial detonation shock interface is initialized by setting the $\phi(x,y,t=0)$ level set function.

The next section of code is the main solver loop. This solves the internal boundary points, meaning it will check for sonic and subsonic angles and, if needed, solve / update the internal boundary nodes according to the DSD boundary conditions. These steps are repeated twice in one iteration loop, since a two step Runge Kutta solver is used. After each complete Runge Kutta step is performed, the code computes new D_n values using Eq.(21), re-computes level set gradients, records burn times, and other output parameters. To determine if the detonation shock has left the inert confinement (reached the end of the confinement) the maximum value of the level set function $\phi(x,y,t)$ is found. If this value is less than zero then the solver loop stops and exits ending the simulation. Figure 4.3 shows a representative flow chart of the DSD serial code.

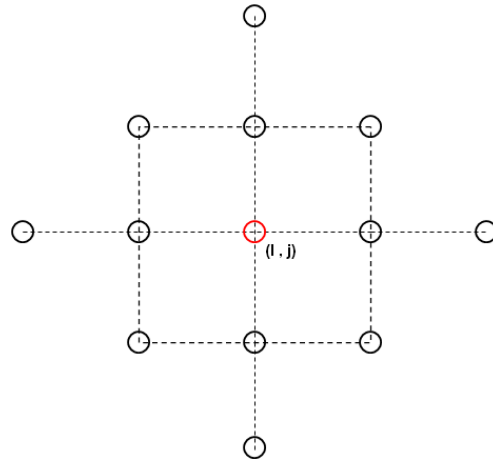
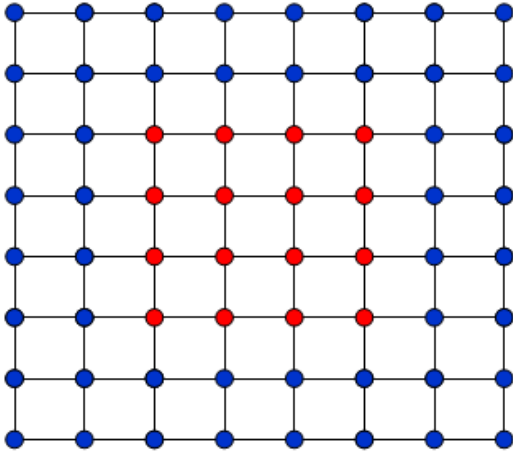


Figure 4.1 (left): Ghost nodes in Full-LS-DSD2D. Two rows of ghost nodes are used on either side of the computational domain. The red dots are computational domain nodes, and the blue dots are ghost nodes.

Figure 4.2 (right): shows the stencil used to solve for shock curvature. This is a nine point stencil plus four nodes used to determining up-winding direction.

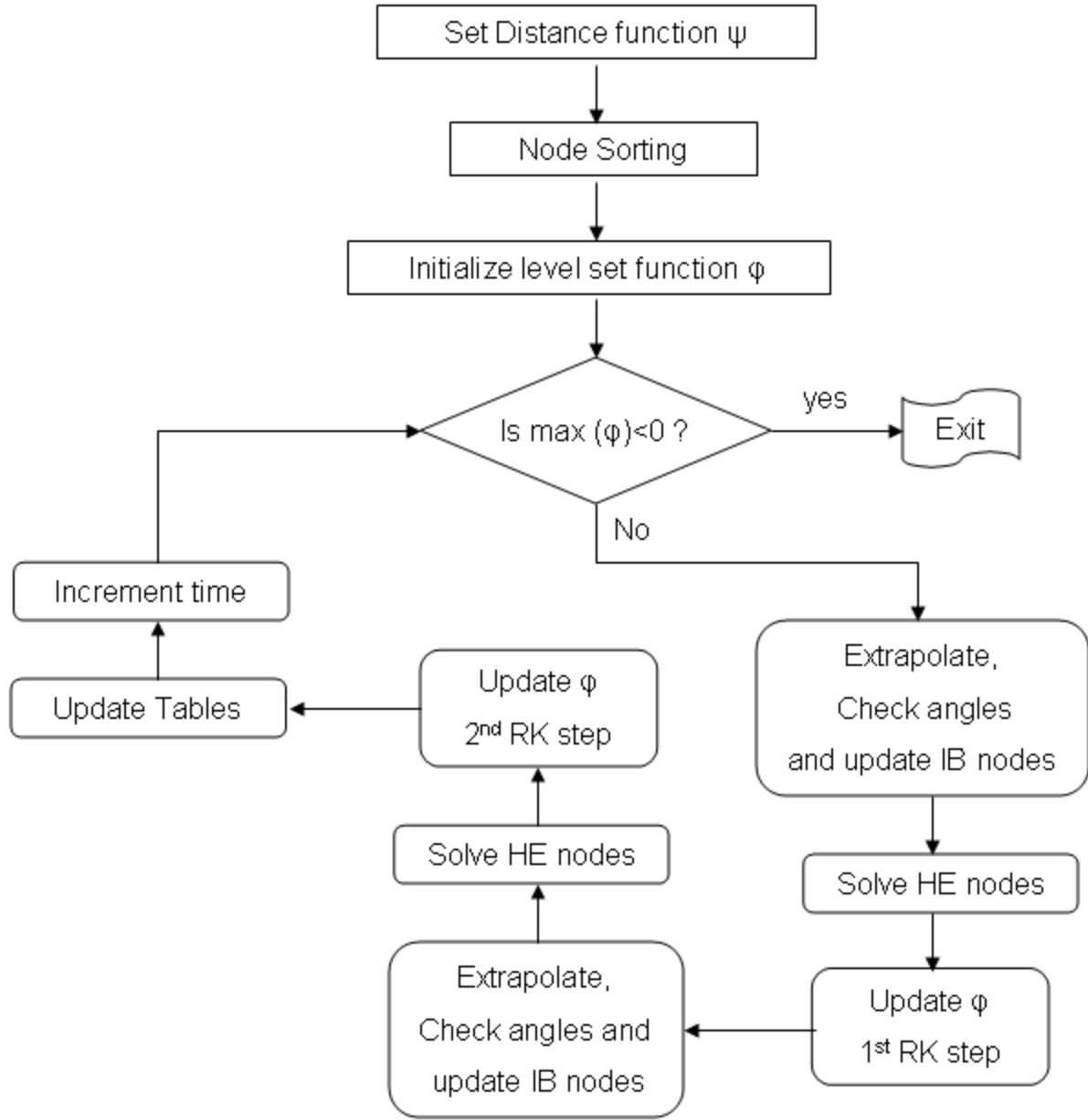


Figure 4.3: Flow chart that sequentially summarizes the step executed by Full-LS-DSD2D during a DSD simulation.

Chapter 5: Parallel Model

The first step before designing a parallel model is to determine if the problem can be made parallel, and if the answer is ‘yes’ then identify which parts of the code would benefit the most from parallelization. Given the characteristics of the serial code described in Chapter 4, the code can be made parallel. To answer the second question, the serial code was run through a profiler. Profilers are programs that show how functions in a code perform: how many times they are called and the time spent executing these functions. Profiling is a very useful tool in designing and optimizing a parallel code, since it will show what section of code would benefit most from parallelism. The Full-LS-DSD2D code was profiled using *Shark* with a 2.4Ghz Intel Dual Core, 3MB L2 cache with a Bus Speed of 800Mhz Apple MacBook Pro laptop. The test problem used was a vertical rate stick problem; table (1) shows functional timing at four problem sizes.

Table 1: Serial Code Profile (percent, % shown)

Size	100x300	200x600	300x900	400x1200
Sorting	0.4	0.2	0.3	0.4
HE Solve	79.8	78.9	75.1	74.4
IB Routines	3.0	2.4	2.0	1.6
Update Tables	3.6	3.8	4.5	4.7
Updating	13.2	14.7	18.1	18.9

There is no significant gain in paralyzing the sorting subroutines since it only counts for around 0.5% of work. The most gain from parallelization is obtained in HE solve and Updating; together these account for 99% of the work done. As the problem size increases, less time is spent on the Internal Boundary routines; this is expected since the work for IB nodes scales as $O(n)$ while for the HE nodes it scales as $O(n^2)$.

Most gains in speedup and optimization would come from the subroutines that work on high explosive nodes. They account for about 99% of the total work done in the serial code. The internal boundary routines accounts for about 1% of the work done, and this value decreases with an increasing problem size.

The parallel design chosen for IPC-DSD2D is a message passing model and the communication interface used was MPI. Let us now divide the model into six groups and describe each one separately and then explain how they interact with each other (design model).

- ☐ Node Sorting Model
- ☐ Domain Decomposition Model
- ☐ Communications Model
- ☐ Internal Boundary Model
- ☐ High Explosive Model
- ☐ I/O Model
- ☐ Design Model

5.1 Node Sorting Model

Node sorting involves all the routines that classify nodes, sort nodes as HE or IB and check what nodes are active or inactive. From table.(1) the percent of time spent on node sorting is less than 0.5%, so making these routines parallel will not effect the overall speedup of the code by any significant amount. There is also a lot of interdependence in these routines making it unfeasible to attempt to paralyze them. Also being that it takes such a small percentage to complete, the communication overhead associated with making this section parallel would be greater than having each processor sort all the nodes in the computational domain. In this parallel model, node sorting will run on all processors as it where a serial code. This means that each processor will sort all the nodes in the computational domain.

5.2 Domain Decomposition Model

The parallel model consists in partitioning the domain into smaller discrete sub-domains and then assigning each sub-domain to a processor. Each processor will only have, and work with the information allocated to it. Sharing of data between processors is archived via the use of ghost nodes that are passed along through point-to-point MPI communications.

The processors are topologically arranged in a two-dimensional rectangular grid of size Xblocks x Yblocks. This arrangement is defined by the user, where Xblocks indicates the number of columns, and Yblocks the number rows. The physical domain is then divided geometrically into rectangular sub-domains that get mapped on to each processor. This

arrangement is illustrated in figure 5.1. Here there is a one-to-one mapping of blocks to processors. In Fig.(5.1) the light blue blocks represent the physical domain, while the light green represent an extra band of blocks that enclose the physical domain. For example a light blue block with a light green block as its top neighbor means that this blue block has no physical neighbor above it, hence it does not need to share information with this block. Each light blue block is assigned a real positive integer that identifies its position in the rectangular arrangement of processors. Each green block is assigned a negative integer.

As explained in section 2.4, it is important to have an efficient load balancing strategy for performance reasons. In this parallel design, each processor roughly does about the same amount of work in terms of operations to be executed. How the active HE and IB nodes are distributed among processors is important. Below are some positive and negative aspects of the domain decomposition model implemented in this code, and how they affect load balancing:

Positive aspects

- ☐ Simpler to implement and understand
- ☐ Requires less communication overhead

Negative aspects

- ☐ Load balancing depends on geometry
- ☐ Might not have equal distribution of active nodes

Figure 5.2 and 5.3 show two example problems with the same number of processor blocks but different geometrical arrangements. The light red are blocks with HE and/or IB nodes. Depending on how many processors are used, and how the domain is split there are some areas that have no HE or IB nodes, in essence processors that receive these geometrical sub domains will do no work. See Fig.(5.2) where there are 4 blocks (light blue) that have no HE or IB nodes. The effects on efficiency are seen through the misuse of resources rather than the overhead associated with processors that do no work. The code treats processors with no active nodes as the light green blocks of Fig.(5.1).

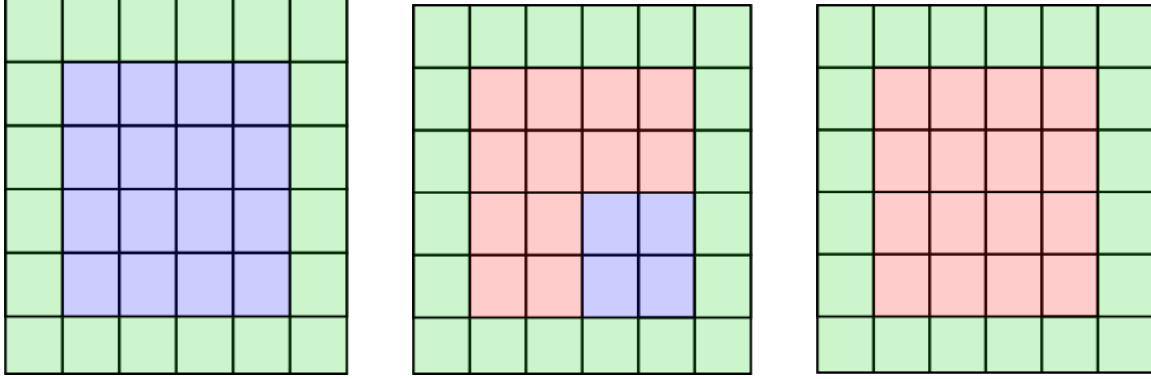


Figure 5.1 (left): Domain decomposition of physical domain. The domain is divided into rectangular sub-domains that map one-to-one with each processor. The light blue blocks represent the physical domain, while the light green represent an extra band of blocks that enclose the physical domain.

Figure 5.2 (middle): Geometrical arrangement with 4 processors that have no HE or IB nodes. The red blocks are processors with HE or IB nodes, the blue block are processors with none.

Figure 5.3 (right): Geometrical arrangement where all processors have HE or IB nodes.

5.3 Communications Model

Communications consist in message passing between processors for sharing information. The interface for communications in this model is MPI. Since each processor only works on its assigned sub-domain, communication between processors are needed. For example to solve HE nodes the stencil needed is a 9 point stencil with an additional 2 points for up-winding. This requires, as in the serial code, two sets of ghost nodes. If an HE node is located at an edge of a sub-domain, then it would need to access nodes from other processors. Figures 5.4 and 5.5 show how ghost nodes are shared between processors. The light red nodes represent the physical domain and the light blue nodes are the ghost nodes for each processor.

The number of MPI sends and receives depend on how the domain is partitioned. Fig.(5.5) shows a partitioned domain that consists of vertical slices; here there is no need to exchange diagonal nodes between processors. This will also be the case if the domain was partitioned with horizontal slices only. Fig.(5.4) shows a domain partitioned in XY blocks; here data is shared vertically, horizontally and diagonally between processors.

Sharing information is necessary each time IB or HE nodes are updated, or when the φ level set function is updated. Communication for this DSD code is complex; the source of the complexity comes from the node sorting routines. As explained in section 3.2, the internal boundary update is in priority order, meaning that to update priority two nodes all priority one nodes had to have been updated first. This implies that information must be shared between processors after each priority update is complete. The IPC-DSD2D code has fourteen calls on the MPI ghost node communication subroutine. All data communication between processors through ghost nodes is asynchronous.

To determine if the solution has reached the end of the domain defined by the ψ level set function each processors needs to know if the φ solution on other processors is less than zero in order to either continue propagating the detonation wave, or stop the simulation. This requires a synchronization point where all processors communicate their respective sub-domain maximum φ value and determine if the simulation exit condition has been met. The procedure to do so is that each processor finds the maximum value of the φ level set field on its sub-domain, and then sends this value to processor 0. Processor 0 determines the largest of the values and broadcasts it to all the other processors. This is done through the use of the MPI_ALLREDUCE function call.

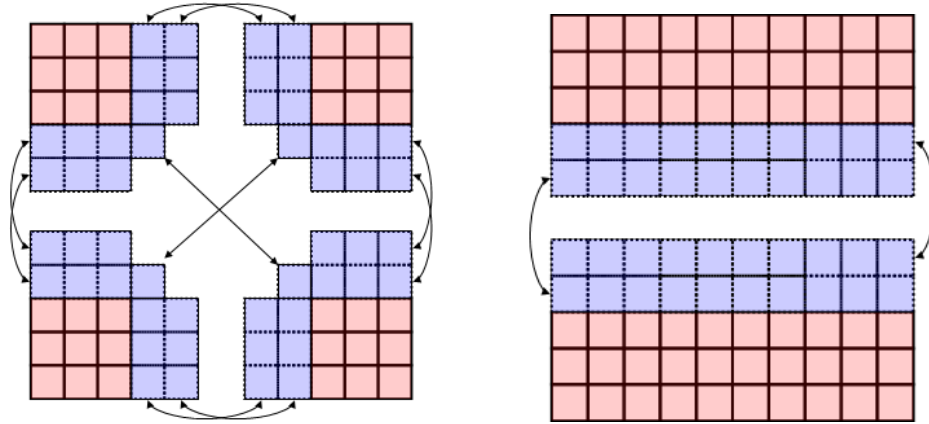


Figure 5.4 (left): Sharing of information through ghost nodes when the domain is divided in XY slices. Blue blocks are ghost nodes and red blocks are computational nodes.

Figure 5.5 (right): Sharing of information through ghost nodes when the domain is divided in Y slices. Same color scheme as Fig.(5.4).

5.4 Internal Boundary Model

Internal Boundary consists in all routines that modify and update internal boundary nodes. As mentioned before in section 5.3, IB nodes need to be updated in priority order, requiring sharing of information between processors after each priority is updated. Each IB related subroutine is now divided in separate loops that correspond to a particular priority, by creating priority sorted lists of internal boundary nodes. Below is a general pseudo-code of the priority splitting of IB loops:

```
1  for i=1 to i = number of priority 1 nodes do
2    Work is done
3  End do
4  Share information with other processors
5  ...
6  ...
7  for i=1 to i = number of priority 5 nodes do
8    Work is done
9  End do
10 Share information with other processors
```

5.5 High Explosive Model

The High Explosive model consists of the HE solver subroutines and all loops that update the HE nodes. These routines are at the core of the parallel design and are the ones that require the least amount of modification to make them run in parallel. The only MPI communication calls needed are to update ghost zones on each sub-domain and share these new values, and a slight modification for the stopping criteria to determine if the shock front has reached the end of the domain (as explained section 5.3).

5.6 I/O Model

It is often the case that a source for bottlenecks in parallel codes is the input and output sections. For simplicity the input file in the parallel DSD code is read by all processors and each processor writes its own output file with information associated to its geometrical sub-domain only.

5.7 Design Model

The design model divides the work (section of code to be executed) depending on the rank of the processor. For example processor 0 refers to the physical processor that was assigned a rank of zero when the code was compiled and the MPI library initialized (call on the MPI_INIT subroutine). In this parallel design one processor (processor 0) is responsible for executing the domain decomposition routines and printing code diagnostics; it also assigns itself a sub-domain and work load. After each processor has run the node sorting routines, processor 0 performs the domain decomposition and sends each processor the following information:

- ☐ Physical dimension of each sub-domain.
- ☐ Relative location of each sub-domain in the rectangular arrangement of processors.
- ☐ Number of nodes in each sub-domain.
- ☐ Number of IB and HE nodes in the sub-domain.
- ☐ List of processors with no active HE nodes.

Processors with no HE or IB nodes are classified as inactive and post no MPI sends or receives. After receiving the information from processor 0, each processor scans each node it was assigned and creates lists that separate the HE from IB nodes. Lists are also made to further divide Internal Boundary nodes in priority order. From now on each processor will work independently on the sub-domain it was assigned and only share information with adjacent processors through ghost node regions.

The time integration solver section of the code is very similar to the serial one; the only differences are the use of synchronization points when finding the maximum value of the ϕ level set function and, as mentioned in Chapter 4 data sharing between processors for updating the RK2 solution sub-steps. Table 2 and figure 5.6 respectively show pseudo code and a flow chart for the parallel design.

Table 2: Pseudo Code for IPC-DSD2D

```
1  Start Program IPC-DSD2D
2  Read input file
3  Initialize psi level set
4  Node sorting subroutines
5  If processor 0 then
6    Partition computational domain (Domain Decomposition)
7    Scan for inactive processors
8    Send domain information
9    Broadcast lists
10 Else
11   Receive domain information
12   Broadcast lists
13 End if
14 Find Neighbors
15 Share information with other processors
16 Calculate time step
17 for solver loop do
18   IB subroutines
19   Solve priority 1 IB nodes
20   Share information with other processors
21   ..
22   Solve priority 5 IB nodes
23   Share information with other processors
24   HE solver
25   1st RK sub-step
26   Update solution
27   Share information with other processors
28   IB subroutines
29   Solve priority 1 IB nodes
30   Share information with other processors
31   ...
32   Solve priority 5 IB nodes
33   Share information with other processors
34   HE solver
35   2nd RK sub-step
36   Update solution
37   Share information with other processors
38   Update tables
39   Find max( $\phi$ ) by an MPI reduction
40   If max( $\phi$ ) < 0.0 stop
41 End do
42 Write output
43 End Program IPC-DSD2D
```

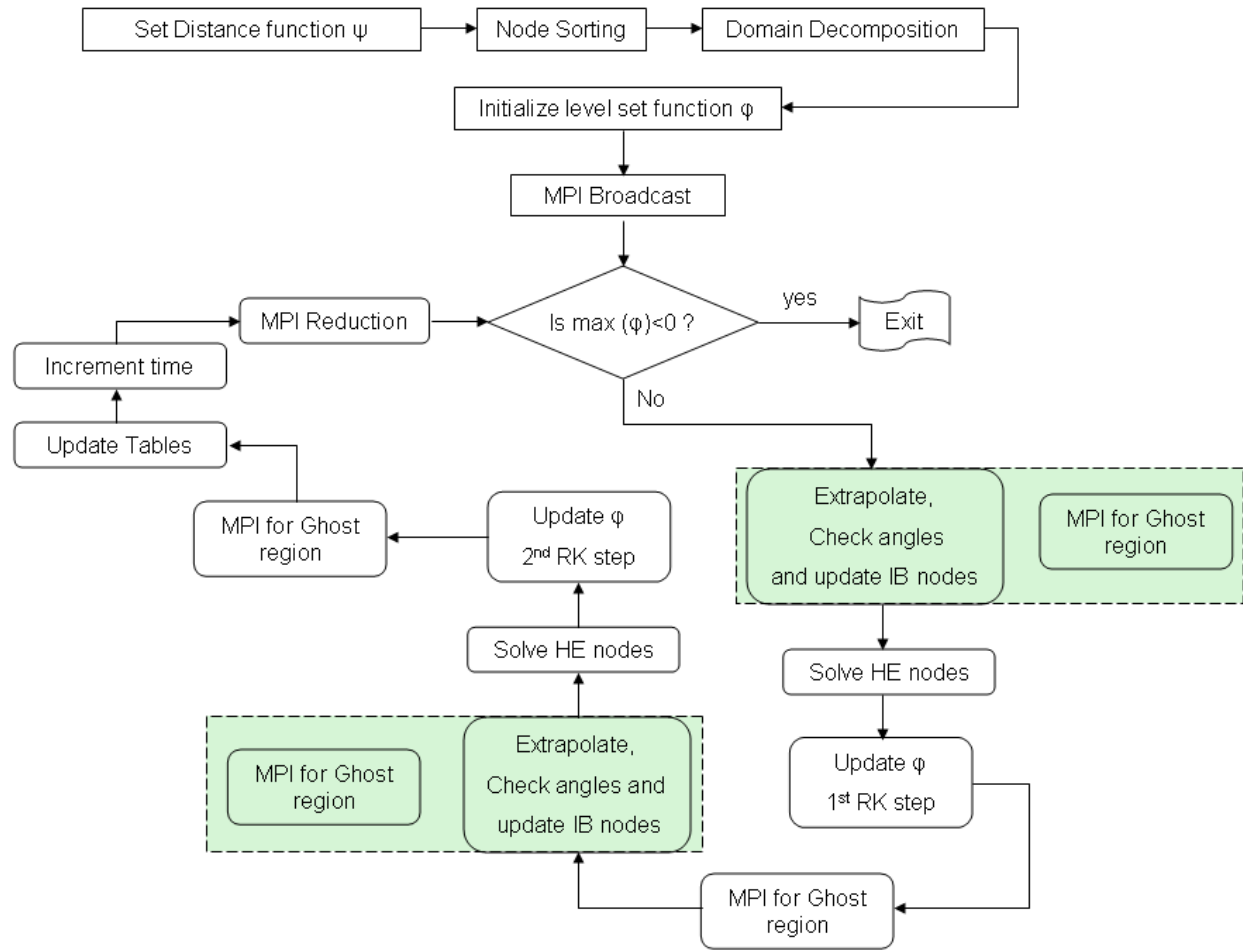


Figure 5.6: Shows a flow chart of the parallel design implementation. Here all processors run the node sorting algorithm and initialize both ψ and ϕ level set functions on the entire domain. After the MPI broadcast step each processor works on the sub-domain that was assigned to it, sharing information between each other through ghost nodes using MPI point-to-point communication (this is referred to as MPI boundary conditions). The green area represents parts of code that have calls to MPI boundary conditions within each blocked (diagram) subroutine.

Chapter 6: Numerical Results

The algorithm used in Full-LS-DSD2D was verified in references [5][6]. Here three test problems: a time dependent rate stick calculation, a cylindrically expanding detonation calculation, and a detonation in an explosive arc problem where used to compare Full-LS-DSD2D with a second order high resolution Maple script that was used to get the exact solution. The results of the numerical comparison showed that Full-LS-DSD2D is of first order convergence (or better) to the exact solution measured by the L1 norm. This was an improvement over narrow-band DSD codes, which show fractional convergence. For further details see reference [6]. The numerical results obtained using the new DSD parallel implementation were identical to the results obtained using the serial program; thus validating IPC-DSD2D.

As mentioned in Chapter 5, some subroutines and aspects of the serial code had to be modified / added to implement the parallel design. One significant improvement was made to the new IPC-DSD2D code that made it roughly two times faster than the original serial code. This applies for problem sizes (memory usage) that do not allow the entire problem to fit in cache. Benchmarking and performance analysis of the parallel code are described in Chapter 7.

The following figures (6.1-6.6) show simulation results using IPC-DSD2D. Three test problems are shown: detonation in a 180° arc, detonation in a square of explosive with a circular hole, and a detonation in an hourglass shaped explosive geometry; using different domain decomposition arrangements. Table 3 shows the common simulation parameters used for these three runs. Parameters such as domain dimension and problem size (nodes used) are given in the caption for each figure. The simulations performed are non-dimensional using unit scaling.

Table 3: Common simulation parameters used in IPC-DSD2D, (angles are in radians)

$D_{CJ} = 1$	$\alpha = 0.1$	$\omega_c = 0.8028500$	$\omega_s = 0.7853982$	$CFL = 0.9$
--------------	----------------	------------------------	------------------------	-------------

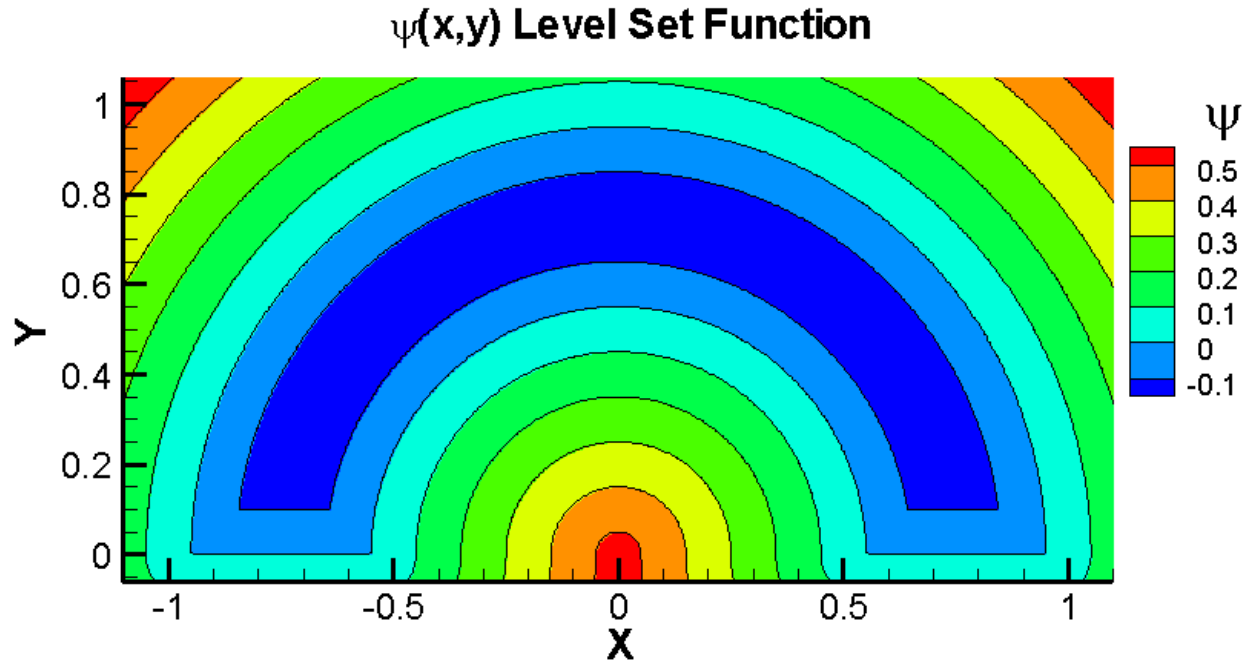


Figure 6.1: Shown is the $\psi(x,y)$ level set function for the detonation wave in a 180° arc problem. The domain size was a 1.05 x 2.1 rectangle, with $n_x = 600$ $n_y = 300$ and $dx = dy = 0.0035$.

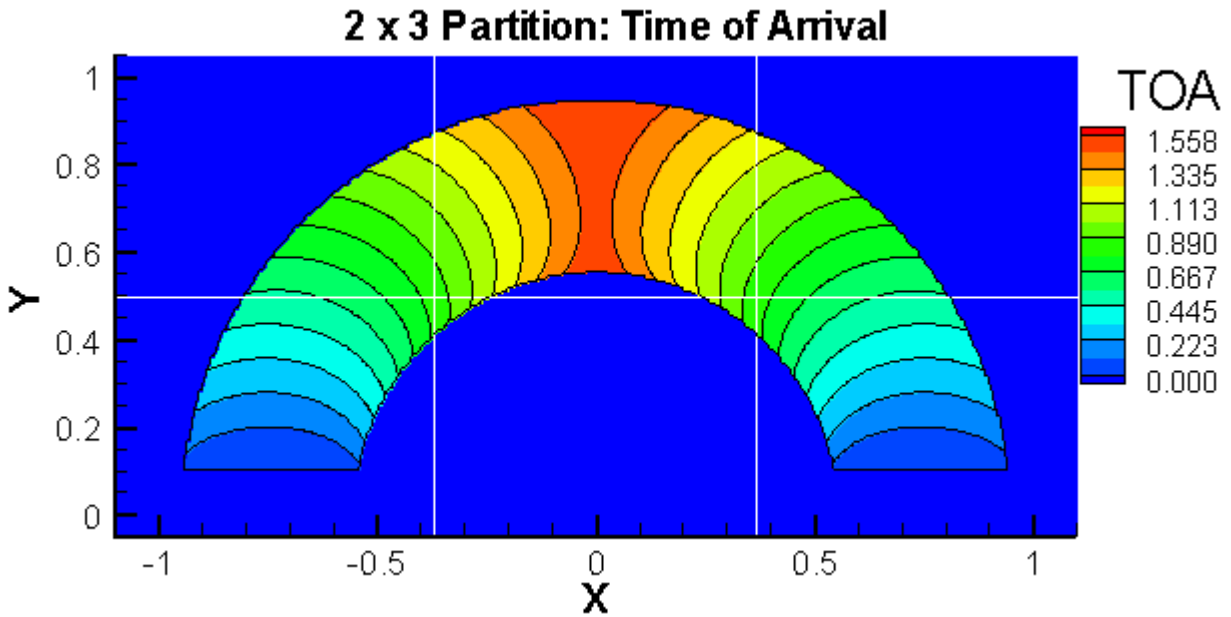


Figure 6.2: Shown are the times of arrival (TOA) with a horizontal (plane) detonation wave initialized at $y = 0.1$ for a 2x3 domain partition of the 180° arc problem. The domain and problem size are the same as in Fig.(6.1).

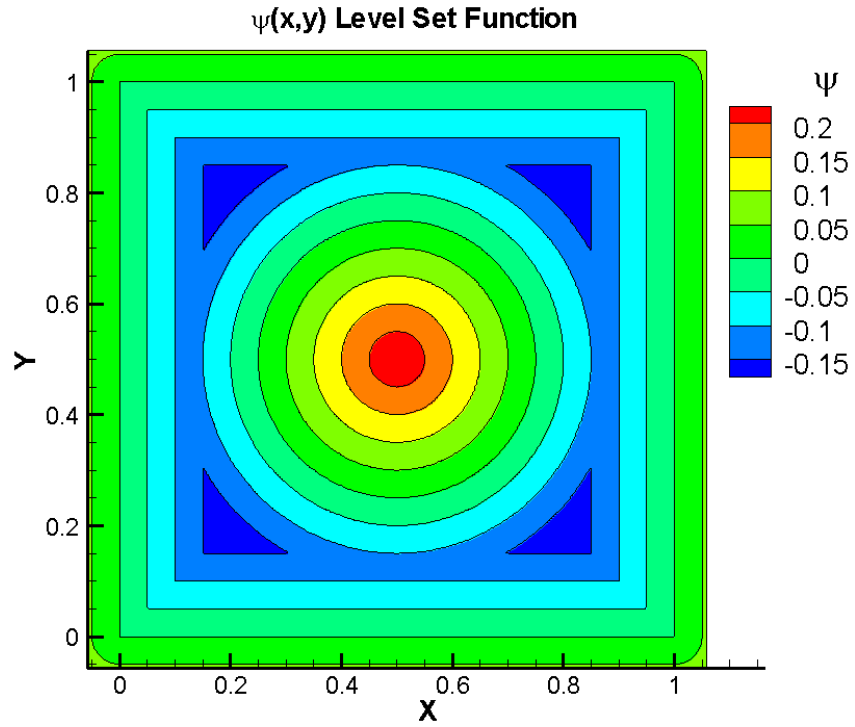


Figure 6.3: Show is the $\psi(x,y)$ level set function for the detonation in a square with a circular hole. The domain size was a 1.05×1.05 square, with $n_x = 600$ $n_y = 600$ and $dx = dy = 0.00175$.

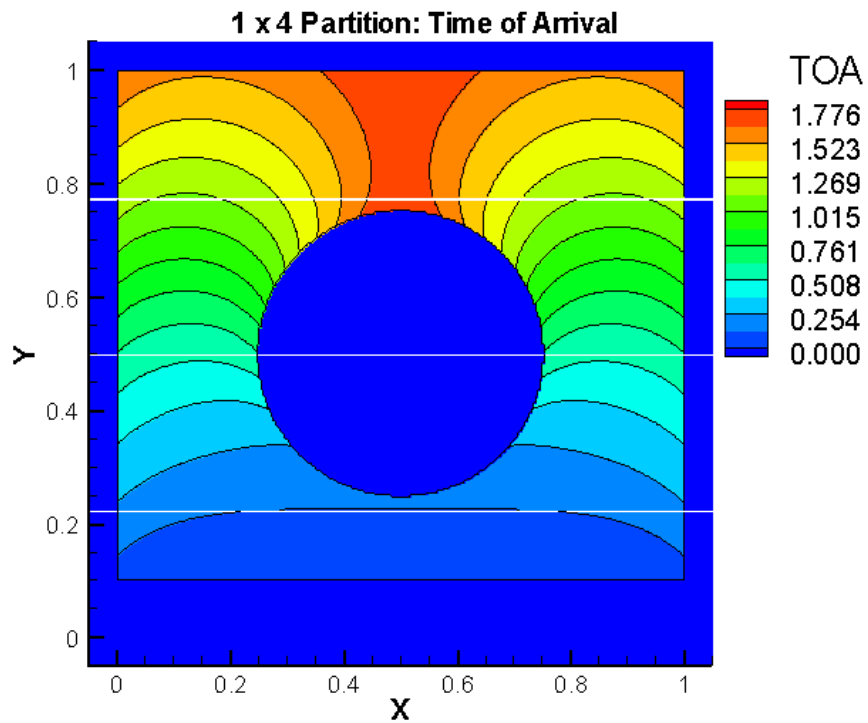


Figure 6.4: Show are the times of arrival (TOA) with a horizontal (plane) detonation wave initialized at $y = 0.1$ for a 1×4 domain partition of the circular hole problem. The domain and problem size are the same as in Fig.(6.3).

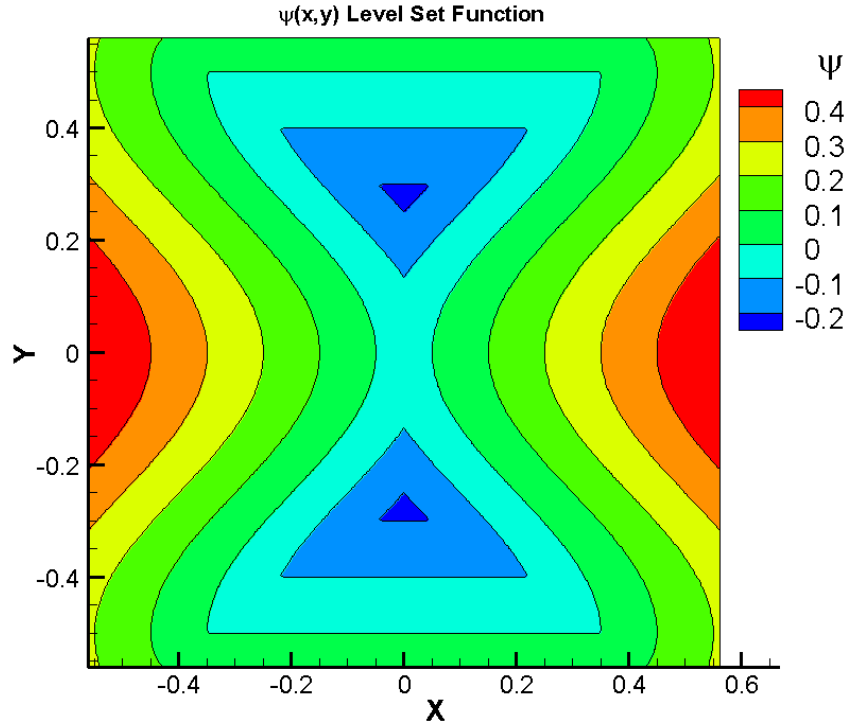


Figure 6.5: Show is the $\psi(x,y,t)$ level set function for the detonation in a hourglass problem. The domain size was a 1.10 x 1.10 square, with $n_x = 600$ $n_y = 600$ and $dx = dy = 0.00183$.

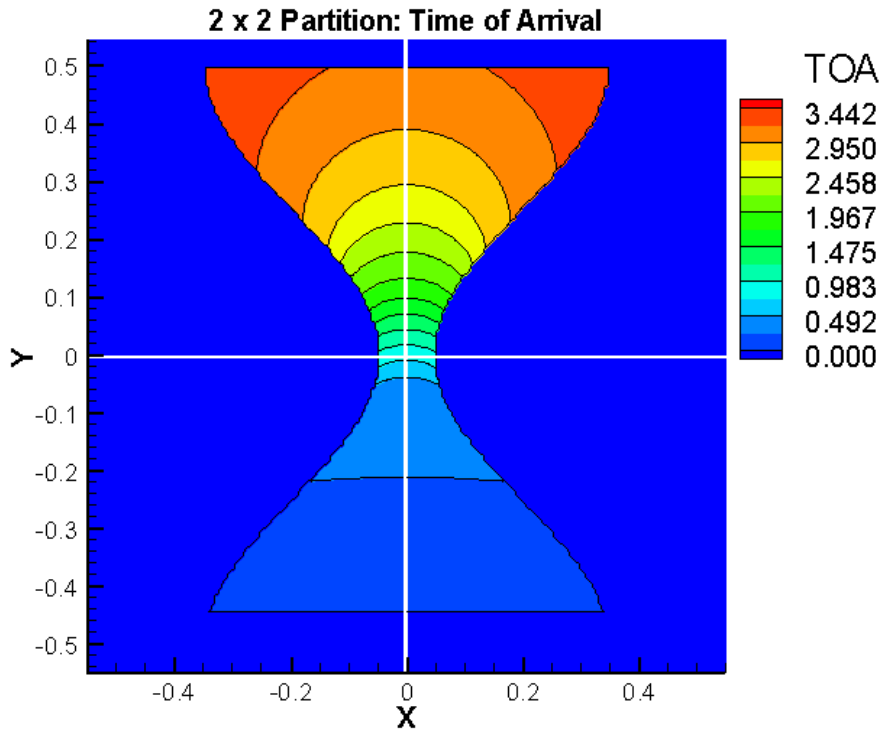


Figure 6.6: Show are the times of arrival (TOA) with a horizontal (plane) detonation wave initialized at $y = -0.45$ for a 2x2 domain partition of the hourglass problem. The domain and problem size are the same as in Fig.(6.5).

Chapter 7: Benchmarking

Benchmarking a parallel code consists of running several performances tests, like the ones described on section 2.4, to assess the overall performance of the parallel program. For IPC-DSD2D the Vertical Rate Stick test problem was chosen to benchmark the parallel code. This problem gives the best possible load balancing and optimal resource usage since every node is either an IB or a HE node. The domain was decomposed into vertical slices and combinations of vertical and horizontal slices. Vertical slices produce higher gains in performance, since they involve MPI sends of ghost node regions that are contiguous in memory. From table.(1), 0.4% of the serial code execution time was spent on node sorting. This is the portion of the code that was not made parallel. Because this is such a low percent, it is assumed that one hundred percent of the code can be made parallel for estimating the maximum theoretical speedup predicted by Amdahl's law, which for this case is linear.

Three problem sizes where used for benchmarking the parallel code: 100x300, 200x600 and 300x900. All jobs ran on three computer arrangements: the Turing Cluster at the University of Illinois Urbana-Champaign, an eight core Macintosh Mac Pro, and the NCSA SGI Altix (Cobalt).

- Turing Cluster: The Turing cluster consists of 768 Apple Xserves, each with two 2 GHz G5 processors and 4GB of RAM, for a total of 1536 processors. Since each node uses a minimum of two processors, the performance analysis was based on the two processor simulation run.
- Mac Pro: The tower used here was a Mac Pro with two Quad-Core Intel Xeon 2.93 GHz processors, for a total of eight cores. 256 KB L2 cache per core, and 8MB L3 cache per processors. The processors interconnect speed is 6.4 GT/s.
- SGI Altix (Cobalt): consists of 1,024 Intel Itanium 2 processors (1.6GHz , 1.66 GHz dual core) shared memory system. L3 cache per processors is from 8MB to 9MB.

7.1 Performance Analysis Tables and Plots

Table 4: Turing Cluster

Processors	Wall time (seconds)			Speedup			Efficiency		
	100x300	200x600	300x900	100x300	200x600	300x900	100x300	200x600	300x900
2	388	7123	40200	1.00	1.00	1.00	1.00	1.00	1.00
4	209	3108	18394	1.86	2.29	2.19	0.93	1.15	1.09
6	151	2119	11302	2.57	3.36	3.56	0.94	0.84	0.89
8	124	1525	8026	3.13	4.67	5.01	0.52	0.78	0.83
10	107	1223	6339	3.63	5.82	6.34	0.45	0.73	0.79
12	105	1058	5139	3.7	6.73	7.82	0.37	0.67	0.78
14	95	934	4380	4.04	7.63	9.18	0.29	0.64	0.76
16	92	952	3840	4.22	8.36	10.47	0.26	0.60	0.75

Table 5: Mac Pro

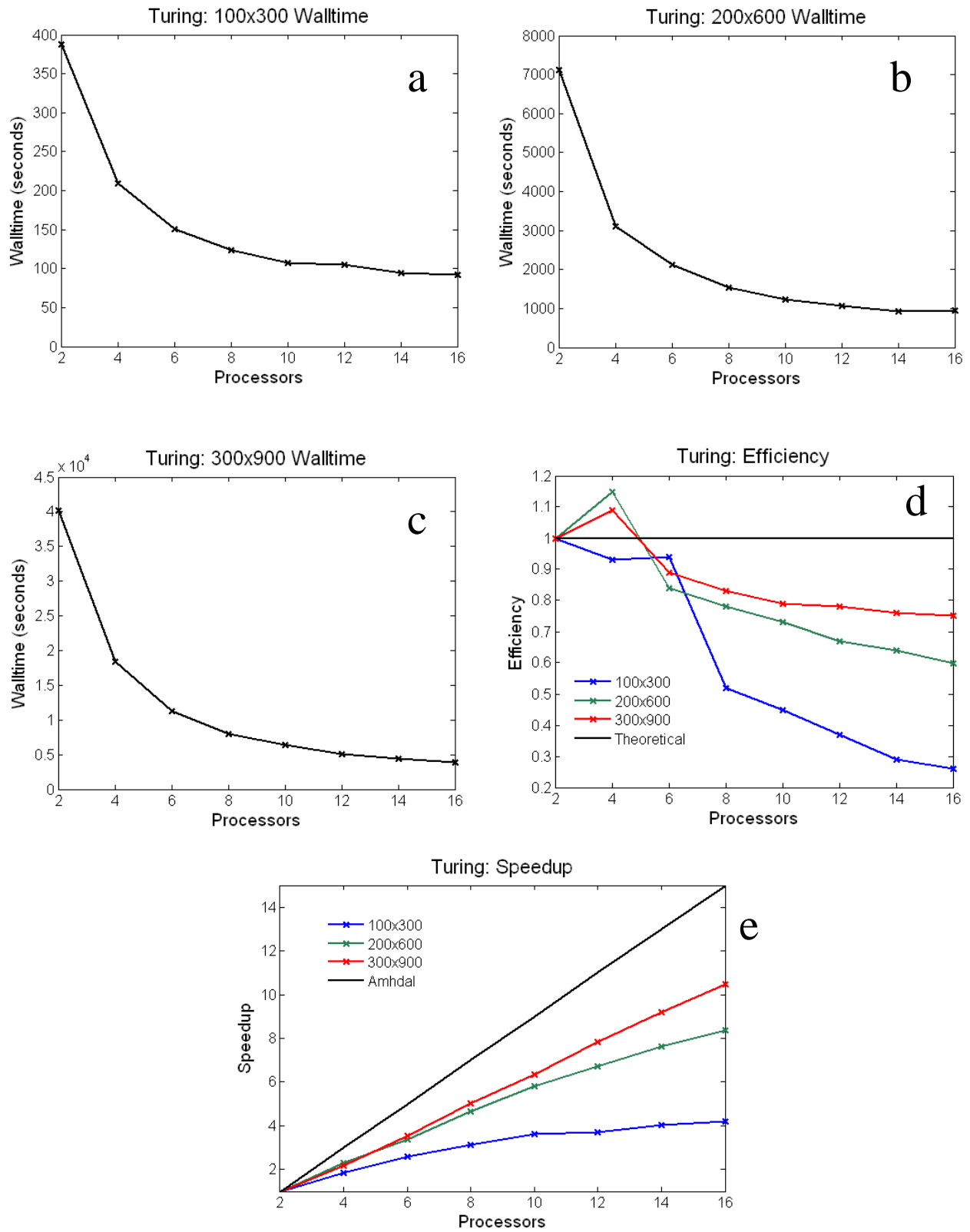
Processors	Wall time (seconds)			Speedup			Efficiency		
	100x300	200x600	300x900	100x300	200x600	300x900	100x300	200x600	300x900
1	353	5042	75149	1.00	1.00	1.00	1.00	1.00	1.00
2	184	2567	38412	1.92	1.96	1.96	0.96	0.98	0.98
4	98	1416	19606	3.60	3.56	3.83	0.90	0.89	0.96
6	71	943	13613	4.97	5.35	5.52	0.83	0.89	0.92
8	57	725	10350	6.19	6.95	7.26	0.77	0.87	0.91

Table 6: SGI Altix (Cobalt)

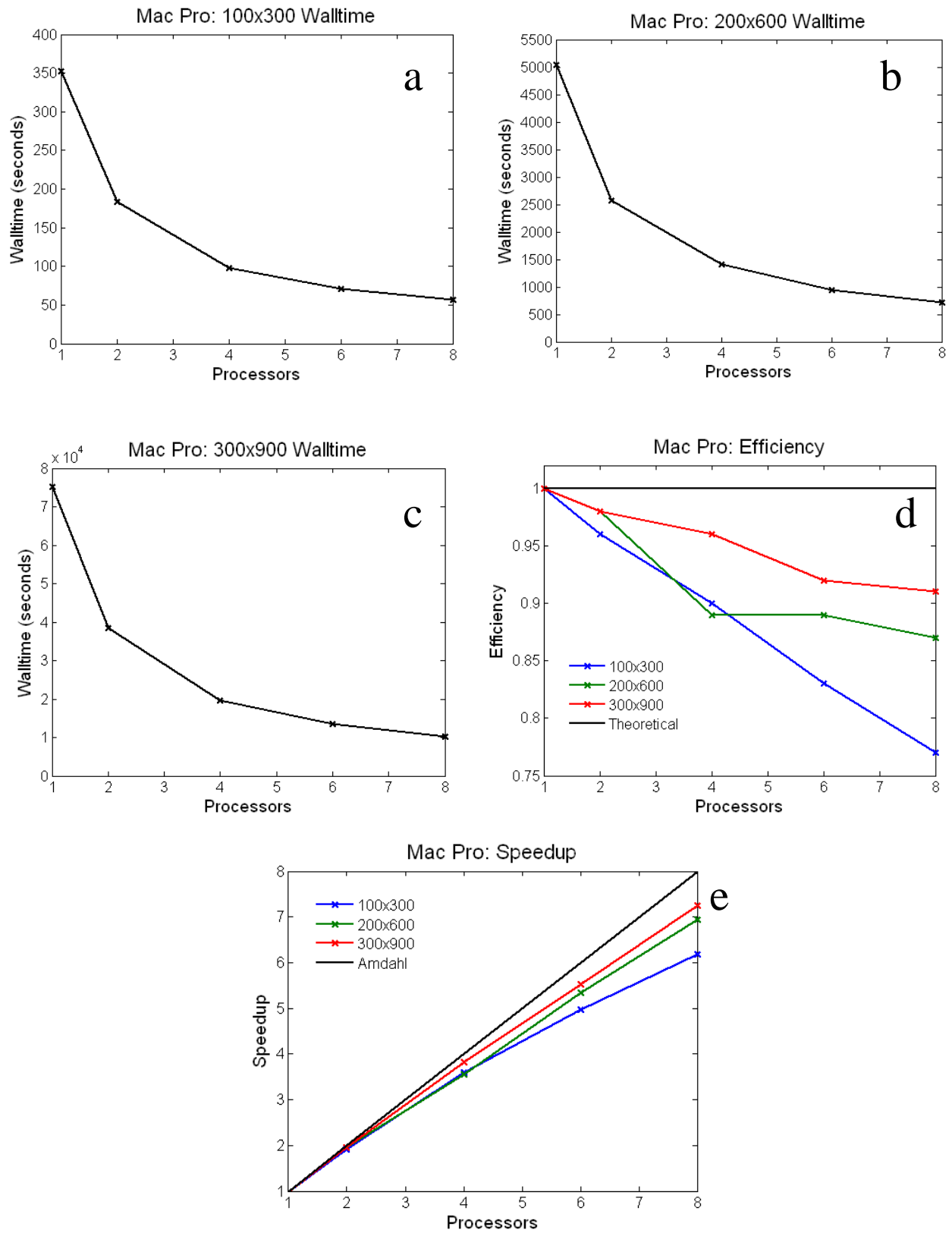
Processors	Wall time (seconds)			Speedup			Efficiency		
	100x300	200x600	300x900	100x300	200x600	300x900	100x300	200x600	300x900
1	745	10620	51660	1.00	1.00	1.00	1.00	1.00	1.00
2	384	5280	25620	1.94	2.01	2.02	0.97	1.01	1.01
4	217	2682	13200	3.43	3.96	3.91	0.86	0.99	0.98
6	144	1978	9060	5.17	5.36	5.70	0.86	0.89	0.95
8	117	1530	6960	6.37	6.94	7.42	0.80	0.87	0.93
10	97	1170	5340	7.68	9.07	9.67	0.77	0.91	0.99
12	86	1092	4620	8.66	9.72	11.18	0.72	0.81	0.93
14	79	968	4200	9.43	10.97	12.30	0.67	0.78	0.88
16	71	885	3780	10.49	12.00	13.67	0.66	0.75	0.85

Figure 7.1 - 7.3 show plots for the wall time, speedup and efficiency of the three test cases using Turing, the Mac pro and Cobalt. The following convention is used for the figures:

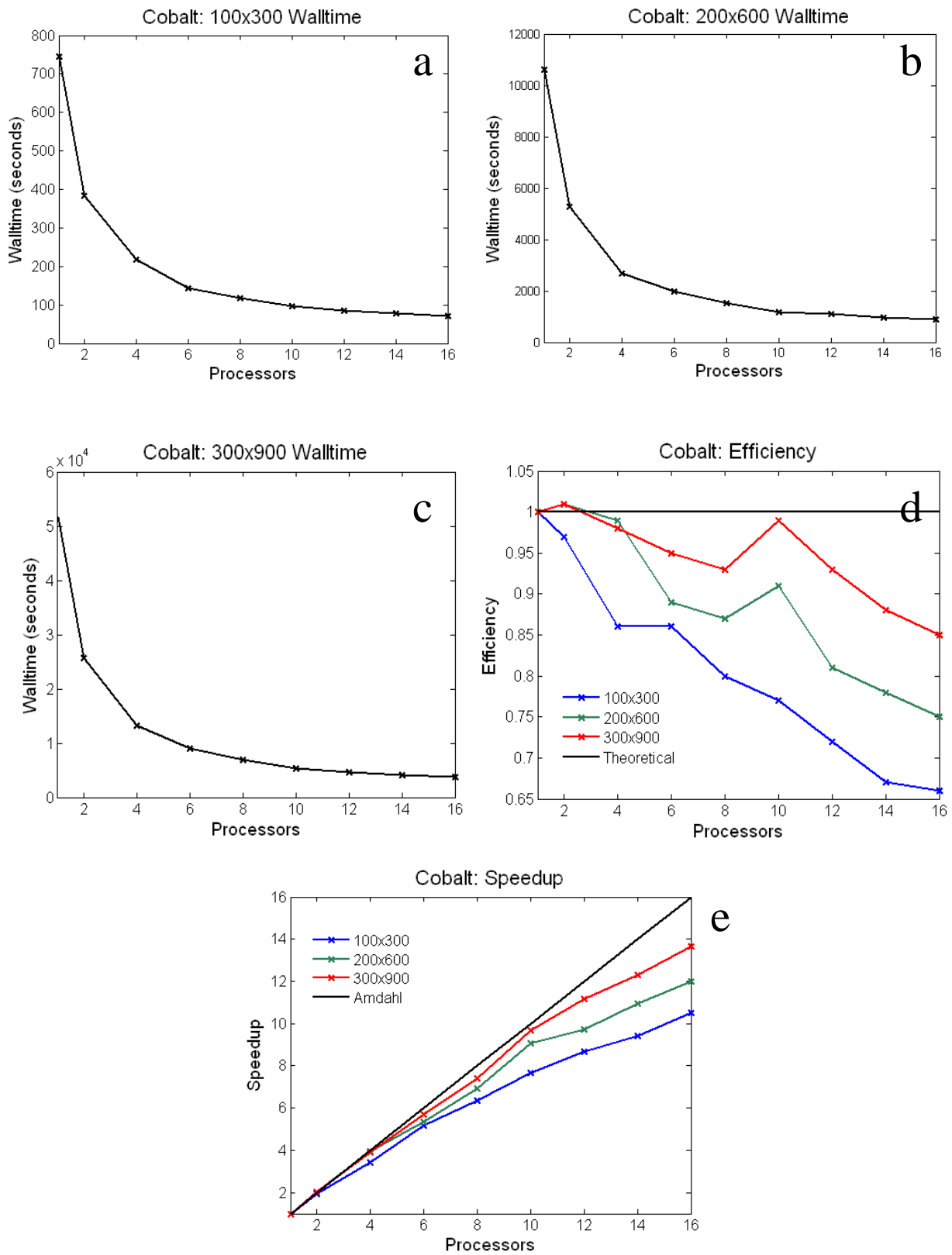
- Is the wall time for the 100x300 case
- Is the wall time for the 200x600 case
- Is the wall time for the 300x900 case
- Compares the efficiency for all the test problem sizes
- Compares the speedup for all the test problem sizes



Figures 7.1: Performance analysis using the Turing Cluster.



Figures 7.2: Performance analysis using the Mac Pro.



Figures 7.3: Performance analysis using SGI Altix (Cobalt).

Because how Full-LS-DSD2D scales (in terms of execution time) when a problem size is increased, it was necessary to implement a parallel design if the code were to be used to solve complex problems. The benchmarking of the code showed very good performance metrics which greatly enhance the effectiveness and usability of Full-LS-DSD2D. The speedup and efficiency were high, and behaved in a stable and predictable pattern. The speedup observed on all three computer arrangements was sub-linear, which was expected. The parallel design also obeyed Amdahl's law, which states that as more processors are used to solve a fixed problem size the speedup increases. It can also be seen from Fig.(7.1-7.3) that as the number of processors increase for a fixed problem size the speedup decreases compared with the maximum achievable value predicted by Amdahl's law.

As mentioned in section 2.4.1 a parallel program's performance is limited by many factors, for example: computer architecture, communications overhead, memory-cpu bus bandwidth, available resources, problem size, problem complexity and load balance. This was observed when analyzing the performance of the code while running on the three different computer arrangements. The Mac Pro had the best hardware characteristics. For this reason, the walltimes were shorter compared to the other two clusters and the speedup and efficiency were higher. For the 100x300 test problem, where the problem fits in cache the execution time was roughly two times faster than that of Cobalt. But the 300x900 test problem the execution time on Cobalt was faster than the Mac Pro. This problem size does not fit in cache in the Mac Pro, but the aggregate cache memory in Cobalt was greater than the Macintosh explaining the faster execution time. For the Turing cluster all the performance analysis was based on the two processor simulation. For this reason the speedup shown on Fig.(7.1) is lower than on Cobalt. But from table.(4) the execution times on the Turing Cluster were similar to the ones on Cobalt when using 2 or more processors.

Chapter 8: Shaped Charge Optimization Using DSD

This chapter describes how a multicomponent optimization system to generate optimal shaped charge designs using Detonation Shock Dynamics instead of the traditional Hydro-code (DNS model). As mentioned in Chapter 1, DSD simulations do not solve the reactive Euler Equations, but solve a PDE given by Eq. (9), this results in simulations that take less time and fewer computational resources to solve a problem compared with DNS models. Accurate estimates of pressure, velocity and other flow properties can be obtained from DSD simulations. The main idea is to use DSD to estimate the pressure along a shaped charge liner and the normal shock velocity at the apex of the liner. These flow variables are then used as inputs for a Lagrangian finite element code to determine the shape of the jet that is formed by the high pressures caused by the detonation shock crushing the liner.

This chapter is divided in three sections. Section 8.1 defines shape charges and their physical characteristics. Section 8.2 describes how to estimate the pressure along the liner and normal shock velocity using IPC-DSD2D. Finally section 8.3 describes an optimization system to find optimal shape charge designs. This thesis only describes the optimization system, and did not simulate the design loop. See reference [1] for work on the optimization of shaped charge design by the Parallel Computational Sciences Department at Sandia National Laboratories.

8.1 Shaped Charges

A shaped charge is an explosive that has been pressed and compacted to a shape that focuses the release of energy once the explosive is ignited. These types of charges are used in the mining and oil industry to penetrate rock and to extract hydrocarbons or mineral ore. These types of shaped charges are also known as perforating, since they are designed to achieve material penetration. Most shaped charges are composed of a primer and main explosive, an outer casing, a liner and a detonation cord. Figure 8.1 was taken from reference [8] and shows a representative shaped charge and the time sequence of a charge detonation. The detonation cord ignites the primer which detonates the main explosive creating a detonation wave that travels through the device at high velocity. When the wave hits the liner apex it pushes it outward; the force exerted on the liner due to the high pressures collapse the liner and forms a high-velocity jet of fluidized

metal that travels along the charge axis. The performance of a shape charge mainly depends on the type of explosive used and depends on the formation of the jet shape of the [8].

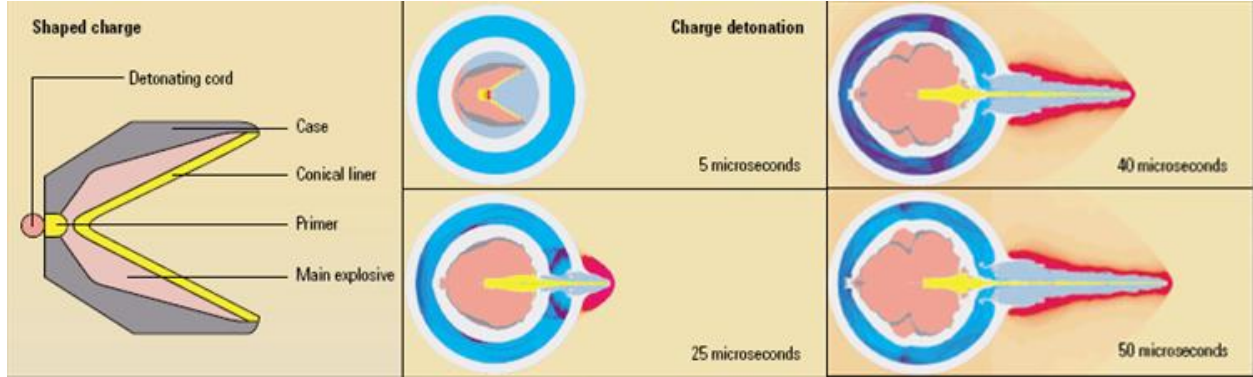


Figure 8.1: Schematic representation of a shaped charge, and a time plot of the reaction process that occurs in the shaped charge as soon as the detonation cord ignites the primer explosive. The figure also shows the jet formation.

8.2 DSD Shaped Charge Simulation

This section focuses on how to model a shaped charge using IPC-DSD2D, and how to obtain estimates of liner pressure and normal shock velocity. Figure 8.2 shows a simplified model of a perforating shaped charge used for the DSD simulation. The geometry of the shape charge is described by the design parameters shown on the same figure. The length D is constrained by the angles ζ , Φ , and β and parameter L . The length of H is constrained by angles Φ and β , and parameters B and L .

Since multi-material capabilities are not currently installed in Full-LS-DSD2D, only the main explosive is modeled instead of both the main charge and primer. Also since the point of interest is to estimate the pressure along the liner, the material used for the confinement and sonic angle pairs was that of the liner. Full-LS-DSD2D uses a rectangular grid to discretize the computational domain. Because of this, the representation of the liner would have a step like shape due to the slope defined by the angle Φ . To reduce the numerical noise that would be associated with this step like representation, the shape charge design was rotated clockwise by Φ such that the upper liner wall was aligned with the horizontal axis at $Y = 0$. Figure 8.3 shows this rotated shape; the red circle represents the initial shock location $\phi(x,y,0)$ which models a point detonation charge. Figure 8.4 shows the ψ level set field, where $\psi = 0$ is the inert/explosive interface.

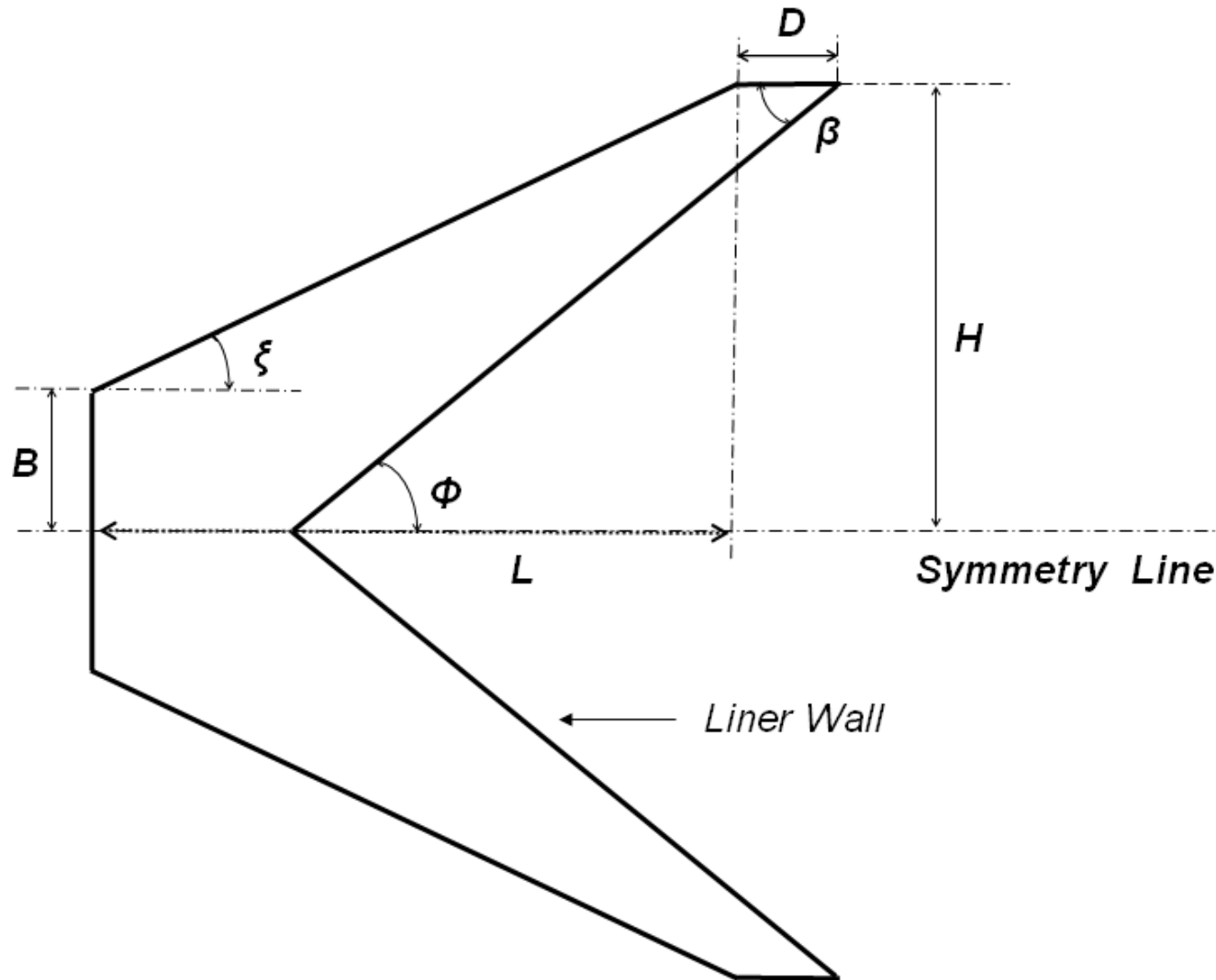


Figure 8.2: shows a simplified model of a perforating shaped charge used for the DSD simulation. The geometry of the shape charge is described by the design parameters shown on the same figure. The bottom half of the figure is horizontally symmetric.

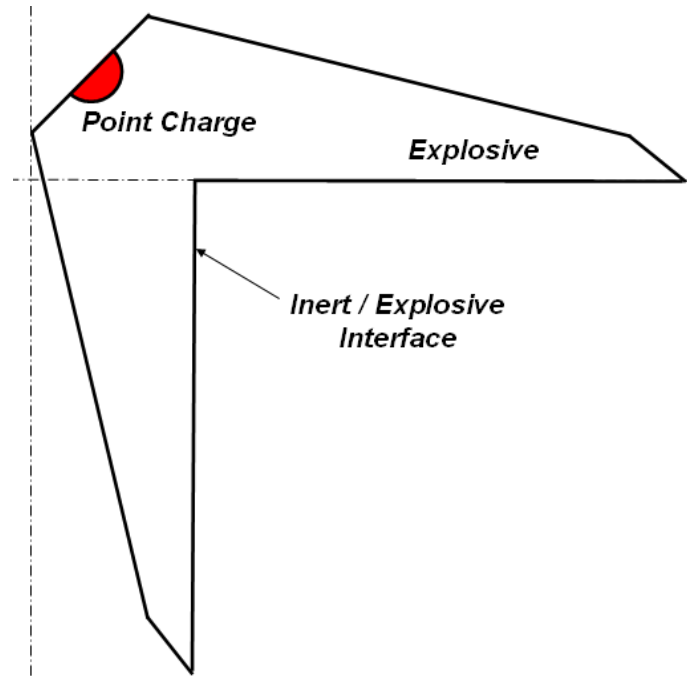


Figure 8.3: Shaped charge geometry rotated by x -degrees to minimize numerical noise associated with the step like representation of the linear wall of Fig.(8.1). The red circle represents the φ level set function initialization of the detonation front.

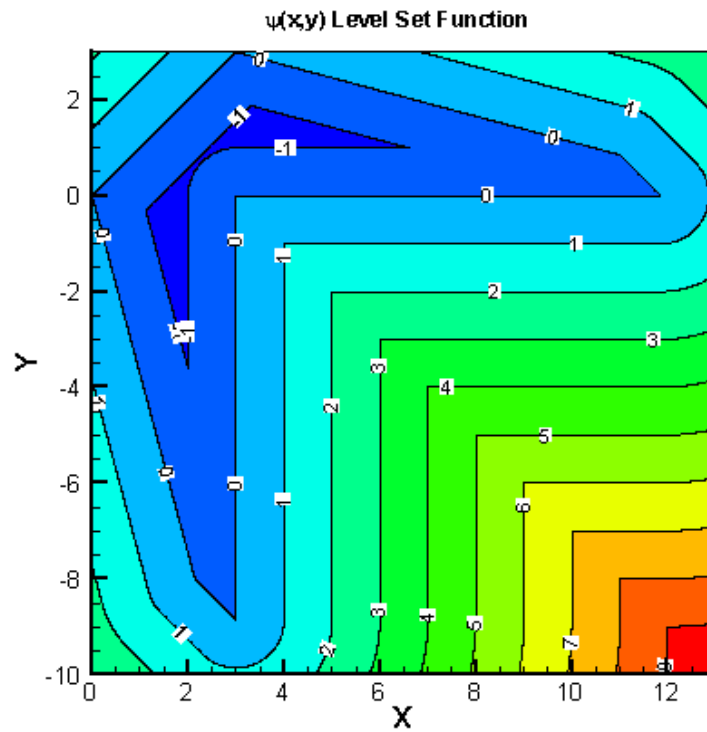


Figure 8.4: Shows the $\psi(x,y)$ level set field for the rotated shape charge design using the geometrical parameters shown in table.(8). Labels indicate the level curves of ψ .

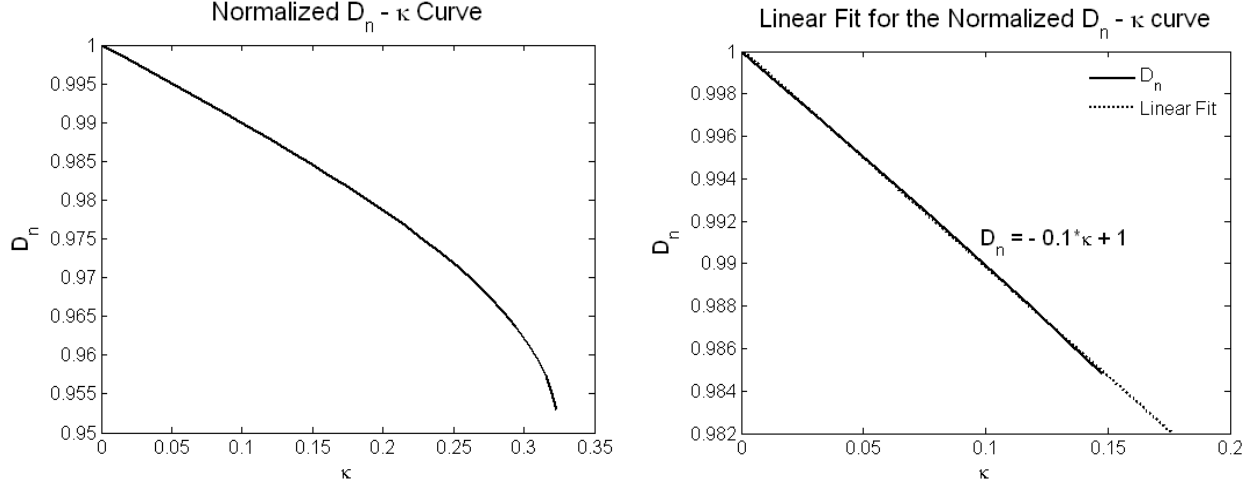


Figure 8.5 (left): Normalized upper branch of the D_n - κ curve for the high explosive used in the DSD simulation.
Figure 8.6 (right): Linear fit for small curvature in the vicinity of D_{CJ} .

For the simulation the inert material used was copper, which is common for fabricating liners. The Explosive D_n - κ curve is non-linear and was calibrated using the Wide Ranging Equation of State. The sonic and confinement angle pairs were found using DSDTOOLS. DSDTOOLS is an engineering software tool that presents the user with an interface that allows a choice of fitting parameters for the EOS and its rate models and the computation of reaction zone propagation in the shock-attached frame. This software was developed at the University of Illinois by Dr. Sunhee Yoo and Professor Donald S. Stewart. Figure 8.5 shows the normalized and non-dimensional upper branch of the D_n - κ curve.

In most engineering applications, only small values of curvature around the D_{CJ} state are used [12]. To simulate the shaped charge using IPC-DSD2D a linear approximation of the D_n - κ curve of Fig.(8.5) was obtained by taking small values of curvature close to the CJ state to estimate the slope α of Eq.(21). This linear fit yielded a slope of 0.1. Figure 8.6 shows the linear approximation and Eq.(21) shows this linear fit where, D_{CJ} is equal to one. Table 7 lists all the simulation parameters used for the shaped charge DSD problem. Ymin, Ymax, Xmax and Xmin define the computational domain which is non-dimensional. NX and NY are the number of nodes used for the simulation.

$$D_n = D_{CJ} - 0.1\kappa \quad (22)$$

Table 8 shows the initial geometric parameters used for the simulation. Normally the liner has a conical apex shape, but to simplify the design and distance function representation of the shape charge, a 90 ° degree angle between the two liner walls was used.

Table 7: Computational parameters used in IPC-DSD2D

$D_{CJ} = 1$	$\alpha = 0.1$	$w_c = 50.8^\circ$	$w_s = 75.8^\circ$	$CFL = 0.9$
$NX = 1300$	$NY = 1300$	$Xmin = 0.0$	$Xmax = 13.0$	$Ymin = -10.0$
$Ymax = 3.0$				

Table 8: Shape Charge Geometric Parameters

$B = 1.5$	$L = 11.0$	$H = f(\beta, \xi, B, L)$	$D = f(\beta, \xi, \Phi, L)$	$\beta = 45^\circ$	$\xi = 15^\circ$	$\Phi = 45^\circ$
-----------	------------	---------------------------	------------------------------	--------------------	------------------	-------------------

After running the simulation with IPC-DSD2D, a Time of Arrival (TOA) and a D_n plot were obtained. Figure 8.7 shows the time of arrival plot and figure 8.8, the normal shock velocity, D_n . The normal shock velocity is computed by taking the magnitude of the gradient of the TOA field, and then taking its inverse. For the two-dimensional case, D_n is found by equation (23), where the Greek letter μ represents time of arrival

$$D_n = \frac{1}{\sqrt{\left(\frac{\partial \mu}{\partial x}\right)^2 + \left(\frac{\partial \mu}{\partial y}\right)^2 + \left(\frac{\partial \mu}{\partial z}\right)^2}} \quad (23)$$

The pressure along the liner for a detonation in a polytropic gas is related to the shock normal velocity by equations (24-27) [13]. Here γ is the ratio of specific heats for a given gas, λ is the reaction progress variable where $\lambda = 1$ is a completely reacted material and $\lambda = 0$ is an unreacted material. P_{CJ} is the shock pressure at the CJ state and D_n is found by Eq.(3). Here D_n is scaled relative to D_{CJ} , which in the simulation is equal to one. Using the fact that at the shock $\lambda = 0$ and that $D_{CJ} = 1$, a simplified expression for pressure is obtained as equation (27).

$$P_{CJ} = \rho_o D_{CJ}^2 / (\gamma + 1) \quad (24)$$

$$f = \left(\frac{D_n}{D_{CJ}} \right)^2 \quad (25)$$

$$g = \left(1 - \frac{\lambda}{f}\right)^{1/2} \quad (26)$$

$$P = fP_{CJ}(1 \pm g) \quad (27)$$

$$P = \frac{\rho_o}{\gamma + 1} D_n^2 \quad (28)$$

For a non ideal equation of state the analytical expression, Eq.(28) does not hold. Thus a numerical solution to the Rankine-Hugoniot equations needs to be performed (for some cases) to solve for pressure. But, near D_{CJ} , where curvature is small, a linear relationship of the form $P \approx D_n^2$ will correlate almost exactly with the shock pressure. Using the D_n profile from the DSD simulation and the $P \approx D_n^2$ correlation for small curvature, an approximation of the pressure along the liner was obtained. Figure 8.9 shows the shock pressure along the liner when varying the geometric parameter ζ by $\pm 5^\circ$. This shows the effect of only perturbing one design parameter on the pressure distribution along the liner; for the optimization system that will be described in the next section all design parameters can be modified.

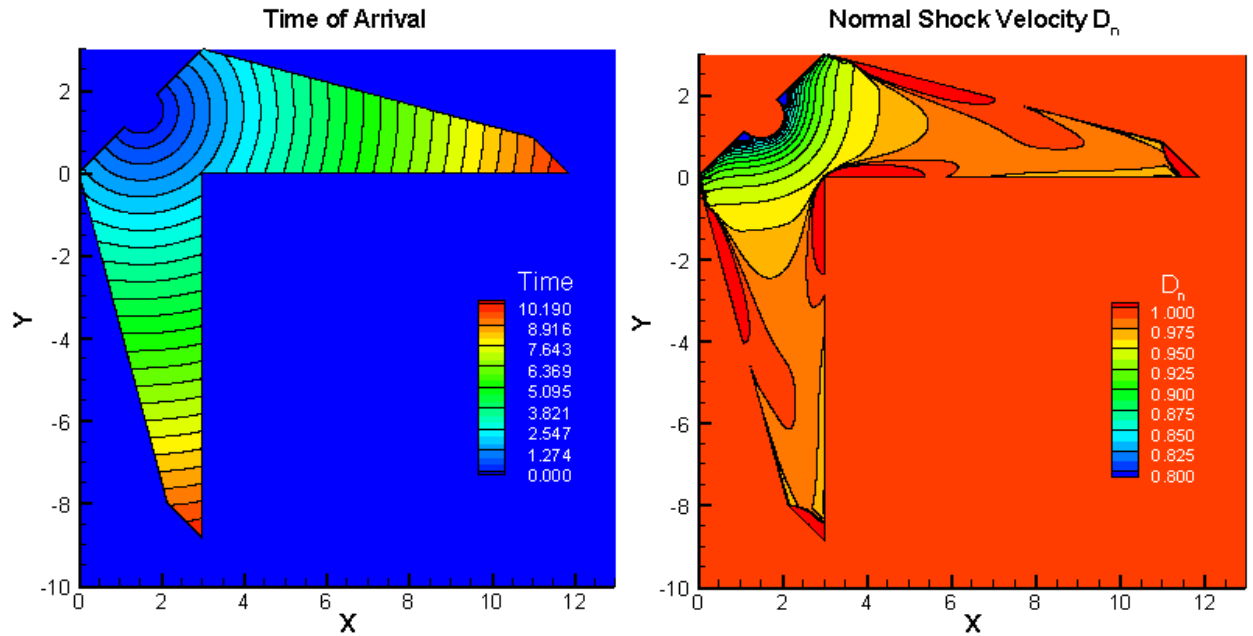


Figure 8.7 (left): Time of Arrival (TOA) field for the DSD simulation using IPC-DSD2D.

Figure 8.8 (right): Normal shock velocity D_n field for the DSD simulation using IPC-DSD2D.

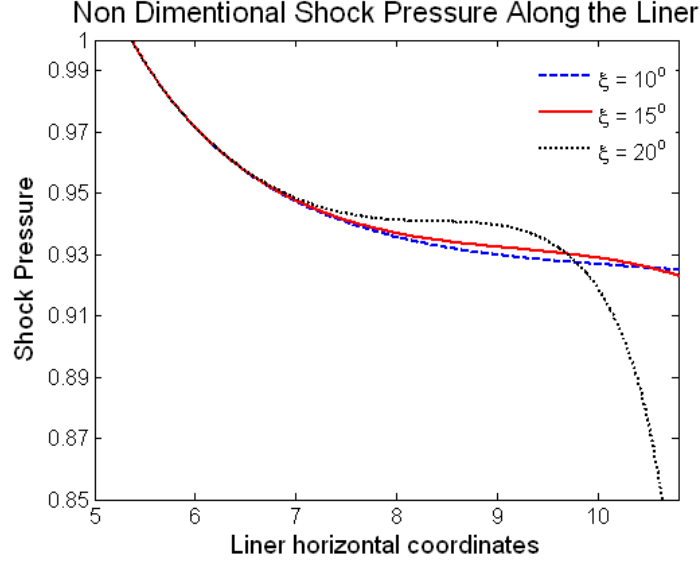


Figure 8.9: Shock pressure along the liner when varying the geometric parameter ξ by $\pm 5^\circ$.

8.3 Shape Charge Optimization

As described in section 8.1, shaped charges are designed to achieve a determined perforation depth and blast hole diameter. These properties are influenced by the shape of the jet formed by the liner. The goal of the design process is to optimize the geometrical parameters that define the shape of charge such that the best possible jet geometry characteristics are achieved. In particular the jet diameter and jet length are the focus of the optimization system. Here jet length refers to how far the jet travels before losing momentum and consequent jet shape. Depending on the application, it might be advantageous to have a small blast hole diameter and deep material perforation, or a large blast diameter with little perforation, or a combination of both. To optimize the shape charge geometry taking into account these two design goals a weighted linear combination of two objective functions is proposed: Equation (29) maximizes jet length, Equation (31) maximizes jet diameter at some specific point in the domain, and Equation (30) is the linear combination of the previous two. The weighting constant w can be modified to account for preferences of either jet length or jet diameter.

$$OBJ_1 = \max_{geometry \in \eta} Jet_{length}(geometry, Liner Pressure, Apex shock Velocity) \quad (29)$$

$$OBJ_2 = \max_{geometry \in \eta} Jet_{Diameter}(geometry, Liner Pressure, Apex shock Velocity) \quad (30)$$

$$W_{OBJ} = \max_{geometry \in \eta} wOBJ_1 + (1-w)OBJ_2 \quad (31)$$

Here OBJ_1 and OBJ_2 are functions of pressure and normal shock velocity obtained from the DSD simulation, and the geometrical parameters from Fig.(8.2). These objective functions are subject to the geometrical constraints, η that were mentioned in section 8.2 as well as physical constraints imposed by the problem specification. For example, the shape charge has to be of a certain length and/or height.

If a specific perforation depth and /or blast hole diameter pair is required, then the objective function can be expressed as a minimization problem. The problem would be to minimize the difference between the specified jet characteristics and the jet characteristic obtained through the numerical simulation. Using the same linear combination of two functions approach, the objective function can be expressed as equation (32).

$$W_{OBJ} = \min_{geometry \in \eta} w(JL_g - JD_m) + (1-w)(JD_g - JD_m) \quad (32)$$

Here JL and JD stand for Jet Length and Jet Diameter respectively. The subscript g indicates the specified solution goals, and subscript m indicates properties found using the numerical simulation.

The simulation processes would be iterative. After each design cycle the objective function would be evaluated and a logical operation would decide to continue or stop the processes. Each design iteration of the optimization system will consist of the following steps:

1. Define geometrical parameters to model the shaped charge.
2. Using IPC-DSD2D solve for the normal shock velocity at the liner apex and pressure along the liner wall.
3. Use the output from the IPC-DSD2D simulation to estimate the jet shape using a Lagrangian finite element code.

4. Define an objective function to use as an optimization metric: determine jet shape by taking into consideration jet diameter and length subject to the objective function given by either Eq.(31) or Eq.(32).
5. Use a non-linear optimization algorithm to maximize the objective function by perturbing the design parameter of step 1.

The optimization process will stop when the difference between the value of two consecutive objective function evaluations is below a user defined tolerance, or after a preset number of iterations is reached. Figure 8.10 shows a schematic representation of the proposed optimization system.

As was shown in Fig.(8.9), varying the angle ζ by $\pm 5^\circ$ changes the pressure profile along the liner wall. By running the above system, a correlation between the design goals and design parameters can be established. In particular for a giving problem, a response curve can be created showing the sensitivity of the DSD simulation output (liner pressure and normal shock velocity approximation) on the formation of the jet. This response curve can then be used as a metric for designing shaped charges without having to run the optimization system.

The quality of the optimal design solution will depend on the robustness of the optimizing algorithm used. Taking the minimization problem described by Eq.(32) as an example: the solution space for the mathematical problem might have a global minimum and many local minimum. Most optimization methods use local information (derivatives or Taylor series expansion) that are designed to only find local minima [10]. If there exists a global minimum, then it is recommended to run the optimization system using various design parameter initial value guesses scattered throughout the feasible design space.

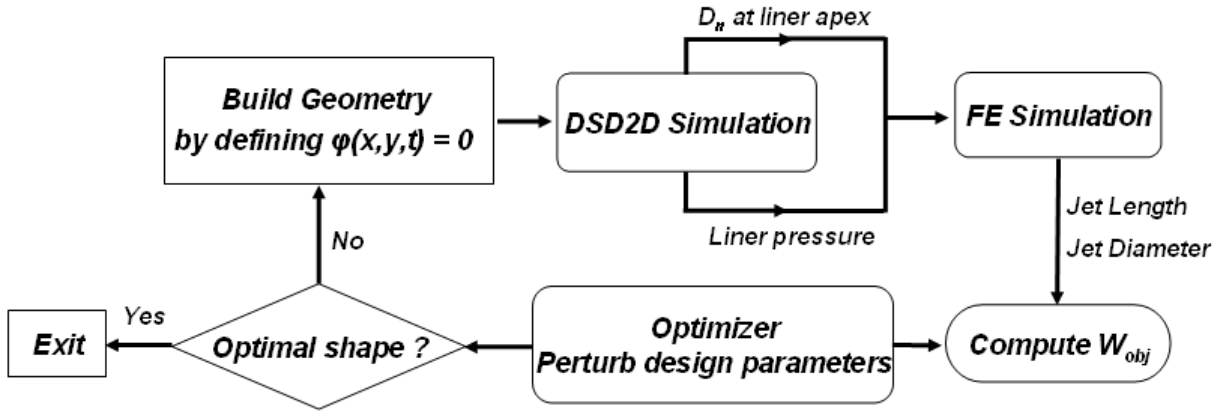


Figure 8.10: schematic representation of the proposed optimization system. The shape charge geometry is defined by specifying the design parameters shown on Fig.(8.2). This design is then used in the DSD simulation, where the shock pressure along the liner wall and normal shock velocity at the liner apex are obtained. These variables are then fed to a FE code to determine the resulting jet shape. An optimizer than perturbs the design parameter by either maximizing Eq.(31) or minimizing Eq.(32) until an optimal shape that meets the design goals is achieved.

Chapter 9: Parallel extension for a 3D Version of IPC-DSD2D

The first step before making a three-dimensional extension of IPC-DSD2D is to have a better domain decomposition routine. As was mentioned in section 5.2, the current domain partitioning routine has the following drawbacks; the load balancing depends on the explosive geometry and generally would not have equal distribution of active nodes to processors. A new algorithm is proposed to fix these drawbacks. It consists of breaking the domain into many fixed size blocks and then grouping them together using space filling curves. This method will break the one-to-one mapping between processors and computational sub-domains. Below are the steps needed to implement this new algorithm:

1. Divide the domain into a series of small rectangles (blocks) of a fixed size (making sure that the entire domain is filled),
2. Scan each block, and mark the ones that are active (have HE and/or IB nodes). The inactive nodes will be treated as empty space like the green blocks of Fig.(5.1),
3. Given the number of processors that are available, decide how many active blocks will be assigned to a particular processor. Trying to distribute equal number of blocks to each processor,
4. Using space filling curves to group blocks that are at least joined by one edge, decide what blocks are to be assigned to each processor,
5. Add lists that identify each block to a processor (needs to be broadcast to all processors),
6. Need to add lists to control what blocks require either information from other processors or information already stored in the processors current memory. A list with the following 3 flags:
 - -99: means neighbor is an inactive node or boundary node,
 - -10: means neighbor is on the same processor,
 - 0 to # blocks: number indicates a blocks' neighbor (for example, Right = 5 means the right neighbor is a block with an ID = 5), and
7. Ghost zone sharing between blocks stored on the same processor could be done via simple copy/pasting of edges, since information is already stored in local processor

memory. The sharing between different processors would be the same as the current parallel implementation.

This strategy would be relatively simple to implement and would keep most of the current domain decomposition and MPI boundary condition subroutines the same. A drawback of the proposed model is that the MPI overhead associated with sharing information between blocks stored on different processors would increase. Figure 9.1 shows why this new complexity would come into effect. Having a one-to-one mapping between processors and computational sub-domains, all ghost node information was shared with other processors via only one call to the MPI boundary conditions routine. As mentioned in Chapter 5, rows and columns of data are sent in the current 2D model, and each processor has the same number of ghost nodes along a given edge. Now, since the block edges of Fig.(9.1) do not have equal sizes, a call to the MPI boundary conditions would need to be made for communicating each individual blocks' ghost nodes.

By having an efficient domain decomposition strategy, one can then focus on how to modify the existing parallel framework to accommodate the third dimension. The current parallel design in IPC-DSD2D was built such that a future extension into three dimensions would be simple and use most of the subroutines and serial codes changes already found in the current 2D design. As far as the parallel design, there are some key aspects that would need to be changed. But these should be straightforward to implement. Below are some of the key points that would need to be added / changed from the parallel design of IPC-DSD2D:

1. The domain would need to be partitioned into cubes instead of rectangles,
2. Each block would have 18 neighbors that it could share information with,
3. The user would need to specify the number of slices in the z-coordinate direction. A Zblocks parameter needs to be added. The block ID array size would then be (Xblocks x Yblocks x Zblocks),
4. Instead of sending rows or columns of data, MPI sharing of data between processors would also require sending and receiving planes of data. Each plane (the faces of the cube) and each edge of the cube would now need to be shared, see figure 9.2, and,
5. MPI boundary conditions would now be applied for sending information in 3 directions (x, y and z).

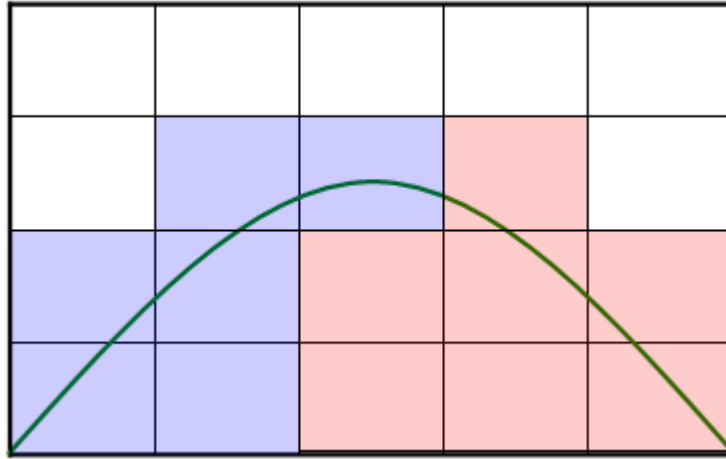


Figure 9.1: The blue blocks are mapped to processor 0 and the red blocks are mapped to processor 1. Each block has its own dimension and ghost nodes. Sharing of information between block on the same processor is done by simply copying the ghost node edges since they are stored in the same memory pool. Exchanging data between the two processors is more involved since not all blocks share the same edges. Each block on the interface between processor 0 and processor 1 need to share data separately with its neighboring blocks.

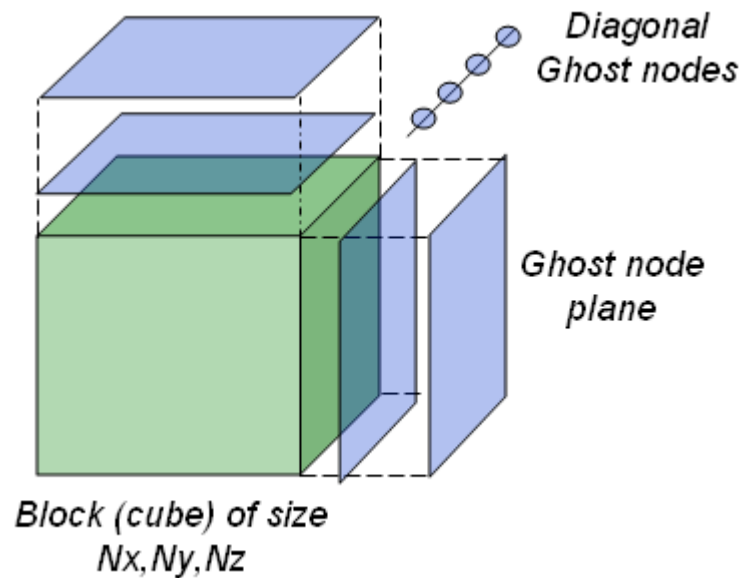


Figure 9.2: The computational sub-domain is the green cube. Each face has a plane of ghost nodes. Since in 2D two ghosts node were needed, in 3D two planes of ghost nodes are now needed (blue rectangles). Diagonal set of nodes are now sent instead of only one node as in the 2D case.

Chapter 10: Conclusions

The goal of this project was to develop a parallel extension for a Detonation Shock Dynamics code (Full-LS-DSD2D), and to demonstrate how it can be applied for solving engineering problems in detonation physics. Full-LS-DSD2D is an explicit code written in Fortran 77 that handles DSD boundary conditions both accurately and robustly. This code was modified to allow a parallel extension using a message passing model with an MPI interface, IPC-DSD2D. The serial code was verified in [5][6] for accuracy and convergence. In this study the serial code was slightly modified, which produced a significantly faster serial code. The parallel code produced identical numerical results as the serial code. This parallel design was then benchmarked, and the results of this performance analysis showed both very high and stable metrics, such as efficiency and speedup when running under the three computer platforms considered here.

IPC-DSD2D was then used for the development of a design optimization system to find optimal geometries that would result in ideal jet formation properties for shaped charges. By using DSD instead of conventional DNS models, the solution process could be both accurate and take less run time, since DSD does not solve the full Euler equation given by Eq.(1). A simulation was made using IPC-DSD2D giving the time of arrival and normal shock velocity fields across the shaped charge. From these flow variables the shock pressure along the liner wall and the normal shock velocity at the liner apex were computed. The design parameter ζ was then perturbed to show how the shock pressure along the liner is influenced by geometric changes in the shaped charge. The simulation successfully showed how DSD could be used in the optimization system described in section 8.3.

To further improve on IPC-DSD2D, a new domain decomposition algorithm was described that would extract more gains from parallelism. These new ideas would make a parallel implementation of a three-dimensional version of IPC-DSD2D a more efficient tool with which to solve explosive design-type problems. The parallel extension to 3D would not involve significant modifications since the new code was written and designed to be easily extensible to 3D.

References

- [1] D. R. Gardner and C. T. Vaughan, “The Optimization of a Shaped-Charge Design Using Parallel Computers” (Sandia National Laboratories, November 1999), SAND99-2953
- [2] D. S. Stewart, S. Yoo and B. L. Wescott “High-order numerical simulation and modeling of the interaction of energetic and inert materials”
Combustion Theory and Modeling Vol.11, No. 2, April 2007, 305-332
- [3] J. A Sethian *Level Set Methods and Fast Marching Methods*
(Cambridge University Press, New York, 1999)
- [4] J. B. Bdzil and D. S. Stewart, “The Dynamics of Detonation in Explosive Systems”
Annual Review of Fluid Mechanics Volume 39, 2007
- [5] J. B. Bdzil, T. Aida et al. “Test Problems for DSD3D”
(Los Alamos National Laboratory, July 2007), LA-14336
- [6] J. B. Bdzil, R. J. Henninger et al. Test Problems for DSD2D
(Los Alamos National Laboratory, July 2006), LA-14277
- [7] J. L. Gustadson “Fixed Time, Tiered Memory, and Superlinear Speedup”
Ames Laboratory – USDOE Ames, IA 50011
- [8] L. Behrmann, J.E. Brooks, et al. “Oilfield Review: Perforating Practices that Optimize Productivity” (Oilfield Review volume 12, issue 1, March 1st, 200)
- [9] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*
(McGrawHill, New York, 2004)
- [10] M. T. Heath, *Scientific Computing An Introductory Survey second edition*
(McGraw-Hill. New York, 2002)
- [11] P. S. Pacheco, *Parallel Programming with MPI*
(Morgan Kaufmann, San Francisco, 1997)
- [12] T. D. Aslam, J. B. Bdzil, and D. S. Stewart “Level Set Methods Applied to Modeling Detonation Shock Dynamics” J. Comput. Phys. 126, 390-409 (1996)
- [13] W. Fickett and W.C. Davis *Detonation Theory and Experiment*
(Dover Publications. New York, 1979)