NON-OVERLAPPING DOMAIN DECOMPOSITION PARALLEL ALGORITHMS FOR CONVECTION-DIFFUSION PROBLEMS

by

Shawn J.A. Chiappetta

A Dissertation Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

in Mathematics

at

The University of Wisconsin-Milwaukee

May 2009

NON-OVERLAPPING DOMAIN DECOMPOSITION PARALLEL ALGORITHMS FOR CONVECTION-DIFFUSION PROBLEMS

by

Shawn J.A. Chiappetta

A Dissertation Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

in Mathematics

at

The University of Wisconsin-Milwaukee

May 2009

Major Professor	Date
Graduate School Approval	Date

ABSTRACT

NON-OVERLAPPING DOMAIN DECOMPOSITION PARALLEL ALGORITHMS FOR CONVECTION-DIFFUSION PROBLEMS

by

Shawn J.A. Chiappetta

The University of Wisconsin-Milwaukee, 2009
Under the Supervision of Dr. B. A. Wade

Domain decomposition has many benefits that, with the plethora of fast and cheap computers, allow for large problems to be solved in parallel at speeds that were previously unattainable. This dissertation builds on known algorithms, extends and expands methods of solving the general convection-diffusion equation with non-overlapping domain decomposition by numerical methods through modifying the designed interface computation. Maximum principles are employed for stability and error estimates of the algorithms and several numerical studies of performance are developed.

Major Professor Date

Table of Contents

1 Introduction **1**

1.1 The Partial Differential Equation 5

1.2 Parallel Computing 8

1.3 Domain Decomposition 12

1.4 Current Results 18

2 Algorithms **31**

2.1 Algorithm 1, CEIDD-Hyb 38

2.2 Algorithm 2, CEIDD-Exp1 50

2.3 Algorithm 3, CEIDD-Exp2 55

2.4 Algorithm 4, CEIDD-Lex 63

3 Numerical Results **67**

3.1 Setup 67

3.2 Example 1 69

3.3 Example 2 72

3.4 Example 3 76

3.5 Example 4 79

3.6 Conclusions 82

4 Open Problems **85**

4.1 Open Problems 85

4.2 Algorithm 5, CEIDD-Exp3d 87

A MPICH **98**

A.1 MPICH 98

B Source Code **103**

B.1 be2d.c 103

B.2 ceidd_hyb.c 112

B.3 ceidd_exp1.c 128

B.4 ceidd_exp2.c 143

B.5 ceidd_lex.c 159

List of Figures

1.1 Overlapping Domains 14

1.2 Non-Overlapping Domains 16

1.3 Explicit Computation of Interface Value Using Forward Euler 18

2.1 1D Stencil for Daoud, Khaliq, and Wade algorithm [11] 32

2.2 Five-Point Stencil. 33

2.3 Stencils used for CEIDD-Hyb. 33

2.4 Stencils used for CEIDD-Exp1. 35

2.5 Stencils used for CEIDD-Exp2. 36

2.6 Stencils used for CEIDD-Lex. 37

2.7 Orientation of the subdomains. 39

2.8 Stencil for predictor of CEIDD-Hyb. 40

2.9 Stencil for corrector of CEIDD-Hyb. 41

2.10 Stencils used for the predictor of CEIDD-Exp1. 51

2.11 Stencil for interior predictor of CEIDD-Exp2. 55

2.12 Stencil for corrector of CEIDD-Exp2. 57

2.13 Stencils used for predictor with CEIDD-Lex. 65

3.1 Graph of $u(x, y, t) = e^{-2t} \sin(\pi x) \sin(\pi y)$ at $T = 1.0$. 70

3.2 Graph of $u(x, y, t) = e^{-2t}\cos(3x + y)$ at $T = 1.0$. 73

3.3 Graph of $u(x, y, t) = e^{t}\sin(2\pi x)\sin(\pi y)$ at $T = 1.0$. 77

3.4 Graph of $u(x, y, t) = e^{-2t}\sin(x)\sin(y)$ at $T = 1.0$. 80

4.1 Stencil produced by using CEIDD-Exp3D. 88

List of Tables

3.1 Maximum Errors for Example 1 at $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors. 70

3.2 Maximum Errors for Example 1 at $T = 1.0$ and $\triangle t = 0.001$ with 4 Processors. 70

3.3 Processing Times (in seconds) for Example 1 with $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors. 71

3.4 Maximum Errors for Example 1 at $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors. 71

3.5 Maximum Errors for Example 1 at $T = 1.0$ and $\triangle t = 0.5h$ with 4 Processors. 72

3.6 Processing Times (in seconds) for Example 1 with $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors. 72

3.7 Maximum Errors for Example 2 at $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors. 74

3.8 Maximum Errors for Example 2 at $T = 1.0$ and $\triangle t = 0.001$ with 4 Processors. 74

3.9 Processing Times (in seconds) for Example 2 with $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors. 75

3.10 Maximum Errors for Example 2 at $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors. 75

3.11 Maximum Errors for Example 2 at $T = 1.0$ and $\triangle t = 0.5h$ with 4 Processors. 75

3.12 Processing Times (in seconds) for Example 2 with $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors. 76

3.13 Maximum Errors for Example 3 at $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors. 77

3.14 Maximum Errors for Example 3 at $T = 1.0$ and $\triangle t = 0.001$ with 4 Processors. 77

3.15 Processing Times (in seconds) for Example 3 with $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors. 78

3.16 Maximum Errors for Example 3 at $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors. 78

3.17 Maximum Errors for Example 3 at $T = 1.0$ and $\triangle t = 0.5h$ with 4 Processors. 78

3.18 Processing Times (in seconds) for Example 3 with $T = 1.0$ and $\triangle t = 0.5h$ with 2 rocessors. 79

3.19 Maximum Errors for Example 4 at $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors. 80

3.20 Maximum Errors for Example 4 at $T = 1.0$ and $\triangle t = 0.001$ with 4 Processors. 81

3.21 Processing Times (in seconds) for Example 4 with $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors. 81

3.22 Maximum Errors for Example 4 at $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors. 81

3.23 Maximum Errors for Example 4 at $T = 1.0$ and $\triangle t = 0.5h$ with 4 Processors. 82

3.24 Processing Times (in seconds) for Example 4 with $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors. 82

4.1 Errors and Timings for Example 1 with 2 Processors Varying the Coarse Grid Size. 86

Chapter 1

Introduction

It is highly desirable to find numerical methods to solve partial differential equations that do not have stability restrictions on the relationship between the spatial and temporal discretizations, like forward Euler. If this cannot be accomplished, it would be acceptable to have a method that minimizes the restriction as much as possible. Along with standard methods of numerical computation, there is movement to find methods that can be parallelized to take advantage of advances in computer processing power. One would like to take advantage of the lowered cost of processing power and couple with parallel code to run numerical experiments at a fraction of the time and cost as the identical serial code would take. Unfortunately, transforming serial code into parallel code uncovers issues focused on handling the interface between the subdomains created by the domain decomposition.

We would like to solve partial differential equations of convection-diffusion type numerically by using domain decomposition methods. The trouble one faces when implementing domain decomposition methods is precisely the handling of the interface, also called the artificial boundary, between two or more subdomains. The primary problem that must be overcome is that natural methods to resolve the interface, like forward Euler, deal with implementing some type of explicit computation which imposes an undesirable constraint on the size of the time step in order to maintain stability.

The interface problem has been looked at from two major perspectives. The first deals with the interface by using overlapping subdomains. In this case, the direction of current research deals with handling nonmatching grids in the overlapped zones (see, for example, [9, 25, 44]). The second, and more relevant perspective for purposes of the dissertation, is dealing with the interface using non-overlapping subdomains.

From this perspective, we look to find algorithms that minimize the information that must be sent to compute the interface whether it be for the elliptic or parabolic problem. The variety of methods that have been implemented range from using Lagrangian multiplier techniques [29, 43] to finite element methods [18, 30]. In our case, we are interested in extending and expanding the results from [13]. The paper introduces a method of computing the interface by implementing a coarse grid as a way to reduce the affect of using an explicit computation which imposes a constraint on the size of the time step. The notion of the coarse grid and the proof techniques introduced in [13] allowed for the ability to use maximum principles to prove error estimates on the parallel algorithms.

The coarse grid was then used to apply the concept of the corrected explicit-implicit domain decomposition [11, 38, 48]. In each of these papers, the authors work to extend the work in [13] by introducing proof techniques to handle the interface to varying degree. In [48], the authors attempt to rework the results from [13] using operators in hopes to create a more robust method of proof while using the same course grid. Unfortunately, as pointed out in [38], the proofs were incomplete and decreased the flexibility of the domain partitioning. While [38] was able to prove unconditional stability, the authors need to implement an explicit-implicit alternating domain decomposition which increases the complexity of the algorithm and can only be shown to work for the heat equation and not a more general differential equation. Another result that we look to is [15] which give methods using a multistep second-order explicit scheme and a one-step high-order scheme. These schemes use the proof technique from [13] as a basis.

An issue that arises with the results of [13], along with extensions like [15, 2], is the dependency of constructing a Green's function to push their proofs through. Whereas [11] introduces a method of proof that removes the need for the construction of Green's function and also extends the problem type to the convection-diffusion equation rather than the limited heat equation. The algorithms introduced in [11] were different from [13] with respect to needing the addition of an implicit corrector in addition to the coarse grid explicit predictor. The algorithms were also shown only for the one-dimensional case.

The aim of the dissertation is to develop and implement four new domain decomposition methods for the convection-diffusion PDE in two-dimensions. The four new methods are constructed by implementing an explicit predictor/implicit corrector to update the interface between the artificial boundaries for each pair of adjacent subdomains at each time step all the while using the coarse grid to minimize the effect of the explicit scheme used for the predictor. The algorithms improve on the implementation of the coarse grid introduced in [13] by removing the need of constructing a Green's function, not requiring strip domains and extending the classes of problems one can solve to the convection-diffusion equations. In regards to [11], we modify the algorithms and proof techniques to work with two-dimensional problems. The last algorithm introduced improves on the results from [15] by creating an algorithm that keeps the order of the scheme the same, but the implementation is made easier by using a simple linear extrapolation for the explicit predictor phase.

The dissertation is structured with Section 1.1 introducing the convection-diffusion equation and lays the theoretical groundwork for using the maximum principle as a basis for proving the presented results. Section 1.2 introduces some history of parallel computing along with some expectations one must keep when dealing with parallel computations. Section 1.3 gives a background on domain decomposition, including the difference between overlapping and non-overlapping domain decomposition. We then give a summary of results from Dawson, Du and Dupont [13] and Daoud, Khaliq and Wade [11]. Finishing the chapter, we look at other results related to domain decomposition methods and discuss strengths and weaknesses of those methods in regards to the convection-diffusion problem.

Chapter 2 introduces four new parallel algorithms that differ in their handling of the interface between the subdomains. Section 2.1 gives an extension of the one-dimensional method of [11] with proofs of a maximum principle and an error estimate. The algorithm employs a computation on the interface that is part implicit and part explicit. The method allows for a relaxation of the temporal constraint arising from the explicit portion by employing the course grid originated in [13]. Section 2.2 gives a modification of the first scheme by changing the explicit interface computation to one that extends the use of the explicit coarse grid computation to more of the interface. Moving to Section 2.3, we introduce, state and prove an error estimate for an algorithm that rotates the five-point Laplacian finite difference formulas so as to use the diagonal elements off of the interface. The last section in Chapter 2 introduces an algorithm where linear extrapolation is used for the explicit interface computation with an improved constraint on the relationship between the spatial and temporal values than what is currently published and proves a corollary showing the interrelated nature of two of the algorithms with this last one.

Chapter 3 begins with a short description of the setup of a Beowulf cluster along with a short introduction of the protocol used in the programs. Four test problems are introduced and results are compared using their l_∞ errors as well as looking at timings between the different algorithms versus the standard backward Euler serial algorithm. We finish with Chapter 4 where suggestions are made for further research. One direct consequence is given as a direct extension of one of the two-dimensional algorithms to three-dimensions. The Appendix, for those interested, gives a more technical detail to the Message Passing Interface (MPI) used, MPI-Chameleon (MPICH) [24], along with the source code used to run the experiments.

1.1 The Partial Differential Equation

We start by considering the following parabolic convection-diffusion problem: Find $u(x,t)$, such that

$$\begin{aligned} \partial u/\partial t + Lu &= f && \text{in } \Omega \times [0,T], \\ u(x,t) &= 0 && \text{on } \partial\Omega \times [0,T], \\ u(x,0) &= u_0(x) && \text{in } \Omega, \end{aligned} \tag{1.1}$$

where $\Omega \subset \mathbb{R}^d$, $d = 1, 2$ or 3 is a bounded, connected set with continuous boundary $\partial\Omega$ and L has the following form

$$Lu = -\sum_{i,j=1}^{d} \frac{\partial}{\partial x_i}(\alpha_{ij}(x,t)\frac{\partial u}{\partial x_j}) + \sum_{i=1}^{d} \beta_i(x,t)\frac{\partial u}{\partial x_i} + \gamma(x,t)u. \tag{1.2}$$

We also need L to be uniformly elliptic, meaning for any point $x \in \Omega$ there exists a constant $\mu_0 > 0$ such that

$$\sum_{i,j=1}^{n} \alpha_{i,j}(x,t)\xi_i\xi_j \geq \mu_0 \sum_{i=1}^{n} \xi_i^2$$

for all n-tuples of real numbers $\xi = (\xi_1, \xi_2, \ldots, \xi_n)$. Also, we assume the coefficients $\alpha_{i,j}(x,t)$, $\beta_i(x,t)$ and $\gamma(x,t)$ are continuous in $\Omega \times [0,T]$ and $\gamma(x,t) \leq 0$ in $\Omega \times [0,T]$. Let the matrix formed by $\alpha_{i,j}(x,t)$ be positive definite. Lastly, it is assumed that $u(x,t)$ has continuous second derivatives in space and continuous first derivative in time. On this point, we will adopt the notation of [17] which defines that $u(x,t) \in C_1^2$ precisely when u satisfies the conditions just stated.

A useful tool that will be used in the dissertation is that of the maximum principle. This principle generalizes the notion that any twice-differentiable function $f(x)$ which satisfies the inequality $f'' > 0$ on an interval achieves its maximum value at one of the endpoints of the interval. The maximum principle allows us to obtain information about the solutions of differential equations without any explicit knowledge about the solution themselves which is exactly the situation we look to remedy by using numerical methods to find an approximate solution to the differential equation.

For the remainder of this section, we present a weak maximum principle and Harnack's Inequality to set up a strong maximum principle and conclude with a result in the form of a statement of existence and uniqueness of the solution to the the parabolic convection-diffusion equation. We introduce the notation $\Omega_T := \Omega \times (0,T]$ and $\Sigma = \Omega \times \{t = 0\} \cup \partial\Omega \times [0,T]$.

Theorem 1.1 (Weak Maximum Principle for $\gamma \geq 0$, [17]). *Let Γ be a bounded, connected subset in $\mathbb{R}^d$, d =1, 2 or 3. Assume $u \in C_1^2(\Omega_T) \cap C(\overline{\Omega}_T)$ and $\gamma \geq 0$ in Ω_T.*

1. If $\partial u/\partial t + Lu \leq 0$, in Ω_T, then

$$\max_{\overline{\Omega_T}} u(x,t) \leq \max_{\Sigma} u(x,t).$$

2. Likewise, if $\partial u/\partial t + Lu \geq 0$, in Ω_T, then

$$\min_{\overline{\Omega}_T} u(x,t) \geq \min_{\Sigma} u(x,t).$$

To obtain a stronger version of the previous theorem, we need Harnack's Inequality for the parabolic problem. Harnack's Inequality states if u is a nonnegative solution of the parabolic PDE, then the maximum of u in some interior region can be estimated by the minimum of u in the same region at a later time.

Theorem 1.2 (Harnack's Inequality, [17]). *Assume $u \in C_1^2(\Omega_T)$ solves*

$$\partial u/\partial t + Lu = 0, \textit{in } \Omega_T, \tag{1.3}$$

and $u \geq 0$ in Ω_T. Suppose $V \subset\subset \Omega$ is connected. Then for each $0 < t_1 < t_2 \leq T$, there exists a constant C such that

$$\sup_V u(\cdot, t_1) \leq C \inf_V u(\cdot, t_2). \tag{1.4}$$

The constant C depends only on V, t_1, t_2 and the coefficients of L.

Using Harnack's Inequality, the following result can be shown:

Theorem 1.3 (Strong maximum principle for $\gamma \geq 0$, [17]). *Assume $u \in C_1^2(\Omega_T) \cap C(\overline{\Omega}_T)$ and $\gamma \geq 0$ in Ω_T. Suppose also that Ω is bounded and connected.*

1. *If $\partial u/\partial t + Lu \leq 0$, in Ω_T, and u attains a nonnegative maximum over $\overline{\Omega}_T$ at a point $(x_0, t_0) \in \Omega_T$, then u is constant on Ω_T.*

2. *Likewise, if $\partial u/\partial t + Lu \geq 0$, in Ω_T, and u attains a nonpositive minimum over $\overline{\Omega}_T$ at a point $(x_0, t_0) \in \Omega_T$, then u is constant on Ω_T.*

The culmination of these theorems yields the following statement:

Theorem 1.4 (Friedman, [20]). *Let L be elliptic in a connected set Ω and the coefficients of L be continuous. Then there exists at most one solution to the boundary value problem*

$$\begin{aligned} \partial u/\partial t + Lu &= f && \text{in } \Omega \times [0,T], \\ u(x,t) &= 0 && \text{on } \partial\Omega \times [0,T], \\ u(x,0) &= u_0(x) && \text{in } \Omega. \end{aligned} \tag{1.5}$$

To review, instead of using energy methods to show existence and uniqueness of a solution to (1.5), we can use the maximum principle to obtain alternate results. The reason one would choose this method is to be able to use the pointwise characteristics of the problem to attain our solutions.

1.2 Parallel Computing

The concept of parallel computing can be considered a natural one. Particularly when viewed that the human mind models the notion of parallel computing as the neurons in the human brain work in concert handling many pieces of information simultaneously. In this section, we give a brief history of parallel computing with a short discussion describing the different hardware architectures for parallel processing, as well as compiler solutions. We finish with some cautions one must keep in mind when using parallel computing.

The history of parallel computing dates back to the late 1950s when S. Gill discussed parallel programming and the need for branching and waiting [21]. While in 1967, Amdahl and Slotnick published a debate [1] about the feasibility of parallel processing. In this debate, the term Amdahl's Law was put into the vernacular as a way to define the limit of speed-up to to parallelism, a topic we will return to as a caution. The C.mmp was an early multiprocessor system developed at Carnegie-Mellon University in 1971 [32] that used sixteen PDP-11 minicomputers as the processing elements. Another project that was started at Caltech and ran from 1983 to 1990 was the Caltech Concurrent Computation Program (C3P). In a multidisciplinary collaboration, a 64-node machine was create and dubbed the Cosmic Cube [19].

While the first commercial clustering product, called ARCnet, was developed in 1977. It was not until an open source software was introduced in 1989, called Parallel Virtual Machine (PVM) and developed by Oak Ridge National Laboratory, Emory University, and the University of Tennessee. With PVM and the subsequent advent of inexpensive networked PCs Donald Becker and Thomas Sterling, in late 1993, outlined a commodity-based cluster system, now known as a Beowulf cluster. These clusters came to overshadow and eventually displace the concept of parallel computing on a supercomputer. This was notably due to NASA's Goddard Space Flight Center, when in 1994, developed the first Beowulf cluster using 16 Intel 100-MHz PCs that were connected by dual 10-Mbps Ethernet LANs [33]. The components, being off-the-shelf, and the developed software tools allowed users to show the performance gains as wells as the cost effectiveness of the Beowulf system for real-world scientific applications. Currently, because consumers continue to become more mobile and energy conscious, manufactures continue to research and develop new methods to add more cores which is more energy efficient than increasing the clock frequencies of the processors [24].

The core elements of parallel processing are CPUs based on the number of instructions and data streams that can be processed simultaneously. Systems can be classified into one of the following four categories: Single Instruction, Single Data (SISD), Single Instruction, Multiple Data (SIMD), Multiple Instruction, Single Data (MISD), and Multiple Instruction, Multiple Data (MIMD).

A SISD system is a single processor machine that is capable of executing a single instruction which operates on a single data stream. Examples of these types of systems are early IBM-PC and Macintoshes. The speed of processing in a SISD system is limited by the rate at which the computer can transfer information internally. A SIMD system, like the CRAY's vector processing machine T80, is a multiprocessor machine that can execute the same instructions on all the CPUs, but operate on different data streams. The MISD system is capable of executing different instructions on different processors, but operates on the same data set. The MISD systems have been built, but since they are not as useful in most applications none have been commercially produced. One such system is the experimental Carnegie-Mellon C.mmp computer. The last category, MIMD, is the group of multiprocessor machines that can be found in off-the-shelf hardware at many stores, like the Intel Core 2 Duo. With the MIMD, the multiprocessor machine can execute multiple instructions on multiple data sets. Since each processor has separate instructions and data streams, the model is well suited for any kind of application. Examples of MIMDs are symmetrical multiprocessors and clusters, like Beowulf clusters.

More discussion is warranted to breakdown the two types of MIMD machines: shared memory and distributed memory. Most machines that are currently sold to consumers fall into that of the shared memory system. In a shared memory model, all the processors have access to the global memory. The communication that takes place goes through the shared memory and is visible to all other processors. Other characteristics of shared-memory MIMD machines can be summarized as follows:

- Easy to build. Conventional operating systems can be easily adapted.
- Easy to program and does not involve much communication overhead.
- Since memory is shared, any issue involving the memory affects the entire system.
- Adding more processors has repercussions due to memory contention. This means that when different processors all want to read or write into the main memory, there is a delay until the memory is free.

The distributed memory MIMD model machine is created so that each processor has its own dedicated memory where communication occurs through the interconnected network hence also given the name "loosely-coupled" multiprocessor system. A similar summary of characteristics are:

- Easy to build, but needs light-weight operating system.
- More difficult to program than shared-memory systems, but is well suited for real-time applications.
- Component failures can be isolated so as not to affect the entire system.
- Adding more processors is much easier when designing the system.

The hardware solutions are only a part of the evolution of parallel computing. Parallel programs are harder to write in contrast to sequential ones. The difficulties arise from the necessary synchronization and communication needed to take place between the tasks that are to be completed. Some standards have emerged starting in the mid 1990s. For massively parallel processors and clusters, a number of application programming interfaces moved to the standard known as Message Passing Interface (MPI). For shared memory multiprocessor computing, a convergence occurred around two standards occurred the late 1990s called pthreads and OpenMP. All of these standards are modifications to conventional, non-parallel languages like C.

We finish the section with a word of caution regarding the possible benefits of parallel computing. One might expect that given n processors that the speed should be increased by n times. Unfortunately, when designing parallel code, one must acknowledge Amdahl's Law which says the maximum speedup, S, one can expect from a parallel algorithm given that a proportion of the code, f, must be computed sequentially is given by

$$S \leq \frac{1}{f + (1-f)/N} \tag{1.6}$$

where N is the number of processors [28]. Since any algorithm using a domain decomposition method for our type of problem will have some serial code, like the subdomain solves, we will never have f approach zero and hence always have a speedup that is less than optimal. With this note, we turn our attention to developing the notions behind domain decomposition.

1.3 Domain Decomposition

Given that parallel computing can obtain performance gains higher than serial computing, we turn our attention to giving the background of how to take our differential equation problem apart and take advantage of our multiprocessor system. Any domain decomposition method is based on taking a given computation domain, Ω, and partitioning into subdomains, Ω_i, $i = 1, \ldots, M$, which may or may not overlap. The original problem is then reformulated upon each subdomain, Ω_i, so that a family of subproblems are created that have reduced size and are coupled to one another through the values of the unknown solution at the subdomain interfaces.

Even with the general definition of domain decomposition there is some interpretation of the statement within different subareas of computational mathematics. For instance, in parallel computing, domain decomposition is referred to as the techniques for decomposing a data structure and can be independent of the numerical solution methods. In preconditioning methods, domain decomposition is the process of subdividing the solution of a large linear system into smaller problems whose problems can be used to create a solver (preconditioner) for the system of equations that results from the discretization of the PDE on the entire domain. Lastly, to those dealing with asymptotic analysis, it means the separation of the physical domain into regions that can be modeled with different equations. In this case, the interface between the domains are handled by various conditions, like continuity. Given these different interpretations, it is very likely that a program that works with one of the subareas does address others as well.

To elaborate on the differences in overlapping and non-overlapping domain decomposition, we look first at describing a simple overlapping algorithm, the alternating Schwarz method. This earliest known domain decomposition was introduced by H.A. Schwarz in 1870, but was not originally intended to be a numerical method. To simplify the description of the Schwarz alternating method, we discuss the method in the context of the linear elliptic PDE

$$-\triangle \mathbf{u} = \mathbf{f} \qquad \text{in } \Omega \tag{1.7}$$

$$\mathbf{u} = \mathbf{g} \qquad \text{on } \partial\Omega. \tag{1.8}$$

using Dirichlet boundary conditions. This point is justified for our discussion since, at a fixed time level, the solving of the parabolic PDE is equivalent to that of an elliptic problem which depends on a time step parameter.

Consider the domain given in Figure 1.1 with $\Omega = \Omega_1 \cup \Omega_2$ for which we want to solve our PDE. Let $\partial\Omega$ denote the boundary of Ω and note that the domains Ω, Ω_1, and Ω_2 do not include their boundaries. Also let $\overline{\Omega} = \Omega \cup \partial\Omega$ denote the closure of the domain. The artificial boundaries, Γ_i, are the part of the boundary of Ω_i that is interior to Ω. We denote the points on $\partial\Omega_i$ that are not on Γ_i by $\partial\Omega_i/\Gamma_i$.

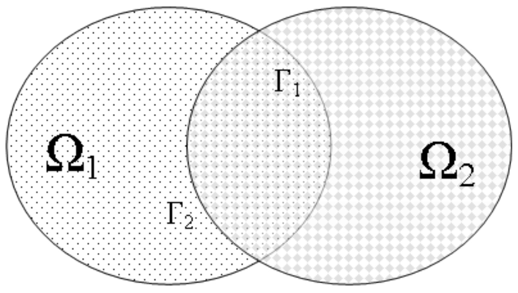


Figure 1.1: Overlapping Domains

Next, let $\mathbf{u}_i^n$ denote the approximate solution on $\overline{\Omega}_i$ after n iterations and $\mathbf{u}_1^n|_{\Gamma_2}$ be the restriction of $\mathbf{u}_1^n$ to Γ_2 and similarly define $\mathbf{u}_2^n|_{\Gamma_1}$. The Schwarz alternating method begins by selecting an initial guess $\mathbf{u}_2^0$, for values in Ω_2. Then, iteratively for

$n = 1, 2, 3, \ldots$, one solves the boundary value problem,

$$\begin{aligned} -\triangle \mathbf{u}_1^n &= \mathbf{f} && \text{in } \Omega_1 \\ \mathbf{u}_1^n &= \mathbf{g} && \text{on } \partial\Omega_1/\Gamma_1, \\ \mathbf{u}_1^n &= \mathbf{u}_2^{n-1}|_{\Gamma_1} && \text{on } \Gamma_1 \end{aligned} \tag{1.9}$$

for $\mathbf{u}_1^n$. This is followed by the solution of the boundary value problem,

$$\begin{aligned} -\triangle \mathbf{u}_2^n &= \mathbf{f} && \text{in } \Omega_2 \\ \mathbf{u}_2^n &= \mathbf{g} && \text{on } \partial\Omega_2/\Gamma_2, \\ \mathbf{u}_2^n &= \mathbf{u}_2^{n-1}|_{\Gamma_1} && \text{on } \Gamma_2. \end{aligned} \tag{1.10}$$

This process reduces to stating that at each half-step, we solve the elliptic boundary value problem on the subdomain Ω_i with the given boundary values, $\mathbf{g}$, on the true boundary $\partial\Omega_i/\Gamma_i$, and the previous approximate solution on the interior boundary Γ_i.

While this method can be parallelized, one immediately runs into a computational bottleneck. Since each half-step is dependent on the previous half-step, meaning there is a need to move a large amount of information from one subdomain to another. A possible solution would be to remove the overlap. To illustrate this non-overlapping domain decomposition method consider the domain given in Figure 1.2 again with $\Omega = \Omega_1 \cup \Omega_2$ for which we want to solve our PDE. In this situation, we are sharing a boundary Γ_1 and have that $\Omega_1 \cap \Omega_2 = \emptyset$. Begin by selecting an initial guess $\mathbf{u}_i^0$ for values in both Ω_1 and Ω_2. Then, iteratively for $n = 1, 2, 3, \ldots$, one solves the boundary value problem,

$$\begin{aligned} -\triangle \mathbf{u}_i^n &= \mathbf{f} && \text{on } \Gamma_1 \\ \mathbf{u}_i^n &= \mathbf{g} && \text{in } \partial\Omega_i/\Gamma_1, \\ -\triangle \mathbf{u}_i^n &= \mathbf{f} && \text{in } \Omega_i \text{ in parallel.} \end{aligned} \tag{1.11}$$

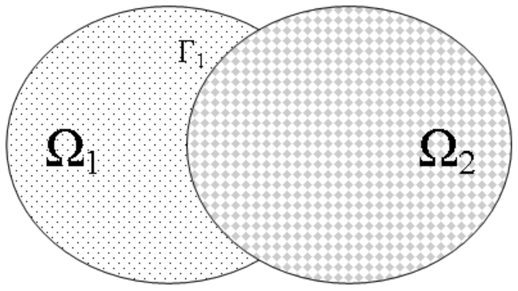


Figure 1.2: Non-Overlapping Domains

The non-overlapping method has the benefit that the dependency on the other subdomain to be solved for the previous half-step has been removed allowing for true parallelism in the implementation of the algorithm. While we are able to describe the difference between the overlapping and non-overlapping schemes relatively easy, hidden is the computation of the interface values. The overlapping methods handle the computation of the interface by creating a larger problem to solve where the interface is interior of the overlap and then update the interface. Whereas, in non-overlapping domain decompositions, we do not have the larger data set available to compute the interface. In these cases, we must implement a scheme that directly computes the interface using available data. An example of this can be explained using the one-dimensional heat equation.

$$\begin{aligned} u_t - u_{xx} &= 0 && x \in (0,1), t \in (0,T], \\ u(x,0) &= u_0(x) && x \in (0,1), \\ u(0,t) = u(1,0) &= 0 && t \in (0,T]. \end{aligned} \tag{1.12}$$

If we were to solve this problem numerically on a serial machine, we might choose to use backward Euler to take advantage of its stability and lack of a constraint on the step size in time. Suppose we split the domain into two pieces, say $(0, x_p)$ and $(x_p, 1)$ and define x_p as our interface value. Our goal would be to compute the next time step on each subdomain independently of each other. This seems relatively straightforward until one realizes that a value for our interface is needed before the subdomain solve, as the value becomes our artificial boundary condition, a necessity to solve the problem. A natural choice under these conditions would to use an explicit scheme to compute the interface. In Figure 1.3, we see what the stencil of the predictor using forward Euler. This particular method gives one the value for the artificial boundary and allows us to proceed to solve for the interior subdomain values. This type of solution has the downside of introducing a constraint on the size of the time step due to the explicit nature of the remedy. In particular, we have now introduced the constraint that

$$\triangle t \leq \frac{1}{2} h^2,$$

where $\triangle t$ is our time step and h is our space step. By having this constraint we lose much of the benefit of implementing a domain decomposition method due to the necessity of requiring more time steps to guarantee stability and the constraint negates the use of backward Euler for the subdomain solves because we could just use forward Euler for the entire domain, a much simpler proposition.

The next section introduces some of the relevant results that are directed at dealing

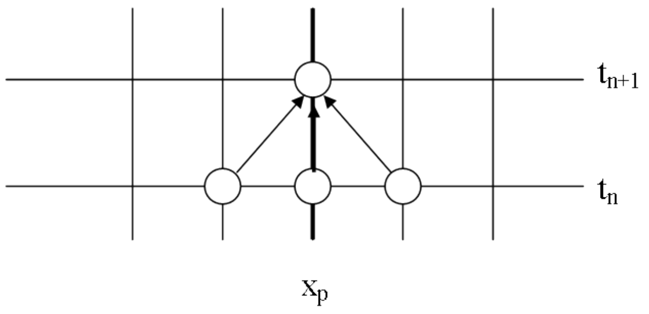


Figure 1.3: Explicit Computation of Interface Value Using Forward Euler

with the interface computation in ways that minimize the problems produced by the explicit time step constraint on the interface.

1.4 Current Results

In 1991, Dawson, Du and Dupont [13] introduced a non-overlapping domain decomposition method for the heat equation in the one-dimensional case

$$\begin{aligned} u_t - u_{xx} &= 0 && x \in (0,1), t \in (0,T], \\ u(x,0) &= u_0(x) && x \in (0,1), \\ u(0,t) = u(1,0) &= 0 && t \in (0,T]. \end{aligned} \tag{1.13}$$

The purpose of their inquiries was to create a finite difference method to address the non-overlapping domain decomposition rather than the finite element methods used at the time [5, 12, 22]. They were also interested in finding a method to handle the interface computation to reduce the impact of the inherited constraint from the explicit computation. The case they present is a much simpler situation, dealing only with the heat equation, than what we would like to focus in this dissertation, but the results that Dawson, Du and Dupont attained are directly related to the direction we wish to take handling the convection-diffusion equation.

It is necessary to introduce notations and naming conventions that will reoccur throughout the dissertation. For a positive integer N, let $h = 1/N$ and take $x_i = i * h$, $i = 0, 1, \ldots, N$. Define our interface value $\overline{x} \in (0, 1)$ and assume that $\overline{x}$ and N are such that $\overline{x} = x_K > 0$ for some integer $K < N$. A related parameter H is defined so that $H/h = q \geq 1$ is an integer and $0 < H < \min\{\overline{x}, 1 - \overline{x}\}$, giving a *coarse grid.* Take $k = T/M$, where M is a positive integer, and $t_n = n * k$. Take u_i^n to be the discrete set of approximations to $u(x_i, t_n)$ as defined by the algorithms. Define the difference operators

$$\begin{aligned}
\delta_k u_i^n &= (u_i^n - u_i^{n-1})/k \\
\nabla_{x,h} u_i^n &= (u_{i+1}^n - u_{i-1}^n)/2h \\
\triangle_{x,h} u_i^n &= (u_{i+1}^n - 2u_i^n + u_{i-1}^n)/h^2.
\end{aligned}$$

We will refer to the points $(x_i, t_n) \in \partial\Omega$ as *boundary points.* The artificial boundary will again be denoted as Γ_i where $(x_i, t_n) \in \Gamma_i$ where $x_i = \overline{x}$ are *interface points.* Otherwise, they are *interior points.*

The algorithm that Dawson, Du and Dupont introduced in [13] is defined as:

$$\begin{aligned}
u_i^n &= u(x_i, t_n) && \text{on } \partial\Omega \\
\delta_k u_i^n - \triangle_H u_i^{n-1} &= 0 && \text{on } \Gamma_i \\
\delta_k u_i^n - \triangle_h u_i^n &= 0 && \text{in } \Omega_i
\end{aligned} \tag{1.14}$$

The approximation is given by an explicit forward difference formula on the interface, while on the interior of the subdomains the algorithm satisfy an implicit backward difference equation. Because of the use of an explicit method on the interface, the paper, [13], proves the algorithm has the constraint

$$k \leq \frac{1}{2}H^2. \tag{1.15}$$

Notice that the constraint is based on size of the coarse grid parameter H not the fine grid parameter h. Some information from the adjacent subdomain is necessary to compute the value of the interface, but once computed, they create two decoupled backward difference problems to be solved, both of which can be done in parallel.

In [13], the authors proved the following theorem.

Theorem 1.5 (Dawson, Du, Dupont, [13]). *Suppose* $\frac{1}{2}|\partial^2 u/\partial t^2|$ *and* $\frac{1}{12}|\partial^4 u/\partial t^4|$ *are bounded by* C_0 *on* $\Omega \times [0,T]$. *Suppose also that* $k \leq H^2/2$. *Then*

$$\max_{i,n} |u(x_i,t_n) - u_i^n| \leq \frac{C_0}{4}(k + h^2 + H^3). \tag{1.16}$$

The introduction of (1.15) given in [13] is important because the constraint is now dependent on the coarse grid defined by H and not the fine grid defined by h. Notice that by the constraint of the explicit computation on the fine grid, we are forced to choose $k = O(h^2)$ which destroys the benefits of the implicit scheme used on the subdomains. We would like to let both k and h go to zero with $k = O(h)$ to regain those benefits. In Theorem 1.5, we are left with an estimate of $O(k+h^2+H^3)$ which, because $h << H$, yields a worst case scenario of $O(k + H^3)$. This fact allows us choose $H \approx \sqrt[3]{k}$ and decouple the driving parameter k from the fine grid h which is where the decrease in performance arose from.

The crux of the proof relies on two concepts. The first is to prove the maximum principle stated in Lemma 1.6 below and the ability to construct a specialized Green's function to push the result through.

Lemma 1.6 (Dawson, Du, Dupont, [13]). *Suppose that $k \leq H^2/2$ and that z_i^n satisfied the following relations:*

$$\begin{aligned} z_i^n &\leq 0 && \textit{at boundary points,} \\ \delta_k z_i^n - \triangle_{x,H} z_i^{n-1} &\leq 0 && \textit{at interface points,} \\ \delta_k z_i^n - \triangle_{x,h} z_i^n &\leq 0 && \textit{at interior points.} \end{aligned}$$

Then, for each i and n,

$$z_i^n \leq 0.$$

Proof of Theorem 1.5. Let $e_i^n = u(x_i, t_n) - u_i^n$ be the discrete error, then Dawson, Du and Dupont shows the scheme gives the error equations:

$$\begin{aligned} e_i^n &\leq 0 && \text{on } \partial\Omega_i, \\ \delta_k e_i^n - \triangle_{x,H} e_i^{n-1} &= K_i^n(k + H^2) && \text{on } \Gamma_i, \\ \delta_k e_i^n - \triangle_{x,h} e_i^n &= K_i^n(k + h^2) && \text{in } \Omega_i. \end{aligned}$$

where $|K_i^n| \leq C_0$.

They proceed to solve for the Green's functions θ_i and β_i constrained by

$$-\partial_{x,h}^2 \theta_i = 1 \qquad 0 < i < N$$

and

$$-\partial_{x,h}^2 \beta_i = 0 \qquad 0 < i < N, i \neq K$$
$$-\partial_{x,H}^2 \beta_K = 1$$

where K is the interface index. Note that β_i's have requirements on the interface and in the subdomains.

Next, bounds are found for the functions. In particular,

$$\begin{aligned} 0 \leq \theta_i \leq \tfrac{1}{8} \\ 0 \leq \beta_i \leq \tfrac{H}{4}. \end{aligned} \tag{1.17}$$

Coupled with the maximum principle, the bounds from (1.17) give an upper estimate of the discrete error of the form

$$\xi_i = C_0[\theta_i(k + h^2) + \beta_i(k + H^2)] \tag{1.18}$$

yielding the result in (1.16). $\square$

Dawson, Du and Dupont extended their algorithm to approximate numerical solutions, u, of the heat equation on $\Omega = (0,1)^2$ satisfying

$$\begin{aligned} u_t - \triangle u &= 0 && (x,y) \in \Omega, t \in (0,T] \\ u(x,y,0) &= u_0(x,y) && (x,y) \in \Omega \\ u(x,y,t) &= 0 && (x,y) \in \partial\Omega, \end{aligned} \tag{1.19}$$

where $\triangle u = u_{xx} + u_{yy}$, by defining the algorithm using the following relations:

$$\begin{aligned} u_{i,j}^n &= u(x_i, y_j, t_n) && \text{on } \partial\Omega_i \\ \delta_k u_{i,j}^n - \triangle_{x,H} u_{i,j}^{n-1} - \triangle_{y,h} u_{i,j}^n &= 0 && \text{on } \Gamma_i \\ \delta_k u_{i,j}^n - \triangle_{x,h} u_{i,j}^n - \triangle_{y,h} u_{i,j}^n &= 0 && \text{in } \Omega_i. \end{aligned} \tag{1.20}$$

where $\triangle_{y,h} u_{i,j}^n$ is defined by

$$\triangle_{y,h} u_{i,j}^n = (u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n)/h^2. \tag{1.21}$$

Notice on the interface, the computation is partially implicit (in the y-direction) due to the implementation of $\triangle_{y,h}^{n} u_{i,j}^{n}$. By using a hybrid scheme from the implicit portion of the corrector, we add the task of finding a solution of a tridiagonal set of equations at each time step. Compared to the relative sizes of the subdomain solves, these interface computations are relatively small (solving an $n \times n$ matrix for the tridiagonal system versus an $n^2 timesn^2$ matrix for the subdomain solves). The proof of the error estimate of this algorithm uses an analogue of Lemma 1.6 and, because of the construction of the algorithm, the same one-dimensional Green's functions are used.

By using the the coarse grid, Dawson, Du and Dupont were able to relax the stability constraint involving the relationship of the spatial and temporal mesh sizes from using the explicit method on the interface, a major problem when using explicit schemes. While the method Dawson, Du and Dupont introduced had many benefits, there were some issues that others tried to overcome. One such issue comes from trying to extend the scheme to the general convection-diffusion equation (1.5). Because of the need to create a discrete Green's function for the proof method in [13], trying to extend this method to other types of PDEs becomes difficult since, in essence, we are trying to find a left inverse operator for L (further discussions can be seen in [14, 23, 31]). Dawson, Du and Dupont did obtain some results in this direction by limiting their extension to modifying the computation along the interface to one that is partially implicit and partially explicit. Unfortunately, one would like to create a method that allows for a completely explicit computation along the interface.

Daoud, Khaliq and Wade [11] were able to take the results of Dawson, Du and Dupont, extend their methodology and overcome some of the limitations. In particular, they were able to extend the algorithm from the heat equation to the more general convection-diffusion equation. Next, they introduced a technique of proof that removed the need for the discrete Green's function, so pivotal in [13]. A side effect caused by the new method of proof is the change in relationship between the fine mesh, coarse mesh and the time step sizes. As seen in [13], Dawson, Du and Dupont showed, in the worst case, one would have a relationship that

$$k \approx H^3, \tag{1.22}$$

whereas, Daoud, Khaliq and Wade [11] yield the relationship, again in the worst case, that

$$k \approx H^2. \tag{1.23}$$

As mentioned previously, the relationship between k and H controls the constraint arising from the explicit computation on the interface. So (1.22) allows us to choose $k = O(H^3)$ versus (1.23) which lets us choose $k = O(H^2)$, the latter being a little worse constraint, but still much better than not using the coarse grid since $h << H$ and we would be forced to choose $k = O(h^2)$ from forward Euler on the fine grid. In either case, by incorporating the coarse grid overlay, we are able to take advantage of increasing the time step without detriment to the computation.

To describe the scheme of Daoud, Khaliq and Wade [11], we introduce the following finite difference operators:

$$\begin{aligned} \triangle_{x,h} u_i^n &= (u_{i-1}^n - 2u_i^n + u_{i+1}^n)/h^2 && (1.24) \\ \bigtriangledown_{x,h} u_i^n &= (u_{i+1}^n - u_{i-1}^n)/2h && (1.25) \\ \delta_k u_i^n &= (u_i^n - u_i^{n-1})/k && (1.26) \end{aligned}$$

and

$$L_{x,h}u_i^n = -\alpha_i^n \triangle_{x,h} u_i^n + \beta_i^n \cdot \nabla_{x,h} u_i^n + \gamma_i^n u_i^n \tag{1.27}$$

where $\alpha_i^n = \alpha(x_i, t_n)$, $\beta_i^n = \beta(x_i, t_n)$, $\gamma_i^n = \gamma(x_i, t_n)$ and $f_i^n = f(x_i, t_n)$. Then the method of Daoud, Khaliq and Wade [11] is defined as:

First, set $u_i^0 = g$. Then for $n = 1, 2, \ldots, N$

1. Compute the interface using an explicit predictor on the coarse grid and assign the value to the temporary placeholder u_*^n.

$$u_p^n = u_p^{n-1} - kL_{x,H}u_p^{n-1} + kf_p^{n-1} \tag{1.28}$$

2. Solve each subdomain $\Omega_1, \Omega_2, \ldots$ in parallel using an implicit solver with a fine grid.

$$\delta_k u_i^n + L_{x,h}u_i^n = f_i^n \tag{1.29}$$

3. Discard the predicted interface u_*^n and correct the interface using information from the newly computed subdomains using an implicit method on the fine grid.

$$\delta_k u_p^n + L_{x,h}u_p^n = f_p^n. \tag{1.30}$$

The differences of the implementation of this method in contrast to [13] materialize as the generalization of the type of problem one is able to handle (now, the convection-diffusion equation) and that a corrector is implemented in the last step, creating an explicit predictor/implict corrector scheme. Recall that Dawson, Du and Dupont [13] only solved for the heat equation and not for any other type of parabolic PDE. The method from Daoud, Khaliq and Wade [11], on the other hand, can deal with the more general convection-diffusion PDE.

Before stating the result from [11], remember the requirements on the coefficients for the parabolic PDE are that the functions must be continuous and bounded on the domain. This results in the existence of constants $\overline{\alpha}$, $\underline{\alpha}$, $\overline{\beta}$ and $\overline{\gamma}$ such that for all $x \in \Omega$ and $t \in (0, T]$

$$\underline{\alpha} \leq \alpha(x,t) \leq \overline{\alpha}, \tag{1.31}$$

$$\beta(x,t) \leq \overline{\beta}, \text{ and} \tag{1.32}$$

$$\gamma(x,t) \leq \overline{\gamma}. \tag{1.33}$$

We now state the main result from [11].

Theorem 1.7 (Daoud, Khaliq, Wade, [11]). *Suppose u is sufficiently smooth and the following conditions hold:*

$$k\overline{\gamma} < 1, H \geq \sqrt{\frac{2\underline{\alpha}k}{1-\overline{\gamma}k}}, \text{ and } h \leq H \leq \frac{2\underline{\alpha}}{\overline{\beta}}. \tag{1.34}$$

Then there exists a constant c, independent of the grid, such that for $(x_i, t_n) \in \Omega$

$$|u(x_i,t_n) - u_i^n| \leq c(k + h^2 + H^2). \tag{1.35}$$

Just as Dawson, Du and Dupont needed a maximum principle to prove their error estimate, [11] did also.

Lemma 1.8 (Daoud, Khaliq, Wade, [11]). *Suppose for $n = 1, 2, \ldots, N$*

$$\begin{aligned} \delta_k z_i^n + L_{x,h} z_i^n \quad &= \quad g_i^n, \\ z_0^n = a_n \quad \text{and} \quad &z_p^n = b_n, \end{aligned} \tag{1.36}$$

Then for each time level t_n the following estimate must hold:

$$\max_i |z_i^n| \leq \max_i \{|a^n|, |b^n|\} + k \max_i \{|g_i^n|\}. \tag{1.37}$$

Even though the scheme was designed for the one-dimensional convection-diffusion equation, the proof techniques used in [11] are more generally applicable and give hope of a reasonable extension to higher dimensional cases.

In 2001, Du, Mu and Wu [15] used the method of proof from Dawson, Du and Dupont [13] and created an algorithm that handles the interface computation with a higher order extrapolation technique for the one-dimensional heat equation. The method is defined similar to [13] by

$$\begin{aligned} u_i^n &= u(x_i, t_n) && \text{on } \partial\Omega_i \\ \delta_k u_i^n &= a\triangle_{x,H} u_p^{n-1} + b\triangle_{x,2H} u_p^{n-1} && \text{on } \Gamma_i \\ \delta_k u_i^n - \triangle_{x,h} u_i^n &= 0 && \text{in } \Omega_i \end{aligned} \tag{1.38}$$

where $a = 4/3$ and $b = -1/12$.

The paper shows that the method is stable in the sense of Osher for $0 < \omega \leq 4/11$ shown in the following theorem.

Theorem 1.9 (Du, Mu, Wu, [15])**.** *For* (1.38)*, under the stability condition* $0 < \omega \leq 4/11$*, there exists a constant* c*, independent of* k *and* h*, such that*

$$|u(x_i, t_n) - u_i^n| \leq c(k + h^2 + H^5) \quad \forall x_i \in \Omega. \tag{1.39}$$

Due to the fact that a higher order scheme is needed to achieve the stability, one hopes to find an intermediate solution that allows for an interface method that is of similar order as those in [13] and [11]. The method in Du, Mu and Wu [15] also suffers the same limitations as Dawson, Du and Dupont [13] as the method presented only works with the heat equation in one- and two-dimensions instead of the general convection-diffusion equations due to the need to solve for similar discrete Green's functions found in [13].

One area that domain decomposition researchers have worked on is a standardized notation and terminology. In Zhuang and Sun [48], the authors introduced similar schemes as Dawson, Du and Dupont [13]. What was changed was notation and terminology, instead of a corrector, the authors defined a 'stabilizer' which is supposed to be chosen based on the predictor in such a way that the propagation of errors would be counteracted. Zhuang and Sun [48] worked very hard to try to prove the stability using more of a semi-group method, but were unsuccessful in their attempt, as noted in [47]. The notation the authors introduced became restrictive when dealing with the characteristic functions used to describe the elements. While it would have been beneficial to be able to use the notation from the paper, difficulty quickly arose when deciding how the operators interacted with each other.

Rivera, Zhu and Huddleston [36] published a survey of three different domain decomposition methods to show the advantages of the non-overlapping schemes available at the time. The authors showed experimental results of the one-dimensional heat equation. Of particular note were the schemes labeled as the "Explicit Predictor" (EP) and "Explicit Predictor/Implicit Corrector" (EPIC). The EP algorithm was defined as the algorithm Dawson, Du and Dupont [15] introduced without the coarse grid. The EPIC algorithm was the method given in Daoud, Khaliq and Wade [11] without the coarse grid. Without using the coarse grid, the methods are bound to the constraint

$$k \leq \frac{1}{2}h^2 \tag{1.40}$$

which, in essence, removes the benefits of the domain decomposition because we are forced to choose $k = O(h^2)$, something we are not willing to accept. The authors showed the numerical results in which EPIC outperformed the EP method and used the methods to look at an application to aerospace engineering.

To review, Dawson, Du and Dupont [13] implemented an algorithm and method of proof that, by implementing a coarse grid, allowed the scheme to regain some of the benefits of the implicit solves on the subdomain by allowing the choice of the time step k to be constrained by the coarse grid rather than the fine grid (destroying the use of backward Euler). The results are useful, but was only able to extend to the heat equation and not a more general equation, like convection-diffusion, caused by the difficulty of constructing a discrete Green's function used to prove their error estimates. Du, Mu and Wu [15] used similar proof techniques as [13] and obtained a result that implemented a higher-order extrapolation scheme for the predictor of interface values. Since it used the same Green's functions that method also was unable to be extended to more a more general setting. Lastly, Daoud, Khaliq and Wade [11] used the same coarse grid as Dawson, Du and Dupont, but presented an alternate approach to develop the maximum principle necessary to prove their error estimate. The result of their analysis, in one-dimension, yield a slightly weaker relationship between the time step and coarse grid size, but allowed for approximating more general partial differential equations, like the convection-diffusion equation.

We would like to now work to extend the results of [13, 15, 11] retaining their benefits and removing some of their limitations. In the next chapter, we introduce the four algorithms that use the coarse grid, with the coarse grid constraint, and can be applied to convection-diffusion equations in one- or two-dimensions.

Chapter 2

Algorithms

From the last chapter, we showed current results that work to handle the interface of the domain decomposition problem for PDEs. We also noted that there were places that one might improve on the available information. One place in particular is the extension to two-dimensions. Recall that in Section 1.4, we introduced the one-dimension explicit predictor/implicit corrector method from Daoud, Khaliq and Wade [11] in the following manner:

First, set $u_i^0 = g$. Then for $n = 1, 2, \ldots, N$

1. Compute the interface using an explicit predictor on the coarse grid and assign the value to the temporary placeholder u_*^n.

$$u_p^n = u_p^{n-1} - kL_{x,H}u_p^{n-1} + kf_p^{n-1} \tag{2.1}$$

2. Solve each subdomain $\Omega_1, \Omega_2, \ldots$ in parallel using an implicit solver with a fine grid.

$$\delta_k u_i^n + L_{x,h}u_i^n = f_i^n \tag{2.2}$$

3. Discard the predicted interface u_*^n and correct the interface using information from the newly computed subdomains using an implicit method on the fine grid.

$$\delta_k u_p^n + L_{x,h} u_p^n = f_p^n. \tag{2.3}$$

In Figure 2.1, we give a visual interpretation of the computations by displaying the stencils for the predictor and corrector.

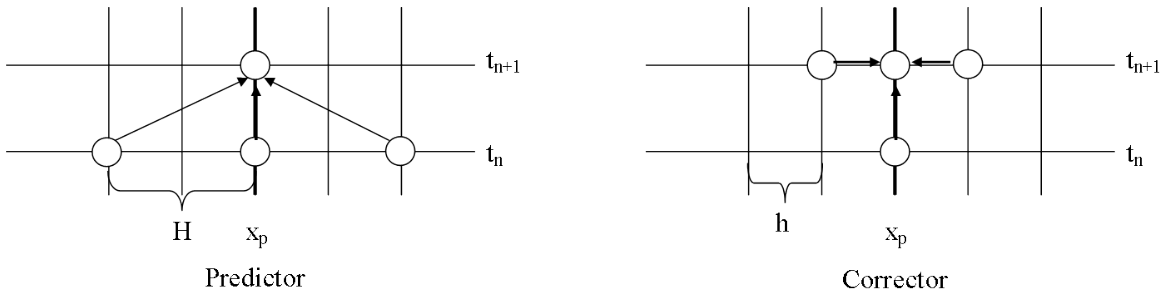


Figure 2.1: 1D Stencil for Daoud, Khaliq, and Wade algorithm [11]

The trouble of handling the interface in two dimensions deals with trying to find a manner to handle the set of points that lie on the interface at each time step rather than a single point in the one-dimension case. Because of this, we introduce four two-dimensional algorithms. Each algorithm uses a different method to to compute the interface values. We point out to the reader that while the two-dimensional algorithms are related to the one-dimensional algorithms, it is hard to relate the two types because of the extra care required for handling the second space variables. To have a point of reference, we show in Figure 2.2 the standard five-point stencil used by the finite difference method of the spatial variable for the heat equation.

The first algorithm in Section 2.1 (Figure 2.3) is a hybrid algorithm of similar ilk as the algorithm of Dawson, Du and Dupont [13] but uses a new proof technique to

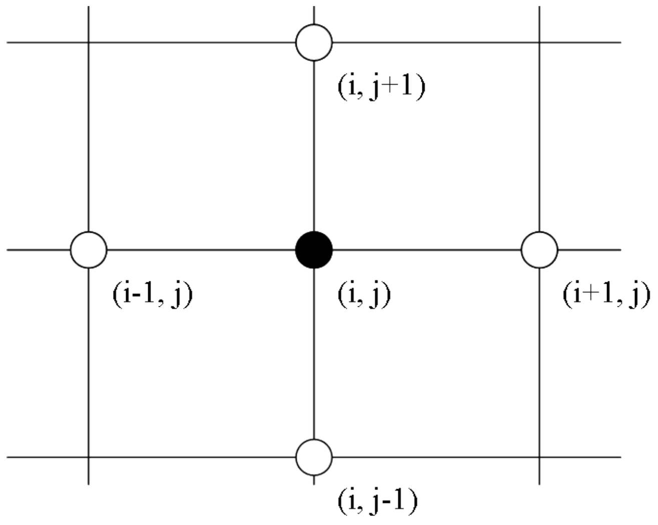


Figure 2.2: Five-Point Stencil.

obtain a similar error estimate and constraint on the coarse grid as Daoud, Khaliq and Wade [11]. This algorithm, which is given the name CEIDD-Hyb, where CEIDD stands for Corrective Explicit/Implicit Domain Decomposition as we also implement a corrector to the scheme, like the one-dimension scheme noted above.

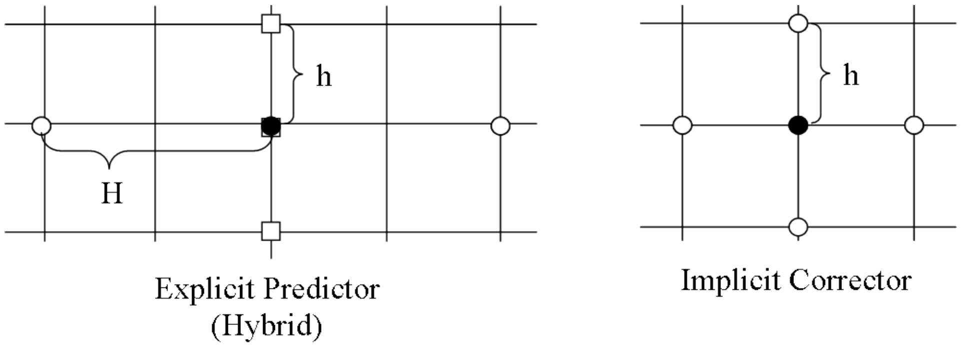


Figure 2.3: Stencils used for CEIDD-Hyb.

In Section 2.2 (Figure 2.4), the second algorithm, named CEIDD-Exp1, changes CEIDD-Hyb use as much of an explicit computation for the interface as allowed with the coarse grid and for the remaining interface values uses CEIDD-Hyb. Section 2.3 introduces CEIDD-Exp2 (Figure 2.5), a modification of CEIDD-Exp1 created by reformulating the finite difference equations to rotate the standard the five-point Laplacian in an effort to use values off the artificial boundary for computation of the interface values. For each of these algorithms, a detailed explanation is given then present statements and proofs of necessary maximum principles and error estimates.

To finish the chapter, the last parallel algorithm is introduced, CEIDD-Lex (Figure 2.6), an algorithm that implements linear extrapolation along the interface. The requisite proof for this algorithm is given where it is shown that CEIDD-Exp1 is equivalent to CEIDD-Lex.

Before delving further into the algorithms, we introduce more notation in an effort to be clear throughout our discussion. We have already introduced the finite difference operator in the one-dimensional case (1.27):

$$L_{x,h}u_i^n = -\alpha_i^n \triangle_{x,h} u_i^n + \beta_i^n \cdot \nabla_{x,h} u_i^n + \gamma_i^n u_i^n.$$

We will define $L_{x,y,h}^{Abc}$ as the appropriate operator for the two-dimensional case defined in each section, where '*Abc*' is the abbreviation of the scheme name. For instance, the algorithm for CEIDD-Hyb would be labeled as $L_{x,y,h}^{Hyb}$. The actual definition of the operator will be introduced at the beginning of each section. We will describe a split of our domain into only two strip subdomains for ease of explanation. There is no limitation of the algorithms in generalizing to multiple strip subdomains. Because our domain is two-dimensional, we should use $u_{*,j}^n$ as the predicted interface value, but since our computation is for each individual element and the j-value will be clear in context, we will continue to use u_*^n.

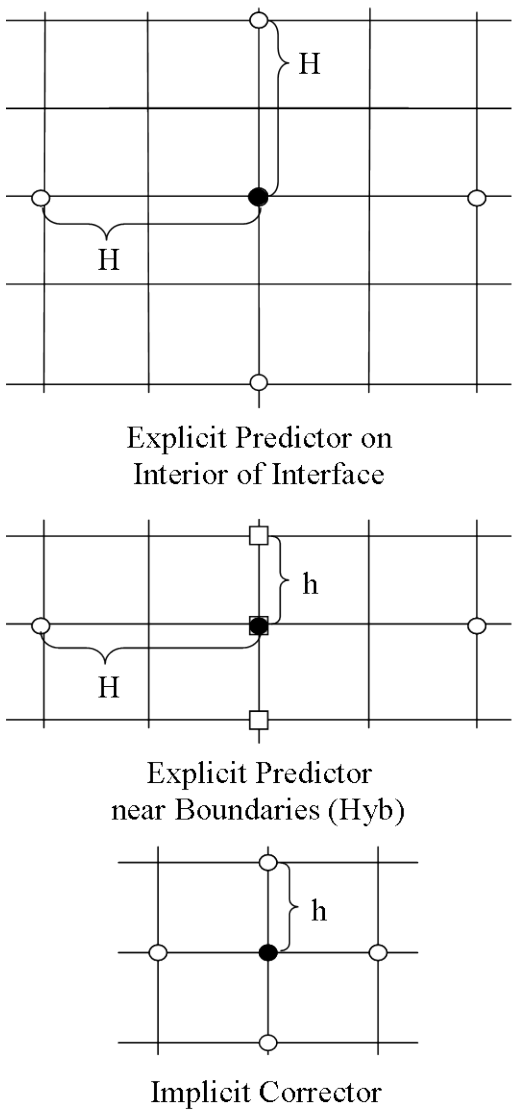


Figure 2.4: Stencils used for CEIDD-Exp1.

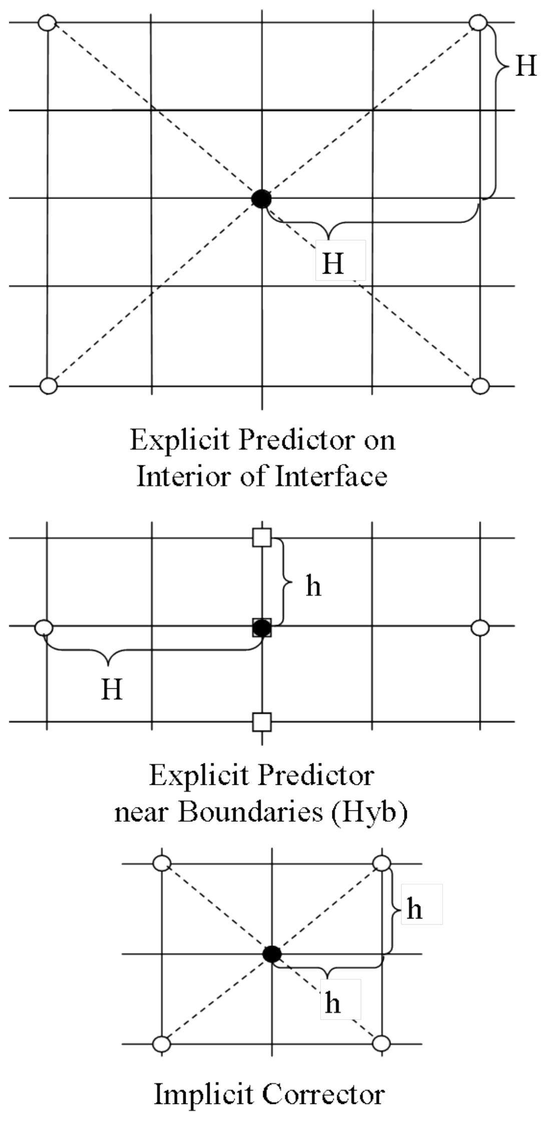


Figure 2.5: Stencils used for CEIDD-Exp2.

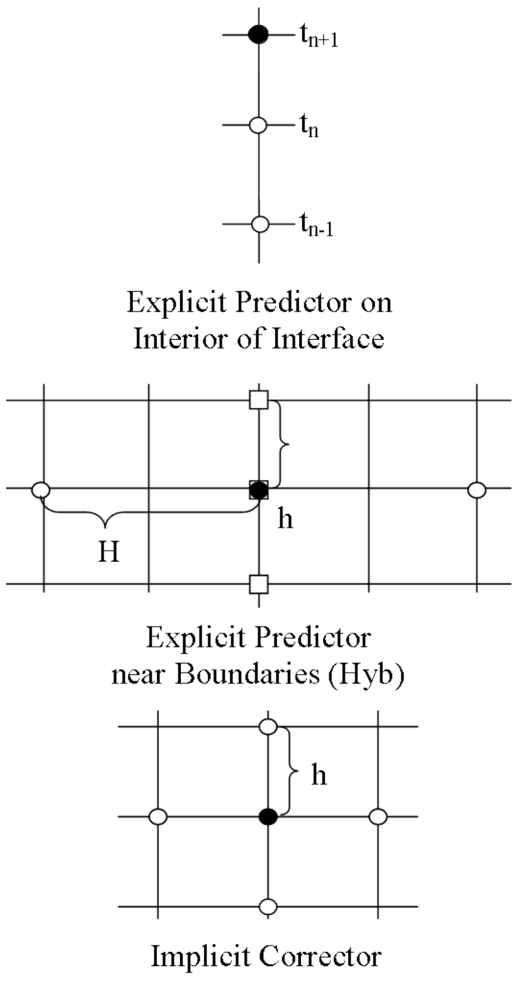

t_{n+1}
t_n
t_{n-1}
Explicit Predictor on Interior of Interface
h
H
Explicit Predictor near Boundaries (Hyb)
h
Implicit Corrector

Figure 2.6: Stencils used for CEIDD-Lex.

2.1 Algorithm 1, CEIDD-Hyb

As stated in the introduction, CEIDD-Hyp modifies the algorithm given in [13] by using a proof technique similar to that of Daoud, Khaliq and Wade [11] so that we are able to use the algorithm on the more general convection-diffusion equation (1.5), which is not a self-adjoint operator. This new extension also allows for us to approximate the solution to the two-dimensional PDE rather than only the one-dimensional scenario [11] worked with.

We will take our domain Ω and split it into two rectangular subdomains with the artificial interface Γ being oriented at $x = x_p$ shown in Figure 2.7. The algorithm starts by predicting on the interface with a hybrid explicit/implicit non-overlapping domain decomposition method. The predictor uses forward Euler in the x-direction (perpendicular to the interface) using the coarse grid (H) and backward Euler in the y-direction (along the interface) using the fine grid (h). Once the interface is computed, each subdomain is solved using backward Euler on the fine grid. To finish the computation for the time step, the interface values are 'corrected' by discarding the predicted values and recomputed using the new subdomain values and backward Euler of the fine grid. We know that backward Euler is unconditionally stable which guarantees that using values from the subdomain solves to recompute the predicted interface will give us a more reliable approximation of the interface values for the next time step.

We describe CEIDD-Hyb as follows:

Set $u^0_{i,j} = g$. For $n = 1, 2, \ldots, N$:

1. Compute the interface using a hybrid explicit/implicit predictor on the coarse

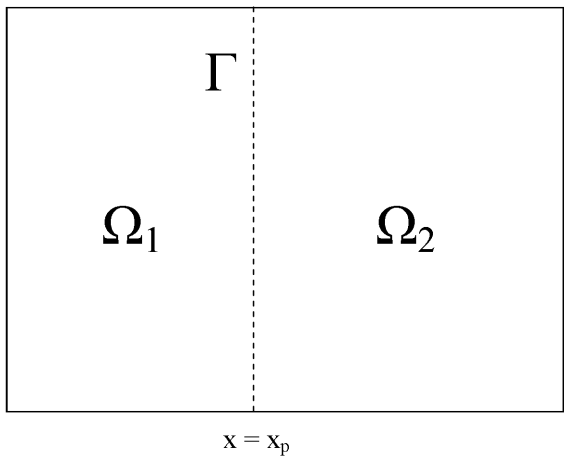


Figure 2.7: Orientation of the subdomains.

grid (Figure 2.8) using

$$\delta_k u^n_{p,j} + L^{Hyb}_{x,y,H} u^{n-1}_{p,j} = f^{n-1}_{p,j} \tag{2.4}$$

which allows us to solve for $u^n_{p,j}$ and then assign the interface values to the temporary placeholder u^n_* where

$$\begin{aligned} L^{Hyb}_{x,y,H} u^{n-1}_{p,j} &= -\alpha^{n-1}_{p,j}(\triangle_{x,H} u^{n-1}_{p,j} + \triangle_{y,h} u^n_{p,j}) \\ &\quad +\beta^{n-1}_{p,j} \cdot (\nabla_{x,H} u^{n-1}_{p,j} + \nabla_{y,h} u^n_{p,j}) \\ &\quad +\gamma^{n-1}_{p,j} u^{n-1}_{p,j}. \end{aligned}$$

2. Solve on each subdomain Ω_1 and Ω_2 in parallel using an implicit solver with a fine grid where the u^n_* makes up the Dirichlet boundary on the side of the

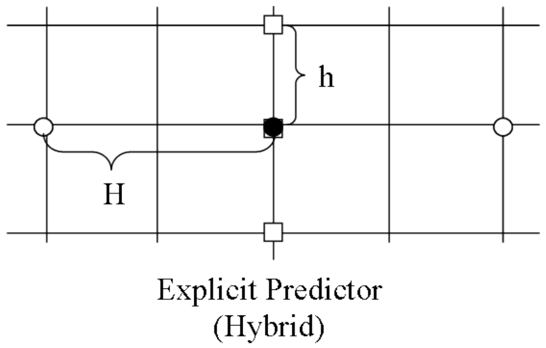


Figure 2.8: Stencil for predictor of CEIDD-Hyb.

appropriate subdomain.

$$\delta_k u^n_{i,j} + L^{BE}_{x,y,h} u^n_{i,j} = f^n_{i,j} \tag{2.5}$$

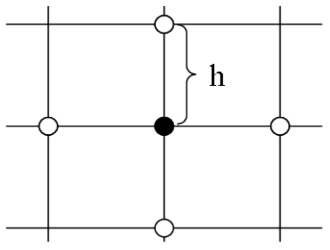
where

$$\begin{aligned} L^{BE}_{x,y,h} u^n_{i,j} &= -\alpha^n_{i,j}(\triangle_{x,h} u^n_{i,j} + \triangle_{y,h} u^n_{i,j}) \\ &\quad +\beta^n_{i,j} \cdot (\nabla_{x,h} u^n_{i,j} + \nabla_{y,h} u^n_{i,j}) \\ &\quad +\gamma^n_{i,j} u^n_{i,j}. \end{aligned}$$

3. Discard the predicted interface u^n_* and correct the interface using information from the newly computed subdomains using an implicit method on the fine grid (Figure 2.9).

$$\delta_k u^n_{p,j} + L^{BE}_{x,y,h} u^n_{p,j} = f^n_{p,j}. \tag{2.6}$$

It may be disconcerting that in the last step we are discarding the information along the interface (u^n_*), but the values were temporary values needed to get through



Implicit Corrector

Figure 2.9: Stencil for corrector of CEIDD-Hyb.

the implicit solve on the subdomains. As stated earlier, the corrector, because it uses an unconditional method, will yield a more stable approximation for the next time step. Even though we are using an implicit scheme in both subdomains and the corrected interface, we still used an explicit method for the predictor which leaves us with a weak constraint on the size of the time step yet will still converge though not unconditionally.

It is important to reiterate that for the interface we are using both an implicit method, to correct for accuracy and preserve stability, and an explicit method, to compute rapidly in a non-overlapping fashion. This scheme improves on the method of Dawson, Du and Dupont [13] by creating a scheme that works for the general convection-diffusion equation, rather than just the simpler heat equation. It also improves the method of Daoud, Khaliq and Wade [11] by extending the algorithm to the two-dimensional problem. As stated in Section 1.4, the algorithm does require a small set of tridiagonal equations, a by-product of the implicit portion of the hybrid predictor, that gives some extra computational work, but this is rather small compared to those involved in the subdomains, something we shall note in the numerical experiments.

Recall from Section 1.1, the coefficients α, β, and γ are assumed to be continuous and thus exists real numbers $\overline{\alpha}$, $\underline{\alpha}$, $\overline{\beta}$ and $\overline{\gamma}$ such that the following equalities hold for all points in $\overline{\Omega}$ where Ω is a bounded, connected region of $\mathbb{R}^2$ and $t \in [0, T)$:

$$\begin{gathered} 0 < \underline{\alpha} < \alpha < \overline{\alpha} \\ |\beta| \leq \overline{\beta} \\ 0 \leq \gamma \leq \overline{\gamma} \end{gathered} \tag{2.7}$$

Theorem 2.1 (Error Estimate for CEIDD-Hyb)**.** *Suppose $u(x, y, t) \in C_1^2(\Omega)$ and the following conditions hold:*

$$k\overline{\gamma} < 1, H \geq \sqrt{\frac{2\underline{\alpha}k}{1 - \overline{\gamma}k}}, \ \textit{and}\ h \leq H \leq \frac{2\underline{\alpha}}{\overline{\beta}} \tag{2.8}$$

Then, for CEIDD-Hyb there exists a constant c, independent of the grid, such that for $1 \leq i, j \leq M$ and $1 \leq n \leq N$

$$|u(x_i, y_j, t_n) - u_{i,j}^n| \leq c(k + h^2 + H^2). \tag{2.9}$$

The reader will notice that the error estimate for Dawson, Du and Dupont is slightly better with respect to the coarse grid, $O(k + h^2 + H^3)$, versus the result of this theorem, $O(k + h^2 + H^2)$, as k is the limiting factor. The new algorithm is much more applicable, allowing for the use with convection-diffusion equations at the cost of a fractional power in the size of the temporal mesh.

Before the proof of the error estimate, we need a discrete maximum principle.

Lemma 2.2 (Maximum Principle for CEIDD-Hyb)**.** *Under the assumptions of* (2.8), *suppose on one subdomain, say* $\Omega_1 := (0, \overline{x}) \times (0,1)$, *and* $n = 1, \dots, N$

$$\delta_k z^n_{i,j} + L^{BE}_{x,y,h} z^n_{i,j} = g^n_{i,j} \quad \textit{for interior points} \tag{2.10}$$

with $z^n_{0,j} = a_n$, $z^n_{p,j} = b_n$, $z^n_{i,0} = c_n$ *and* $z^n_{i,M} = d_n$, *where* a_n, b_n, c_n *and* d_n *are constants which arise from the boundary values. Then the following estimate must hold at each time level* t_n*:*

$$\max_{\Omega_1} |z^n_{i,j}| \quad \leq \quad \max\{|a_n|, |b_n|, |c_n|, |d_n|, \max_{\Omega_1} |z^{n-1}_{i,j}|\} + k \max_{\Omega_1} |g^n_{i,j}|. \tag{2.11}$$

Proof. Suppose we have an index (r, s) is such that

$$|z^n_{r,s}| = \max_{\Omega_1} |z^n_{i,j}|. \tag{2.12}$$

If $r = 0$ or p or $s = 0$ or M, the result is immediate. Therefore, suppose that (r, s) is such that $z_{r,s}$ is an interior point in Ω_1 is chosen. We consider the following cases:

1. Case 1: $z^n_{r,s} \geq 0$

   Define $\alpha_h = \frac{k}{h^2}\alpha^n_{r,s}$ and $\beta_h = \frac{k}{2h}\beta^n_{r,s}$, then beginning with

$$\delta_k z^n_{r,s} + L^{BE}_{x,y,h} z^n_{r,s} = g^n_{r,s}, \tag{2.13}$$

$$\begin{aligned} z^n_{r,s} \quad = \quad & z^{n-1}_{r,s} + (\alpha_h - \beta_h) z^n_{r+1,s} + (\alpha_h + \beta_h) z^n_{r-1,s} \\ & + (-4\alpha_h - k\gamma^n_{i,j}) z^n_{r,s} + (\alpha_h - \beta_h) z^n_{r,s-1} + (\alpha_h + \beta_h) z^n_{r,s+1} \\ & + k g^n_{r,s} \end{aligned}$$

By the assumptions (2.8), $h \leq 2\overline{\alpha}/\underline{\beta}$ can be rewritten as

$$|\beta_{i,j}^n|h \leq \overline{\beta}h \leq 2\underline{\alpha} < 2\alpha. \tag{2.14}$$

The implication of this statement along with the assumption of the case yields the following results

$$|(\alpha_h \pm \beta_h)z_{r+1,s}^n| \leq (\alpha_h \pm \beta_h)|z_{r,s}^n|. \tag{2.15}$$

Using (2.14) and (2.15),

$$\begin{aligned}
|z_{r,s}^n| &\leq |z_{r,s}^{n-1}| + (\alpha_h - \beta_h)|z_{r,s}^n| + (\alpha_h + \beta_h)|z_{r,s}^n| \\
&\quad + (-4\alpha_h - k\gamma_{i,j}^n)z_{r,s}^n + (\alpha_h - \beta_h)|z_{r,s}^n| + (\alpha_h + \beta_h)|z_{r,s}^n| \\
&\quad + k|g_{r,s}^n| \\
&= |z_{r,s}^{n-1}| + 4\alpha_h|z_{r,s}^n| + (-4\alpha_h - k\gamma_{i,j}^n)|z_{r,s}^n| + k|g_{r,s}^n| \\
&= |z_{r,s}^{n-1}| - k\gamma_{i,j}^n|z_{r,s}^n| + k|g_{r,s}^n| \\
&\leq |z_{r,s}^{n-1}| + k|g_{r,s}^n| \qquad (2.16)
\end{aligned}$$

2. Case 2: $z_{r,s}^n < 0$

   In this case, the assumption implies $|z_{r,s}^n| = -z_{r,s}^n$, and similar to the first case,

$$\begin{aligned}
-z_{r,s}^n &= -z_{r,s}^{n-1} + (\alpha_h - \beta_h)(-z_{r+1,s}^n) + (\alpha_h + \beta_h)(-z_{r-1,s}^n) \\
&\quad + (-4\alpha_h - k\gamma_{i,j}^n)(-z_{r,s}^n) + (\alpha_h - \beta_h)(-z_{r,s-1}^n) + (\alpha_h + \beta_h)(-z_{r,s+1}^n) \\
&\quad + k(-g_{r,s}^n)
\end{aligned}$$

which yields

$$\begin{aligned}
|z_{r,s}^n| &\leq |z_{r,s}^{n-1}| + (\alpha_h - \beta_h)|z_{r,s}^n| + (\alpha_h + \beta_h)|z_{r,s}^n| \\
&\quad + (-4\alpha_h - \gamma_{i,j}^n)z_{r,s}^n + (\alpha_h - \beta_h)|z_{r,s}^n| + (\alpha_h + \beta_h)|z_{r,s}^n| \\
&\quad + k|g_{r,s}^n| \\
&\leq |z_{r,s}^{n-1}| + k|g_{r,s}^n| \qquad (2.17)
\end{aligned}$$

Combining (2.16) and (2.17), along with the trivial cases along the boundaries, yields the desired result. □

Attention now returns to proving Theorem 2.1.

Proof of Theorem 2.1. We first define the error at a point (x_i, y_j) as

$$e_{i,j}^n = u(x_i, y_j, t_n) - u_{i,j}^n, \qquad (2.18)$$

and we use the Taylor series in one variable about a point a for the discretization of time. Similarly, we use the Taylor series in two variables about a point (a, b) for the discretization in space

$$f(x,y) = \sum_{n_1=0}^{\infty} \sum_{n_2=0}^{\infty} \frac{\partial^{n_1}}{\partial x^{n_1}} \frac{\partial^{n_2}}{\partial y^{n_1}} \frac{f(a,b)}{n_1! n_2!} (x-a)^{n_1} (y-b)^{n_2}. \qquad (2.19)$$

Since we need only the first few terms for our computation and using the fact that our function is in $C_1^2(\Omega)$, we can bound the remainder which gives a portion of the error. The error must be computed for each step in our algorithm and combining the definition of the error, Taylor polynomials and the remainder bound, we obtain the following error equations:

- To compute the error of $\delta_k u^n_{i,j}$, we use the Taylor expansion for one variable, noting $k = t_n - t_{n-1}$ and see

$$\begin{aligned} u^n_{i,j} &= u^{n-1}_{i,j} + \partial/\partial t u^n_{i,j}(k) + O(k^2) && (2.20)\\ \partial/\partial t u^n_{i,j}(k) &= u^n_{i,j} - u^{n-1}_{i,j} + O(k^2) && (2.21)\\ \partial/\partial t u^n_{i,j} &= (u^n_{i,j} - u^{n-1}_{i,j})/k + O(k). && (2.22)\end{aligned}$$

- To compute the error of $\triangle_{x,h} u^n_{i,j} + \triangle_{y,h} u^n_{i,j}$, we use the Taylor expansion for two variables, noting $h = x_n - x_{n-1}$ to find

$$\begin{aligned} u^n_{i+1,j} &= u^n_{i,j} + \frac{\partial}{\partial x}u^n_{i,j}(h) + \frac{\partial^2}{\partial x^2}u^n_{i,j}\frac{h^2}{2} + \frac{\partial^3}{\partial x^3}u^n_{i,j}\frac{h^3}{6} + O(h^4) && (2.23)\\ u^n_{i-1,j} &= u^n_{i,j} + \frac{\partial}{\partial x}u^n_{i,j}(-h) + \frac{\partial^2}{\partial x^2}u^n_{i,j}\frac{h^2}{2} + \frac{\partial^3}{\partial x^3}u^n_{i,j}\frac{-h^3}{6} + O(h^4) && (2.24)\\ u^n_{i,j+1} &= u^n_{i,j} + \frac{\partial}{\partial y}u^n_{i,j}(h) + \frac{\partial^2}{\partial y^2}u^n_{i,j}\frac{h^2}{2} + \frac{\partial^3}{\partial y^3}u^n_{i,j}\frac{h^3}{6} + O(h^4) && (2.25)\\ u^n_{i,j-1} &= u^n_{i,j} + \frac{\partial}{\partial y}u^n_{i,j}(-h) + \frac{\partial^2}{\partial y^2}u^n_{i,j}\frac{h^2}{2} + \frac{\partial^3}{\partial y^3}u^n_{i,j}\frac{-h^3}{6} + O(h^4) && (2.26)\end{aligned}$$

When the left side of the equalities are summed we are left with

$$u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} = 4u^n_{i,j} + \frac{\partial^2}{\partial x^2}u^n_{i,j}(h^2) + \frac{\partial^2}{\partial y^2}u^n_{i,j}(h^2) + O(h^4) \quad (2.27)$$

or

$$\frac{\partial^2}{\partial x^2}u^n_{i,j}(h^2) + \frac{\partial^2}{\partial y^2}u^n_{i,j}(h^2) = u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} - 4u^n_{i,j} + O(h^4). \quad (2.28)$$

The final simplification arises from dividing by h^2.

$$\frac{\partial^2}{\partial x^2}u_{i,j}^n + \frac{\partial^2}{\partial y^2}u_{i,j}^n = (u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n)/h^2 + O(h^2). \quad (2.29)$$

- Lastly, we compute the error of $\nabla_{x,h}^n u_{i,j}^n + \nabla_{y,h}^n u_{i,j}^n$, we use the expansion to find

$$\begin{aligned} u_{i+1,j}^n &= u_{i,j}^n + \frac{\partial}{\partial x}u_{i,j}^n(h) + \frac{\partial^2}{\partial x^2}u_{i,j}^n\frac{h^2}{2} + O(h^3) && (2.30)\\ u_{i-1,j}^n &= u_{i,j}^n + \frac{\partial}{\partial x}u_{i,j}^n(-h) + \frac{\partial^2}{\partial x^2}u_{i,j}^n\frac{h^2}{2} + O(h^3) && (2.31)\\ u_{i,j+1}^n &= u_{i,j}^n + \frac{\partial}{\partial y}u_{i,j}^n(h) + \frac{\partial^2}{\partial y^2}u_{i,j}^n\frac{h^2}{2} + O(h^3) && (2.32)\\ u_{i,j-1}^n &= u_{i,j}^n + \frac{\partial}{\partial y}u_{i,j}^n(-h) + \frac{\partial^2}{\partial y^2}u_{i,j}^n\frac{h^2}{2} + O(h^3) && (2.33) \end{aligned}$$

Similarly, by looking at the sum of $u_{i+1,j}^n - u_{i-1,j}^n$ and $u_{i,j+1}^n - u_{i,j-1}^n$ we are left with

$$u_{i+1,j}^n - u_{i-1,j}^n + u_{i,j+1}^n - u_{i,j-1}^n = \frac{\partial}{\partial x}u_{i,j}^n(2h) + \frac{\partial}{\partial y}u_{i,j}^n(2h) + O(h^3) \quad (2.34)$$

which when simplified yields

$$\frac{\partial}{\partial x}u_{i,j}^n + \frac{\partial}{\partial y}u_{i,j}^n = (u_{i+1,j}^n - u_{i-1,j}^n)/2h + (u_{i,j+1}^n - u_{i,j-1}^n)/2h + O(h^2). \quad (2.35)$$

Combining these derivations allows us to compute the error equations as:

$$\begin{aligned} e_*^n &= e_{p,j}^{n-1} - kL_{x,y,H}^{Hyb}e_{p,j}^{n-1} + kK_{*,j}^n(k + h^2 + H^2)\\ \delta_k e_{i,j}^n + L_{x,y,h}^{BE}e_{i,j}^n &= K_{i,j}^n(k + h^2) \qquad (2.36)\\ \delta_k e_{p,j}^n + L_{x,y,h}^{BE}e_{p,j}^n &= K_{p,j}^n(k + h^2) \end{aligned}$$

with $e^n_{0,j} = e^n_{M,j} = e^n_{i,0} = e^n_{i,M} = 0$. $K^n_{i,j}$ and $K^n_{*,j}$ represent real numbers depending on u, h, H, and k and are uniformly bounded, meaning their bounds are independent of the grid parameters. Let

$$\begin{aligned} C_1 &= \max_{\Omega_1} \{|K^n_{i,j}| : 1 \leq n \leq N\} \\ C_* &= \max_{\Gamma} \{|K^n_{*,j}| : 1 \leq n \leq N\} \end{aligned}$$

where the constants are also independent of the grid.

Applying the estimate from Lemma 2.2 to the error equations, we obtain:

$$\max_{\Omega_1} |e^n_{i,j}| \leq \max \{|e^n_{*,j}|, \max_{\Omega_1} |e^{n-1}_{i,j}|\} + C_1 k(k + h^2) \tag{2.37}$$

Just as in [11], the error estimate of the correction is included since the correction is based on the same implicit Euler step. What is left is to find a bound on the interface error e^n_*.

To that end, it is claimed that if

$$z^n_* = z^{n-1}_{p,j} - kL^{Hyb}_{x,y,H} z^{n-1}_{p,j} + kg^{n-1}_{p,j}, \tag{2.38}$$

then

$$|z^n_*| \leq \max_{1 \leq i \leq p-1} |z^{n-1}_{i,j}| + k|g^{n-1}_{p,j}|. \tag{2.39}$$

To see this, note the following

$$\begin{aligned}
z_*^n &= z_{p,j}^{n-1} - kL_{x,y,H}^{Hyb} + kg_{p,j}^n \\
&= z_{p,j}^{n-1} + \alpha_H(z_{p-q,j}^{n-1} - 2z_{p,j}^{n-1} + z_{p+q,j}^{n-1}) + \alpha_h(z_{p,j-1}^n - 2z_{p,j}^n + z_{p,j+1}^n) \\
&\quad -\beta_H(z_{p+q,j}^{n-1} - z_{p-q,j}^{n-1}) - \beta_h(z_{p,j+1}^n - z_{p,j-1}^n) \\
&\quad -k\gamma_{p,j}^{n-1} z_{p,j}^{n-1} + kg_{p,j}^{n-1} \\
&= (\alpha_H + \beta_H)\, z_{p-q,j}^{n-1} + (1 - 2\alpha_H - k\gamma_{p,j}^{n-1}) z_{p,j}^{n-1} + (\alpha_H - \beta_H)\, z_{p+q,j}^{n-1} + \\
&\quad (\alpha_h + \beta_h)\, z_{p,j-1}^n - 2\alpha_h z_{p,j}^n + (\alpha_h - \beta_h)\, z_{p,j+1}^n + kg_{p,j}^{n-1}
\end{aligned}$$

Rewriting the equation using the assumptions yields

$$\begin{aligned}
|z_*^n| &\le \left[(\alpha_H + \beta_H) + (\alpha_H - \beta_H) + (1 - 2\alpha_H - k\gamma_{p,j}^n)\right] \max_{i=p-q,p,p+q} \left\{|z_{i,j}^{n-1}|\right\} \\
&\quad + [(\alpha_h + \beta_h) + (\alpha_h - \beta_h) - 2\alpha_h] \max_{j=j-1,j,j+1} |z_{p,j}^n| + k|g_{p,j}^{n-1}| \\
&\le \max |z_{p,j}^{n-1}| + k|g_{p,j}^{n-1}| \qquad (2.40)
\end{aligned}$$

Applying the result of the claim to e_*^n leaves

$$|e_*^n| \le \max_{1\le i\le p-1} |e_{p,j}^{n-1}| + kC_*(k + h^2 + H^2) \qquad (2.41)$$

and hence

$$\begin{aligned}
\max_{\Omega_1} |e_{i,j}^n| &\le \max_{\Omega_1} |e_{i,j}^0| + \sum_{w=1}^{n} C'k(k + h^2 + H^2) \\
&\le \max_{\Omega_1} |e_{i,j}^0| + C'nk(k + h^2 + H^2) \qquad (2.42)
\end{aligned}$$

Assuming no error with the initial time and $nk \le T$, we have successfully shown that

$||e^n_{\cdot,\cdot}||_\infty = \max\{u(x_i, y_j, t_n) - u^n_{i,j}\}$ has the following bound:

$$||e^n_{\cdot,\cdot}||_\infty \leq C(k + h^2 + H^2) \tag{2.43}$$

uniformly for $1 \leq n \leq N$. □

2.2 Algorithm 2, CEIDD-Exp1

Computationally there is a desire to try to find an algorithm that does not rely so heavily on the implicit solve during the predictor phase of the computation. Doing so should allow for a faster computation during the predictor phase of the scheme by removing the small tridiagonal solver. In an attempt to remove some of the implicit computation (backward Euler) on the interface necessary for the hybrid explicit/implicit predictor, a modification of the previous scheme is now introduced. Since the course grid introduced in Section 2.1 is used as a basis for the new algorithm, we have chosen to adopt a name for the new algorithm to relate its lineage, CEIDD-Exp1.

Set $u^0_{i,j} = g$. For $n = 1, 2, \ldots, N$

1. Given that the interface coordinates are denoted as (x_p, y_j) with $j = 0, \ldots, M$ and $H = q * h$ for appropriate values of q we compute the interface using an explicit predictor on the coarse grid.

   (a) For $j = q, \ldots, M - q$, solve the system arising from

$$u^n_{p,j} = u^{n-1}_{p,j} - kL^{Exp1}_{x,y,H} u^{n-1}_{p,j} + kf^{n-1}_{p,j} \tag{2.44}$$

and assign the values to the temporary placeholders $u^n_{*,j}$ where

$$\begin{aligned}
L^{Exp1}_{x,y,H} u^{n-1}_{p,j} \quad = \quad & -\alpha^{n-1}_{p,j}(\triangle_{x,H} u^{n-1}_{p,j} + \triangle_{y,H} u^{n-1}_{p,j}) \\
& +\beta^{n-1}_{p,j} \cdot (\nabla_{x,H} u^{n-1}_{p,j} + \nabla_{y,H} u^{n-1}_{p,j}) \\
& +\gamma^{n-1}_{p,j} u^{n-1}_{p,j}.
\end{aligned}$$

(b) For $j = 1, \ldots, q-1$ and $j = M-(q+1), \ldots, M-1$, solve the system arising from

$$u^n_{p,j} = u^{n-1}_{p,j} - kL^{Hyb}_{x,y,H} u^{n-1}_{p,j} + kf^{n-1}_{p,j}. \tag{2.45}$$

and assign the values to the temporary placeholders u^n_*. The reason for the two different approaches arises from the fact that the new predictor uses the coarse grid in both the x- and y-directions, as seen in Figure 2.10, which causes a portion of the interface to require CEIDD-Hyb for a relatively small number of values.

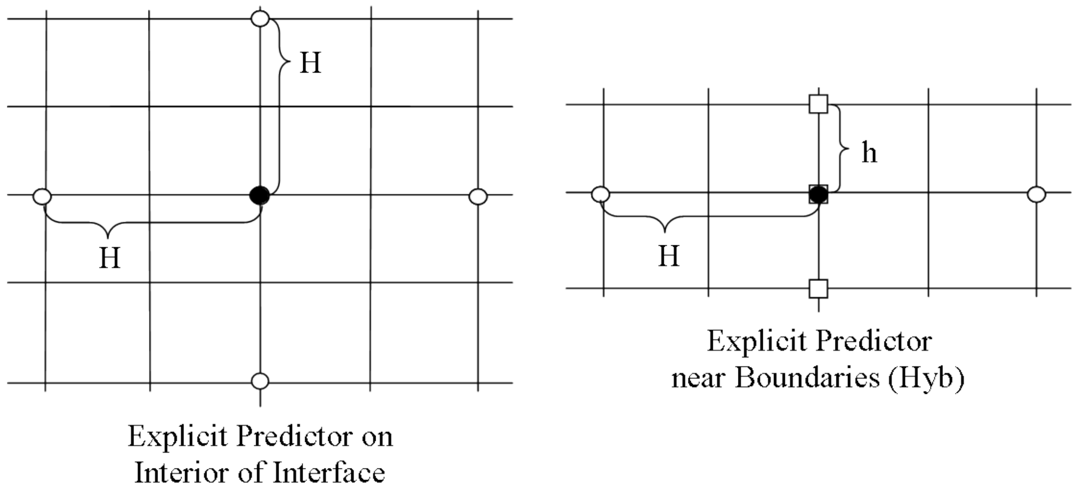


Figure 2.10: Stencils used for the predictor of CEIDD-Exp1.

2. Solve each subdomain Ω_1 and Ω_2 in parallel using an implicit solver with a fine grid.

$$\delta_k u_{i,j}^n + L_{x,y,h}^{BE} u_{i,j}^n = f_{i,j}^n. \tag{2.46}$$

3. Discard the predicted interface u_*^n and correct the interface using information from the newly computed subdomains using an implicit method on the fine grid

$$\delta_k u_{p,j}^n + L_{x,y,h}^{BE} u_{p,j}^n = f_{p,j}^n. \tag{2.47}$$

We are able to use the same maximum principle from CEIDD-Hyb (Lemma 2.2) leaving us to state and prove the following error estimate for CEIDD-Exp1.

Theorem 2.3 (Error Estimate for CEIDD-Exp1)**.** *Suppose $u(x,y,t) \in C_1^2(\Omega)$ and the following conditions hold:*

$$k\overline{\gamma} < 1, H \geq \sqrt{\frac{4\underline{\alpha}k}{1-\overline{\gamma}k}}, \text{ and } h \leq H \leq \frac{2\underline{\alpha}}{\overline{\beta}} \tag{2.48}$$

Then, for CEIDD-Exp1 there exists a constant c, independent of the grid, such that for $1 \leq i,j \leq M$ and $1 \leq n \leq N$

$$|u(x_i, y_j, t_n) - u_{i,j}^n| \leq c(k + h^2 + H^2). \tag{2.49}$$

Proof. Given that we computed the error estimate for CEIDD-Hyb in Theorem 2.1, we only concern ourselves with the error from the new interface predictor. The error equation for those interface values, e_*^n, are computed again using the Taylor series expansion for the one- and two-variable functions as in Theorem 2.1 to obtain:

$$e_*^n = e_{p,j}^{n-1} - kL_{x,y,H}^{Exp1} e_{p,j}^{n-1} + kK_{*,j}^n (k + H^2). \tag{2.50}$$

with $e_{0,j}^n = e_{M,j}^n = e_{i,0}^n = e_{i,M}^n = 0$. $K_{i,j}^n$ and $K_{*,j}^n$ represent real numbers depending on u, h, H, and k. Again, these numbers are uniformly bounded with constants independent of the grid parameters. Let

$$\begin{aligned} C_1 &= \max_{\Omega_1} \{|K_{i,j}^n| : 1 \leq n \leq N\} \\ C_* &= \max_{\Gamma} \{|K_{*,j}^n| : 1 \leq n \leq N\}. \end{aligned}$$

A bound on e_*^n again remains and we make that claim if

$$z_*^n = z_{p,j}^{n-1} - kL_{x,y,H}^{Exp1} z_{p,j}^{n-1} + kg_{p,j}^{n-1}, \tag{2.51}$$

then

$$|z_*^n| \leq \max_{1 \leq i \leq p-1} |z_{i,j}^{n-1}| + k|g_{p,j}^{n-1}|. \tag{2.52}$$

To see this,

$$\begin{aligned}
z_*^n &= z_{p,j}^{n-1} - kL_{x,y,H}^{Exp1} z_{p,j}^{n-1} + kg_{p,j}^{n-1} \\
&= z_{p,j}^{n-1} + \alpha_H (z_{p-q,j}^{n-1} + z_{p+q,j}^{n-1} - 4z_{p,j}^{n-1} + z_{p,j-q}^{n-1} + z_{p,j+q}^{n-1}) + \\
&\quad -\beta_H (z_{p+q,j}^{n-1} - z_{p-q,j}^{n-1} + z_{p,j+q}^{n-1} - z_{p,j-q}^{n-1}) \\
&\quad -k\gamma_{p,j}^{n-1} z_{p,j}^{n-1} + kg_{p,j}^{n-1} \\
&= (\alpha_H + \beta_H)\, z_{p-q,j}^{n-1} + (\alpha_h - \beta_h)\, z_{p+q,j}^{n-1} \\
&\quad +(1 - 4\alpha_H - k\gamma_{p,j}^{n-1}) z_{p,j}^{n-1} + (\alpha_H + \beta_H)\, z_{p,j-q}^{n-1} \\
&\quad + (\alpha_h - \beta_h)\, z_{p,j+q}^{n-1} + kg_{p,j}^{n-1}
\end{aligned}$$

The assumptions of the theorem lead similarly to

$$|z_*^n| \le \max_{\Omega_1} |z_{i,j}^{n-1}| + k|g_{p,j}^{n-1}| \tag{2.53}$$

Applying the result to e_*^n leaves

$$|e_*^n| \le \max_{\Omega_1} |e_{p,j}^{n-1}| + kC_*(k + H^2) \tag{2.54}$$

and hence

$$\begin{aligned}
\max_{\Omega_1} |e_{i,j}^n| &\le \max_{(i,j)\in\Gamma_1} |e_{i,j}^0| + \sum_{w=1}^{n} C'k(k + h^2 + H^2) \\
&\le \max_{\Omega_1} |e_{i,j}^0| + C'nk(k + h^2 + H^2)
\end{aligned} \tag{2.55}$$

The mixture of h^2 and H^2 comes from using the semi-explicit scheme near the boundaries.

Assuming no error with the initial time and $nk \leq T$, it has been shown again that

$$||e^n_{\cdot,\cdot}||_\infty \leq C(k + h^2 + H^2) \tag{2.56}$$

uniformly for $1 \leq n \leq N$. □

2.3 Algorithm 3, CEIDD-Exp2

With this algorithm, we address a question that might arise regarding the effects of a predictor that uses a majority of values along the interface. In an attempt to address this possible concern of overuse of the predicted interface, we introduce a modification of CEIDD-Exp1 by rotating the stencil so that only the center of the node remains in the stencil and the other values come from elements in the subdomains (Figure 2.3). We will are still required to use CEIDD-Hyb near the boundaries once again.

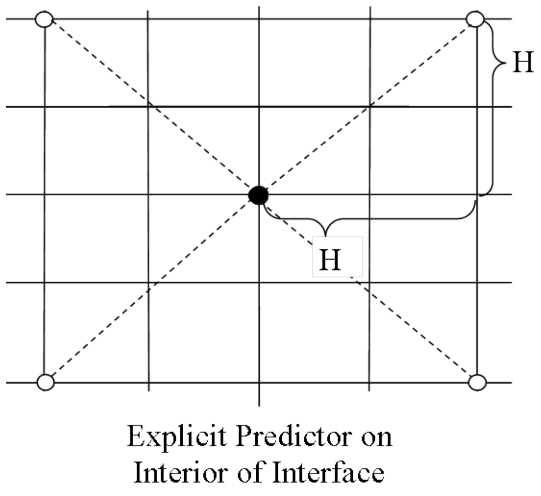


Figure 2.11: Stencil for interior predictor of CEIDD-Exp2.

A new maximum principle is proved along with a similar error estimate, but first an explanation of the scheme CEIDD-Exp2.

Set $u^0_{i,j} = g$. For $n = 1, 2, \ldots, N$:

1. Given that the interface coordinates are denoted as (x_p, y_j) with $j = 0, \ldots, M$ and $H = q * h$ we compute the interface using an explicit predictor on the coarse grid.

   (a) For $j = q, \ldots, M - q$, solve the system arising from

   $$u^n_{p,j} = u^{n-1}_{p,j} - kL^{Exp2}_{x,y,H} u^{n-1}_{p,j} + kf^{n-1}_{p,j} \tag{2.57}$$

   and assign the values to the temporary placeholders $u^n_{*,j}$ where

   $$\begin{aligned} L^{Exp2}_{x,y,H} u^{n-1}_{p,j} &= -\alpha^{n-1}_{p,j}(u^{n-1}_{p+q,j+q} + u^{n-1}_{p-q,j+q} - 4u^{n-1}_{p,j} + u^{n-1}_{p-q,j+q} + u^{n-1}_{p-q,j-q})/H^2 \\ &\quad + \beta^{n-1}_{p,j} \cdot (u^{n-1}_{p+q,j+q} + u^{n-1}_{p-q,j-q} + u^{n-1}_{p-q,j+q} + u^{n-1}_{p+q,j-q})/2H \\ &\quad + \gamma^{n-1}_{p,j} u^{n-1}_{p,j}. \end{aligned}$$

   (b) For $j = 1, \ldots, q - 1$ and $j = M - (q + 1), \ldots, M - 1$, solve the system arising from

   $$u^n_{p,j} = u^{n-1}_{p,j} - kL^{Hyb}_{x,y,H} u^{n-1}_{p,j} + kf^{n-1}_{p,j}. \tag{2.58}$$

   and assign the values to the temporary placeholders u^n_*.

2. Solve each subdomain Ω_1 and Ω_2 in parallel using an implicit solver with a fine grid.

   $$\delta_k u^n_{i,j} + L^{BE}_{x,y,h} u^n_{i,j} = f^n_{i,j}. \tag{2.59}$$

3. Discard the predicted interface u^n_* and correct the interface using information from the newly computed subdomains using an implicit method on the fine grid

(Figure 3).

$$\delta_k u^n_{p,j} + L^{BE}_{x,y,h} u^n_{p,j} = f^n_{p,j}. \tag{2.60}$$

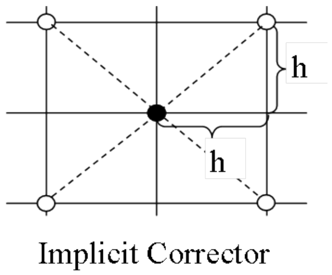


Figure 2.12: Stencil for corrector of CEIDD-Exp2.

Theorem 2.4 (Error Estimate for CEIDD-Exp2). *Suppose $u(x, y, t) \in C^2_1(\Omega)$ and the following conditions hold:*

$$k\overline{\gamma} < 1, H \geq \sqrt{\frac{4\underline{\alpha}k}{1-\overline{\gamma}k}}, \text{ and } h \leq H \leq \frac{2\underline{\underline{\alpha}}}{\overline{\overline{\beta}}} \tag{2.61}$$

Then, for CEIDD-Exp2 there exists a constant c, independent of the grid, such that for $1 \leq i, j \leq M$ and $1 \leq n \leq N$

$$|u(x_i, y_j, t_n) - u^n_{i,j}| \leq c(k + h^2 + H^2). \tag{2.62}$$

The reader should notice that the second constraint is more restrictive by a factor of $\sqrt{2}$. This is due to the construction of the rotated five-point Laplacian. We turn our attention to a maximum principle.

Lemma 2.5 (Maximum Principle for CEIDD-Exp2). *Under the assumptions from*

Theorem 2.4, suppose on one subdomain $\Omega_1 := (0, \overline{x}) \times (0, 1)$ *and* $n = 1, \ldots, N$

$$\delta_k z^n_{i,j} + L^{Exp2}_{x,y,h} z^n_{i,j} = g^n_{i,j} \quad \textit{for interior points} \tag{2.63}$$

with $z^n_{0,j} = a_n$, $z^n_{p,j} = b_n$, $z^n_{i,0} = c_n$ *and* $z^n_{i,M} = d_n$ *where* a_n, b_n, c_n *and* d_n *are constants. Then the following estimate must hold at each time level* t_n*:*

$$\max_{\Omega_1} |z^n_{i,j}| \leq \max \{|a_n|, |b_n|, |c_n|, |d_n|, \max_{\Omega_1} |z^{n-1}_{i,j}|\} + k \max_{\Omega_1} |g^n_{i,j}| \tag{2.64}$$

with $1 \leq n \leq N$.

Proof. Suppose the index (r, s) is such that

$$|z^n_{r,s}| = \max_{\Omega_1} |z^n_{i,j}|. \tag{2.65}$$

If $r = 0$ or p or $s = 0$ or M, the result is immediate. Therefore, suppose that (r, s) is such that (x_r, y_s) is an interior point in Ω_1 is chosen. Consider

1. Case 1: $z^n_{r,s} \geq 0$

   Define $\alpha_h = \frac{k}{h^2}\alpha^n_{i,j}$ and $\beta_h = \frac{k}{2h}\beta^n_{i,j}$, then starting with

$$\delta_k z^n_{i,j} + L^{Exp2}_{x,y,h} z^n_{i,j} = g^n_{i,j}, \tag{2.66}$$

$$\begin{aligned} z^n_{r,s} &= z^{n-1}_{r,s} + (\alpha_h - \beta_h) z^n_{r+1,s+1} + (\alpha_h + \beta_h) z^n_{r-1,s+1} \\ &\quad + (-4\alpha_h - k\gamma^n_{i,j}) z^n_{r,s} + (\alpha_h - \beta_h) z^n_{r+1,s-1} + (\alpha_h + \beta_h) z^n_{r-1,s+1} \\ &\quad + k g^n_{r,s} \end{aligned}$$

Using the assumptions from Theorem 2.4,

$$\begin{aligned}
|z^n_{r,s}| &\leq |z^{n-1}_{r,s}| + (\alpha_h - \beta_h)|z^n_{r,s}| + (\alpha_h + \beta_h)|z^n_{r,s}| \\
&\quad + (-4\alpha_h - k\gamma^n_{i,j})z^n_{r,s} + (\alpha_h - \beta_h)|z^n_{r,s}| + (\alpha_h + \beta_h)|z^n_{r,s}| \\
&\quad + k|g^n_{r,s}| \\
&= |z^{n-1}_{r,s}| + 4\alpha_h|z^n_{i,j}| + (-4\alpha_h - k\gamma^n_{i,j})|z^n_{r,s}| + k|g^n_{r,s}| \\
&= |z^{n-1}_{r,s}| - k\gamma^n_{i,j}|z^n_{r,s}| + k|g^n_{r,s}| \\
&\leq |z^{n-1}_{r,s}| + k|g^n_{r,s}| \qquad (2.67)
\end{aligned}$$

2. Case 2: $z^n_{r,s} < 0$

   Similar to the first case and Lemma 2.2,

$$\begin{aligned}
-z^n_{r,s} &= -z^{n-1}_{r,s} + (\alpha_h - \beta_h)(-z^n_{r+1,s+1}) + (\alpha_h + \beta_h)(-z^n_{r-1,s+1}) \\
&\quad + (-4\alpha_h - k\gamma^n_{i,j})(-z^n_{r,s}) + (\alpha_h - \beta_h)(-z^n_{r+1,s-1}) + (\alpha_h + \beta_h)(-z^n_{r-1,s+1}) \\
&\quad + k(-g^n_{r,s})
\end{aligned}$$

   which yields

$$\begin{aligned}
|z^n_{r,s}| &\leq |z^{n-1}_{r,s}| + (\alpha_h - \beta_h)|z^n_{r,s}| + (\alpha_h + \beta_h)|z^n_{r,s}| \\
&\quad + (-4\alpha_h - k\gamma^n_{i,j})z^n_{r,s} + (\alpha_h - \beta_h)|z^n_{r,s}| + (\alpha_h + \beta_h)|z^n_{r,s}| \\
&\quad + k|g^n_{r,s}| \\
&\leq |z^{n-1}_{r,s}| + k|g^n_{r,s}| \qquad (2.68)
\end{aligned}$$

Combining the trivial cases with (2.67) and (2.68) yields the result stated in (2.64) $\square$

Proof of Theorem 2.4. With a new stencil, we need to derive new error equations similar to what was done in the proof of Theorem 2.1, but only for those in space.

- To compute the error of $\triangle_{x,h}u^n_{i,j} + \triangle_{y,h}u^n_{i,j}$, we now use different points for the expansion:

$$\begin{aligned}
u^n_{i+1,j+1} &= u^n_{i,j} + \frac{\partial}{\partial x}u^n_{i,j}(h) + \frac{\partial}{\partial y}u^n_{i,j}(h) \\
&\quad + \frac{\partial^2}{\partial x^2}u^n_{i,j}\frac{h^2}{2} + \frac{\partial}{\partial x}\frac{\partial}{\partial y}u^n_{i,j}\frac{h^2}{2} + \frac{\partial^2}{\partial y^2}u^n_{i,j}\frac{h^2}{2} + \cdots + O(h^4) \quad (2.69)\\
u^n_{i+1,j-1} &= u^n_{i,j} + \frac{\partial}{\partial x}u^n_{i,j}(h) + \frac{\partial}{\partial y}u^n_{i,j}(-h) + \\
&\quad \frac{\partial^2}{\partial x^2}u^n_{i,j}\frac{h^2}{2} + \frac{\partial}{\partial x}\frac{\partial}{\partial y}u^n_{i,j}\frac{-h^2}{2} + \frac{\partial^2}{\partial y^2}u^n_{i,j}\frac{h^2}{2} + \cdots + O(h^4) \quad (2.70)\\
u^n_{i-1,j+1} &= u^n_{i,j} + \frac{\partial}{\partial x}u^n_{i,j}(-h) + \frac{\partial}{\partial y}u^n_{i,j}(h) + \\
&\quad \frac{\partial^2}{\partial x^2}u^n_{i,j}\frac{h^2}{2} + \frac{\partial}{\partial x}\frac{\partial}{\partial y}u^n_{i,j}\frac{-h^2}{2} + \frac{\partial^2}{\partial y^2}u^n_{i,j}\frac{h^2}{2} + \cdots + O(h^4) \quad (2.71)\\
u^n_{i-1,j-1} &= u^n_{i,j} + \frac{\partial}{\partial x}u^n_{i,j}(-h) + \frac{\partial}{\partial y}u^n_{i,j}(-h) + \\
&\quad \frac{\partial^2}{\partial x^2}u^n_{i,j}\frac{h^2}{2} + \frac{\partial}{\partial x}\frac{\partial}{\partial y}u^n_{i,j}\frac{h^2}{2} + \frac{\partial^2}{\partial y^2}u^n_{i,j}\frac{h^2}{2} + \cdots + O(h^4) \quad (2.72)
\end{aligned}$$

When the left side of the equalities are summed we are left with

$$u^n_{i+1,j}+u^n_{i-1,j}+u^n_{i,j+1}+u^n_{i,j-1} = 4u^n_{i,j}+\frac{\partial^2}{\partial x^2}u^n_{i,j}(2h^2)+\frac{\partial^2}{\partial y^2}u^n_{i,j}(2h^2)+O(h^4) \quad (2.73)$$

or

$$\frac{\partial^2}{\partial x^2}u^n_{i,j}(2h^2)+\frac{\partial^2}{\partial y^2}u^n_{i,j}(2h^2) = u^n_{i+1,j}+u^n_{i-1,j}+u^n_{i,j+1}+u^n_{i,j-1}-4u^n_{i,j}+O(h^4). \quad (2.74)$$

The final simplification arises from dividing by $2h^2$.

$$\frac{\partial^2}{\partial x^2}u^n_{i,j}+\frac{\partial^2}{\partial y^2}u^n_{i,j}=(u^n_{i+1,j}+u^n_{i-1,j}+u^n_{i,j+1}+u^n_{i,j-1}-4u^n_{i,j})/(2h^2)+O(h^2). \quad (2.75)$$

- We also compute the error of $\nabla^n_{x,h}u^n_{i,j}+\nabla^n_{y,h}u^n_{i,j}$, we use the expansion to find

$$\begin{aligned}
u^n_{i+1,j+1} &= u^n_{i,j}+\frac{\partial}{\partial x}u^n_{i,j}(h)+\frac{\partial}{\partial y}u^n_{i,j}(h)+ && (2.76)\\
&\quad \frac{\partial^2}{\partial x^2}u^n_{i,j}\frac{h^2}{2}+\frac{\partial}{\partial x}\frac{\partial}{\partial y}u^n_{i,j}\frac{h^2}{2}+\frac{\partial^2}{\partial y^2}u^n_{i,j}\frac{h^2}{2}+O(h^3) && (2.77)\\
u^n_{i-1,j-1} &= u^n_{i,j}+\frac{\partial}{\partial x}u^n_{i,j}(-h)+\frac{\partial}{\partial y}u^n_{i,j}(-h)+ && (2.78)\\
&\quad \frac{\partial^2}{\partial x^2}u^n_{i,j}\frac{h^2}{2}+\frac{\partial}{\partial x}\frac{\partial}{\partial y}u^n_{i,j}\frac{h^2}{2}+\frac{\partial^2}{\partial y^2}u^n_{i,j}\frac{h^2}{2}+O(h^3) && (2.79)
\end{aligned}$$

By looking at the difference of $u^n_{i+1,j+1}-u^n_{i-1,j-1}$, we are left with

$$u^n_{i+1,j+1}-u^n_{i-1,j-1}=\frac{\partial}{\partial x}u^n_{i,j}(2h)+\frac{\partial}{\partial y}u^n_{i,j}(2h)+O(h^3) \quad (2.80)$$

which when simplified yields

$$\frac{\partial}{\partial x}u^n_{i,j}+\frac{\partial}{\partial y}u^n_{i,j}=(u^n_{i+1,j+1}-u^n_{i-1,j-1})/2h+O(h^2). \quad (2.81)$$

Combining these derivations yields error equations as:

$$\begin{aligned}
e^n_* &= e^{n-1}_{p,j}-kL^{Hyb}_{x,y,H}e^{n-1}_{p,j}+kK^n_{*,j}(k+h^2+H^2)\\
\delta_k e^n_{i,j}+L^{BE}_{x,y,h}e^n_{i,j} &= K^n_{i,j}(k+h^2)\\
\delta_k e^n_{p,j}+L^{BE}_{x,y,h}e^n_{p,j} &= K^n_{p,j}(k+h^2)
\end{aligned} \quad (2.82)$$

with $e^n_{0,j}=e^n_{M,j}=e^n_{i,0}=e^n_{i,M}=0$. $K^n_{i,j}$ and $K^n_{*,j}$ represent real numbers depending on u, h, H, and k and are uniformly bounded.

Once again, we only concern ourselves with the error from the new interface predictor as the subdomain solve is backward Euler and the corrector is similar as the predictor. To that end we look at:

$$e_*^n = e_{p,j}^{n-1} - kL_{x,y,H}^{Exp2} e_{p,j}^{n-1} + kK_{*,j}^n (k + H^2). \tag{2.83}$$

Let

$$\begin{aligned} C_1 &= \max_{\Omega_1} \{|K_{i,j}^n| : 1 \leq n \leq N\} \\ C_* &= \max_{\Gamma} \{|K_{*,j}^n| : 1 \leq n \leq N\} \end{aligned}$$

where the constants are independent of the grid.

To obtain our bound on the interface error this time,

$$\begin{aligned} z_*^n &= z_{p,j}^{n-1} - kL_{x,y,H}^{Exp2} + kg_{p,j}^n \\ &= z_{p,j}^{n-1} + \alpha_H (z_{p-q,j+q}^{n-1} + z_{p+q,j+q}^{n-1} - 4z_{p,j}^{n-1} + z_{p+q,j-q}^{n-1} + z_{p-q,j+q}^{n-1}) + \\ &\quad -\beta_H (z_{p+q,j+q}^{n-1} - z_{p-q,j-q}^{n-1} + z_{p-q,j+q}^{n-1} - z_{p+q,j-q}^{n-1}) \\ &\quad -k\gamma_{p,j}^{n-1} z_{p,j}^{n-1} + kg_{p,j}^{n-1} \\ &= (\alpha_H + \beta_H)\, z_{p-q,j+q}^{n-1} + (\alpha_h - \beta_h)\, z_{p+q,j+q}^{n-1} \\ &\quad +(1 - 4\alpha_H - k\gamma_{p,j}^{n-1}) z_{p,j}^{n-1} + (\alpha_H + \beta_H)\, z_{p-q,j-q}^{n-1} \\ &\quad + (\alpha_h - \beta_h)\, z_{p-q,j+q}^{n-1} + kg_{p,j}^{n-1} \end{aligned}$$

The assumptions (2.61) lead to

$$|z_*^n| \leq \max_\Gamma |z_{i,j}^{n-1}| + k|g_{p,j}^{n-1}|. \tag{2.84}$$

Applying the result to e_*^n leaves

$$|e_*^n| \leq \max_{\Omega_1} |e_{p,j}^{n-1}| + kC_*(k + H^2) \tag{2.85}$$

and hence

$$\begin{aligned} \max_{\Omega_1} |e_{i,j}^n| &\leq \max_{\Omega_1} |e_{i,j}^0| + \sum_{w=1}^{n} C'k(k + h^2 + H^2) \\ &\leq \max_{\Omega_1} |e_{i,j}^0| + C'nk(k + h^2 + H^2) \end{aligned} \tag{2.86}$$

The mixture of h^2 and H^2 comes from using the semi-explicit scheme near the boundaries. Therefore, assuming no error with the initial time and $nk \leq T$, it has been shown that

$$||e_{\cdot,\cdot}^n||_\infty \leq C(k + h^2 + H^2) \tag{2.87}$$

uniformly for $1 \leq n \leq N$. □

2.4 Algorithm 4, CEIDD-Lex

Up to this point, we have introduced three two-dimensional algorithms which work by modifying the computation of the predictor on the interface of the artificial boundary Γ. Ideally, one would hope that there might be a simpler method to compute the interface. Recall from Section 1.4 that Mu, Du and Wu [15] did find a method that allowed for extrapolation, a much simpler computation, along the interface.

Unfortunately, they were unable to obtain linear extrapolation. At this point we introduce CEIDD-Lex, a scheme that allows use of linear extrapolation for most points along the interface, and state the following corollary:

Corollary 2.6 (Equivalence of CEIDD-Lex and CEIDD-Exp1)**.** *The scheme, defined by using the predictor on the interface of linear extrapolation is equivalent to the predictor found in CEIDD-Exp1.*

We state the algorithm CEIDD-Lex in its two-dimensional form.

Set $u^0_{i,j} = g$. For $n = 1, 2, \ldots, N$:

1. Similar to CEIDD-Exp1 and CEIDD-Exp2, we have a two-part predictor using CEIDD-Hyb near the boundaries (Figure 2.13):

   (a) For $i = 1, \ldots, q-1$ and $i = M-(q+1), \ldots, M-1$, solve the system arising from

   $$u^n_{p,j} = u^{n-1}_{p,j} - kL^{Hyb}_{x,y,H} u^{n-1}_{p,j} + kf^{n-1}_{p,j} \tag{2.88}$$

   and assign the values to the temporary placeholders u^n_*.

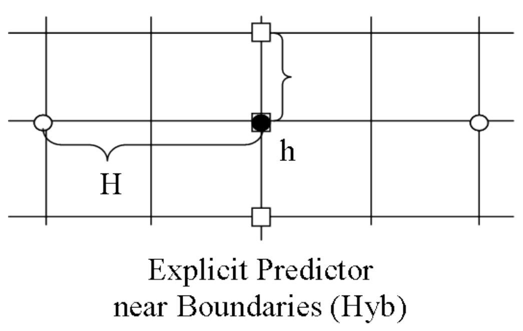
   (b) For $i = q, \ldots, M-q$, solve the system arising from

   $$u^n_{p,j} = 2u^{n-1}_{p,j} - u^{n-2}_{p,j} + kf^{n-1}_{p,j} \tag{2.89}$$

   and assign the values to the temporary placeholders u^n_*.

2. Solve each subdomain Ω_1 and Ω_2 in parallel using an implicit solver with a fine grid.

   $$\delta_k u^n_{i,j} + L^{BE}_{x,y,h} u^n_{i,j} = f^n_{i,j}. \tag{2.90}$$



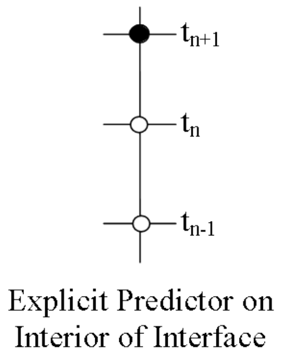


Figure 2.13: Stencils used for predictor with CEIDD-Lex.

3. Discard the predicted interface u_*^n and correct the interface using information from the newly computed subdomains using an implicit method on the fine grid.

$$\delta_k u_{p,j}^n + L^* u_{p,j}^n = f_{p,j}^n. \tag{2.91}$$

This corollary allows for the use of a no-communication predictor on the interior portion of the interface ($j = q, \dots, M - q$). Again, while not perfect, the scheme tries to take advantage of the ability to be applied to convection-diffusion equations something that the extrapolation technique of Du, Mu and Wu [15] were not able to accomplish.

Proof of Corollary 2.6. If we look at the predictor from CEIDD-Exp1

$$u^n = u^{n-1} + kL^{Exp1}u^{n-1} + kg^{n-1}. \tag{2.92}$$

The previous time-step correction has the form

$$u^{n-1} = u^{n-2} + kL^{Exp1}u^{n-1} + kg^{n-1}. \tag{2.93}$$

Solving the last equation for $kL^{Exp1}u^{n-1}$ yields

$$kL^{Exp1}u^{n-1} = u^{n-1} - u^{n-2} - kg^{n-1}. \tag{2.94}$$

Substituting (2.94) into (2.92) gives

$$\begin{aligned} u^n &= u^{n-1} + \left[u^{n-1} - u^{n-2} - kg^{n-1}\right] + kg^{n-1} \\ &= 2u^{n-1} - u^{n-2}. \end{aligned} \tag{2.95}$$

Hence, the conclusion that CEIDD-Exp1 is equivalent to linear extrapolation (2.95).

$\square$

Because we are using the pointwise manipulation of the interface, we are able to arrive at a scheme that shares the properties of maximum principle and error estimate by recognizing the relationship between the predictor and corrector of different time levels.

Chapter 3

Numerical Results

3.1 Setup

In this section, we look at experimental results of using each of the algorithms presented in Chapter 2 using four model problems. The experiments were run using MPICH, version 1.2.7p via the GCC compiler under LinuxMint 5. The timings were run on a machine using an Intel Core 2 Duo CPU running at 2.4 GHz and having 3 GB of memory. Due to circumstances beyond control, the original cluster of machines that was to be used for the experiments was decommissioned prior to the code being finished. While this caused some limitation to the size of mesh refinement we could use in the experiments, we were able to gather enough data to to make conclusions regarding the results.

Because we are using a single machine with a dual-core processor, we do not have some of the communication cost that would come with a cluster which would have the nodes connected via ethernet. This is due to the decreased communication cost moving data from one node to another. In a dual-core setup as ours, the communication is minimized because we using a shared memory architecture. This drop in communication should manifest itself in the computational timings of the examples coming close to halving the time necessary to run the experiment when doubling the number of subdomains (processors).

To demonstrate the performance of the new algorithms, we recall the general parabolic differential equation which we have been working:

$$\partial u/\partial t + Lu \quad = \quad f \tag{3.1}$$

where L is a uniformly elliptic operator which has the following form

$$Lu = -\sum_{i,j=1}^{d} \frac{\partial}{\partial x_i}(\alpha_{ij}(x)\frac{\partial u}{\partial x_j}) + \sum_{i=1}^{d} \beta_i(x)\frac{\partial u}{\partial x_i} + \gamma(x)u \tag{3.2}$$

and the coefficients are continuous on the domain and $\alpha_{ij}(x) = \alpha_{ji}(x)$ for all i, j and x.

The four test problems were culled from other papers or were problems that were slightly modified to better fit the memory constraints of the hardware. The model problems start with an example of the heat equation moving to two reaction-diffusion equations, one stable and the other unstable. We then conclude with an example of a convection-diffusion problem. Each example will give a brief description of the problem, including a graph of the solution. Then tables of the maximum errors are given for each of the five algorithms (backward Euler, CEIDD-Hyb, Exp1, Exp2 and Lex).

The experiments were run with two and four processors by first fixing the time step at $\triangle t = 0.001$ and letting $h \to 0$ to see the effect of the coarse grid and then looking at the results of letting $h \to 0$ while defining the time step as $\triangle t = h^2$ to make sure the convergence is what was predicted in Chapter 2. Each section then gives a table relating the run times of each algorithm under the same conditions and finishes with observations obtained from the experiment.

3.2 Example 1

The first example to be examined was chosen to test the algorithms with a known set of results. In this case, a standard heat equation with homogeneous boundary conditions was used as seen in [48]:

$$\begin{aligned} u_t &= \frac{1}{\pi^2}\triangle u \\ u(x,y,0) &= u_0(x,y) \end{aligned}$$

on the square domain of $\Omega = [0,1]\times[0,1]$ with true solution $u(x,y,t) = e^{-2t}\sin(\pi x)\sin(\pi y)$. We look at the results at $T = 1.0$ and see the graph of the solution in Figure 3.1.

We see in Table 3.1 and Table 3.2 that the errors keep pace to that of the serial backward Euler using either two or four processors. In particular, CEIDD-Exp2 lags behind in performance compared to the other schemes. Notice the necessity to adjust the coarse grid as $h \to 0$ by looking at the change of q ($q = H/h$). This control of the coarse grid allows us to be less concerned with the size of the time step for the explicit portion of the predictor. Lastly, we point out that in this example, CEIDD-Lex outperforms the other schemes including backward Euler. This is a very welcomed result.

Next, we turn our attention to examine the times in Table 3.3 and see that the parallelized algorithms cut the run times by close to 45% of the serial backward Euler. The 5% discrepancy from the optimal halving can be attributed to the communication

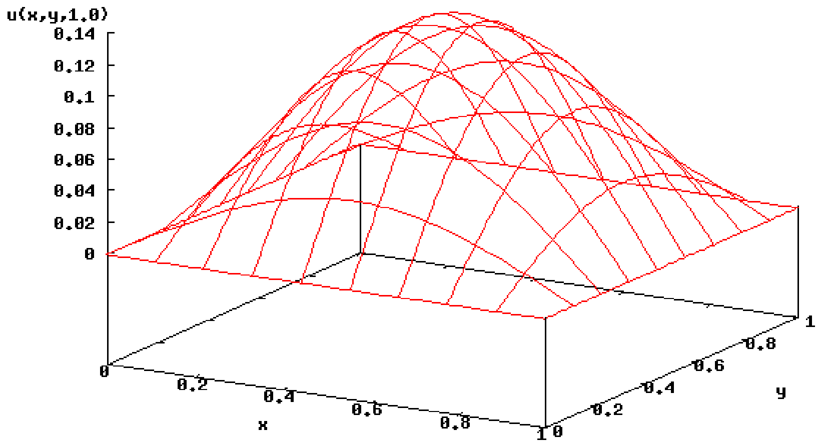


Figure 3.1: Graph of $u(x,y,t) = e^{-2t}\sin(\pi x)\sin(\pi y)$ at $T = 1.0$.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.20	2	2.508e-03	2.520e-03	2.532e-03	2.630e-03	2.506e-03
0.05	0.10	2	8.278e-04	8.329e-04	8.379e-04	8.967e-04	8.229e-04
0.025	0.05	2	4.098e-04	4.087e-04	4.076e-04	4.373e-04	3.993e-04
0.01	0.05	5	2.929e-04	2.927e-04	2.926e-04	3.552e-04	2.660e-04
0.005	0.045	9	2.761e-04	2.735e-04	2.708e-04	3.463e-04	2.236e-04

Table 3.1: Maximum Errors for Example 1 at $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.05	0.10	2	8.278e-04	8.367e-04	8.456e-04	9.492e-04	8.192e-04
0.025	0.05	2	4.098e-04	4.079e-04	4.060e-04	4.572e-04	3.918e-04
0.01	0.05	5	2.929e-04	2.926e-04	2.924e-04	3.989e-04	2.472e-04
0.005	0.045	9	2.761e-04	2.716e-04	2.671e-04	3.949e-04	1.860e-04

Table 3.2: Maximum Errors for Example 1 at $T = 1.0$ and $\triangle t = 0.001$ with 4 Processors.

necessary for the computation of the interface values between the two processors as mentioned in 1.2.

$\triangle x$	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0	0	0	0	0
0.05	0	0	1	0	0
0.025	1	1	0	1	1
0.01	24	14	14	14	14
0.005	309	176	175	176	175

Table 3.3: Processing Times (in seconds) for Example 1 with $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors.

When we look at running the algorithms with a time step that is scaling with the change in the spatial mesh ($\triangle t = 0.5h$), we see similar results. Those being CEIDD-Hyb and Exp1 being close in performance to the serial algorithm, CEIDD-Exp2 falling behind in performance and Lex outperforming all the algorithms. It seems that reducing the use subdomain elements actually improves the performance of the algorithm as seen in Table 3.4 and Table 3.5.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.40	4	1.554e-02	1.620e-02	1.667e-02	2.439e-02	1.260e-02
0.05	0.25	5	7.267e-03	7.288e-03	7.308e-03	1.717e-02	4.537e-03
0.025	0.175	7	3.508e-03	3.510e-03	3.511e-03	8.178e-03	2.042e-03
0.01	0.10	10	1.373e-03	1.342e-03	1.310e-03	2.472e-03	7.670e-04
0.005	0.075	15	6.817e-04	6.760e-04	6.703e-04	1.184e-03	3.772e-04

Table 3.4: Maximum Errors for Example 1 at $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors.

In any use of the algorithms, we see in Table 3.6 similar performance gains by implementing the parallel code on two processors.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.40	4	1.554e-02	3.341e-02	3.368e-02	6.359e-02	4.182e-02
0.05	0.25	5	7.267e-03	7.303e-03	7.336e-03	1.152e-02	2.540e-03
0.025	0.175	7	3.508e-03	3.511e-03	3.512e-03	1.133e-02	9.224e-04
0.01	0.10	10	1.373e-03	1.320e-03	1.266e-03	3.229e-03	2.985e-04
0.005	0.075	15	6.817e-04	6.720e-04	6.624e-04	1.530e-03	1.414e-04

Table 3.5: Maximum Errors for Example 1 at $T = 1.0$ and $\triangle t = 0.5h$ with 4 Processors.

$\triangle x$	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0	0	0	0	0
0.05	0	0	0	0	0
0.025	1	1	1	1	1
0.01	19	11	11	11	11
0.005	287	160	160	160	160

Table 3.6: Processing Times (in seconds) for Example 1 with $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors.

3.3 Example 2

The next example examined is a reaction-diffusion equation. The reaction-diffusion equation takes the parabolic differential equation with $\beta = 0$. These type of equations model how the concentration of one or more substances distributed in space change under the influence of the local chemical reaction in which substances are converted into one another and the diffusion which cause the spread of substances into space.

The can be used in a variety of situations from biology (Fisher's equation that was used to describe the spread of biological processes) to recent developments in pattern formations (finding fronts, spirals, targets, hexagons and stripes).

The example chosen is similar to that in [48] adjusted to utilize the square domain of $\Omega = [0,1] \times [0,1]$:

$$\begin{aligned} u_t &= \triangle u + 8u \\ u(x,y,0) &= u_0(x,y) \end{aligned}$$

and the true solution is $u(x,y,t) = e^{-2t}\cos(3x+y)$. We look at the results at $T = 1.0$ and see the graph of the solution in Figure 3.3.

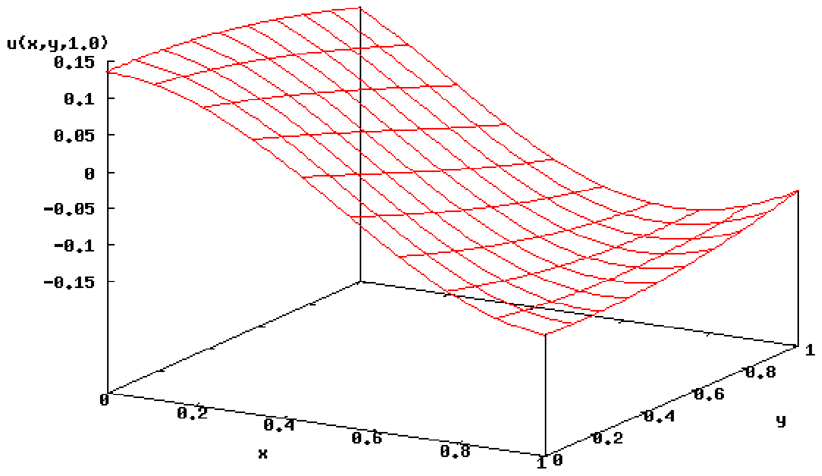


Figure 3.2: Graph of $u(x,y,t) = e^{-2t}\cos(3x+y)$ at $T = 1.0$.

Looking at Tables 3.7 and 3.8, we see that our parallel algorithms keep up with the serial backward Euler, but not with the same precision as in Example 3.2. Again, the one result that yields a positive response is the implementation of the CEIDD-Lex algorithm. The errors come in under those obtained from the serial program and easily outperforms the other parallel algorithms.

We take a moment to look at the relationship with coarse grid under the new problem. We notice that the ability to modify the coarse grid is necessary as the reaction term forces us to implement a more coarse grid to satisfy the constraints of the theorems. This ability to adapt to a given type of problem is a positive aspect that should not be underestimated. To do this with the different types of parabolic PDEs is another aspect that the reader should be reminded of as previously published results do not allow for these extensions.

With the constraints and implementation of the coarse grid, we are able to obtain consistency without adding the extra time steps that occur from the explicit portion of the algorithms giving the user the flexibility to tailor the time steps with what is reasonable. This is particularly impressive considering the fact that we are using linear extrapolation in the last algorithm.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.20	2	5.941e-04	6.437e-04	6.476e-04	6.954e-04	5.967e-04
0.05	0.10	2	1.647e-04	1.782e-04	1.841e-04	2.048e-04	1.630e-04
0.025	0.075	3	5.408e-05	6.357e-05	7.292e-05	9.202e-05	5.040e-05
0.01	0.07	7	2.315e-05	3.482e-05	4.884e-05	7.157e-05	1.474e-05
0.005	0.065	13	1.873e-05	2.730e-05	4.181e-05	6.231e-05	1.165e-05

Table 3.7: Maximum Errors for Example 2 at $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.05	0.10	2	1.674e-04	2.000e-04	2.154e-04	3.288e-02	1.601e-04
0.025	0.075	3	5.408e-05	7.800e-05	9.972e-05	2.647e-04	4.710e-05
0.01	0.07	7	2.315e-05	5.039e-05	7.732e-05	1.389e-04	1.262e-05
0.005	0.065	13	1.873e-05	3.895e-05	6.670e-05	1.203e-04	1.125e-05

Table 3.8: Maximum Errors for Example 2 at $T = 1.0$ and $\triangle t = 0.001$ with 4 Processors.

In the timings found in Table 3.9, we find ourselves again at an approximately

45% decrease in the times necessary for the computations.

$\triangle x$	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0	0	0	0	0
0.05	1	1	0	1	0
0.025	6	3	4	3	3
0.01	186	101	101	101	102
0.005	2783	1481	1473	1476	1473

Table 3.9: Processing Times (in seconds) for Example 2 with $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors.

This experiment yields the greatest variance of errors out of the four examples and when we allow for $\triangle t$ to be equal to $0.5h$, only CEIDD-Hyb comes close to the result of the serial computation and it is even worse off when using four processors as can be seen in Table 3.11.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.60	6	1.466e-03	1.905e-02	1.905e-02	3.402e-02	1.905e-02
0.05	0.40	8	5.889e-04	3.462e-03	4.492e-03	1.126e-02	2.337e-03
0.025	0.25	10	2.547e-04	9.208e-04	1.425e-03	3.876e-03	2.440e-04
0.01	0.15	15	9.247e-05	2.269e-04	3.842e-04	7.945e-04	2.209e-04
0.005	0.105	21	4.467e-05	8.929e-05	1.491e-04	2.736e-04	1.266e-04

Table 3.10: Maximum Errors for Example 2 at $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.60	6	1.466e-03	4.787e-02	4.878e-02	4.386e-02	4.516e-02
0.05	0.40	8	5.889e-04	3.052e-02	2.808e-02	6.318e-03	2.365e-02
0.025	0.25	10	2.547e-04	1.496e-03	2.450e-03	5.960e-02	7.112e-04
0.01	0.15	15	9.247e-05	3.612e-04	6.613e-04	1.326e-03	4.958e-04
0.005	0.105	21	4.467e-05	1.368e-04	2.498e-04	4.468e-04	2.716e-04

Table 3.11: Maximum Errors for Example 2 at $T = 1.0$ and $\triangle t = 0.5h$ with 4 Processors.

$\triangle x$	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0	0	0	0	0
0.05	0	1	0	0	0
0.025	4	1	2	2	2
0.01	168	80	79	79	79
0.005	2677	1342	1345	1341	1336

Table 3.12: Processing Times (in seconds) for Example 2 with $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors.

3.4 Example 3

We use an unstable reaction-diffusion problem to gauge the algorithms ability to handle a situation that the solution is not uniformly bounded. In this case, we look to the following problem:

$$\begin{aligned} u_t &= \triangle u + (1 + 5\pi^2)u \\ u(x, y, 0) &= u_0(x, y) \end{aligned}$$

on the square domain of $\Omega = [0, 1]\times[0, 1]$ with true solution $u(x, y, t) = e^t \sin(2\pi x) \sin(\pi y)$. Again, we look at the results at $T = 1.0$ and see the graph of the solution in Figure 3.4.

Since one solution is $u(x, y, t) = e^t \sin(2\pi x) \sin(\pi y)$, we see that there is no uniform bound due to the exponential term as time progresses and hence is unstable.

What is seen in Table 3.13 is, with a relatively conservative step size, the parallel algorithms outperform backward Euler by a large margin after the initial grid size of $\triangle x = 0.10$. Yet, the stability of the solution affects the results when we increase the number of processors. While CEIDD-Lex continues to yield similar results with four processors, CEIDD-Exp2 cannot seem to regain its efficiency.

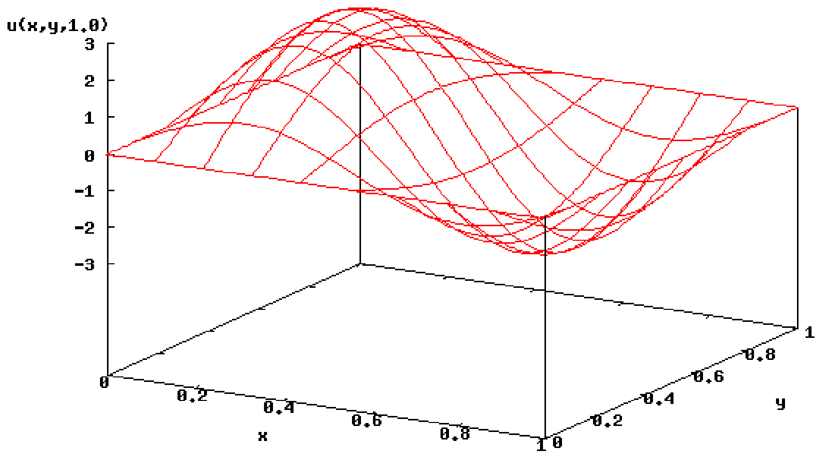


Figure 3.3: Graph of $u(x,y,t) = e^t \sin(2\pi x) \sin(\pi y)$ at $T = 1.0$.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.20	2	7.547e+00	7.543e+00	7.543e+00	7.543e+00	7.543e+00
0.05	0.10	2	1.130e+00	1.117e+00	1.121e+00	1.121e+00	1.121e+00
0.025	0.075	3	3.238e-01	2.453e-01	2.474e-01	2.474e-01	2.473e-01
0.01	0.07	7	2.753e-01	3.787e-02	3.801e-02	4.038e-02	3.924e-02
0.005	0.65	13	3.429e-01	1.252e-02	1.231e-02	1.234e-02	1.131e-02

Table 3.13: Maximum Errors for Example 3 at $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.20	2	7.547e+00	9.870e+11	9.337e+11	6.284e+11	5.604e+11
0.05	0.10	2	1.130e+00	1.663e+00	1.645e+00	2.803e+00	1.121e+00
0.025	0.075	3	3.238e-01	6.622e-01	6.470e-01	1.392e+00	2.479e-01
0.01	0.07	7	2.753e-01	5.131e-01	4.995e-01	1.307e+00	4.149e-02
0.005	0.65	13	3.429e-01	4.349e-01	4.209e-01	1.124e+00	1.156e-02

Table 3.14: Maximum Errors for Example 3 at $T = 1.0$ and $\triangle t = 0.001$ with 4 Processors.

In regards to the timings in Table 3.15, the results are consistent to what has been shown in Example 1 and 2.

$\triangle x$	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0	0	0	0	1
0.05	0	1	0	0	0
0.025	6	3	3	3	3
0.01	172	93	95	94	94
0.005	2513	1314	1312	1314	1315

Table 3.15: Processing Times (in seconds) for Example 3 with $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors.

This example had some interesting results, as can be seen in Table 3.16 and Table 3.17, where the errors performance of every parallel algorithm outperformed their serial counterpart. Particularly noting the improvement when using four processors, the algorithms were able to handle the unstable problem quite well.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.30	3	9.586e+00	8.320e+00	7.820e+00	7.813e+00	7.808e+00
0.05	0.15	3	1.160e+00	1.125e+00	1.125e+00	1.125e+00	1.125e+00
0.025	0.075	3	2.915e-01	2.467e-01	2.468e-01	2.468e-01	2.468e-01
0.01	0.03	3	8.552e-02	3.856e-02	3.852e-02	3.850e-02	3.855e-02
0.005	0.015	3	5.648e-02	1.019e-02	1.019e-02	1.019e-02	1.019e-02

Table 3.16: Maximum Errors for Example 3 at $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0.30	3	9.586e+00	7.346e+16	1.477e+14	2.090e+14	2.063e+13
0.05	0.15	3	1.160e+00	5.470e+00	4.959e+00	1.863e+01	1.211e+00
0.025	0.075	3	2.915e-01	5.395e-01	5.372e-01	1.056e+00	2.461e-01
0.01	0.03	3	8.552e-02	6.260e-02	4.800e-02	8.900e-02	4.615e-02

Table 3.17: Maximum Errors for Example 3 at $T = 1.0$ and $\triangle t = 0.5h$ with 4 Processors.

$\triangle x$	BEuler	Hyb	Exp1	Exp2	Lex
0.10	0	0	0	0	0
0.05	1	1	0	1	0
0.025	6	3	3	3	3
0.01	248	128	128	129	128
0.005	3895	2046	2042	2058	2062

Table 3.18: Processing Times (in seconds) for Example 3 with $T = 1.0$ and $\triangle t = 0.5h$ with 2 rocessors.

3.5 Example 4

Lastly, the convection-diffusion problem describes the physical phenomena where particles of energy are transferred inside a physical system due to two processes: diffusion and convection. Convection refers to the movement of molecules within fluids. In one of the two major types of heat convection, the heat is carried passively by a fluid motion (sometimes called heat advection). The other type of convection occurs when the heating itself causes the fluid motion by expansion or buoyancy force (called natural convection). The last example presented is a convection-diffusion problem based on an example from [48]:

$$\begin{aligned} u_t &= \triangle u + 9.9\sin(x)u_x - 9.9\cos(x)u \\ u(x,y,0) &= u_0(x,y) \end{aligned}$$

on the domain $\Omega = [0, 2\pi] \times [0, \pi]$ and the true solution is $u(x,y,t) = e^{-2t}\sin(x)\sin(y)$. The graph of the solution is given in Figure 3.4 at $T = 1.0$.

In this example, we do not see much variation in the errors in comparison to the serial version. In fact, with two processors for $\triangle t = 0.001$, eight decimal places were needed to see any difference in the error. It is not until we look at Table 3.20 with

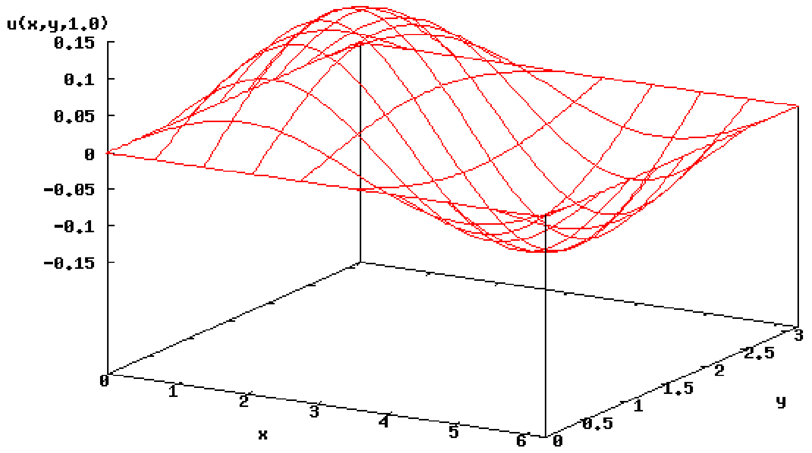


Figure 3.4: Graph of $u(x, y, t) = e^{-2t} \sin(x) \sin(y)$ at $T = 1.0$.

four processors that we see some variations in the errors. In this case, each of the parallel algorithms performs better with error on time with CEIDD-Lex once again working the best.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
$\pi/10$	$\pi/5$	2	2.176e-02	2.176e-02	2.177e-02	2.176e-02	2.176e-02
$\pi/20$	$\pi/10$	2	5.318e-03	5.318e-03	5.318e-03	5.318e-03	5.318e-03
$\pi/40$	$\pi/20$	2	1.514e-03	1.514e-03	1.514e-03	1.514e-03	1.514e-03
$\pi/100$	$3\pi/10$	3	4.683e-04	4.684e-04	4.683e-04	4.683e-04	4.683e-04
$\pi/200$	$\pi/40$	5	3.199e-04	3.199e-04	3.199e-04	3.199e-04	3.199e-04

Table 3.19: Maximum Errors for Example 4 at $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors.

Even with a larger domain, the results of the algorithms hold up well in comparison to the serial algorithm yielding a increase of speed close to 47%.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
$\pi/10$	$\pi/5$	2	2.176e-02	2.176e-02	2.176e-02	2.175e-02	2.176e-02
$\pi/20$	$\pi/10$	2	5.318e-03	5.318e-03	5.318e-03	5.318e-03	5.318e-03
$\pi/40$	$\pi/20$	2	1.514e-03	1.514e-03	1.514e-03	3.588e-03	1.514e-03
$\pi/100$	$3\pi/10$	3	4.683e-04	4.677e-04	4.672e-04	5.763e-03	4.662e-04
$\pi/200$	$\pi/40$	5	3.199e-04	3.190e-04	3.182e-04	3.182e-04	3.159e-04

Table 3.20: Maximum Errors for Example 4 at $T = 1.0$ and $\triangle t = 0.001$ with 4 Processors.

$\triangle x$	BEuler	Hyb	Exp1	Exp2	Lex
$\pi/10$	1	0	0	0	0
$\pi/20$	1	1	1	1	0
$\pi/40$	3	1	1	1	2
$\pi/100$	57	28	28	29	28
$\pi/200$	703	380	375	375	381

Table 3.21: Processing Times (in seconds) for Example 4 with $T = 1.0$ and $\triangle t = 0.001$ with 2 Processors.

In Table 3.22 and Table 3.23 we see that the results hold, but Exp2 continues to have problems scaling to more processors. While the errors are decreasing it is not near the rate of the other algorithms.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
$\pi/10$	$\pi/5$	2	3.445e-02	3.455e-02	3.455e-02	3.455e-02	3.455e-02
$\pi/20$	$\pi/10$	2	1.172e-02	1.175e-02	1.175e-02	1.175e-02	1.175e-02
$\pi/40$	$\pi/20$	3	4.606e-03	4.607e-03	4.607e-03	4.607e-03	4.607e-03
$\pi/100$	$\pi/50$	5	1.548e-03	1.548e-03	1.548e-03	1.548e-03	1.548e-03
$\pi/200$	$\pi/100$	7	7.254e-04	7.254e-04	7.254e-04	7.254e-04	7.254e-04

Table 3.22: Maximum Errors for Example 4 at $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors.

$\triangle x$	H	q	BEuler	Hyb	Exp1	Exp2	Lex
$\pi/10$	$\pi/5$	2	3.445e-02	3.472e-02	3.494e-02	4.339e-02	3.506e-02
$\pi/20$	$\pi/10$	2	1.172e-02	1.224e-02	1.203e-02	3.090e-02	1.169e-02
$\pi/40$	$\pi/20$	3	4.606e-03	4.657e-03	4.597e-03	2.564e-02	4.573e-03
$\pi/100$	$\pi/50$	5	1.548e-03	1.538e-03	1.531e-03	1.957e-02	1.470e-03
$\pi/200$	$\pi/100$	7	7.254e-04	7.213e-04	7.182e-04	1.372e-02	6.830e-04

Table 3.23: Maximum Errors for Example 4 at $T = 1.0$ and $\triangle t = 0.5h$ with 4 Processors.

$\triangle x$	BEuler	Hyb	Exp1	Exp2	Lex
$\pi/10$	3	2	1	2	0
$\pi/20$	0	0	0	0	0
$\pi/40$	1	0	1	0	1
$\pi/100$	42	22	21	22	22
$\pi/200$	635	348	348	348	347

Table 3.24: Processing Times (in seconds) for Example 4 with $T = 1.0$ and $\triangle t = 0.5h$ with 2 Processors.

3.6 Conclusions

In this chapter, we have worked to see how our four algorithms (CEIDD-Hyb, Exp1, Exp2 and Lex) faired with four different test problems that started with the heat equation, progressed to a stable reaction-diffusion equation and then to an unstable reaction-diffusion equation. We concluded the experiments with a convection-diffusion example. Each of the examples were run on a dual-core processor, which while limited by memory size, was still able to test a reasonable size discretization in the spatial variable.

Each algorithm was implemented using C, augmented by MPICH as the library of functions necessary to implement the message passing interface needed to send and receive information from each subdomain, and uses the same iterative solver for the subdomain solution at each time step. What varied was the computation of the interface for each algorithm. CEIDD-Hyb uses a predictor that requires the use of an explicit finite difference scheme in the x-direction, while using an implicit scheme along the interface.

CEIDD-Exp1, as well as the remaining algorithms, required two different schemes to compute the interface. To implement Exp1, one must first use CEIDD-Hyb for the first and last p nodes along the interface, while using an explicit forward Euler scheme with the coarse grid for the remaining nodes on the interface. What changes for CEIDD-Exp2 is the computation of the remaining interface after using Hyb. In this instance, we created a five-point stencil that is, effectively, rotated 45°. Everything else is the same as the previous scheme. We then finished with CEIDD-Lex, a scheme that is implemented by using standard linear extrapolation along the interior of the interface.

The examples illustrate the notion of the importance of the coarse grid to offset the fact that we are using an explicit computation along the interface. We see this by looking at the necessity p takes on to meet the constraints of the algorithms, for example in Tables 3.1 or Table 3.19. When looking at the results, we see that the further we shrink the relationship of $\triangle t \approx h$, the larger the coarse overlay must be to compensate.

While running the experiments, the extra requirement that manifests itself in CEIDD-Exp2 with a more strict constraint of

$$H \geq \sqrt{\frac{4\underline{\alpha}k}{1-\overline{\gamma}k}}$$

causes less give regarding the coarse grid and also achieved less than stellar results. These results arose in subpar errors compared to the other schemes and also came through as a lack of scalability of the algorithm as seen in the results shown in Table 3.8 or Table 3.14.

In the end, it would be recommended that a user wishing to implement a domain decomposition method should use either CEIDD-Hyb or CEIDD-Lex. While CEIDD-Exp1 works adequately, the amount of work to modify the code would be better spent modifying the code to use CEIDD-Lex and yield better results both in error and scalability.

Chapter 4

Open Problems

4.1 Open Problems

One immediate problem that one could investigate deals with the types of differential equations we worked with, in this case linear. It would be extremely useful to extend these results to their most general instance with the inclusion of non-linear equations. Developing these ideas non-linear problems in mind would increase the usefulness of the algorithms to a much larger class of problems.

Another immediate consequence that should be explored is the sensitivity of the constraints to the problems. In Table 4.1, we look at CEIDD-Hyb and CEIDD-Lex with the heat equation from Example 1 at $\triangle t = 0.001$, $h = 0.005$ and vary the value of $q = H/h$ which controls the coarse grid. Recall from Chapter 3, the actual value to be assigned to q in this case should have been $q = 9$.

From Table 4.1, it seems that our constraint on the value of H may be a little stronger than needed for our problem. We do have reinforcement that the coarse grid is important as we see in the difference of the first two lines in the table. While

q	Hyb Error	Hyb Timing	Lex Error	Lex Timing
1	1.353e-01	449	1.353e-01	282
2	2.675e-04	179	2.235e-04	174
3	2.673e-04	175	2.235e-04	176
4	2.670e-04	178	2.235e-04	176
5	2.673e-04	178	2.235e-04	174
6	2.680e-04	177	2.235e-04	176
7	2.680e-04	177	2.236e-04	175
8	2.711e-04	176	2.236e-04	177
9	2.764e-04	175	2.236e-04	176

Table 4.1: Errors and Timings for Example 1 with 2 Processors Varying the Coarse Grid Size.

CEIDD-Lex seems a little more resilient to those issues (smaller error and less time) either method benefitted from the implementation of the coarse grid.

Another direction of this work is to use the coarse grid to extend other methods that use the coarse grid notion. In Zhang and Shen [47], the authors propose another method of computing the interface of the one-dimensional heat equation by developing Saul'yev's asymmetric scheme. This scheme computes the left and right interface points of the subdomains by

$$\begin{aligned} \delta_k u_i^n - (1/H)(\partial_{x,H} u_{i+1}^{n-1} - \partial_{x,H} u_i^n) &= 0 \\ \delta_k u_i^n - (1/H)(\partial_{x,H} u_{i+1}^{n} - \partial_{x,H} u_i^{n-1}) &= 0 \end{aligned}$$

The methods the authors used to prove their claim are very similar to Dawson, Du and Dupont [13]. Because of this, the methods presented in this dissertation have a good probability of extending this method to not only the more general convection-diffusion equation, but also to a two-dimensional differential equation.

One last suggestion for extension deals with the dimension of the problem. One would like to think that extending the ideas to three-dimensions might be simple.

Again, the interface values near the boundary must require the analogue of CEIDD-Hyb which has issues when showing the maximum principle if the same proof technique is used. What has been accomplished, and is shown in the next section, is part of the extension of CEIDD-Exp2 into three-dimensions. A nine-point Laplacian that has been rotated similarly as the two-dimensional case is implemented to take values off the artificial boundary. While the two-diminsional algorithm did not achieve the most consistent results, being able to prove this higher-dimensional case lends hope that one might find a proof with CEIDD-Exp1 and through a similar result, CEIDD-Lex.

4.2 Algorithm 5, CEIDD-Exp3d

We finish our exploration of open problems with an extension into three dimensions by showing a proof of the error estimate for CEIDD-Exp2. In three dimensions, our interface is now a plane. Because we have a larger interface and want to use a rotated nine-point stencil (Figure 4.1), we need to have the analogue of CEIDD-Hyb prior to completely extending this algorithm.

To reach a partial result, we will split the three-dimensional bounded and connected domain along the plane $x = x_p$. In three dimensions, we define the discrete operators as the three-dimensional analogues obtained from their respective Taylor expansions:

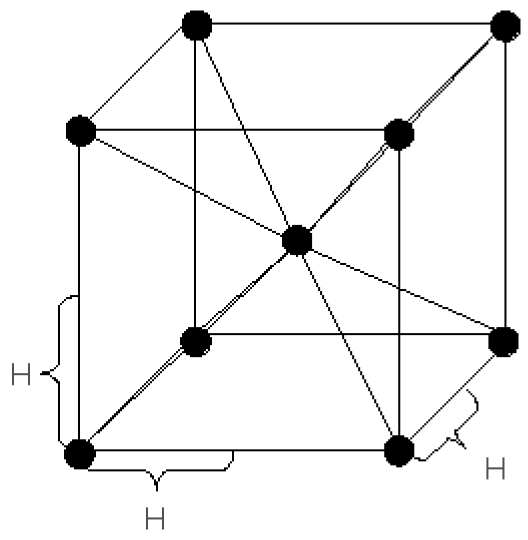


Figure 4.1: Stencil produced by using CEIDD-Exp3D.

$$
\begin{aligned}
L_h^{Exp3D} &= -\alpha_{i,j,l}^n \triangle_{3,h} + \beta_{i,j,l} \cdot \nabla_{3,h} + \gamma_{i,j,l}^n \\
\triangle_{3,h} u_{i,j,l}^n &= (u_{i+1,j+1,l+1}^n + u_{i+1,j+1,l-1}^n + u_{i+1,j-1,l+1}^n \\
&\quad + u_{i-1,j+1,l+1}^n - 8u_{i,j,l}^n + u_{i-1,j-1,l+1}^n + u_{i-1,j+1,l-1}^n \\
&\quad + u_{i+1,j-1,l-1}^n + u_{i-1,j-1,l-1}^n)/4h^2 \\
\nabla_{3,h} u_{i,j,l}^n &= (u_{i+1,j+1,l+1}^n - u_{i+1,j+1,l-1}^n + u_{i+1,j+1,l+1}^n - u_{i+1,j+1,l-1}^n \\
&\quad + u_{i+1,j+1,l+1}^n - u_{i+1,j+1,l-1}^n + u_{i+1,j+1,l+1}^n - u_{i+1,j+1,l-1}^n)/4h \\
\triangle_{3,H} u_{i,j,l}^n &= (u_{i+q,j+q,l+q}^n + u_{i+q,j+q,l-q}^n + u_{i+q,j-q,l+q}^n \\
&\quad + u_{i-q,j+q,l+q}^n - 8u_{i,j,l}^n + u_{i-q,j-q,l+q}^n + u_{i-q,j+q,l-q}^n \\
&\quad + u_{i+q,j-q,l-q}^n + u_{i-q,j-q,l-q}^n)/4H^2
\end{aligned}
$$

Next, we define the algorithm as:

Set $u_{i,j,l}^0 = g$, then for $n = 1, 2, \ldots, N$:

1. Compute the interface using an explicit predictor on the coarse grid and then assign the values to a temporary placeholder u_*^n.

$$u_{p,j,l}^n = u_{p,j,l}^{n-1} - kL_H^{Exp3D} u_{p,j,l}^{n-1} + kf_{p,j,l}^{n-1} \tag{4.1}$$

2. Solve each subdomain $\Omega_1, \Omega_2, \dots$ in parallel using an implicit solver with a fine grid.

$$\delta_k u_{i,j,l}^n + L_h^{BE} u_{i,j,l}^n = f_{i,j,l}^n. \tag{4.2}$$

3. Discard the predicted interface u_*^n and correct the interface using information from the newly computed subdomains using an implicit method on the fine grid.

$$\delta_k u_{p,j,l}^n + L_h^{C3D} u_{p,j,l}^n = f_{p,j,l}^n. \tag{4.3}$$

The three-dimensional analogue of Theorem 2.4 is given as:

Theorem 4.1. *Suppose $u(x,t) \in C_1^3(\Omega)$ and the following conditions hold:*

$$k\overline{\gamma} < 1, H \geq \sqrt{\frac{2\underline{\alpha}k}{1-\overline{\gamma}k}}, \text{ and } h \leq H \leq \frac{2\underline{\alpha}}{\overline{\overline{\beta}}} \tag{4.4}$$

Then, for CEIDD-C3D there exists a constant c, independent of the grid, such that for $1 \leq i,j,l \leq M$ and $1 \leq n \leq N$

$$|u(x_i, y_j, z_l, t_n) - u_{i,j,l}^n| \leq c(k + h^2 + H^2). \tag{4.5}$$

Next, the maximum principle for our new stencil.

Lemma 4.2. *Under the similar assumptions from Theorem 2.4, suppose on one subdomain* $\Omega_1 := (0, \overline{x}) \times (0,1)^2$ *and* $n = 1, \ldots, N$

$$\delta_k z^n_{i,j,l} + L_h^{Exp3D} z^n_{i,j,l} = g^n_{i,j,l} \quad \textit{for interior points} \tag{4.6}$$

with $z^n_{0,j,l} = a_{1,n}$, $z^n_{p,j,l} = a_{2,n}$, $z^n_{i,0,l} = a_{3,n}$, $z^n_{i,M,l} = a_{4,n}$, $z^n_{i,j,0} = a_{5,n}$, *and* $z^n_{i,j,M} = a_{6,n}$. *Then the following estimate must hold at each time level* t_n *with* $1 \leq n \leq N$*:*

$$\begin{aligned} \max_{(i,j,l)\in\Gamma_1} |z^n_{i,j,l}| \quad &\leq \quad \max\{\max_{1\leq m\leq 6} |a_{m,n}|, \max_{(i,j,l)\in\Gamma_1} |z^{n-1}_{i,j,l}|\} \\ &\qquad + k \max_{(i,j,l)\in\Gamma_1} |g^n_{i,j,l}|. \end{aligned} \tag{4.7}$$

Proof. Suppose $(r, s, w) \in \Gamma_1$ is such that

$$|z^n_{r,s,w}| = \max_{(i,j,l)\in\Gamma_1} |z^n_{i,j,l}|. \tag{4.8}$$

If $r = 0$ or p or $s, w = 0$ or M, the result is clear. Therefore, suppose that (r, s, w) is such that (x_r, y_s, z_w) is an interior point in Ω_1 is chosen. Consider

1. Case 1: $z^n_{r,s,w} \geq 0$

   Define $\alpha_h = \frac{\alpha^n_{i,j,l} k}{4h^2}$ and $\beta_h = \frac{\beta^n_{i,j,l} k}{4h}$, then starting with

$$\delta_k z^n_{i,j,l} + L_h^{Exp3D} z^n_{i,j,l} = g^{n-1}_{i,j,l}, \tag{4.9}$$

$$\begin{aligned} z^n_{r,s,w} &= z^{n-1}_{r,s,w} \\ &+(\alpha_h - \beta_h)\left[z^n_{r+1,s+1,w+1} + z^n_{r+1,s+1,w-1} + z^n_{r+1,s-1,w+1} + z^n_{r-1,s+1,w+1}\right] \\ &+(\alpha_h + \beta_h)\left[z^n_{r+1,s-1,w-1} + z^n_{r-1,s+1,w-1} + z^n_{r-1,s-1,w+1} + z^n_{r-1,s-1,w-1}\right] \\ &+(-8\alpha_h - \gamma^n_{i,j,l})z^n_{r,s,w} + kg^n_{r,s,w} \end{aligned}$$

Using this result on the theorem,

$$\begin{aligned} |z^n_{r,s,w}| &\leq |z^{n-1}_{r,s,w}| + (\alpha_h - \beta_h) \\ &\quad \left[|z^n_{r+1,s+1,w+1}| + |z^n_{r+1,s+1,w-1}| + |z^n_{r+1,s-1,w+1}| + |z^n_{r-1,s+1,w+1}|\right] \\ &\quad +(\alpha_h + \beta_h) \qquad (4.10) \\ &\quad \left[|z^n_{r+1,s-1,w-1}| + |z^n_{r-1,s+1,w-1}| + |z^n_{r-1,s-1,w+1} + |z^n_{r-1,s-1,w-1}|\right] \\ &\quad +(-8\alpha_h - \gamma^n_{i,j,l})|z^n_{r,s,w}| + k|g^n_{r,s,w}| \\ &\leq |z^{n-1}_{r,s,w}| + k|g^n_{r,s,w}| \qquad (4.11) \end{aligned}$$

2. Case 2: $z^n_{r,s,w} < 0$

   Again using the first case, along with the assumption that

$$|z^n_{r,s,w}| = -z^n_{r,s,t}, \tag{4.12}$$

   we find

$$|z^n_{r,s,w}| \leq |z^{n-1}_{r,s,w}| + k|g^n_{r,s,w}|. \tag{4.13}$$

Once again, with the trivial cases, combining (4.11) and (4.13), we obtain the result (4.7). □

Theorem 4.1. The error equation for the new interface value can be shown to be

$$e_*^n = e_{p,j,l}^{n-1} - kL_H^{Exp3D} e_{p,j,k}^{n-1} + kK_{*,j,l}^n (k + H^2) \tag{4.14}$$

with $e_{0,j,l}^n = e_{M,j,l}^n = e_{i,0,l}^n = e_{i,M,l}^n = e_{i,j,0}^n = e_{i,j,M}^n = 0$. $K_{i,j,l}^n$ and $K_{*,j,l}^n$ represent real numbers depending on u, h, H, and k. Let

$$\begin{aligned} C_1 &= \max_{(i,j,l)\in\Gamma_1} \{|K_{i,j,l}^n| : 1 \le n \le N\} \\ C_* &= \max_{(*,j,l)\in\Gamma_1} \{|K_{*,j,l}^n| : 1 \le n \le N\} \end{aligned}$$

where the constants are independent of the grid.

Using similar steps from the previous schemes, a bound on e_*^n again remains.

To obtain our bound on the interface error,

$$\begin{aligned} |z_*^n| \le\ & |z_{p,s,w}^{n-1}| + (\alpha_H - \beta_H) \\ & \left[|z_{p+q,s+q,w+q}^{n-1}| + |z_{p+q,s+q,w-q}^{n-1}| + |z_{p+q,s-q,t+q}^{w-1}| + |z_{p-q,s+q,w+q}^{n-1}|\right] \\ & +(\alpha_H + \beta_H)\left[|z_{p+q,s-q,w-q}^{n-1}| + |z_{p-1,s+q,w-q}^{n-1}| + |z_{p-q,s-q,w+q}^{n-1} + |z_{p-q,s-q,w-q}^{n-1}|\right] \\ & +(-8\alpha_H - \gamma_{p,s,w}^{n-1})|z_{p,s,w}^{n-1}| + k|g_{p,s,w}^{n-1}| \\ \le\ & (\alpha_H - \beta_H)\left[|z_{p+q,s+q,w+q}^{n-1}| + |z_{p+q,s+q,w-q}^{n-1}| + |z_{p+q,s-q,w+q}^{n-1}| + |z_{p-q,s+q,w+q}^{n-1}|\right] \\ & +(\alpha_H + \beta_H)\left[|z_{p+q,s-q,w-q}^{n-1}| + |z_{p-1,s+q,w-q}^{n-1}| + |z_{p-q,s-q,w+q}^{n-1} + |z_{p-q,s-q,w-q}^{n-1}|\right] \\ & +(1 - 8\alpha_H - \gamma_{p,s,w}^{n-1})|z_{p,s,w}^{n-1}| + k|g_{p,s,w}^{n-1}| \end{aligned}$$

The assumptions given in (4.4) give

$$|z_*^n| \le \max_{(i,j,l)\in\Gamma_1} |z_{i,j,l}^{n-1}| + k|g_{p,j,l}^{n-1}|. \tag{4.15}$$

Applying the result to e_*^n leaves

$$|e_*^n| \leq \max e_{p,j,l}^{n-1} + kC_*(k + h^2 + H^2) \tag{4.16}$$

and hence

$$\begin{aligned} \max_{(i,j,l)\in\Gamma} |e_{i,j,k}^n| &\leq \max_{(i,j,l)\in\Gamma} |e_{i,j,k}^0| + \sum_{w=1}^{n} C'k(k + h^2 + H^2) \\ &\leq \max_{(i,j,l)\in\Gamma} |e_{i,j,k}^0| + C'nk(k + h^2 + H^2) \end{aligned}$$

Assuming no error with the initial time and $nk \leq T$, we have shown

$$||e_{\cdot,\cdot}^n||_\infty \leq C(k + h^2 + H^2) \tag{4.17}$$

uniformly for $1 \leq n \leq N$. $\square$

Bibliography

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] D. Amitai, A. Averbuch, M. Israeli, and S. Itzikowitz. Implicit-explicit parallel asynchronous solver of parabolic pdes. *SIAM Journal on Scientific Computing*, 19(4):1366–1404, 2006.

[3] K.E. Atkinson. *An Introduction to Numerical Analysis.* Wiley and Sons, 1989.

[4] H. Blum, S. Lisky, and R. Rannacher. A domain splitting algorithm for parabolic problems. *Computing*, 49:11–49, 1992.

[5] J.H. Bramble, J.E. Pasciak, and A.H. Schatz. The construction of preconditioners for elliptic problems by substructuring i. *Mathematics of Computation*, 47:103–134, 1986.

[6] J.H. Bramble, J.E. Pasciak, and A.T. Vassilev. Analysis of non-overlapping domain decomposition algorithms with inexact solves. *Mathematics of Computation*, 67(221):1–19, 1998.

[7] R.L. Burden and J.D. Faires. *Numerical Analysis.* Thomson/Brooks/Cole, 2005.

[8] X. Cai. Additive schwarz algorithms for parabolic convection-diffusion equations. *Numerical Mathematics*, 60:41–62, 1990.

[9] X.C. Cai, T.P. Mathew, and M.V. Sarkis. Maximum nom analysis of overlapping nonmatching grid discretizations of elliptic equations. *SIAM Journal on Numerical Analysis*, 37(5):1709–1728, 2000.

[10] S.C. Chapra and R.P. Canale. *Numerical Methods for Engineers.* McGraw-Hill, 2006.

[11] D.S. Daoud, A.Q.M. Khaliq, and B.A. Wade. A non-overlapping implicit predictor-corrector sheme for parabolic equations. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 15–19. CSREA Press, 2000.

[12] C.N. Dawson and Q. Du. A finite element domain decomposition method for parabolic equation. Technical Report TR90-25, Rice University.

[13] C.N. Dawson, Q.Du, and T.F. Dupont. A finite difference domain decomposition algorithm for numerical solution of the heat equation. *Mathematics of Computation*, pages 63–71, 1991.

[14] C.R. Deeter and G.J. Gray. The discrete green's function and the discrete kernel function. *Discrete Mathematics*, 10:29–42, 1974.

[15] Q. Du, M. Mu, and Z.N. Wu. Efficient parallel algorithms for parabolic problems. *SIAM Journal of Numerical Analysis*, pages 1469–1487, 2001.

[16] G.F.D. Duff and D. Naylor. *Differential Equations of Applied Mathematics.* Wiley and Sons, 1966.

[17] L. Evans. *Partial Differential Equations*, volume 19 of *Graduate Studies in Mathematics.* American Mathematical Society, 1998.

[18] X. Feng. Analysis of finite element methods and domain decomposition algorithms for a fluid-solid interaction problem. *SIAM Journal on Numerical Analysis*, 38(4):1312–1336, 2001.

[19] G.C. Fox, R.D. Williams, and P.C. Messina. *Parallel Computing Works.* Morgan Kaufmann Publishers, 1994.

[20] A. Friedman. *Partial Differential Equations of Parabolic Type.* Prentice-Hall, 1964.

[21] S. Gill. Parallel programming. *The Computer Journal*, 1(1):2–10, 1958.

[22] R. Glowinski and M.F. Wheeler. Domain decomposition and mixed finite element methods for elliptic problems. In *Proceedings for the First International Symposium on Domain Decomposition Methods for Partial Differential Equations.* SIAM, 1988. Philadelphia, PA.

[23] R.D. Grigorieff. Convergence of discrete green's functions for finite difference schemes. *Applicable Analysis*, 19(4):233–250, 1985.

[24] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message-passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[25] Y. Huang and J. Xu. A conforming finite element method for overlapping and nonmatching grids. *Mathematics of Computation*, 72(243):1057–1066, 2003.

[26] F. John. *Partial Differential Equations*, volume 1 of *Applied Mathematical Sciences*. Springer-Verlag, 1971.

[27] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Cambridge University Press, 1987.

[28] T.G. Lewis and H. El-Rewini. *Introductionto Parallel Computing*. Prentice-Hall, 1992.

[29] G. Liang and J.H. He. The non-conforming domain decomposition method for elliptic problems with lagrangian muliplier. *Chinese Journal of Numerical Mathematics and Applications*, 15(1):8–19, 1993.

[30] L. Marcinkowski. Domain decomposition methods for mortar finite element discretizations of plate problems. *SIAM Journal on Numerical Analysis*, 39(4):1097–1114, 2002.

[31] G.T. McAllister and E.F. Sabotka. Discrete green's function. *Mathematics of Computation*, 27:59–80, 1973.

[32] D.A. Patterson and J.L. Hennessy. *Computer Organization and Desing*. Morgan Kaufmann Publishers, 1998.

[33] G.F. Pfister. *In Search of Clusters*. Prentice-Hall, 2nd edition, 1998.

[34] M.H. Protter and H.F. Weinberger. *Maximum Principles in Difference Equations*. Prentice-Hall, 1967.

[35] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Numerical Mathematics and Scientific Computation. Oxford Science Publications, 1999.

[36] W. Rivera, J. Zhu, and D. Huddleston. An efficient parallel algorithm with application to computational fluid dynamics. *Journal of Computers and Mathematics with Applications*, 45:165–188, 2003.

[37] P.E. Bjørstad, M.S. Espedal, and D.E. Keyes, editors. *Overlapping Schwarz for Parabolic Problems*. Ninth International Conference on Domain Decomposition Methods, 1998.

[38] H.-S. Shi and H.-L. Liao. Unconditional stability of corrected explicit-implicit domain decomposition algorithms for parallel approximation of heat equations. *SIAM Journal on Numerical Analysis*, 44(4):1584–1611, 2006.

[39] B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, 1996.

[40] G.D. Smith. *Numerical Solution of Partial Differential Equations.* Oxford Mathematical Handbooks. Oxford University Press, 1965.

[41] R. Sperb. *Maximum Principles and Their Applications*, volume 157 of *Mathematics in Science and Engineering.* Academic Press, 1981.

[42] J.C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations.* Brooks/Cole, 1989.

[43] P. Le Tallec and T. Sassi. Domain decomposition with nonmatching grids: Augmented lagrangian approach. *Mathematics of Computation*, 64(15):1367–1396, 1995.

[44] P. Le Tallec and M.D. Tidriri. Convergence analysis of domain decomposition algorithms with full overlapping for advection-diffusion problems. *Mathematics of Computation*, 68(226):585–606, 1999.

[45] V. Thomée. *Galerkin Finite Element Methods for Parabolic Problems*, volume 25 of *Springer Series in Computational Mathematics.* Springer, 1997.

[46] D. U. von Rosenberg. *Methods for the Numerical Solution of Partial Differential Equations.* American Elsevier Publishing Company, 1969.

[47] B. Zhang and W. Shen. Notes on finite differene domain decomposition algorithm for the solution of heat equation. *Journal of Numerical Method and Computation Applications*, 23:2:81–90, 2002.

[48] Y. Zhuang and X.H. Sun. Stabilized explicit-implicit domain decomposition methods for the numerical solution of parabolic equations. *SIAM Journal of Scientific Computing*, 24(1):335–358, 2002.

Appendix A

MPICH

A.1 MPICH

This section is meant to give the reader some understanding of how programs are written and implemented using the MPICH v1.2.7p1 standard [24]. While not a exhaustive explanation of the MPICH standard, it is hoped that the reader will see the underlying strategy and structure used to program in parallel in the context of solving the non-overlapping domain decomposition of partial differential equations numerically. The programs that are in the appendix use these strategies and structures.

The programs are written in standard C code, but the standard can work in a variety languages (with appropriate changes in syntax). There are a variety of sources that state how to install the libraries based on the operating system used, in particular, the Argonne National Laboratory website (`http://www-unix.mcs.anl.gov/mpi/`) houses the official installation files and a lot of documentation. In regard to the actual programming, the program itself will look very similar to one that was written in serial with a few extra blocks of code discussed below. This makes sense because the program will be working on its own portion of the domain independently of the other subdomains.

To write parallel code using MPICH, there are only a few pieces that change the structure of a serial program. Those changes can be broken down into the following categories: header files, starting up the MPI environment, communication commands and closing down the MPI environment. One more header must be inserted into the code to use the MPI libraries which allow for usage of the MPI environment. The header to be added is the `mpi.h`.

Starting up the MPI environment is, at minimum, a set of four commands that opens the MPI environment and gathers information that is typically for addressing the other processors. A typical block of MPI start-up code would look like

```
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

`MPI_Status` is a variable used to make sure that communication between the processors are working as desired during the program run. The second line, `MPI_Init`, initializes the MPI execution environment. While the last two lines of the block of code find out how many processors there are available to work with (`np`) and assign a unique ID number to that particular processor (`id`). This block of code must be in every parallel program using MPICH and must be placed prior to any MPI commands.

Once the environment is open and working properly the program looks identical to a serial program. It is only until the user needs to get information from another processor, like information from one subdomain to compute the predictor or corrector, that we have to add another block of code. The code has a ‘send’ piece for the processor sending information and a ‘receive’ piece for the processor waiting to obtain the sent information. In context of the problem of the paper, a subdomain would send their data to an adjacent subdomain so the processor associated with the subdomain could compute the interface value. Once the interface was computed, the processor would then send back the interface to the original sender and replace their interface with the new one. This means we have a send/receive pair every time the interface needed to be computed. The two commands are:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
```

and

```
int MPI_Recv(void  *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
```

where

- `buf` is the initial address of send buffer (choice)
- `count` is a number of elements in send/receive buffer (nonnegative integer)
- `datatype` is a datatype of each send/receive buffer element (handle)
- `dest` or `source` is the rank of destination/source (integer)
- `tag` is a message tag (integer)
- `comm` is a communicator (handle)
- `MPI_Status` checks status of MPI environment.

Truly, one of the only warnings to be given, is to make sure that the send/receive pairs are constructed so that information is sent to the correct processor and that processor receives it. Without that perpetual problem, there are very few other parallel programming difficulties.

Once the program has finished and is ready to end, one more command must be given to successfully exit the MPI environment and close down the connections to the other processors. That command is

```
int MPI_Finalize();
```

which can be placed immediately preceding the end of `main`.

In the algorithms shown in the dissertation, we used rectangular domains and decomposed them into strip subdomains in a way so the maximum number of interfaces on each subdomain was two. This allowed for easier logistics in programming the necessary communications because each send/receive pair could be done simultaneously for every pair of adjacent subdomains. As an example, we would instruct the even processors to send their necessary values used in the interface computation to the odd subdomain that is adjacent and to the right of the even one. Simultaneously, we would instruct the odd processors to wait for those values from the even subdomain adjacent and to the left. One benefit of constructing the subdomains this way is the timing, in essence, only involves one send/receive command as all the processor pairs are accomplishing their task in concert. After the information is sent, the odd processors make the computation and the process is repeated in reverse to share the results of the computation back to the even processors.

In terms of other issues that arise when programming in parallel, there are actually very few that would not surface in the serial versions of the programs. Of the ones that might come about and is of merit is the issue of trying to make the code scalable. A couple issues that must be worked out for programs if the number of subdomains is not fixed can be stated as (1) setting up the processors/subdomains appropriately and

(2) obtaining the correct information from the correct subdomain for the interface computations. In the case of setting up the processors/subdomains, the difficulty arises from making sure that each subdomain only has their respective portion of the overall domain. This requires some care in the indices of arrays and counters used to fill those arrays. Luckily, once this is accomplished for one program it can be reused for similar shaped subdomains (rectangles in the examples used in the dissertation).

A similar, but not equivalent problem occurs when attempting to obtain the correct information from adjacent subdomains. The code, to be scalable, had to be written in such a way that it accounted for either the scenario that it was on a receiving processor (one that computed the interface) or it was the sending processor. Once a processor started it had to have the instructions for the adjacent processor and had to be able to state which values it needed for the computation. The similarity of this issue with the previous deals with the necessity to reliably know the adjacent subdomain's structure. Again, once it was worked out once the code was able to be reused.

Appendix B

Source Code

B.1 be2d.c

```
/*  This program will solve the 2-d convection-diffusion equation
    using Backward Euler to solve and Gauss-Seidel as the interative
    matrix solver
*/

#include "stdio.h"
#include "math.h"
#include "time.h"

double uxyt(); double alpha(); double beta();
double gam(); double fct();

/* begin main */
main()
{
    /* variables */
        double a[6][85200], u[405][405];
        double x[405], y[405];
        double v[85200], v_new[85200];
        double h[7]={0.1,0.05,0.025,0.01,0.005,0.0025,0.001};
        int i, j, n, mx, my, mt, p;
        int dum, ii, loop;
        int iter, sent, imax=20000;

        double dx, dt, dy, lenx, leny, lent, H;
        double alpha_coef, beta_coef;
        double diff, maxerr, time1;
        double dummy, old, sum, ea, es = 0.0000000000000001;

        double x0, xf, y0, yf, t0, tf;

        time_t start, end;

        int ex = 3;
        /* domain declarations */
        switch(ex)
            {
            case 1: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 1 */
                break;
            case 2: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 2 */
                break;
            case 3: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 3 */
                break;
            case 4: x0 = 0.0; xf = 2.0*M_PI;
                y0 = 0.0; yf = 1.0*M_PI;
                t0 = 0.0; tf = 1.0;     /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
        /* end domain declarations */
    /* end variables */

    lenx = xf - x0; leny = yf - y0; lent = tf - t0;
```

```
    for(loop=0;loop<=4;loop++)
    {
        /* grid assignment */
        dx = /*M_PI**/h[loop]; dy = dx;
        mx = ceil(lenx/dx); my = ceil(leny/dy);
        n = (mx-1)*(my-1);
        dt = 0.01/*pow(h[loop],2.0)*/; mt = ceil(lent/dt);
        p = ceil(pow(2.0*dt,0.5)/dx); H=p*dx;
        /* end grid assignment */

        /* coefficients */
        alpha_coef = dt/pow(dx,2);
        beta_coef = dt/(2.0*dx);
        /* end coefficients */

        /* grid points */
        for(i=0;i<=mx;i++) x[i] = x0 + i*dx;
        for(i=0;i<=my;i++) y[i] = y0 + i*dy;
        /* end grid points */

printf("x[0]=%f x[mx]=%f\ny[0]=%f y[my]=%f\n\n",x[0],x[mx],y[0],y[my]);

        /* initial conditions */
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                u[i][j] = uxyt(x[i],y[j],t0);
            }
        }
        /* end initial conditions */

        /* boundary initialization */
        /* lower and upper boundary */
            for(i=0;i<=mx;i++){
                u[i][0] = uxyt(x[i],y[0],t0);
                u[i][my] = uxyt(x[i],y[my],t0);
            }
        /* end lower and upper boundary */
        /* left and right boundary */
            for(i=0;i<=my;i++){
                u[0][i] = uxyt(x[0],y[i],t0);
                u[mx][i] = uxyt(x[mx],y[i],t0);
    }
/* end left and right boundary */
/* end boundary initialization */

/* initializing coefficient matrix a */
for(i=1;i<=n;i++){
    a[3][i] = 1.0 + 4.0*alpha_coef*
        alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1) + 1]) -
        dt*gam(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1) + 1]);
}
for(i=1;i<=n;i++){
    dum=i%(mx-1);
    if(dum!=1){
        a[2][i] = beta_coef*
            beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
            + 1])-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
    } else a[2][i] = 0.0;
    if(dum!=0){
        a[4][i] = -beta_coef*
            beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
            + 1])-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
    } else a[4][i] = 0.0;
}
for(i=1;i<=n;i++){
        a[1][i] = /*beta_coef*
            beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
            + 1])*/-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1]
            ,y[(i-1)/(mx-1)
             + 1]);
        a[5][i] = /*-beta_coef*
            beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
            + 1])*/-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
}
for(i=0;i<=mx-2;i++){
        a[1][i+1] = 0.0;
        a[5][n-i] = 0.0;
}
/* end intializing coefficient matrix a */


start = time(NULL);

for(ii=1;ii<=mt;ii++){
    dum = 1;

    for(j=1;j<=my-1;j++){
        for(i=1;i<=mx-1;i++){
            v[dum] = u[i][j] +
             dt*fct(x[i],y[j],t0+ii*dt);
            dum++;
        }
    }
/* boundary values */
        for(i=1;i<=mx-1;i++){
        /* horizontal - bottom */
            v[i] = v[i] + (alpha_coef*
             alpha(x[i],y[1],t0+ii*dt) )*
                uxyt(x[i],y0,t0+ii*dt);
        /* horizontal - top */
            v[(n-mx+1+i)] = v[(n-mx+1+i)] +
             (alpha_coef*
                alpha(x[i],y[my-1],t0+ii*dt)/*+
                beta_coef*
                beta(x[i],y[my-1],t0+ii*dt)*/)*
                uxyt(x[i],yf,t0+ii*dt);
        }

        for(i=1;i<=my-1;i++){
        /* vertical - left */
            v[1+((i-1)*(mx-1))] =
             v[1+((i-1)*(mx-1))] +
             (alpha_coef*
                alpha(x[1],y[i],t0+ii*dt) -
                beta_coef*beta(x[1],
                y[i],t0+ii*dt))*
                uxyt(x0,y[i],t0+ii*dt);
        /* vertical - right */
            v[(i*(mx-1))] = v[(i*(mx-1))] +
             (alpha_coef*
                alpha(x[mx-1],y[i],t0+ii*dt) +
                beta_coef*beta(x[mx-1],
                y[i],t0+ii*dt))*
                uxyt(xf,y[i],t0+ii*dt);
        }
/* end boundary values */
/* gauss-seidel solver */
    for(i=1;i<=n;i++) v_new[i]=v[i];

    iter = 1;
    sent = 0;
    while((sent != 1) && (iter <= imax)){
        sent = 1;
        maxerr = 0.0;
        for(i=1;i<=n;i++){
            old=v_new[i];
            sum = 0.0;
        sum = a[1][i]*v_new[i-(mx-1)] +
         a[2][i]*v_new[i-1] +
            a[4][i]*v_new[i+1] +
            a[5][i]*v_new[i+(mx-1)];
            v_new[i] = (v[i]-sum)/a[3][i];
            if((sent = 1) && (v_new[i] != 0.0)){
                ea = fabs((v_new[i]-old)/v_new[i]);
                if(ea > maxerr) maxerr = ea;
            }
        }
        iter = iter + 1;
        if(maxerr > es) sent = 0;
    }
/* end gauss-seidel solver */
/* putting v back into u */
```

```
            dum = 1;
            for(j=1;j<=my-1;j++){
                for(i=1;i<=mx-1;i++){
                    u[i][j] = v_new[dum];
                    dum++;
                }
            }
        /* end putting v back into u */
        }                       /* end solver */
        end = time(NULL);
        time1 = end - start;
        /* update boundaries */
            /* lower and upper boundary */
                for(i=0;i<=mx;i++){
                    u[i][0] = uxyt(x[i],y0,t0+(ii-1)*dt);
                    u[i][my] = uxyt(x[i],yf,t0+(ii-1)*dt);
                }
            /* end lower and upper boundary */
            /* left and right boundary */
                for(i=0;i<=my;i++){
                    u[0][i] = uxyt(x0,y[i],t0+(ii-1)*dt);
                    u[my][i] = uxyt(xf,y[i],t0+(ii-1)*dt);
                }
            /* end left and right boundary */
        /* ending update boundaries */

        /* error computation */
        maxerr = 0.0;
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                diff = fabs(uxyt(x[i],y[j],t0+(ii-1)*dt)
                -u[i][j]);
                if(diff > maxerr) maxerr = diff;
            }
        }
        printf("time=%.1f maxerr= %.16e\n",time1,maxerr);
        /* end error computation */

        /* output loop */
/*      for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                printf("%.5f\t%.5f\t%.10f\t%.10f\t%.16e\n"
                ,x[i],y[j],
                uxyt(x[i],y[j],(ii-1)*dt),u[i][j],
                (uxyt(x[i],y[j],(ii-1)*dt)-u[i][j]));
            }
        }
        /* end output loop */
        printf("\n\n");
    }                           /* end increment loop */

}
/* end main */

/* function definitions */
double uxyt(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return
             exp(-2.0*t)*sin(M_PI*x)*sin(M_PI*y);
                break;
            case 2: return exp(-2.0*t)*cos(3.0*x+y);
                break;
            case 3: return
             exp(t)*sin(2.0*x*M_PI)*sin(y*M_PI);
                break;
            case 4: return exp(-2.0*t)*sin(x)*sin(y);
                break;
            default:  printf("fix the example number\n");
            }
    }

double alpha(double x, double y)
    {

        int ex = 3;

        switch(ex)
```

```
            {
            case 1: return 1.0/(M_PI * M_PI);
                    /* example 1 */
                break;
            case 2: return 1.0;             /* example 2 */
                break;
            case 3: return 1.0;             /* example 3 */
                break;
            case 4: return 1.0;             /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double beta(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 9.9*sin(x);      /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double gam(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
```

```
                break;
            case 2: return 8.0;             /* example 2 */
                break;
            case 3: return (1.0+5.0*M_PI*M_PI);
            /* example 3 */
                break;
            case 4: return -9.9*cos(x);
            /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double fct(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 0.0;             /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }
```

B.2 ceidd_hyb.c

```
/*  This program will solve the 2-d convection-diffusion equation
    using Backward Euler to solve and Gauss-Seidel as the interative
    matrix solver and hyb on the interface
*/

#include "mpi.h"
```

```
#include "stdio.h"
#include "math.h"
#include "time.h"

double uxyt(); double alpha(); double beta();
double gam(); double fct();

/* begin main */
main(int argc, char *argv[])
{
    /* variables */
        double a[6][72200], u[405][405];
        double x[405], y[405];
        double v[72200], v_new[72200];
        double
         h[7]={0.1,0.05,0.025,0.01,0.005,0.0025,0.001};

        /* parallel variables */
        double b[4][405], left[405], mid[405], right[405];
        double send[405], recv[405], v1[405], v1_new[405];
        double predict[405], temp[405];
        double alphaH, betaH;
        int np, id;
        /* end parallel variables */

        int i, j, n, mx, my, mt, p;
        int dum, ii, loop;
        int iter, sent, imax=20000;

        double dx, dt, dy, lenx, leny, lent, H;
        double alpha_coef, beta_coef;
        double diff, maxerr, time1;
        double dummy, old, sum, ea, es =
         0.0000000000000001;

        double x0, xf, y0, yf, t0, tf;

        time_t start, end;

        int ex = 3;
        /* domain declarations */
        switch(ex)
            {
            case 1: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 1 */
                break;
            case 2: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 2 */
                break;
            case 3: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 3 */
                break;
            case 4: x0 = 0.0; xf = 2.0*M_PI;
                y0 = 0.0; yf = 1.0*M_PI;
                t0 = 0.0; tf = 1.0;     /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
        /* end domain declarations */
    /* end variables */

    lenx = xf - x0; leny = yf - y0; lent = tf - t0;

    /* MPI startup */
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    /* end MPI startup */

    for(loop=0;loop<=5;loop++)
    {
        /* grid assignment */
        dx = /*M_PI**/h[loop]; dy = dx;
        mx = ceil(lenx/(np*dx)); my = ceil(leny/dy);
        n = (mx-1)*(my-1);
        dt = 0.01/*pow(h[loop],2.0)*/; mt = ceil(lent/dt);
        p = ceil(pow(2.0*dt,0.5)/dx); H=p*dx;
        /* end grid assignment */
```

```
        /* coefficients */
        alpha_coef = dt/pow(dx,2);
        beta_coef = dt/(2.0*dx);
        alphaH = dt/pow(H,2);
        betaH = dt/(2.0*H);
        /* end coefficients */

        /* grid points */
        for(i=0;i<=mx;i++) x[i] = x0 + (id*lenx)/np + i*dx;
        for(i=0;i<=my;i++) y[i] = y0 + i*dy;
        /* end grid points */

printf("x[0]=%f x[mx]=%f\ny[0]=%f
 y[my]=%f\n\n",x[0],x[mx],y[0],y[my]);

        /* initial conditions */
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                u[i][j] = uxyt(x[i],y[j],t0);
            }
        }
        /* end initial conditions */

        /* boundary initialization */
        /* lower and upper boundary */
            for(i=0;i<=mx;i++){
                u[i][0] = uxyt(x[i],y[0],t0);
                u[i][my] = uxyt(x[i],y[my],t0);
            }
        /* end lower and upper boundary */
        /* left and right boundary */
            for(i=0;i<=my;i++){
                if(id==0) u[0][i] = uxyt(x[0],y[i],t0);
                if(id==(np-1)) u[mx][i] =
                 uxyt(x[mx],y[i],t0);
            }
        /* end left and right boundary */
        /* end boundary initialization */

        /* initializing coefficient matrix a */
```

```
for(i=1;i<=n;i++){
    a[3][i] = 1.0 + 4.0*alpha_coef*
        alpha(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
        + 1]) -
        dt*gam(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
        + 1]);
}
for(i=1;i<=n;i++){
    dum=i%(mx-1);
    if(dum!=1){
        a[2][i] =
         beta_coef*beta(x[(i-1)%(mx-1)+1],
         y[(i-1)/(mx-1)
         + 1])-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
    } else a[2][i] = 0.0;
    if(dum!=0){
        a[4][i] =
        -beta_coef*beta(x[(i-1)%(mx-1)+1],
        y[(i-1)/(mx-1)
        + 1])-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
    } else a[4][i] = 0.0;
}
for(i=1;i<=n;i++){
        a[1][i] = -alpha_coef*
                    alpha(x[(i-1)%(mx-1)+1],
                    y[(i-1)/(mx-1)
                    + 1]);
        a[5][i] = -alpha_coef*
                    alpha(x[(i-1)%(mx-1)+1],
                    y[(i-1)/(mx-1)
                     + 1]);
}
for(i=0;i<=mx-2;i++){
        a[1][i+1] = 0.0;
        a[5][n-i] = 0.0;
```

```
}
/* end intializing coefficient matrix a */


start = time(NULL);

for(ii=1;ii<=mt;ii++){
    dum = 1;

    for(j=1;j<=my-1;j++){
        for(i=1;i<=mx-1;i++){
            v[dum] = u[i][j] +
             dt*fct(x[i],y[j],t0+ii*dt);
            dum++;
        }
    }
/* begin predictor */
    /* predictor send/recv 1 */
    if(id!=(np-1)){
        for(i=0;i<=my;i++) send[i] = u[mx-p][i];
    }
    if(id!=(np-1))
     MPI_Send(&send,my+1,MPI_DOUBLE,
     id+1,9,MPI_COMM_WORLD);
    if(id!=0)
    MPI_Recv(&left,my+1,MPI_DOUBLE,
    id-1,9,MPI_COMM_WORLD,&status);
    /* end predictor send/recv 1 */
        /* putting values in vectors */
        for(i=0;i<=my;i++){
            mid[i] = u[0][i];
            right[i] = u[p][i];
        }
        /* end putting values in vectors */
        /* explicit portion of predictor */
        if(p!=1){
            for(i=1;i<=my-1;i++){
                predict[i] =
                 (alpha(x[0],y[i])*alphaH -
                    beta(x[0],y[i])*betaH)*left[i]
                     +(1.0 - 2.0*alpha(x[0],y[i])*
                     alphaH)*mid[i] +
                    (alpha(x[0],y[i])*alphaH +
                    beta(x[0],y[i])*betaH)*right[i]
                    +dt*fct(x[0],y[i],t0+ii*dt);
            }
        }
        /* end explicit portion of predictor */
        /* setting up b matrix */
        for(i=1;i<=my-1;i++){
            b[1][i] =
            (-alpha(x[0],y[i])*alpha_coef);
            b[2][i] =
            1.0 + 2.0*alpha(x[0],y[i])*alpha_coef-
                dt*gam(x[0],y[i]);
            b[3][i] =
            (-alpha(x[0],y[i])*alpha_coef);
        }
        b[1][1] = 0.0;
        b[3][my-1] = 0.0;
        /* end setting up b matrix */
        /* setting up v1 vector (ax = v1) */
        for(i=1;i<=my-1;i++) v1[i] = predict[i];
        v1[1] = v1[1] +
         (alpha_coef*alpha(x[0],y[1]))
         *uxyt(x[0],y[0],t0+ii*dt);
        v1[my-1] = v1[my-1] +
             (alpha_coef*alpha(x[0],y[my-1]))
             *uxyt(x[0],y[my],t0+ii*dt);
        /* end setting up v1 vector (ax = v1) */
        /* gauss-seidel for v1_new */
            for(i=1;i<=my-1;i++) v1_new[i]=v1[i];

            iter=1;
            sent=0;
            while((sent!=1) && (iter<=imax)){
                sent=1;
                maxerr=0.0;
                for(i=1;i<=my-1;i++){
                    old=v1_new[i];
                    sum=0.0;
                    sum = b[1][i]*v1_new[i-1]
```

```
                    +b[3][i]*v1_new[i+1];
                    v1_new[i]=(v1[i]-sum)/b[2][i];
                    if(sent=1 && v1_new[i]!=0.0){
                        ea = fabs((v1_new[i]
                        -old)/v1_new[i]);
                        if(ea > maxerr) maxerr=ea;
                    }
                }
                iter=iter+1;
                if(maxerr > es) sent = 0;
            }
        /* end gauss-seidel for v1_new */
    /* replacing variables for next time */
    if(id!=0){
        for(i=1;i<=my-1;i++){
            predict[i] = v1_new[i];
            temp[i] = u[0][i];
            u[0][i] = predict[i];
        }
    }
    /* end replacing variables for next time */

    /* predictor send/recv 2 */
    if(id!=0)
     MPI_Send(&predict,my+1,MPI_DOUBLE,id-1,
    9,MPI_COMM_WORLD);
    if(id!=(np-1)){
MPI_Recv(&recv,my+1,MPI_DOUBLE,id+1,
9,MPI_COMM_WORLD,&status);
        for(i=1;i<=my-1;i++){
            u[mx][i] = recv[i];
        }
    }
    /* end predictor send/recv 2 */
/* end predictor */
/* boundary values */
        for(i=1;i<=mx-1;i++){
        /* horizontal - bottom */
            v[i] = v[i] + alpha_coef
            *alpha(x[i],y[1],t0+ii*dt)*
                uxyt(x[i],y[0],t0+ii*dt);
```

```
/* horizontal - top */
    v[(n-mx+1+i)] = v[(n-mx+1+i)] +
     (alpha_coef*
        alpha(x[i],y[my-1],t0+ii*dt))*
        uxyt(x[i],y[my],t0+ii*dt);
}
/* vertical - left */
for(i=1;i<=my-1;i++){
    if(id!=0){
        v[1+((i-1)*(mx-1))] =
         v[1+((i-1)*(mx-1))] +
            (alpha_coef*alpha(x[1]
            ,y[i],t0+ii*dt) -
            beta_coef*beta(x[1],
            y[i],t0+ii*dt))*
            u[0][i];
    } else {
        v[1+((i-1)*(mx-1))] =
         v[1+((i-1)*(mx-1))] +
            (alpha_coef*alpha(x[1],
            y[i],t0+ii*dt) -
            beta_coef*beta(x[1],
            y[i],t0+ii*dt))*
            uxyt(x[0],y[i],t0+ii*dt);
    }
}
/* vertical - right */
for(i=1;i<=my-1;i++){
    if(id!=(np-1)){
        v[(i*(mx-1))] = v[(i*(mx-1))] +
         (alpha_coef*
            alpha(x[mx-1],y[i],t0+ii*dt) +
            beta_coef*beta(x[mx-1],
            y[i],t0+ii*dt))*
            u[mx][i];
    } else {
        v[i*(mx-1)] = v[i*(mx-1)] +
         (alpha_coef*
            alpha(x[mx-1],y[i],t0+ii*dt) +
            beta_coef*beta(x[mx-1],
            y[i],t0+ii*dt))*
```

```
                    uxyt(x[mx],y[i],t0+ii*dt);
            }
        }
/* end boundary values */
/* gauss-seidel solver */
    for(i=1;i<=n;i++) v_new[i]=v[i];

    iter = 1;
    sent = 0;
    while((sent != 1) && (iter <= imax)){
        sent = 1;
        maxerr = 0.0;
        for(i=1;i<=n;i++){
            old=v_new[i];
            sum = 0.0;
        sum = a[1][i]*v_new[i-(mx-1)] +
        a[2][i]*v_new[i-1] +
              a[4][i]*v_new[i+1] +
               a[5][i]*v_new[i+(mx-1)];
            v_new[i] = (v[i]-sum)/a[3][i];
            if((sent = 1) && (v_new[i] != 0.0)){
                ea = fabs((v_new[i]-old)/v_new[i]);
                if(ea > maxerr) maxerr = ea;
            }
        }
        iter = iter + 1;
        if(maxerr > es) sent = 0;
    }
/* end gauss-seidel solver */
/* begin corrector */
    /* corrector send/recv pair 1 */
    if(id!=(np-1)){
        for(i=0;i<=my;i++) send[i] =
         v_new[i*(mx-1)];
    }
    if(id!=(np-1))
     MPI_Send(&send,my+1,MPI_DOUBLE,id+1,
    9,MPI_COMM_WORLD);
    if(id!=0)
 MPI_Recv(&left,my+1,MPI_DOUBLE,id-1,
 9,MPI_COMM_WORLD,&status);
```

```
/* end corrector send/recv pair 1 */
/* putting values into vectors */
    for(i=0;i<=my;i++){
        mid[i] = temp[i];
        right[i] = v_new[1+(i-1)*(mx-1)];
    }
/* end putting values into vectors */
/* explicit correction */
    if(p!=1){
        for(i=1;i<=my-1;i++){
            predict[i] =
             (alpha(x[0],y[i])*alpha_coef -
                beta(x[0],y[i])*beta_coef)
                *left[i] +
                mid[i] +
                (alpha(x[0],y[i])*alpha_coef +
                beta(x[0],y[i])*beta_coef)
                *right[i]
                +dt*fct(x[0],y[i],t0+ii*dt);
        }
    }
/* end explicit correction */
/* setting up b matrix */
    for(i=1;i<=my-1;i++){
        b[1][i] =
        (-alpha(x[0],y[i])*alpha_coef);
        b[2][i] =
        1.0 + 4.0*alpha(x[0],y[i])*alpha_coef
            - dt*gam(x[0],y[i]);
        b[3][i] =
        (-alpha(x[0],y[i])*alpha_coef);
    }
    b[1][1] = 0.0;
    b[3][my-1] = 0.0;
/* end setting up b matrix */
/* setting up v1 vector (bx=v1) */
    for(i=1;i<=my-1;i++) v1[i] = predict[i];
    v1[1] = v1[1] +
        (alpha_coef*alpha(x[0],y[1]))
        *uxyt(x[0],y[0],t0+ii*dt);
    v1[my-1] = v1[my-1] +
        (alpha_coef*alpha(x[0],y[my-1]))
        *uxyt(x[0],y[my],t0+ii*dt);
/* end setting up v1 vector (bx=v1) */
/* gauss-seidel for corrector */
        for(i=1;i<=my-1;i++) v1_new[i]=v1[i];

        iter=1;
        sent=0;
        while((sent!=1) && (iter<=imax)){
            sent=1;
            maxerr=0.0;
            for(i=1;i<=my-1;i++){
                old=v1_new[i];
                sum=0.0;
                sum = b[1][i]*v1_new[i-1]
                +b[3][i]*v1_new[i+1];
                v1_new[i]=(v1[i]-sum)/b[2][i];
                if(sent=1 && v1_new[i]!=0.0){
                    ea = fabs((v1_new[i]-
                    old)/v1_new[i]);
                    if(ea > maxerr) maxerr=ea;
                }
            }
            iter=iter+1;
            if(maxerr > es) sent = 0;
        }
/* end gauss-seidel for corrector */
/* replacing variables prior to resend */
if(id!=0){
    for(i=1;i<=my-1;i++){
        predict[i] = v1_new[i];
        u[0][i] = predict[i];
    }
}
/* end replacing variables prior to resend */

/* corrector send/recv pair 2 */
if(id!=0)
 MPI_Send(&predict,my+1,MPI_DOUBLE,id-1,
9,MPI_COMM_WORLD);
if(id!=(np-1)){
        MPI_Recv(&recv,my+1,MPI_DOUBLE,id+1,
        9,MPI_COMM_WORLD,&status);
        for(i=1;i<=my-1;i++) u[mx][i] = recv[i];
    }
    /* end corrector send/recv pair 2 */
/* end corrector */
/* putting v back into u */
    dum = 1;
    for(j=1;j<=my-1;j++){
        for(i=1;i<=mx-1;i++){
            u[i][j] = v_new[dum];
            dum++;
        }
    }
/* end putting v back into u */
/* updating boundaries */
    for(i=0;i<=mx;i++){
        u[i][0] = uxyt(x[i],y[0],t0+ii*dt);
        /* bottom */
        u[i][my] = uxyt(x[i],y[my],t0+ii*dt);
        /* top */
    }
    if(id==0){
        for(i=0;i<=my;i++)
        u[0][i] = uxyt(x[0],y[i],t0+ii*dt);
    }
    if(id==(np-1)){
        for(i=0;i<=my;i++)
        u[mx][i] = uxyt(x[mx],y[i],t0+ii*dt);
    }
/* end updating boundaries */
}                       /* end solver */
end = time(NULL);
time1 = end - start;

/* error computation */
maxerr = 0.0;
for(j=0;j<=my;j++){
    for(i=0;i<=mx;i++){
        diff = fabs(uxyt(x[i],y[j],t0+(ii-1)*dt)
        -u[i][j]);
```

```
                if(diff > maxerr) maxerr = diff;
            }
        }
        printf("time=%.1f maxerr= %.3e\n",time1,maxerr);
        /* end error computation */

        /* output loop */
/*      for(j=0;j<=mx;j++){
            for(i=0;i<=my;i++){
                printf("%.5f\t%.5f\t%.10f\t%.10f\t%.16e\n",
                x[j],y[i],
                uxyt(x[j],y[i],(ii-1)*dt),u[j][i],
                (uxyt(x[j],y[i],(ii-1)*dt)-u[j][i]));
            }
        }
        /* end output loop */
        printf("\n\n");
    }                           /* end increment loop */

    /* closing MPI */
    MPI_Finalize();
    return 0;
    /* end closing MPI */

}
/* end main */

/* function definitions */
double uxyt(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return
             exp(-2.0*t)*sin(M_PI*x)*sin(M_PI*y);
                break;
            case 2: return exp(-2.0*t)*cos(3.0*x+y);
                break;
            case 3: return
```

```
              exp(t)*sin(2.0*x*M_PI)*sin(y*M_PI);
                 break;
             case 4: return exp(-2.0*t)*sin(x)*sin(y);
                 break;
             default:  printf("fix the example number\n");
             }
     }

double alpha(double x, double y)
     {

         int ex = 3;

         switch(ex)
             {
             case 1: return 1.0/(M_PI * M_PI);
             /* example 1 */
                 break;
             case 2: return 1.0;             /* example 2 */
                 break;
             case 3: return 1.0;             /* example 3 */
                 break;
             case 4: return 1.0;             /* example 4 */
                 break;
             default:  printf("fix the example number\n");
             }
     }

double beta(double x, double y)
     {

         int ex = 3;

         switch(ex)
             {
             case 1: return 0.0;             /* example 1 */
                 break;
             case 2: return 0.0;             /* example 2 */
                 break;
             case 3: return 0.0;             /* example 3 */
                 break;
```

```
            case 4: return 9.9*sin(x);      /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double gam(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 8.0;             /* example 2 */
                break;
            case 3: return (1.0+5.0*M_PI*M_PI);
                /* example 3 */
                break;
            case 4: return -9.9*cos(x);
            /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double fct(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 0.0;             /* example 4 */
```

```
                break;
            default:  printf("fix the example number\n");
            }
    }
```

B.3 ceidd_exp1.c

```
/*  This program will solve the 2-d convection-diffusion equation
    using Backward Euler to solve and Gauss-Seidel as the interative
    matrix solver and exp1 on the interface
*/

#include "mpi.h"
#include "stdio.h"
#include "math.h"
#include "time.h"

double uxyt(); double alpha(); double beta();
double gam(); double fct();

/* begin main */
main(int argc, char *argv[])
{
    /* variables */
        double a[6][72200], u[405][405];
        double x[405], y[405];
        double v[72200], v_new[72200];
        double
         h[7]={0.1,0.05,0.025,0.01,0.005,0.0025,0.001};

        /* parallel variables */
        double b[4][405], left[405], mid[405], right[405];
        double send[405], recv[405], v1[405], v1_new[405];
        double predict[405], temp[405];
        double alphaH, betaH;
        int np, id;
        /* end parallel variables */

        int i, j, n, mx, my, mt, p;
        int dum, ii, loop;
        int iter, sent, imax=20000;
        double dx, dt, dy, lenx, leny, lent, H;
        double alpha_coef, beta_coef;
        double diff, maxerr, time1;
        double dummy, old, sum, ea, es =
         0.0000000000000001;

        double x0, xf, y0, yf, t0, tf;

        time_t start, end;

        int ex = 3;
        /* domain declarations */
        switch(ex)
            {
            case 1: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 1 */
                break;
            case 2: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 2 */
                break;
            case 3: x0 = 0.0; xf = 1.0;
                y0 = 0.0; yf = 1.0;
                t0 = 0.0; tf = 1.0;     /* example 3 */
                break;
            case 4: x0 = 0.0; xf = 2.0*M_PI;
                y0 = 0.0; yf = 1.0*M_PI;
                t0 = 0.0; tf = 1.0;     /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
        /* end domain declarations */
    /* end variables */

    lenx = xf - x0; leny = yf - y0; lent = tf - t0;

    /* MPI startup */
    MPI_Status status;
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    /* end MPI startup */

    for(loop=0;loop<=5;loop++)
    {
        /* grid assignment */
        dx = /*M_PI**/h[loop]; dy = dx;
        mx = ceil(lenx/(np*dx)); my = ceil(leny/dy);
        n = (mx-1)*(my-1);
        dt = /*0.01*/pow(h[loop],2.0); mt = ceil(lent/dt);
        p = ceil(pow(2.0*dt,0.5)/dx); H=p*dx;
        /* end grid assignment */

        /* coefficients */
        alpha_coef = dt/pow(dx,2);
        beta_coef = dt/(2.0*dx);
        alphaH = dt/pow(H,2);
        betaH = dt/(2.0*H);
        /* end coefficients */

        /* grid points */
        for(i=0;i<=mx;i++) x[i] = x0 + (id*lenx)/np + i*dx;
        for(i=0;i<=my;i++) y[i] = y0 + i*dy;
        /* end grid points */

printf("x[0]=%f x[mx]=%f\ny[0]=%f
 y[my]=%f\n\n",x[0],x[mx],y[0],y[my]);

        /* initial conditions */
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                u[i][j] = uxyt(x[i],y[j],t0);
            }
        }
        /* end initial conditions */

        /* boundary initialization */
        /* lower and upper boundary */
            for(i=0;i<=mx;i++){
                u[i][0] = uxyt(x[i],y[0],t0);
        u[i][my] = uxyt(x[i],y[my],t0);
    }
/* end lower and upper boundary */
/* left and right boundary */
    for(i=0;i<=my;i++){
        if(id==0) u[0][i] = uxyt(x[0],y[i],t0);
        if(id==(np-1)) u[mx][i] =
         uxyt(x[mx],y[i],t0);
    }
/* end left and right boundary */
/* end boundary initialization */

/* initializing coefficient matrix a */
for(i=1;i<=n;i++){
    a[3][i] = 1.0 + 4.0*alpha_coef*
        alpha(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
        + 1]) -
        dt*gam(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
         + 1]);
}
for(i=1;i<=n;i++){
    dum=i%(mx-1);
    if(dum!=1){
        a[2][i] = beta_coef
        *beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1) +
         1])-
            alpha_coef*
            alpha(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
             + 1]);
    } else a[2][i] = 0.0;
    if(dum!=0){
        a[4][i] = -beta_coef
        *beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
        + 1])-
            alpha_coef*
            alpha(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
             + 1]);
    } else a[4][i] = 0.0;
}
for(i=1;i<=n;i++){
        a[1][i] = -alpha_coef*
```

```
                alpha(x[(i-1)%(mx-1)+1]
                ,y[(i-1)/(mx-1) + 1]);
        a[5][i] = -alpha_coef*
                alpha(x[(i-1)%(mx-1)+1]
                ,y[(i-1)/(mx-1) + 1]);
}
for(i=0;i<=mx-2;i++){
        a[1][i+1] = 0.0;
        a[5][n-i] = 0.0;
}
/* end intializing coefficient matrix a */


start = time(NULL);

for(ii=1;ii<=mt;ii++){
    dum = 1;

    for(j=1;j<=my-1;j++){
        for(i=1;i<=mx-1;i++){
            v[dum] = u[i][j]
            + dt*fct(x[i],y[j],t0+ii*dt);
            dum++;
        }
    }
/* begin predictor */
    /* predictor send/recv 1 */
    if(id!=(np-1)){
        for(i=0;i<=my;i++) send[i] = u[mx-p][i];
    }
    if(id!=(np-1))
     MPI_Send(&send,my+1,MPI_DOUBLE,id+1,9
    ,MPI_COMM_WORLD);
    if(id!=0)
         MPI_Recv(&left,my+1,MPI_DOUBLE,id-1,
         9,MPI_COMM_WORLD,&status);
    /* end predictor send/recv 1 */
        /* putting values in vectors */
        for(i=0;i<=my;i++){
            mid[i] = u[0][i];
            right[i] = u[p][i];
        }
        /* end putting values in vectors */
        /* explicit portion of predictor */
        if(p!=1){
            for(i=1;i<=my-1;i++){
                predict[i] =
                 (alpha(x[0],y[i])*alphaH -
                    beta(x[0],y[i])*betaH)*left[i]
                     + (1.0 - 2.0
                    *alpha(x[0],y[i])*
                    alphaH)*mid[i] +
                    (alpha(x[0],y[i])*alphaH +
                    beta(x[0],y[i])*betaH)*right[i]
                    +dt*fct(x[0],y[i],t0+ii*dt);
            }
        }
        /* end explicit portion of predictor */
        /* setting up b matrix */
        for(i=1;i<=my-1;i++){
            b[1][i] = (-alpha(x[0],y[i])*
            alpha_coef);
            b[2][i] = 1.0 + 2.0*alpha(x[0],y[i])*
            alpha_coef-
                dt*gam(x[0],y[i]);
            b[3][i] = (-alpha(x[0],y[i])
            *alpha_coef);
        }
        b[1][1] = 0.0;
        b[3][my-1] = 0.0;
        /* end setting up b matrix */
        /* setting up v1 vector (ax = v1) */
        for(i=1;i<=my-1;i++) v1[i] = predict[i];
        v1[1] = v1[1] +
             (alpha_coef*alpha(x[0],y[1]))
             *uxyt(x[0],y[0],t0+ii*dt);
        v1[my-1] = v1[my-1] +
             (alpha_coef*alpha(x[0],y[my-1]))
             *uxyt(x[0],y[my],t0+ii*dt);
        /* end setting up v1 vector (ax = v1) */
        /* gauss-seidel for v1_new */
            for(i=1;i<=my-1;i++) v1_new[i]=v1[i];
```

```
        iter=1;
        sent=0;
        while((sent!=1) && (iter<=imax)){
            sent=1;
            maxerr=0.0;
            for(i=1;i<=my-1;i++){
                old=v1_new[i];
                sum=0.0;
                sum = b[1][i]*v1_new[i-1]
                +b[3][i]*v1_new[i+1];
                v1_new[i]=(v1[i]-sum)/b[2][i];
                if(sent=1 && v1_new[i]!=0.0){
                    ea = fabs((v1_new[i]
                    -old)/v1_new[i]);
                    if(ea > maxerr) maxerr=ea;
                }
            }
            iter=iter+1;
            if(maxerr > es) sent = 0;
        }
    /* end gauss-seidel for v1_new */
    /* computing explicit predictor
     on coarse grid */
if(id!=0){
    for(i=p;i<=my-p;i++){
        v1_new[i] = (alpha(x[0],y[i])*alphaH)
        *(left[i] + right[i] +
            mid[i-p] + mid[i+p] - 4.0*mid[i])
            + (betaH*
            beta(x[0],y[i]))*(right[i] -
             left[i]) +
            dt*gam(x[0],y[i])*mid[i] +
             dt*fct(x[0],y[i],t0+(ii-1)*dt)
            + mid[i];
    }
}
    /* end computing explicit predictor on
     coarse grid */
/* replacing variables for next time */
if(id!=0){
        for(i=1;i<=my-1;i++){
            predict[i] = v1_new[i];
            temp[i] = u[0][i];
            u[0][i] = predict[i];
        }
    }
    /* end replacing variables for next time */

    /* predictor send/recv 2 */
    if(id!=0)
     MPI_Send(&predict,my+1,MPI_DOUBLE,id-1,
    9,MPI_COMM_WORLD);
    if(id!=(np-1)){
        MPI_Recv(&recv,my+1,MPI_DOUBLE,id+1,
        9,MPI_COMM_WORLD,&status);
        for(i=1;i<=my-1;i++){
            u[mx][i] = recv[i];
        }
    }
    /* end predictor send/recv 2 */
/* end predictor */
/* boundary values */
        for(i=1;i<=mx-1;i++){
        /* horizontal - bottom */
            v[i] = v[i] + alpha_coef
            *alpha(x[i],y[1],t0+ii*dt)*
                uxyt(x[i],y[0],t0+ii*dt);
        /* horizontal - top */
            v[(n-mx+1+i)] = v[(n-mx+1+i)] +
             (alpha_coef*
                alpha(x[i],y[my-1],t0+ii*dt))*
                uxyt(x[i],y[my],t0+ii*dt);
        }
        /* vertical - left */
        for(i=1;i<=my-1;i++){
            if(id!=0){
                v[1+((i-1)*(mx-1))] =
                 v[1+((i-1)*(mx-1))] +
                    (alpha_coef*alpha(x[1],
                    y[i],t0+ii*dt) -
                    beta_coef*beta(x[1],
                    y[i],t0+ii*dt))*
                    u[0][i];
            } else {
                v[1+((i-1)*(mx-1))] =
                 v[1+((i-1)*(mx-1))] +
                    (alpha_coef*alpha(x[1],
                    y[i],t0+ii*dt) -
                    beta_coef*beta(x[1],
                    y[i],t0+ii*dt))*
                    uxyt(x[0],y[i],t0+ii*dt);
            }
        }
        /* vertical - right */
        for(i=1;i<=my-1;i++){
            if(id!=(np-1)){
                v[(i*(mx-1))] = v[(i*(mx-1))] +
                 (alpha_coef*
                    alpha(x[mx-1],y[i],t0+ii*dt) +
                    beta_coef*beta(x[mx-1],
                    y[i],t0+ii*dt))*
                    u[mx][i];
            } else {
                v[i*(mx-1)] = v[i*(mx-1)] +
                 (alpha_coef*
                    alpha(x[mx-1],y[i],t0+ii*dt) +
                    beta_coef*beta(x[mx-1],
                    y[i],t0+ii*dt))*
                    uxyt(x[mx],y[i],t0+ii*dt);
            }
        }
/* end boundary values */
/* gauss-seidel solver */
    for(i=1;i<=n;i++) v_new[i]=v[i];

    iter = 1;
    sent = 0;
    while((sent != 1) && (iter <= imax)){
        sent = 1;
        maxerr = 0.0;
        for(i=1;i<=n;i++){
            old=v_new[i];
```

```
                sum = 0.0;
            sum = a[1][i]*v_new[i-(mx-1)] +
            a[2][i]*v_new[i-1] +
                    a[4][i]*v_new[i+1] +
                     a[5][i]*v_new[i+(mx-1)];
                v_new[i] = (v[i]-sum)/a[3][i];
                if((sent = 1) && (v_new[i] != 0.0)){
                    ea = fabs((v_new[i]-old)/v_new[i]);
                    if(ea > maxerr) maxerr = ea;
                }
            }
            iter = iter + 1;
            if(maxerr > es) sent = 0;
        }
    /* end gauss-seidel solver */
    /* begin corrector */
        /* corrector send/recv pair 1 */
        if(id!=(np-1)){
            for(i=0;i<=my;i++) send[i] =
             v_new[i*(mx-1)];
        }
        if(id!=(np-1))
         MPI_Send(&send,my+1,MPI_DOUBLE,id+1,
        9,MPI_COMM_WORLD);
        if(id!=0)
             MPI_Recv(&left,my+1,MPI_DOUBLE,id-1,
             9,MPI_COMM_WORLD,&status);
        /* end corrector send/recv pair 1 */
        /* putting values into vectors */
            for(i=0;i<=my;i++){
                mid[i] = temp[i];
                right[i] = v_new[1+(i-1)*(mx-1)];
            }
        /* end putting values into vectors */
        /* explicit correction */
            if(p!=1){
                for(i=1;i<=my-1;i++){
                    predict[i] =
                     (alpha(x[0],y[i])*alpha_coef -
                        beta(x[0],y[i])*beta_coef)*
                        left[i] + mid[i] +
                        (alpha(x[0],y[i])*alpha_coef +
                        beta(x[0],y[i])*beta_coef)
                        *right[i]
                        +dt*fct(x[0],y[i],t0+ii*dt);
                }
            }
        /* end explicit correction */
        /* setting up b matrix */
            for(i=1;i<=my-1;i++){
                b[1][i] = (-alpha(x[0],y[i])*
                alpha_coef);
                b[2][i] = 1.0 + 4.0*alpha(x[0],y[i])*
                alpha_coef
                    - dt*gam(x[0],y[i]);
                b[3][i] = (-alpha(x[0],y[i])*
                alpha_coef);
            }
            b[1][1] = 0.0;
            b[3][my-1] = 0.0;
        /* end setting up b matrix */
        /* setting up v1 vector (bx=v1) */
            for(i=1;i<=my-1;i++) v1[i] = predict[i];
            v1[1] = v1[1] +
                 (alpha_coef*alpha(x[0],y[1]))
                 *uxyt(x[0],y[0],t0+ii*dt);
            v1[my-1] = v1[my-1] +
                 (alpha_coef
                 *alpha(x[0],y[my-1]))*
                 uxyt(x[0],y[my],t0+ii*dt);
        /* end setting up v1 vector (bx=v1) */
        /* gauss-seidel for corrector */
                for(i=1;i<=my-1;i++) v1_new[i]=v1[i];

                iter=1;
                sent=0;
                while((sent!=1) && (iter<=imax)){
                    sent=1;
                    maxerr=0.0;
                    for(i=1;i<=my-1;i++){
                        old=v1_new[i];
                        sum=0.0;
                        sum = b[1][i]*v1_new[i-1]
                        +b[3][i]*v1_new[i+1];
                        v1_new[i]=(v1[i]-sum)/b[2][i];
                        if(sent=1 && v1_new[i]!=0.0){
                            ea = fabs((v1_new[i]
                            -old)/v1_new[i]);
                            if(ea > maxerr) maxerr=ea;
                        }
                    }
                    iter=iter+1;
                    if(maxerr > es) sent = 0;
                }
        /* end gauss-seidel for corrector */
        /* replacing variables prior to resend */
        if(id!=0){
            for(i=1;i<=my-1;i++){
                predict[i] = v1_new[i];
                u[0][i] = predict[i];
            }
        }
        /* end replacing variables prior to resend */

        /* corrector send/recv pair 2 */
        if(id!=0)
         MPI_Send(&predict,my+1,MPI_DOUBLE,id-1,
        9,MPI_COMM_WORLD);
        if(id!=(np-1)){
            MPI_Recv(&recv,my+1,MPI_DOUBLE,id+1,
            9,MPI_COMM_WORLD,&status);
            for(i=1;i<=my-1;i++) u[mx][i] = recv[i];
        }
        /* end corrector send/recv pair 2 */
/* end corrector */
/* putting v back into u */
    dum = 1;
    for(j=1;j<=my-1;j++){
        for(i=1;i<=mx-1;i++){
            u[i][j] = v_new[dum];
            dum++;
        }
    }
```

```
        /* end putting v back into u */
        /* updating boundaries */
            for(i=0;i<=mx;i++){
                u[i][0] = uxyt(x[i],y[0],t0+ii*dt);
                /* bottom */
                u[i][my] = uxyt(x[i],y[my],t0+ii*dt);
                /* top */
            }
            if(id==0){
                for(i=0;i<=my;i++)
                u[0][i] = uxyt(x[0],y[i],t0+ii*dt);
            }
            if(id==(np-1)){
                for(i=0;i<=my;i++)
                u[mx][i] = uxyt(x[mx],y[i],t0+ii*dt);
            }
        /* end updating boundaries */
        }                       /* end solver */
        end = time(NULL);
        time1 = end - start;

        /* error computation */
        maxerr = 0.0;
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                diff = fabs(uxyt(x[i],y[j],t0+(ii-1)*dt)-
                u[i][j]);
                if(diff > maxerr) maxerr = diff;
            }
        }
        printf("time=%.1f maxerr= %.3e\n",time1,maxerr);
        /* end error computation */

        /* output loop */
/*      for(j=0;j<=mx;j++){
            for(i=0;i<=my;i++){
                printf("%.5f\t%.5f\t%.10f\t%.10f\t%.16e\n",
                x[j],y[i],
                uxyt(x[j],y[i],(ii-1)*dt),u[j][i],
                (uxyt(x[j],y[i],(ii-1)*dt)-u[j][i]));
            }
        }
        /* end output loop */
        printf("\n\n");
    }                           /* end increment loop */

    /* closing MPI */
    MPI_Finalize();
    return 0;
    /* end closing MPI */

}
/* end main */

/* function definitions */
double uxyt(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return
             exp(-2.0*t)*sin(M_PI*x)*sin(M_PI*y);
                break;
            case 2: return exp(-2.0*t)*cos(3.0*x+y);
                break;
            case 3: return
             exp(t)*sin(2.0*x*M_PI)*sin(y*M_PI);
                break;
            case 4: return exp(-2.0*t)*sin(x)*sin(y);
                break;
            default:  printf("fix the example number\n");
            }
    }

double alpha(double x, double y)
    {

        int ex = 3;

        switch(ex)
```

```
            {
            case 1: return 1.0/(M_PI * M_PI);
            /* example 1 */
                break;
            case 2: return 1.0;             /* example 2 */
                break;
            case 3: return 1.0;             /* example 3 */
                break;
            case 4: return 1.0;             /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double beta(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 9.9*sin(x);      /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double gam(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
```

```
                break;
            case 2: return 8.0;             /* example 2 */
                break;
            case 3: return (1.0+5.0*M_PI*M_PI);
            /* example 3 */
                break;
            case 4: return -9.9*cos(x);
            /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double fct(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 0.0;             /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }
```

B.4 ceidd_exp2.c

```
/*  This program will solve the 2-d convection-diffusion equation
    using Backward Euler to solve and Gauss-Seidel as the interative
    matrix solver and exp2 on the interface
*/

#include "mpi.h"
#include "stdio.h"
```

```
#include "math.h"
#include "time.h"

double uxyt(); double alpha(); double beta();
double gam(); double fct();

/* begin main */
main(int argc, char *argv[])
{
    /* variables */
        double a[6][72200], u[405][405];
        double x[405], y[405];
        double v[72200], v_new[72200];
        double
         h[7]={0.1,0.05,0.025,0.01,0.005,0.0025,0.001};

        /* parallel variables */
        double b[4][405], left[405], mid[405], right[405];
        double send[405], recv[405], v1[405], v1_new[405];
        double predict[405], temp[405];
        double alphaH, betaH;
        int np, id;
        /* end parallel variables */

        int i, j, n, mx, my, mt, p;
        int dum, ii, loop;
        int iter, sent, imax=20000;

        double dx, dt, dy, lenx, leny, lent, H;
        double alpha_coef, beta_coef;
        double diff, maxerr, time1;
        double dummy, old, sum, ea, es =
         0.0000000000000001;

        double x0, xf, y0, yf, t0, tf;

        time_t start, end;

        int ex = 3;
        /* domain declarations */
        switch(ex)
        {
        case 1: x0 = 0.0; xf = 1.0;
            y0 = 0.0; yf = 1.0;
            t0 = 0.0; tf = 1.0;     /* example 1 */
            break;
        case 2: x0 = 0.0; xf = 1.0;
            y0 = 0.0; yf = 1.0;
            t0 = 0.0; tf = 1.0;     /* example 2 */
            break;
        case 3: x0 = 0.0; xf = 1.0;
            y0 = 0.0; yf = 1.0;
            t0 = 0.0; tf = 1.0;     /* example 3 */
            break;
        case 4: x0 = 0.0; xf = 2.0*M_PI;
            y0 = 0.0; yf = 1.0*M_PI;
            t0 = 0.0; tf = 1.0;     /* example 4 */
            break;
        default:  printf("fix the example number\n");
        }
    /* end domain declarations */
/* end variables */

lenx = xf - x0; leny = yf - y0; lent = tf - t0;

/* MPI startup */
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
/* end MPI startup */

for(loop=0;loop<=5;loop++)
{
    /* grid assignment */
    dx = /*M_PI**/h[loop]; dy = dx;
    mx = ceil(lenx/(np*dx)); my = ceil(leny/dy);
    n = (mx-1)*(my-1);
    dt = /*0.001*/pow(h[loop],2.0); mt = ceil(lent/dt);
    p = ceil(pow(2.0*dt,0.5)/dx); H=p*dx;
    /* end grid assignment */
```

```
        /* coefficients */
        alpha_coef = dt/pow(dx,2);
        beta_coef = dt/(2.0*dx);
        alphaH = dt/pow(H,2);
        betaH = dt/(2.0*H);
        /* end coefficients */

        /* grid points */
        for(i=0;i<=mx;i++) x[i] = x0 + (id*lenx)/np + i*dx;
        for(i=0;i<=my;i++) y[i] = y0 + i*dy;
        /* end grid points */

printf("x[0]=%f x[mx]=%f\ny[0]=%f
 y[my]=%f\n\n",x[0],x[mx],y[0],y[my]);

        /* initial conditions */
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                u[i][j] = uxyt(x[i],y[j],t0);
            }
        }
        /* end initial conditions */

        /* boundary initialization */
        /* lower and upper boundary */
            for(i=0;i<=mx;i++){
                u[i][0] = uxyt(x[i],y[0],t0);
                u[i][my] = uxyt(x[i],y[my],t0);
            }
        /* end lower and upper boundary */
        /* left and right boundary */
            for(i=0;i<=my;i++){
                if(id==0) u[0][i] = uxyt(x[0],y[i],t0);
                if(id==(np-1)) u[mx][i] =
                 uxyt(x[mx],y[i],t0);
            }
        /* end left and right boundary */
        /* end boundary initialization */

        /* initializing coefficient matrix a */
        for(i=1;i<=n;i++){
```

```
    a[3][i] = 1.0 + 4.0*alpha_coef*
        alpha(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1) +
         1]) -
        dt*gam(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1) +
         1]);
}
for(i=1;i<=n;i++){
    dum=i%(mx-1);
    if(dum!=1){
        a[2][i] = beta_coef
        *beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1) +
         1])-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
    } else a[2][i] = 0.0;
    if(dum!=0){
        a[4][i] = -beta_coef
        *beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1) +
         1])-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
    } else a[4][i] = 0.0;
}
for(i=1;i<=n;i++){
        a[1][i] = -alpha_coef*
                alpha(x[(i-1)%(mx-1)+1],
                y[(i-1)/(mx-1) + 1]);
        a[5][i] = -alpha_coef*
                alpha(x[(i-1)%(mx-1)+1],
                y[(i-1)/(mx-1) + 1]);
}
for(i=0;i<=mx-2;i++){
        a[1][i+1] = 0.0;
        a[5][n-i] = 0.0;
}
/* end intializing coefficient matrix a */


start = time(NULL);
```

```
for(ii=1;ii<=mt;ii++){
    dum = 1;

    for(j=1;j<=my-1;j++){
        for(i=1;i<=mx-1;i++){
            v[dum] = u[i][j] +
             dt*fct(x[i],y[j],t0+ii*dt);
            dum++;
        }
    }
/* begin predictor */
    /* predictor send/recv 1 */
    if(id!=(np-1)){
        for(i=0;i<=my;i++) send[i] = u[mx-p][i];
    }
    if(id!=(np-1))
     MPI_Send(&send,my+1,MPI_DOUBLE,id+1,
    9,MPI_COMM_WORLD);
    if(id!=0)
         MPI_Recv(&left,my+1,MPI_DOUBLE,id-1,
         9,MPI_COMM_WORLD,&status);
    /* end predictor send/recv 1 */
        /* putting values in vectors */
        for(i=0;i<=my;i++){
            mid[i] = u[0][i];
            right[i] = u[p][i];
        }
        /* end putting values in vectors */
        /* explicit portion of predictor */
        if(p!=1){
            for(i=1;i<=my-1;i++){
                predict[i] =
                 (alpha(x[0],y[i])*alphaH -
                    beta(x[0],y[i])*betaH)*left[i]
                     + (1.0 - 2.0
                    *alpha(x[0],y[i])*
                    alphaH)*mid[i] +
                    (alpha(x[0],y[i])*alphaH +
                    beta(x[0],y[i])*betaH)*right[i]
                    +dt*fct(x[0],y[i],t0+ii*dt);
            }
        }
        /* end explicit portion of predictor */
        /* setting up b matrix */
        for(i=1;i<=my-1;i++){
            b[1][i] = (-alpha(x[0],y[i])*
            alpha_coef);
            b[2][i] = 1.0 + 2.0*alpha(x[0],y[i])*
            alpha_coef-
                dt*gam(x[0],y[i]);
            b[3][i] = (-alpha(x[0],y[i])*
            alpha_coef);
        }
        b[1][1] = 0.0;
        b[3][my-1] = 0.0;
        /* end setting up b matrix */
        /* setting up v1 vector (ax = v1) */
        for(i=1;i<=my-1;i++) v1[i] = predict[i];
        v1[1] = v1[1] +
                 (alpha_coef*alpha(x[0],y[1]))
                 *uxyt(x[0],y[0],t0+ii*dt);
        v1[my-1] = v1[my-1] +
             (alpha_coef*alpha(x[0],y[my-1]))
             *uxyt(x[0],y[my],t0+ii*dt);
        /* end setting up v1 vector (ax = v1) */
        /* gauss-seidel for v1_new */
            for(i=1;i<=my-1;i++) v1_new[i]=v1[i];

            iter=1;
            sent=0;
            while((sent!=1) && (iter<=imax)){
                sent=1;
                maxerr=0.0;
                for(i=1;i<=my-1;i++){
                    old=v1_new[i];
                    sum=0.0;
                    sum = b[1][i]*v1_new[i-1]
                    +b[3][i]*v1_new[i+1];
                    v1_new[i]=(v1[i]-sum)/b[2][i];
                    if(sent=1 && v1_new[i]!=0.0){
                        ea = fabs((v1_new[i]-
                        old)/v1_new[i]);
                        if(ea > maxerr) maxerr=ea;
                    }
                }
                iter=iter+1;
                if(maxerr > es) sent = 0;
            }
        /* end gauss-seidel for v1_new */
    /* computing explicit predictor on
    coarse grid */
    if(id!=0){
        for(i=p;i<=my-p;i++){
            v1_new[i] = (alpha(x[0],y[i])*alphaH +
                    beta(x[0],y[i])*betaH)*
                    right[i+p] +
                    (alpha(x[0],y[i])*alphaH)*
                    left[i+p] +
                    (1.0-4.0*alpha(x[0],y[i])*
                    alphaH +
                    dt*gam(x[0],y[i]))*mid[i] +
                    (alpha(x[0],y[i])*alphaH)*
                    right[i-p] +
                    (alpha(x[0],y[i])*alphaH +
                    beta(x[0],y[i])*betaH)*
                    left[i-p] +
                    dt*fct(x[0],y[i],t0+ii*dt);
        }
    }
    /* end computing explicit predictor
    on coarse grid */
    /* replacing variables for next time */
    if(id!=0){
        for(i=1;i<=my-1;i++){
            predict[i] = v1_new[i];
            temp[i] = u[0][i];
            u[0][i] = predict[i];
        }
    }
    /* end replacing variables for next time */

    /* predictor send/recv 2 */
```

```
    if(id!=0)
     MPI_Send(&predict,my+1,MPI_DOUBLE,id-1,
    9,MPI_COMM_WORLD);
    if(id!=(np-1)){
        MPI_Recv(&recv,my+1,MPI_DOUBLE,id+1,
        9,MPI_COMM_WORLD,&status);
        for(i=1;i<=my-1;i++){
            u[mx][i] = recv[i];
        }
    }
    /* end predictor send/recv 2 */
/* end predictor */
/* boundary values */
        for(i=1;i<=mx-1;i++){
        /* horizontal - bottom */
            v[i] = v[i] + alpha_coef
            *alpha(x[i],y[1],t0+ii*dt)*
                uxyt(x[i],y[0],t0+ii*dt);
        /* horizontal - top */
            v[(n-mx+1+i)] = v[(n-mx+1+i)] +
             (alpha_coef*
                alpha(x[i],y[my-1],t0+ii*dt))*
                uxyt(x[i],y[my],t0+ii*dt);
        }
        /* vertical - left */
        for(i=1;i<=my-1;i++){
            if(id!=0){
                v[1+((i-1)*(mx-1))] =
                 v[1+((i-1)*(mx-1))] +
                    (alpha_coef*alpha(x[1],
                    y[i],t0+ii*dt) -
                    beta_coef*beta(x[1],
                    y[i],t0+ii*dt))*
                    u[0][i];
            } else {
                v[1+((i-1)*(mx-1))] =
                 v[1+((i-1)*(mx-1))] +
                    (alpha_coef*alpha(x[1],
                    y[i],t0+ii*dt) -
                    beta_coef*beta(x[1],
                    y[i],t0+ii*dt))*
                    uxyt(x[0],y[i],t0+ii*dt);
            }
        }
        /* vertical - right */
        for(i=1;i<=my-1;i++){
            if(id!=(np-1)){
                v[(i*(mx-1))] = v[(i*(mx-1))] +
                 (alpha_coef*
                    alpha(x[mx-1],y[i],t0+ii*dt) +
                    beta_coef*beta(x[mx-1],
                    y[i],t0+ii*dt))*
                    u[mx][i];
            } else {
                v[i*(mx-1)] = v[i*(mx-1)] +
                 (alpha_coef*
                    alpha(x[mx-1],y[i],t0+ii*dt) +
                    beta_coef*beta(x[mx-1],
                    y[i],t0+ii*dt))*
                    uxyt(x[mx],y[i],t0+ii*dt);
            }
        }
/* end boundary values */
/* gauss-seidel solver */
    for(i=1;i<=n;i++) v_new[i]=v[i];

    iter = 1;
    sent = 0;
    while((sent != 1) && (iter <= imax)){
        sent = 1;
        maxerr = 0.0;
        for(i=1;i<=n;i++){
            old=v_new[i];
            sum = 0.0;
        sum = a[1][i]*v_new[i-(mx-1)] +
        a[2][i]*v_new[i-1] +
            a[4][i]*v_new[i+1] +
            a[5][i]*v_new[i+(mx-1)];
            v_new[i] = (v[i]-sum)/a[3][i];
            if((sent = 1) && (v_new[i] != 0.0)){
                ea = fabs((v_new[i]-old)/v_new[i]);
                if(ea > maxerr) maxerr = ea;
            }
        }
        iter = iter + 1;
        if(maxerr > es) sent = 0;
    }
/* end gauss-seidel solver */
/* begin corrector */
    /* corrector send/recv pair 1 */
    if(id!=(np-1)){
        for(i=0;i<=my;i++) send[i] =
         v_new[i*(mx-1)];
    }
    if(id!=(np-1))
     MPI_Send(&send,my+1,MPI_DOUBLE,id+1,
    9,MPI_COMM_WORLD);
    if(id!=0)
         MPI_Recv(&left,my+1,MPI_DOUBLE,id-1,
         9,MPI_COMM_WORLD,&status);
    /* end corrector send/recv pair 1 */
    /* putting values into vectors */
        for(i=0;i<=my;i++){
            mid[i] = temp[i];
            right[i] = v_new[1+(i-1)*(mx-1)];
        }
    /* end putting values into vectors */
    /* explicit correction */
        if(p!=1){
            for(i=1;i<=my-1;i++){
                predict[i] = (alpha(x[0],y[i])*
                alpha_coef -
                    beta(x[0],y[i])*
                    beta_coef)*left[i] +
                    mid[i] +
                    (alpha(x[0],y[i])*alpha_coef +
                    beta(x[0],y[i])*beta_coef)*
                    right[i]
                    +dt*fct(x[0],y[i],t0+ii*dt);
            }
        }
    /* end explicit correction */
    /* setting up b matrix */
        for(i=1;i<=my-1;i++){
            b[1][i] =
            (-alpha(x[0],y[i])*alpha_coef);
            b[2][i] = 1.0 +
             4.0*alpha(x[0],y[i])*alpha_coef
                - dt*gam(x[0],y[i]);
            b[3][i] =
            (-alpha(x[0],y[i])*alpha_coef);
        }
        b[1][1] = 0.0;
        b[3][my-1] = 0.0;
    /* end setting up b matrix */
    /* setting up v1 vector (bx=v1) */
        for(i=1;i<=my-1;i++) v1[i] = predict[i];
        v1[1] = v1[1] +
             (alpha_coef*alpha(x[0],y[1]))
             *uxyt(x[0],y[0],t0+ii*dt);
        v1[my-1] = v1[my-1] +
             (alpha_coef*alpha(x[0],y[my-1]))
             *uxyt(x[0],y[my],t0+ii*dt);
    /* end setting up v1 vector (bx=v1) */
    /* gauss-seidel for corrector */
            for(i=1;i<=my-1;i++) v1_new[i]=v1[i];

            iter=1;
            sent=0;
            while((sent!=1) && (iter<=imax)){
                sent=1;
                maxerr=0.0;
                for(i=1;i<=my-1;i++){
                    old=v1_new[i];
                    sum=0.0;
                    sum = b[1][i]*v1_new[i-1]
                    +b[3][i]*v1_new[i+1];
                    v1_new[i]=(v1[i]-sum)/b[2][i];
                    if(sent=1 && v1_new[i]!=0.0){
                        ea = fabs((v1_new[i]-
                        old)/v1_new[i]);
                        if(ea > maxerr) maxerr=ea;
                    }
                }
```

```
                iter=iter+1;
                if(maxerr > es) sent = 0;
            }
        /* end gauss-seidel for corrector */
        /* replacing variables prior to resend */
        if(id!=0){
            for(i=1;i<=my-1;i++){
                predict[i] = v1_new[i];
                u[0][i] = predict[i];
            }
        }
        /* end replacing variables prior to resend */

        /* corrector send/recv pair 2 */
        if(id!=0)
         MPI_Send(&predict,my+1,MPI_DOUBLE,id-1,
        9,MPI_COMM_WORLD);
        if(id!=(np-1)){
            MPI_Recv(&recv,my+1,MPI_DOUBLE,id+1,
            9,MPI_COMM_WORLD,&status);
            for(i=1;i<=my-1;i++) u[mx][i] = recv[i];
        }
        /* end corrector send/recv pair 2 */
    /* end corrector */
    /* putting v back into u */
        dum = 1;
        for(j=1;j<=my-1;j++){
            for(i=1;i<=mx-1;i++){
                u[i][j] = v_new[dum];
                dum++;
            }
        }
    /* end putting v back into u */
    /* updating boundaries */
        for(i=0;i<=mx;i++){
            u[i][0] = uxyt(x[i],y[0],t0+ii*dt);
            /* bottom */
            u[i][my] = uxyt(x[i],y[my],t0+ii*dt);
            /* top */
        }
        if(id==0){
                for(i=0;i<=my;i++)
                u[0][i] = uxyt(x[0],y[i],t0+ii*dt);
            }
            if(id==(np-1)){
                for(i=0;i<=my;i++)
                u[mx][i] = uxyt(x[mx],y[i],t0+ii*dt);
            }
        /* end updating boundaries */
        }                       /* end solver */
        end = time(NULL);
        time1 = end - start;

        /* error computation */
        maxerr = 0.0;
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                diff = fabs(uxyt(x[i],y[j],t0+(ii-1)*dt)
                -u[i][j]);
                if(diff > maxerr) maxerr = diff;
            }
        }
        printf("time=%.1f maxerr= %.3e\n",time1,maxerr);
        /* end error computation */

        /* output loop */
/*      for(j=0;j<=mx;j++){
            for(i=0;i<=my;i++){
                printf("%.5f\t%.5f\t%.10f\t%.10f\t%.16e\n",
                x[j],y[i],
                uxyt(x[j],y[i],(ii-1)*dt),u[j][i],
                (uxyt(x[j],y[i],(ii-1)*dt)-u[j][i]));
            }
        }
        /* end output loop */
        printf("\n\n");
    }                           /* end increment loop */

    /* closing MPI */
    MPI_Finalize();
    return 0;
    /* end closing MPI */
```

```
}
/* end main */

/* function definitions */
double uxyt(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return
             exp(-2.0*t)*sin(M_PI*x)*sin(M_PI*y);
                break;
            case 2: return exp(-2.0*t)*cos(3.0*x+y);
                break;
            case 3: return
             exp(t)*sin(2.0*x*M_PI)*sin(y*M_PI);
                break;
            case 4: return exp(-2.0*t)*sin(x)*sin(y);
                break;
            default:  printf("fix the example number\n");
            }
    }

double alpha(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 1.0/(M_PI * M_PI);
            /* example 1 */
                break;
            case 2: return 1.0;             /* example 2 */
                break;
            case 3: return 1.0;             /* example 3 */
                break;
            case 4: return 1.0;             /* example 4 */
```

```
                break;
            default:  printf("fix the example number\n");
            }
    }

double beta(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 9.9*sin(x);      /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double gam(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 8.0;             /* example 2 */
                break;
            case 3: return (1.0+5.0*M_PI*M_PI);
            /* example 3 */
                break;
            case 4: return -9.9*cos(x);     /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double fct(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 0.0;             /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }
```

B.5 ceidd_lex.c

```
/*  This program will solve the 2-d convection-diffusion equation
    using Backward Euler to solve and Gauss-Seidel as the interative
    matrix solver and use lex on the interface
*/

#include "mpi.h"
#include "stdio.h"
#include "math.h"
#include "time.h"

double uxyt(); double alpha(); double beta();
double gam(); double fct();

/* begin main */
main(int argc, char *argv[])
{
    /* variables */
```

```
double a[6][72200], u[405][405];
double x[405], y[405];
double v[72200], v_new[72200];
double
 h[7]={0.1,0.05,0.025,0.01,0.005,0.0025,0.001};

/* parallel variables */
double b[4][405], left[405], mid[405], right[405];
double send[405], recv[405], v1[405], v1_new[405];
double predict[405], temp[405], old1[405],
 old2[405];
double alphaH, betaH;
int np, id;
/* end parallel variables */

int i, j, n, mx, my, mt, p;
int dum, ii, loop;
int iter, sent, imax=20000;

double dx, dt, dy, lenx, leny, lent, H;
double alpha_coef, beta_coef;
double diff, maxerr, time1;
double dummy, old, sum, ea, es =
 0.0000000000000001;

double x0, xf, y0, yf, t0, tf;

time_t start, end;

int ex = 3;
/* domain declarations */
switch(ex)
    {
    case 1: x0 = 0.0; xf = 1.0;
        y0 = 0.0; yf = 1.0;
        t0 = 0.0; tf = 1.0;     /* example 1 */
        break;
    case 2: x0 = 0.0; xf = 1.0;
        y0 = 0.0; yf = 1.0;
        t0 = 0.0; tf = 1.0;     /* example 2 */
        break;
        case 3: x0 = 0.0; xf = 1.0;
            y0 = 0.0; yf = 1.0;
            t0 = 0.0; tf = 1.0;     /* example 3 */
            break;
        case 4: x0 = 0.0; xf = 2.0*M_PI;
            y0 = 0.0; yf = 1.0*M_PI;
            t0 = 0.0; tf = 1.0;     /* example 4 */
            break;
        default:  printf("fix the example number\n");
        }
    /* end domain declarations */
/* end variables */

lenx = xf - x0; leny = yf - y0; lent = tf - t0;
/* MPI startup */
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
/* end MPI startup */

for(loop=0;loop<=5;loop++)
{
    /* grid assignment */
    dx = /*M_PI**/h[loop]; dy = dx;
    mx = ceil(lenx/(np*dx)); my = ceil(leny/dy);
    n = (mx-1)*(my-1);
    dt = 0.01/*pow(h[loop],2.0)*/; mt = ceil(lent/dt);
    p = ceil(pow(2.0*dt,0.5)/dx); H=p*dx;
    /* end grid assignment */

    /* coefficients */
    alpha_coef = dt/pow(dx,2);
    beta_coef = dt/(2.0*dx);
    alphaH = dt/pow(H,2);
    betaH = dt/(2.0*H);
    /* end coefficients */

    /* grid points */
    for(i=0;i<=mx;i++) x[i] = x0 + (id*lenx)/np + i*dx;
    for(i=0;i<=my;i++) y[i] = y0 + i*dy;
```

```
        /* end grid points */

printf("x[0]=%f x[mx]=%f\ny[0]=%f
 y[my]=%f\n\n",x[0],x[mx],y[0],y[my]);

        /* initial conditions */
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                u[i][j] = uxyt(x[i],y[j],t0);
            }
        }
        /* end initial conditions */

        /* boundary initialization */
        /* lower and upper boundary */
            for(i=0;i<=mx;i++){
                u[i][0] = uxyt(x[i],y[0],t0);
                u[i][my] = uxyt(x[i],y[my],t0);
            }
        /* end lower and upper boundary */
        /* left and right boundary */
            for(i=0;i<=my;i++){
                if(id==0) u[0][i] = uxyt(x[0],y[i],t0);
                if(id==(np-1)) u[mx][i] =
                 uxyt(x[mx],y[i],t0);
            }
        /* end left and right boundary */
        /* end boundary initialization */

        /* initializing coefficient matrix a */
        for(i=1;i<=n;i++){
            a[3][i] = 1.0 + 4.0*alpha_coef*
                alpha(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
                 + 1]) -
                dt*gam(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
                + 1]);
        }
        for(i=1;i<=n;i++){
            dum=i%(mx-1);
            if(dum!=1){
                a[2][i] = beta_coef
        *beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
        + 1])-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
    } else a[2][i] = 0.0;
    if(dum!=0){
        a[4][i] = -beta_coef
        *beta(x[(i-1)%(mx-1)+1],y[(i-1)/(mx-1)
        + 1])-
            alpha_coef*alpha(x[(i-1)%(mx-1)+1],
            y[(i-1)/(mx-1)
             + 1]);
    } else a[4][i] = 0.0;
}
for(i=1;i<=n;i++){
        a[1][i] = -alpha_coef*
                alpha(x[(i-1)%(mx-1)+1],
                y[(i-1)/(mx-1) + 1]);
        a[5][i] = -alpha_coef*
                alpha(x[(i-1)%(mx-1)+1],
                y[(i-1)/(mx-1) + 1]);
}
for(i=0;i<=mx-2;i++){
        a[1][i+1] = 0.0;
        a[5][n-i] = 0.0;
}
/* end intializing coefficient matrix a */


start = time(NULL);

for(ii=1;ii<=mt;ii++){
    dum = 1;

    for(j=1;j<=my-1;j++){
        for(i=1;i<=mx-1;i++){
            v[dum] = u[i][j] +
             dt*fct(x[i],y[j],t0+ii*dt);
            dum++;
        }
    }
/* begin predictor */
    /* predictor send/recv 1 */
    if(id!=(np-1)){
        for(i=0;i<=my;i++) send[i] = u[mx-p][i];
    }
    if(id!=(np-1))
     MPI_Send(&send,my+1,MPI_DOUBLE,id+1,
    9,MPI_COMM_WORLD);
    if(id!=0)
         MPI_Recv(&left,my+1,MPI_DOUBLE,id-1,
         9,MPI_COMM_WORLD,&status);
    /* end predictor send/recv 1 */
        /* putting values in vectors */
        for(i=0;i<=my;i++){
            mid[i] = u[0][i];
            right[i] = u[p][i];
        }
        /* end putting values in vectors */
        /* explicit portion of predictor */
        if(p!=1){
            for(i=1;i<=my-1;i++){
                predict[i] =
                 (alpha(x[0],y[i])*alphaH -
                    beta(x[0],y[i])*betaH)*left[i]
                     +  (1.0 - 2.0
                    *alpha(x[0],y[i])*alphaH)
                    *mid[i] +
                    (alpha(x[0],y[i])*alphaH +
                    beta(x[0],y[i])*betaH)*right[i]
                    +dt*fct(x[0],y[i],t0+ii*dt);
            }
        }
        /* end explicit portion of predictor */
        /* setting up b matrix */
        for(i=1;i<=my-1;i++){
            b[1][i] =
            (-alpha(x[0],y[i])*alpha_coef);
            b[2][i] = 1.0 + 2.0*alpha(x[0],y[i])
            *alpha_coef-
                dt*gam(x[0],y[i]);
            b[3][i] =
            (-alpha(x[0],y[i])*alpha_coef);
        }
        b[1][1] = 0.0;
        b[3][my-1] = 0.0;
        /* end setting up b matrix */
        /* setting up v1 vector (ax = v1) */
        for(i=1;i<=my-1;i++) v1[i] = predict[i];
        v1[1] = v1[1] +
             (alpha_coef*alpha(x[0],y[1]))
             *uxyt(x[0],y[0],t0+ii*dt);
        v1[my-1] = v1[my-1] +
             (alpha_coef
             *alpha(x[0],y[my-1]))*uxyt(x[0],
             y[my],t0+ii*dt);
        /* end setting up v1 vector (ax = v1) */
        /* gauss-seidel for v1_new */
            for(i=1;i<=my-1;i++) v1_new[i]=v1[i];

            iter=1;
            sent=0;
            while((sent!=1) && (iter<=imax)){
                sent=1;
                maxerr=0.0;
                for(i=1;i<=my-1;i++){
                    old=v1_new[i];
                    sum=0.0;
                    sum = b[1][i]*v1_new[i-1]
                    +b[3][i]*v1_new[i+1];
                    v1_new[i]=(v1[i]-sum)/b[2][i];
                    if(sent=1 && v1_new[i]!=0.0){
                        ea = fabs((v1_new[i]
                        -old)/v1_new[i]);
                        if(ea > maxerr) maxerr=ea;
                    }
                }
                iter=iter+1;
                if(maxerr > es) sent = 0;
            }
        /* end gauss-seidel for v1_new */
    /* replacing variables for next time */
    if(id!=0){
        /* begin computation of
        linear extrapolation */
        if(ii==1){
            for(i=1;i<=my-1;i++) old2[i]=u[0][i];
        } else {
            for(i=p;i<=my-p;i++)
            v1_new[i]=2.0*old1[i]-old2[i];
        }
        /* end computation of
        linear extrapolation */
        for(i=1;i<=my-1;i++){
            predict[i] = v1_new[i];
            temp[i] = u[0][i];
            u[0][i] = predict[i];
        }
    }
    /* end replacing variables for next time */

    /* predictor send/recv 2 */
    if(id!=0)
     MPI_Send(&predict,my+1,MPI_DOUBLE,id-1,
    9,MPI_COMM_WORLD);
    if(id!=(np-1)){
        MPI_Recv(&recv,my+1,MPI_DOUBLE,id+1,
        9,MPI_COMM_WORLD,&status);
        for(i=1;i<=my-1;i++){
            u[mx][i] = recv[i];
        }
    }
    /* end predictor send/recv 2 */
/* end predictor */
/* boundary values */
        for(i=1;i<=mx-1;i++){
        /* horizontal - bottom */
            v[i] = v[i] + alpha_coef
            *alpha(x[i],y[1],t0+ii*dt)*
                uxyt(x[i],y[0],t0+ii*dt);
        /* horizontal - top */
            v[(n-mx+1+i)] = v[(n-mx+1+i)] +
             (alpha_coef*
        alpha(x[i],y[my-1],t0+ii*dt))*
        uxyt(x[i],y[my],t0+ii*dt);
}
/* vertical - left */
for(i=1;i<=my-1;i++){
    if(id!=0){
        v[1+((i-1)*(mx-1))] =
         v[1+((i-1)*(mx-1))] +
            (alpha_coef*alpha(x[1],
            y[i],t0+ii*dt) -
            beta_coef*beta(x[1],
            y[i],t0+ii*dt))*
            u[0][i];
    } else {
        v[1+((i-1)*(mx-1))] =
         v[1+((i-1)*(mx-1))] +
            (alpha_coef*alpha(x[1],
            y[i],t0+ii*dt) -
            beta_coef*beta(x[1],
            y[i],t0+ii*dt))*
            uxyt(x[0],y[i],t0+ii*dt);
    }
}
/* vertical - right */
for(i=1;i<=my-1;i++){
    if(id!=(np-1)){
        v[(i*(mx-1))] = v[(i*(mx-1))] +
         (alpha_coef*
            alpha(x[mx-1],y[i],t0+ii*dt) +
            beta_coef*beta(x[mx-1],
            y[i],t0+ii*dt))*
            u[mx][i];
    } else {
        v[i*(mx-1)] = v[i*(mx-1)] +
         (alpha_coef*
            alpha(x[mx-1],y[i],t0+ii*dt) +
            beta_coef*beta(x[mx-1],
            y[i],t0+ii*dt))*
            uxyt(x[mx],y[i],t0+ii*dt);
    }
}
```

```
/* end boundary values */
/* gauss-seidel solver */
    for(i=1;i<=n;i++) v_new[i]=v[i];

    iter = 1;
    sent = 0;
    while((sent != 1) && (iter <= imax)){
        sent = 1;
        maxerr = 0.0;
        for(i=1;i<=n;i++){
            old=v_new[i];
            sum = 0.0;
        sum = a[1][i]*v_new[i-(mx-1)] +
         a[2][i]*v_new[i-1] +
            a[4][i]*v_new[i+1] +
             a[5][i]*v_new[i+(mx-1)];
            v_new[i] = (v[i]-sum)/a[3][i];
            if((sent = 1) && (v_new[i] != 0.0)){
                ea = fabs((v_new[i]-old)/v_new[i]);
                if(ea > maxerr) maxerr = ea;
            }
        }
        iter = iter + 1;
        if(maxerr > es) sent = 0;
    }
/* end gauss-seidel solver */
/* begin corrector */
    /* corrector send/recv pair 1 */
    if(id!=(np-1)){
        for(i=0;i<=my;i++) send[i] =
         v_new[i*(mx-1)];
    }
    if(id!=(np-1))
     MPI_Send(&send,my+1,MPI_DOUBLE,id+1,
    9,MPI_COMM_WORLD);
    if(id!=0)
         MPI_Recv(&left,my+1,MPI_DOUBLE,id-1,
         9,MPI_COMM_WORLD,&status);
    /* end corrector send/recv pair 1 */
    /* putting values into vectors */
        for(i=0;i<=my;i++){
```

```
    mid[i] = temp[i];
    right[i] = v_new[1+(i-1)*(mx-1)];
  }
/* end putting values into vectors */
/* explicit correction */
  if(p!=1){
    for(i=1;i<=my-1;i++){
      predict[i] =
       (alpha(x[0],y[i])*alpha_coef -
         beta(x[0],y[i])*beta_coef)*
         left[i] + mid[i] +
         (alpha(x[0],y[i])*alpha_coef +
         beta(x[0],y[i])*beta_coef
         )*right[i]
         +dt*fct(x[0],y[i],t0+ii*dt);
    }
  }
/* end explicit correction */
/* setting up b matrix */
  for(i=1;i<=my-1;i++){
    b[1][i] =
    (-alpha(x[0],y[i])*alpha_coef);
    b[2][i] = 1.0 + 4.0*alpha(x[0],y[i])*
    alpha_coef
      - dt*gam(x[0],y[i]);
    b[3][i] =
    (-alpha(x[0],y[i])*alpha_coef);
  }
  b[1][1] = 0.0;
  b[3][my-1] = 0.0;
/* end setting up b matrix */
/* setting up v1 vector (bx=v1) */
  for(i=1;i<=my-1;i++) v1[i] = predict[i];
  v1[1] = v1[1] +
     (alpha_coef*alpha(x[0],y[1]))
     *uxyt(x[0],y[0],t0+ii*dt);
  v1[my-1] = v1[my-1] +
     (alpha_coef
     *alpha(x[0],y[my-1]))*
     uxyt(x[0],y[my],t0+ii*dt);
/* end setting up v1 vector (bx=v1) */
/* gauss-seidel for corrector */
        for(i=1;i<=my-1;i++) v1_new[i]=v1[i];

        iter=1;
        sent=0;
        while((sent!=1) && (iter<=imax)){
            sent=1;
            maxerr=0.0;
            for(i=1;i<=my-1;i++){
                old=v1_new[i];
                sum=0.0;
                sum = b[1][i]*v1_new[i-1]
                +b[3][i]*v1_new[i+1];
                v1_new[i]=(v1[i]-sum)/b[2][i];
                if(sent=1 && v1_new[i]!=0.0){
                    ea = fabs((v1_new[i]
                    -old)/v1_new[i]);
                    if(ea > maxerr) maxerr=ea;
                }
            }
            iter=iter+1;
            if(maxerr > es) sent = 0;
        }
/* end gauss-seidel for corrector */
/* setting up for next loop */
    if(ii==1){
        for(i=1;i<=my-1;i++) old1[i]=v1_new[i];
    } else {
        for(i=1;i<=my-1;i++){
            old2[i]=old1[i];
            old1[i]=v1_new[i];
        }
    }
/* end setting up for next loop */
/* replacing variables prior to resend */
if(id!=0){
    for(i=1;i<=my-1;i++){
        predict[i] = v1_new[i];
        u[0][i] = predict[i];
    }
}
```

```
    /* end replacing variables prior to resend */

    /* corrector send/recv pair 2 */
    if(id!=0)
     MPI_Send(&predict,my+1,MPI_DOUBLE,id-1,
    9,MPI_COMM_WORLD);
    if(id!=(np-1)){
        MPI_Recv(&recv,my+1,MPI_DOUBLE,id+1,
        9,MPI_COMM_WORLD,&status);
        for(i=1;i<=my-1;i++) u[mx][i] = recv[i];
    }
    /* end corrector send/recv pair 2 */
/* end corrector */
/* putting v back into u */
    dum = 1;
    for(j=1;j<=my-1;j++){
        for(i=1;i<=mx-1;i++){
            u[i][j] = v_new[dum];
            dum++;
        }
    }
/* end putting v back into u */
/* updating boundaries */
    for(i=0;i<=mx;i++){
        u[i][0] = uxyt(x[i],y[0],t0+ii*dt);
        /* bottom */
        u[i][my] = uxyt(x[i],y[my],t0+ii*dt);
        /* top */
    }
    if(id==0){
        for(i=0;i<=my;i++)
        u[0][i] = uxyt(x[0],y[i],t0+ii*dt);
    }
    if(id==(np-1)){
        for(i=0;i<=my;i++)
        u[mx][i] = uxyt(x[mx],y[i],t0+ii*dt);
    }
/* end updating boundaries */
}                       /* end solver */
end = time(NULL);
time1 = end - start;
```

```
        /* error computation */
        maxerr = 0.0;
        for(j=0;j<=my;j++){
            for(i=0;i<=mx;i++){
                diff = fabs(uxyt(x[i],y[j],t0+(ii-1)*dt)
                -u[i][j]);
                if(diff > maxerr) maxerr = diff;
            }
        }
        printf("time=%.1f maxerr= %.16e\n",time1,maxerr);
        /* end error computation */

        /* output loop */
/*      for(j=0;j<=mx;j++){
            for(i=0;i<=my;i++){
                printf("%.5f\t%.5f\t%.10f\t%.10f\t%.16e\n",
                x[j],y[i],
                uxyt(x[j],y[i],(ii-1)*dt),u[j][i],
                (uxyt(x[j],y[i],(ii-1)*dt)-u[j][i]));
            }
        }
        /* end output loop */
        printf("\n\n");
    }                           /* end increment loop */

    /* closing MPI */
    MPI_Finalize();
    return 0;
    /* end closing MPI */

}
/* end main */

/* function definitions */
double uxyt(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
```

```
            {
            case 1: return
             exp(-2.0*t)*sin(M_PI*x)*sin(M_PI*y);
                break;
            case 2: return exp(-2.0*t)*cos(3.0*x+y);
                break;
            case 3: return
             exp(t)*sin(2.0*x*M_PI)*sin(y*M_PI);
                break;
            case 4: return exp(-2.0*t)*sin(x)*sin(y);
                break;
            default:  printf("fix the example number\n");
            }
    }

double alpha(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 1.0/(M_PI * M_PI);
            /* example 1 */
                break;
            case 2: return 1.0;             /* example 2 */
                break;
            case 3: return 1.0;             /* example 3 */
                break;
            case 4: return 1.0;             /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double beta(double x, double y)
    {

        int ex = 3;

        switch(ex)
```

```
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 9.9*sin(x);      /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double gam(double x, double y)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 8.0;             /* example 2 */
                break;
            case 3: return (1.0+5.0*M_PI*M_PI);
            /* example 3 */
                break;
            case 4: return -9.9*cos(x);     /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }

double fct(double x, double y, double t)
    {

        int ex = 3;

        switch(ex)
            {
            case 1: return 0.0;             /* example 1 */
                break;
            case 2: return 0.0;             /* example 2 */
                break;
            case 3: return 0.0;             /* example 3 */
                break;
            case 4: return 0.0;             /* example 4 */
                break;
            default:  printf("fix the example number\n");
            }
    }
```

CURRICULUM VITAE

Shawn J.A. Chiappetta

Natural Sciences Area
University of Sioux Falls
1101 West 22nd Street
Sioux Falls, South Dakota 57105
e-mail: shawn.chiappetta@usiouxfalls.edu

Education

- University of Wisconsin - Milwaukee, Milwaukee, Wisconsin

  Ph.D., Mathematical Sciences (expected graduation: May 2009)

  - Dissertation Topic: Non-Overlapping Domain Decomposition Parallel Algorithms for Convection-Diffusion Problems
  - Advisor: Dr. Bruce A. Wade
  - Area of Study: Parallel Numerical Algorithms

- Southern Illinois University - Carbondale, Carbondale, Illinois

  M.S., Mathematics, May 1998

  - Research Paper: Operators on the Hilbert Space $L_2[a, b]$
  - Advisor: Dr. Ronald Grimmer

- Carthage College, Kenosha, Wisconsin

  B.A., Mathematics and Accounting, May 1996

Teaching History

- Assistant Professor of Mathematics - University of Sioux Falls, Sioux Falls, South Dakota. Fall 2003-present

  - Teaching responsibilities span the mathematics curriculum and extends to interdisciplinary courses beyond mathematics.

- Visiting Assistant Professor of Mathematics - Carthage College, Kenosha, Wisconsin, Academic Year 2002-2003

  - Taught:

    * Applied Mathematics (liberal arts math course)
    * Principles of Modern Mathematics (for teacher certification)
    * Functions, Graphs, and Analysis (pre-calculus)

- Graduate Teaching Assistant - University of Wisconsin - Milwaukee, Milwaukee, Wisconsin. August 1998-May 2002
  - Taught:
    * College Algebra
    * Combined course of Algebra/Trigonometry and Calculus
- Graduate Teaching Assistant - Southern Illinois University - Carbondale. August 1996-May 1998
  - Taught:
    * Trigonometry
    * Finite Mathematics
  - Led Recitation Sessions:
    * Applied Mathematics (liberal arts math course)
    * Calculus I

Leadership and Service

- Information Officer/Webmaster, Mathematical Association of America/ North Central Section, April 2007-present
- Co-Chair, North Central Section NExT, Mathematical Association of America/ North Central Section, April 2008-present
- Application Committee, Mathematical Association of America/ North Central Section, October 2006-present
- Co-Organizer, Math on the Northern Plains Conference, University of Sioux Falls, Sioux Falls, South Dakota, April 2006
- Coordinator, Mathematical Association of America/North Central Section Fall Section Meeting, University of Sioux Falls, Sioux Falls, SD, October 2003
- Assessment Committee member, University of Sioux Falls, Sioux Falls, SD, Fall 2008-present
  - Chair, August 2008-present

- Committee on Liberal Arts Education, University of Sioux Falls, Sioux Falls, SD, Spring 2004-Spring 2008
  - Chair, August 2005- August 2007
- Director of USF Mathematics Tutoring Center, University of Sioux Falls, Sioux Falls, SD, August 2003-August 2008
- Academic Success Center ad-hoc Committee Member, University of Sioux Falls, Sioux Falls, SD, Fall 2006-Fall 2008
- USF Faculty Club Advisor, University of Sioux Falls, Sioux Falls, SD, Spring 2004-present
  - USF Math/CS Club, 2008-present
  - USF Aviation Club, 2006-present
  - USF Chess Club, 2004-2006
- Search Committees, 2005 (math), 2007(math and finance)

Research Interests

- Numerical analysis, parallel algorithms

Professional Presentations

- "Implementation of a Capstone Course in the Mathematical Sciences," Presented at the MAA/NCS Project NExT Meeting, College of St. Benedict, Collegeville, MN, April 2008
- "Baby Steps: An Introduction to Mathematical Technology in the Classroom," Presented at the MAA/NCS Project NExT Meeting, Bemidji State University, Bemidji, MN, October 2007
- Panelist, "Transition from Undergraduate to Graduate Study in Mathematics," MAA/Wisconsin Section Meeting, April 2003

Professional Activity

- Formal Reviewer, *Elementary Statistics: Looking at the Big Picture* Duxbury/Thomson) by Nancy Pfenning, University of Pittsburgh (expected publication 2009)
- Workshop Participant, Bioinformatics Workshop, University of Sioux Falls, Sioux Falls, SD, August 2005

- Participant, Cell Group for Improved Teaching Methods, University of Sioux Falls, Sioux Falls, SD, Fall 2004-Spring 2006
- Participant, Philosophy of Mathematics, University of Sioux Falls, Sioux Falls, SD, Interim 2005
- Consultant, Education Area, University of Sioux Falls, Sioux Falls, SD
  - Submitted assessment report for recognition of Secondary Mathematics Education program at USF. Approved November 2008
  - Attended SPA Training Activity in preparation for Mathematics Education program at USF for SD Board of Regents (Summer 2006)
  - Created template for assessment of Mathematics Education program at USF for SD Board of Regents (Spring 2006)
  - Coordinator of templates for PRAXIS II mathematics content exam under SD Teacher Quality Enhancement Grant, 2005
- University Supervisor for Mathematics student teachers, University of Sioux Falls, Dell Rapids and Roosevelt High Schools, 2006, 2007
- National Fellow, Mathematical Association of America Project NExT (New Experiences in Teaching A New Faculty Preparation Program), 2003

Undergraduate Research

- As a *mentor*, I have directed several student research projects, including projects that led to the following presentations:
  - "Energy Flow in an Ecosystem," Roxie Truax, presented at Mathematics on the Northern Plains, April 2005
  - "Forest Management: Finding the Optimal Sustainable Yield," Emily Dean, presented at Mathematics on the Northern Plains, April 2005
  - "A Critique of the Leslie Matrix: A Method for Determining Population Growth," Jennifer Buckley, presented at Mathematics on the Northern Plains, April 2005

Courses Taught at USF

MAT111	Elementary Algebra
MAT112	College Algebra and Trigonometry
MAT113L	College Algebra (Recitation)
MAT151	Nature of Mathematics
MAT202	Finite Mathematics
MAT204	Calculus I
MAT205	Calculus II
MAT233	Introduction to Statistics
MAT270	Statistics and Mathematical Functions
MAT283	Mathematician's Toolbox
MAT294	Art, Math and Culture in Italy and Greece
MAT300	Numerical Methods
MAT304	Linear Algebra
MAT310	Calculus III
MAT311	Differential Equations
MAT320	Introduction to Real Analysis
MAT490	Senior Seminar
BUS382	Management Control Systems

Other Curriculum Development

- Researched and implemented large-scale changes in the Mathematics major to align with other programs in our peer group, University of Sioux Falls, Sioux Falls, SD, 2005
- Authored and implemented Mathematica computer laboratories for the calculus curriculum, 2007
- Implemented the use of LaTeX typesetting in MAT283 Mathematicians Toolbox that has matriculated into other courses and into MAT490 Senior Seminar

Computer Skills

- Mathematics Software: *Mathematica*, *Maple*
- Statistical Software: SPSS, Excel
- Programming: Basic, Pascal, Fortran, C, C++, HTML
- Graphing Calculators: TI-83/83+, TI-89
- Technical Typesetting: LaTeX, Microsoft Equation Editor

Memberships

- Mathematical Association of America
- Research in Undergraduate Mathematics Special Interest Group of the MAA
- History of Mathematics Special Interest Group of the MAA (charter member)
- Art in Mathematics Special Interest Group of the MAA (charter member)
- Pi Mu Epsilon

Awards and Honors

- Outstanding Faculty of the Year Award, University of Sioux Falls, 2006
- National Mathematical Association of America Project NExT Fellow, 2003
- GAANN Fellow, 1998-2002

Major Professor Date