



**WICHITA STATE
UNIVERSITY**

UNIVERSITY LIBRARIES

Parallelization and optimization of higher-order Finite-difference methods utilizing MPI

Item Type	Thesis
Authors	Irvin, Lance J.
Publisher	Wichita State University
Rights	Copyright Lance J. Irvin, 2010. All rights reserved
Download date	2026-05-20 13:45:11
Link to Item	http://hdl.handle.net/10057/3724

PARALLELIZATION AND OPTIMIZATION OF HIGHER-ORDER
FINITE-DIFFERENCE METHODS UTILIZING MPI

A Thesis by

Lance J. Irvin

Bachelor of Science, Fort Hays State University, 2007

Bachelor of Science, Fort Hays State University, 2008

Submitted to the Department of Aerospace Engineering
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

December 2010

© Copyright 2010 by Lance J. Irvin

All Rights Reserved

PARALLELIZATION AND OPTIMIZATION OF HIGHER-ORDER
FINITE-DIFFERENCE METHODS UTILIZING MPI

The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science with a major in Aerospace Engineering.

Klaus Hoffmann, Committee Chair

L. Scott Miller, Committee Member

Alan Elcrat, Committee Member

DEDICATIONS

To my loving wife.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Klaus Hoffmann for his several years of guidance and support. I appreciate Dr. Hoffmann, Dr. Scott Miller, and Dr. Alan Elcrat for serving on my thesis committee and taking their valuable time in support of my education. I would also like to thank Arshed Ghulam for his help in understanding his serial WENO model.

Thanks are also due to my wife Jessie for her outstanding editing skills and unwavering support. I would also like to thank my friends in the graduate student office, specifically Niloufar Mahmoudnejad for her friendly advice. I would like to thank the High Performance Computing Center at Wichita State, as well as the TeraGrid team for their support of high performance computing resources. I am grateful to Henry Neeman from the University of Oklahoma's Oklahoma Supercomputing Center for Education and Research for several enlightening conversations about parallel computing and his efforts in the Parallel Computing Workshop series.

I would also like to thank Paul Maseberg. Many concepts and techniques are inspired from his previous work. He also provided the figure on memory architecture. Doug Goddard let me use his AMD Phenom 2 six-core to produce results for viscous flow. My family, especially my parents John and Sharon, have always supported me and have been there when I needed them.

This project was funded by the State of Kansas through NIAR/Industry/State (NIS) and by NASA space grant funding.

ABSTRACT

Parallelization of computational models is key to utilizing the widely available computational resources of today to solve problems in ways the computational science community could not yesterday. The main focus of the work presented is in the parallelization of a Direct Numerical Simulation (DNS) finite-difference Computational Fluid Dynamics (CFD) flow solver based on the Weighted Essentially Non-Oscillatory (WENO) scheme. The parallel DNS model was developed by modifying the existing serial model developed by Arshed Ghulam. An Alternating Direction Implicit (ADI) finite-difference heat solver is also parallelized in an effort to develop and refine parallelization techniques. The Message Passing Interface (MPI) was utilized to manage the communication between processes.

In addition to parallelization many improvements were made to the WENO model. The efficiency was increased by addressing memory locality and allocation. The configurability of the model was increased by including multiple options adjustable within the model at runtime. A key-value configuration file reader was added to read in parameters from a file at runtime.

The model was validated against the published serial version's results. The scalability was analyzed with respect to a set domain size and with respect to constant sub-domain size. For a given number of iterations the WENO model proves to scale perfectly with a constant sub-domain size. The results support that the same number of iterations can be performed on a domain of any size without time lost over that of a smaller domain, assuming the computational resources are available to keep the sub-domain per MPI process the same size as in the smaller analysis.

TABLE OF CONTENTS

Chapter		Page
1.	INTRODUCTION	1
1.1	Computational Models of Interest	1
1.2	Parallelization	2
1.2.1	Parallel Computers and Associated Techniques	2
1.2.2	Scalability	5
1.3	Distributed Parallelization Techniques	7
1.3.1	Boundary Sharing	7
1.3.2	Pipelining	10
1.3.3	MPI Functions	12
1.3.4	Data Output	16
1.4	Literature Review	17
2.	METHODOLOGY	20
2.1	WENO Model Structure	20
2.2	Initialization	23
2.2.1	Topology	23
2.2.2	Grid Division	26
2.2.3	Initial Conditions	26
2.2.4	General Computational Domain	27
2.3	Parallel Modifications	27
2.3.1	Minimizing Code Modification	27
2.3.2	Boundary Update	28
2.3.3	Physical Boundary Conditions	31
2.3.4	Reduction	31
2.3.5	Domain Reconstruction in Data Files	31
2.3.6	Generality of Parallel Functions	32
2.3.7	Communication Visualization	32
3.	IMPROVEMENTS	36
3.1	Efficiency	36
3.1.1	Loop Efficiency	36
3.1.2	Memory Locality	37
3.1.3	Dynamic Allocation	39
3.2	Configuration	39
3.2.1	Configuration Branches	39
3.2.2	Configuration File	40

TABLE OF CONTENTS (continued)

Chapter	Page
4. RESULTS	43
4.1 Validation	43
4.2 Scalability	56
4.3 Conclusions	63
4.4 Recomendations	64
REFERENCES	65
APPENDIX	68
A. MPI FUNCTIONS	69

LIST OF FIGURES

Figure		Page
1.1	Memory Architecture	3
1.2	ADI Stencil	8
1.3	WENO Stencil	9
1.4	Local Grid Representation	10
1.5	Pipelining Stencil	12
1.6	Example of Separate Send and Receive MPI Functions	14
1.7	Example of Send and Receive MPI Function	15
2.1	Flow Chart: Model Overview	21
2.2	Flow Chart: Runge Kutta	22
2.3	Grid Division Topology	24
2.4	Example of MPI Topology Functions	25
2.5	Example of Message Passing Stage	30
2.6	Jumpshot Before Communication Revision	33
2.7	Jumpshot Communication Profile	34
2.8	Communication Time Histogram Comparison	35
3.1	Example of Configuration File	41
3.2	Excerpt of Configuration File Reader	42
4.1	Serial Shock Turbulence Density - JS	44
4.2	Parallel Shock Turbulence Density - JS	44

LIST OF FIGURES (continued)

Figure	Page
4.3	Serial Shock Turbulence Vorticity - JS 45
4.4	Parallel Shock Turbulence Vorticity - JS 45
4.5	Serial Shock Turbulence Density - MA 46
4.6	Parallel Shock Turbulence Density - MA 46
4.7	Serial Shock Turbulence Vorticity - MA 47
4.8	Parallel Shock Turbulence Vorticity - MA 47
4.9	Initial Shock for Inviscid Flow Over Cylinder 48
4.10	Serial Inviscid Shock Over Cylinder - JS: $t=1.24$ 49
4.11	Parallel Inviscid Shock Over Cylinder - JS: $t=1.24$ 49
4.12	Serial Inviscid Shock Over Cylinder - JS: $t=2.5$ 50
4.13	Parallel Inviscid Shock Over Cylinder - JS: $t=2.5$ 50
4.14	Serial Inviscid Shock Over Cylinder - MA: $t=1.24$ 51
4.15	Parallel Inviscid Shock Over Cylinder - MA: $t=1.24$ 51
4.16	Serial Inviscid Shock Over Cylinder - MA: $t=2.5$ 52
4.17	Parallel Inviscid Shock Over Cylinder - MA: $t=2.5$ 52
4.18	Serial Viscous High Speed Flow Over Cylinder 54
4.19	Parallel Viscous High Speed Flow Over Cylinder 54
4.20	Viscous High Speed Flow Over Cylinder - High Resolution 55
4.21	Viscous High Speed Flow Over Cylinder - High Resolution, Full Domain..... 55

LIST OF FIGURES (continued)

Figure		Page
4.22	Notus WENO Strong Scaling	56
4.23	Notus WENO Weak Scaling	57
4.24	Network and Local File Output Comparison	58
4.25	Lonestar WENO Strong Scaling	58
4.26	Lonestar WENO Weak Scaling Normalized	59
4.27	Lonestar WENO Weak Scaling	60
4.28	ADI Weak Scaling	61
4.29	ADI Weak Scaling High Grid Density	61
A.1	MPI Function: MPI_INIT	70
A.2	MPI Function: MPI_FINALIZE	70
A.3	MPI Function: MPI_SEND	71
A.4	MPI & Fortran Data Type Comparison	71
A.5	MPI Function: MPI_RECV	72
A.6	MPI Function: MPI_SENDRECV	73
A.7	MPI Function: MPI_DIMS_CREATE	74
A.8	MPI Function: MPI_CART_CREATE	75
A.9	MPI Function: MPI_CART_GET	76
A.10	MPI Function: MPI_CART_RANK	77
A.11	MPI Function: MPI_COMM_RANK	77

LIST OF FIGURES (continued)

Figure		Page
A.12	MPI Function: MPI_REDUCE	78
A.13	MPI Function: MPI_BCAST	78
A.14	MPI Function: MPI_GATHER	79

CHAPTER 1

INTRODUCTION

Accepting the risk of sounding like all technical papers relating to High Performance Computing (HPC), the steady increase of computational resources demands changes to existing computer models to achieve resource utilization and allows for the creation of computational models deemed unusable before because of the available resources. Higher-order finite-difference models are becoming increasingly popular due to two major factors: HPC resource availability and the ability to distribute the memory requirements via distributed grid parallelization. The most common way to achieve this parallelization is implementation of the Message Passing Interface (MPI).

1.1 Computational Models of Interest

Although the topics presented can be applied to a wide range of finite-difference models, the focus model is the Weighted Essentially Non-Oscillatory (WENO) scheme for solving the Navier-Stokes equations directly [1]. The WENO model is designed to run at high enough resolution to model turbulence without artificial addition produced by a separate turbulence model. Such Computational Fluid Dynamics (CFD) models are known as Direct Numerical Simulations (DNS). The base WENO model is too dispersive to generate such results, but modification at critical points and slope modification at discontinuities is employed to decrease the dispersion. The WENO model utilizes fifth order finite-differences. It is an explicit two-dimensional solver and uses a Runge Kutta time differentiation scheme. The time step is dynamically chosen to satisfy the CFL condition:

$$\Delta t = \frac{CFL}{\frac{\max(|u_x|+c)_\Omega}{\Delta x} + \frac{\max(|u_y|+c)_\Omega}{\Delta y}} \quad (1.1)$$

Problems of interest require a vast number of computations and are slow to run on a serial

processor. Several of the test cases performed with the serial model each took a week to run. In addition, for many problems the model requires large computational domains that are unlikely to fit on a single computer—especially when the model is extended to three dimensions. Finite-difference models are prime candidates for parallelization and explicit models more so than implicit models. Parallelization of a model such as the WENO is key in studying practical problems.

The Alternating Direction Implicit (ADI) heat model was parallelized as a test bed for the parallelization techniques utilized in the WENO model. The WENO model is an excellent choice for developing parallelization techniques. It contains both implicit and explicit stages, providing the need for both boundary sharing and a way of solving a system of equations across the entire domain.

1.2 Parallelization

There are two main types of parallelization: shared and distributed. In shared memory parallel computational work is divided up into threads which can be performed independently by available processing cores. The nomenclature for these parallelization methods spawns from the difference in how the memory and in turn the computational mesh is managed. In distributed memory parallelization the computational mesh is divided and distributed across multiple computers. Computations are performed on each piece of the mesh and communication between computers is utilized when necessary.

The primary advantage of the distributed model is that problem size is no longer limited on the amount of shared memory available. Larger sizes can be achieved by seeking larger HPC clusters.

1.2.1 Parallel Computers and Associated Techniques

There are two prominent types of parallel computer structures: Multiple Instruction, Multiple-Data stream (MIMD) and Single Instruction, Multiple-Data stream (SIMD) [2]. Both shared

(OpenMP) and distributed (MPI) memory systems (and corresponding parallelization methods) fall under MIMD. Shared and distributed memory systems are depicted in figure . Graphics Processing Units (GPU) utilizing Compute Unified Device Architecture (CUDA) and vector computers are examples of SIMD structures. GPU's are great at performing the same set of instructions on multiple data.

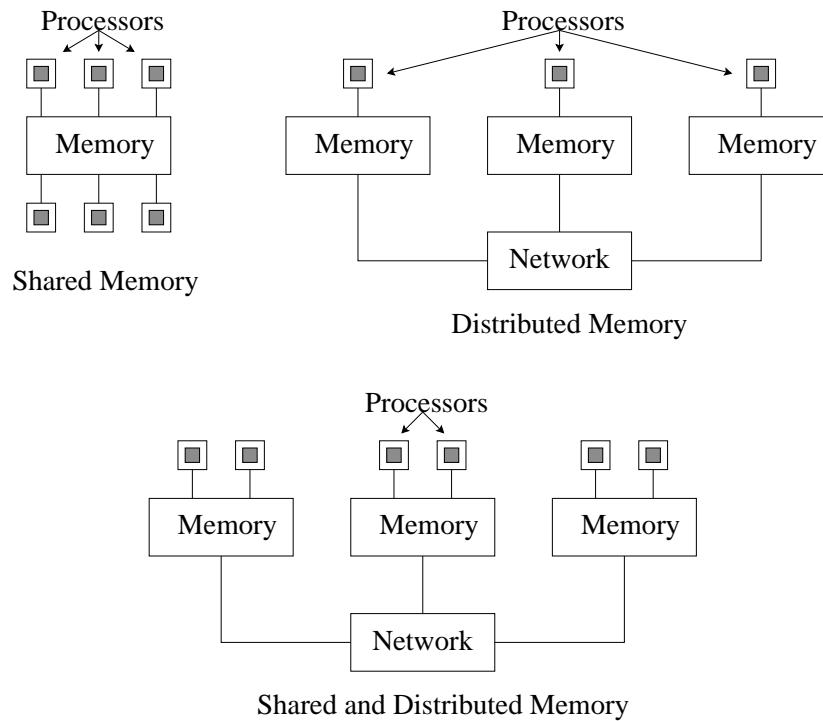


Figure 1.1 Memory Architecture

Shared

On shared memory parallel computers, multiple processing cores have access to the same memory. Parallel code can be written to take advantage of this. In regards to computational science models, the most common way of achieving this is implementation of OpenMP. OpenMP allows a relatively easy way to achieve parallelism within a model. OpenMP can almost automatically achieve loop parallelization by forking threads and dividing the loop into chunks for each thread or processing core to work on. The instructions for this are provided by precompiler

directives. Another advantage of OpenMP is since the memory is shared there are fewer problems with implicit calculations [3].

Large shared memory systems are not as prominent as distributed memory computer clusters. It is true that the amount of total memory on shared memory systems is steadily increasing, but in general shared memory storage of the order of magnitude readily available on distributed memory clusters does not exist. This puts a restriction on the size of problems that can be solved with a shared memory parallel model.

Distributed

The most common type of high performance computing resource available is the distributed memory cluster. This is made up of multiple individual computers tied together over a high speed network. To take full advantage of this type of computational resource, the computational mesh (and therefore the memory requirements) is divided up and spanned across multiple processes. The Message Passing Interface (MPI) is an agreed upon standard for implementing the communication between processes. Many libraries exist that are based on this standard.

Hybrid

With the onset of multi-core processors, hybrid computational models can be advantageous. Many large distributed machines have multiple cores within a cluster node. Each core can be treated as a separate MPI process. However, the multiple MPI processes will all have to share other resources, such as the network card. Hybrid models employ MPI to divide the domain among separate processes, and utilize OpenMP to divide the work on each sub-domain among the available cores.

1.2.2 Scalability

Primarily, there are only two reasons to parallelize a model: to improve runtime and to provide scaling. Speedup is a measure of how much faster a model runs due to parallelization. Similarly, scaling refers to the increase in the ability to analyze larger data sets due to parallelization.

Speedup

Speedup is a measure of the time gain due to parallelization defined as the ratio of single processor runtime to multiple processor runtime [4]:

$$Speedup(n) = \frac{T_1}{T_n} \quad (1.2)$$

In an ideal situation $Speedup(n) = n$.

Speedup Efficiency

A way to relate speedup to ideal speedup is speedup efficiency [2].

$$E(n) = \frac{S(n)}{n} \quad (1.3)$$

An ideal speedup would correspond to $E = 1$.

Amdahl's Law

Speedup is limited by the amount of work in a model that must be done in serial.

$$Speedup(n) = \frac{T_s + T_{p1}}{T_s + T_{pn}} \quad (1.4)$$

where T_s is the time spent on any portion of the program that cannot be parallelized, T_{p1} is the time spent on the parts of the program that can be parallelized utilizing only one process, and T_{pn} is the time on the parallel portions utilizing n processes. This is a mathematical way

of stating that speedup is very limited. For example, if twenty percent of the program is spent doing work that cannot be parallelized, then the maximum speedup of that program is five. This would be the case that the time spent on parallel tasks as been reduced to zero. Since most programs contain sequential portions, Amdahl's law gives a grim outlook on parallelization work. Gene Amdahl argued in 1967 that efforts in "achieving high parallel processing rates is wasted unless accompanied by achievements in sequential processing rates of very nearly the same magnitude" [5].

Scaling

A model is considered scalable if the speedup is close to the number of processors utilized. A Monte Carlo method is scalable in this way. There are sufficient independent calculations available such that no matter how many processors are chosen, close to ideal speedup is achieved. This type of scaling is commonly referred to as strong scaling.

Many models may exhibit reasonable speedup for some problem sizes, but do not exhibit strong scaling. For example in grid division problems, as the number of processes is increased the size of grid on each process decreases. At some point, the amount of computation becomes small compared to the communication overhead [6]. These problems combined with Amdahl's argument lead to a different sort of discussion about parallelization. Instead of focusing on speedup for a certain problem size, the interest is in scaling to larger problems. This sort of scalability, often referred to as weak scaling, is analyzed by keeping the amount of work (calculations or grid size) constant for each of n processes and comparing the times. In an ideal case if the ratio of work to amount of processes is kept the same, then the execution time also remains constant. John L. Gustafson supported looking at scalability in this manner. He stated that generally "we have found that it is the parallel or vector part of a program that scales with the problem size" [7].

Finite-difference models such as the WENO are considered weak scaling models. In practice, this is very useful. If a model scales well in this manner, then the amount of time can remain the

same for a problem with a much denser or larger grid. In general, scalability is defined by weak scaling [4].

1.3 Distributed Parallelization Techniques

In distributed parallelization the domain of the problems is divided into multiple sub-domains. This technique is also known as data decomposition or domain decomposition [8]. Information from other domains is then shared by passing messages. In explicit finite-difference routines the information along the sub-domain boundaries must be updated. Since the communication overhead will be proportional to the length of the boundaries, the choice in domain division will affect the overhead [9]. Efficient distributed parallelization is achieved by minimizing the communication and maintaining load balancing [10]. Load balancing refers to providing equal amount of work to all of the processes. Sometimes minimizing communication means doing redundant calculations among the processors [11]. The more efficient choice is problem dependent. [12]

1.3.1 Boundary Sharing

In finite-difference models the algorithms used for calculating updated values during iterative steps utilize the neighboring points in the grid. This can be represented graphically as a stencil. The number of neighboring points used depends on the order of the finite-difference representations utilized within the algorithms. For a second-order finite-difference model such as the ADI only one point in each direction is required. Figure 1.2 shows the stencil representation for this in two dimensions. The black point represents the grid point in which values are being updated and the points around it are used in those calculations.

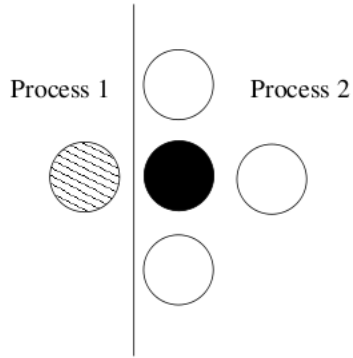


Figure 1.2 ADI Stencil

In distributed grid parallelization special considerations must be made along the boundaries of the processes containing interior portions of the total grid. The line in Figure 1.2 represents the boundary between process 1 and process 2. For the calculations to be performed on the black point, information is required from points contained on process 1. Process 2 could request this information when it needs it during calculation; however, it is much faster computationally to update all of this information (every point along a boundary) outside of the calculation loops. This is accomplished through the use of ghost points. Extra points are allocated along the boundaries to store the information required from processes with neighboring grids. Message passing stages are introduced to update the information on the ghost points from the processes containing adjacent grids in the MPI topology. Every ghost point corresponds to a real grid point on another process and the message passing stages update this information as needed between calculation steps during main iterations.

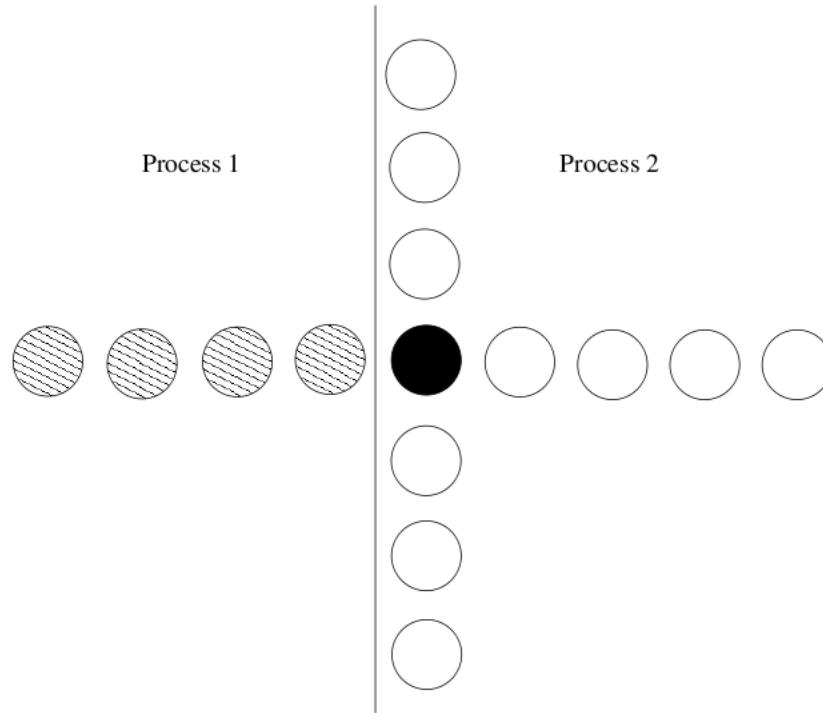


Figure 1.3 WENO Stencil

The WENO model utilizes fifth-order finite-differences and in turn has a much larger stencil. Four points in every direction are required to perform calculations on a specific grid point. Figure 1.3 depicts the stencil for the WENO model. In this case four arrays of ghost points are required on each local boundary shared with another process. Much more information is passed during message passing stages than with a lower order model.

Local Grid

Figure 1.4 shows the grid that an interior process possesses. The intersections of the solid lines represent real points within its computational grid and the intersections of dashed lines represent the ghost points. The process is only responsible for calculations performed on the real grid points, but requires the other points to perform those calculations.

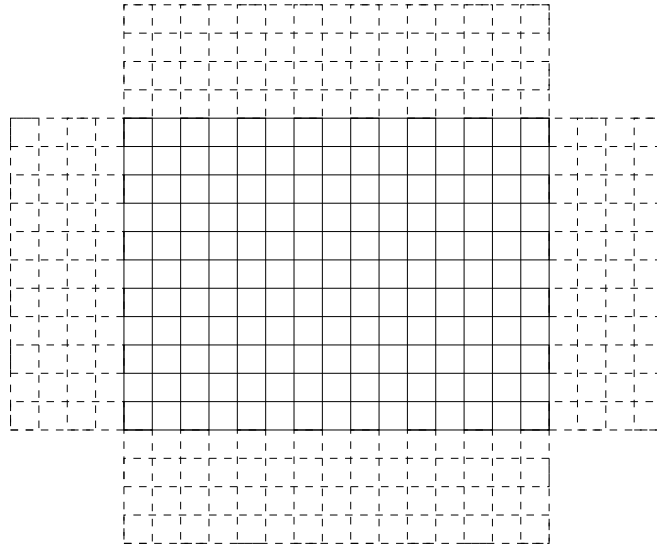


Figure 1.4 Local Grid Representation

1.3.2 Pipelining

Some algorithms such as those in implicit finite-difference models involve solving systems of equations. Many times an efficient recursive algorithm such as the Thomas Algorithm [13] can be used for this. The Thomas Algorithm solves a tridiagonal system of equations like that produced in the ADI model [14]. For implicit algorithms such as Thomas Algorithm, the recursion within the algorithm prevents processes from working independently. There are several ways to still achieve some level of parallelization and one of these is called pipelining. In the Thomas Algorithm a data array (one grid row or column) must be evaluated in a sequential order: starting at the first element and doing calculations in order to the last element. This particular algorithm then marches backwards through the array for its final calculation. For now just consider the first part. Since the preceding element must be considered first there is no way to parallelize just one application of the Thomas Algorithm. However, in most practical applications many instances

of the algorithm are employed and can be solved simultaneously. For example, in each implicit step of the two-dimensional ADI the Thomas Algorithm is used in one direction for every point in the second direction. That is, in a 100x200 sized grid, the Thomas Algorithm is used 100 times during one of the implicit steps and 200 times during the other implicit step. Pipelining is a way to organize the work done so that all the processors can be working at least most of the time. Essentially, pipelining is a way of dividing the work of many serial problems to achieve one parallel problem.

Consider the division of a two-dimensional grid in one direction. For simplicity's sake the grid will only be divided up in one direction, although it is easy to extend this logic to a more flexible division of the grid. If there are only five processors, then each process has one-fifth of the grid in the y direction. Figure 1.5 shows this division. Using a recursive algorithm in the y direction is difficult because each process is waiting on the process before it to proceed. However, if the first process works on a chunk of data and then passes the info onto the next process needed for it to start, then some parallelization can be accomplished. Study figure 1.5. Each data set consists of a number of arrays—say ten. The numbers denote the iteration number in this pipelining scheme. Hence in the first iteration, process 0 is working on data set 1. At the end of this iteration it passes the data needed for process 1 to work on that data set. During iteration 2, process 0 works on data set 2 and process 1 works on data set 1. By the fifth iteration all processes are working simultaneously. As previously mentioned, the Thomas Algorithm has two parts. First it marches recursively through the arrays in the forward direction, storing some temporary values, and then it marches the other direction through the arrays. By iteration 6 the second part of the Thomas Algorithm can be done and the temporary arrays can be destroyed or used for a different data set. In this example process 4 doesn't do any work until the fifth iteration, but during most of the iterations all the processes have an equal amount of work to do. Pipelining has proven to be an effective and fairly efficient way to parallelize recursive algorithms.

Data Set 1	Data Set 2	Data Set 3	Data Set 4	Data Set 5	Data Set 6	Data Set 7	Data Set 8	Data Set 9	Data Set 10	
↓ 1	↓ 2	↓ 3	↓ 4	↓ 5	↓ 6	↓ 7	↓ 8	↓ 9	↓ 10	Process 0
↑ 10	↑ 11	↑ 12	↑ 13	↑ 14	↑ 15	↑ 16	↑ 17	↑ 18	↑ 19	
↓ 2	↓ 3	↓ 4	↓ 5	↓ 6	↓ 7	↓ 8	↓ 9	↓ 10	↓ 11	Process 1
↑ 9	↑ 10	↑ 11	↑ 12	↑ 13	↑ 14	↑ 15	↑ 16	↑ 17	↑ 18	
↓ 3	↓ 4	↓ 5	↓ 6	↓ 7	↓ 8	↓ 9	↓ 10	↓ 11	↓ 12	Process 2
↑ 8	↑ 9	↑ 10	↑ 11	↑ 12	↑ 13	↑ 14	↑ 15	↑ 16	↑ 17	
↓ 4	↓ 5	↓ 6	↓ 7	↓ 8	↓ 9	↓ 10	↓ 11	↓ 12	↓ 13	Process 3
↑ 7	↑ 8	↑ 9	↑ 10	↑ 11	↑ 12	↑ 13	↑ 14	↑ 15	↑ 16	
↓ 5	↓ 6	↓ 7	↓ 8	↓ 9	↓ 10	↓ 11	↓ 12	↓ 13	↓ 14	Process 4
↑ 6	↑ 7	↑ 8	↑ 9	↑ 10	↑ 11	↑ 12	↑ 13	↑ 14	↑ 15	

Figure 1.5 Pipelining Stencil

1.3.3 MPI Functions

For successful domain decomposition parallelization, MPI is utilized to manage the communication between sub-domains. There are a few MPI functions that are necessary for a distributed parallel model. MPI send and receive functions allow data to be passed between processes [15]. Other functions can be useful such as MPI topology functions. A listing of common MPI functions also appears in Appendix A.

MPI Initialize and Finalize

The function `MPI_INIT()` must be called on every MPI process before almost any other MPI functions are used. There are a few exceptions such as `MPI_INITIALIZED()` which checks to see if `MPI_INIT()` has been called or `MPI_GET_VERSION()` which checks what MPI version is installed. Any calls such as `MPI_SEND()` before `MPI_INIT()` will return an error and multiple calls of `MPI_INIT` will also return an error [16].

MPI Topology

MPI has functions to help the user define the topology of sub-domains across the entire domain. `MPI_CART_CREATE()` produces a Cartesian topology based on the desired dimensions and the number of MPI processes. `MPI_CART_GET()` returns information about that Cartesian topology including the processes topology coordinate that called the function. `MPI_CART_RANK()` returns the rank of a MPI process at a given topology coordinate and is useful in locating an MPI process needed for communication by another MPI process.

Send and Receive

The plain `MPI_SEND()` and `MPI_RECV()` functions are considered blocking. The threads that initiate them wait for completion to go on. Figure 1.6 shows an example usage of separate blocking `SEND` and `RECV` functions. The boundary information is loaded into a buffer pass. Then the process calls `MPI_SEND` to pass the boundary information to the correct process.

```

DO j=1, yN+2
  Pass(j)=Temp(N+1,j)
ENDDO
Call MPI_SEND(Pass,yN+2,MPI_DOUBLE_PRECISION,right,tag, &
  MPI_COMM_WORLD,ierr)
Call MPI_RECV(Catch,yN+2,MPI_DOUBLE_PRECISION,left,tag, &
  MPI_COMM_WORLD,MPI_STATUS_IGNORE,ierr)
if(coords(2).NE.0) then
DO j=1, yN+2
  TEMP(1,j)=Catch(j)
ENDDO
endif

```

Figure 1.6 Example of Separate Send and Receive MPI Functions

Figure 1.6 is an example from a right wave. So it is passing to the process right of it in the MPI topology and receiving from the process left of it with respect to the grid topology. The MPI_SEND() function takes arguments for send buffer, size of buffer, type of data, destination process, a reference tag, the communicator, and error status, respectively. The communicator can be MPI_COMM_WORLD for all sends and receives unless multiple communicators or communication name-spaces are needed within the model. The process then calls a MPI_RECV() function. The MPI_RECV() function takes arguments for a receive buffer, size of buffer, data type, a reference tag, a communicator, a status, and error status. The data is then loaded out of catch (the receive buffer) into the proper boundary location.

This sort of separate blocking send and receive functions can cause deadlock issues. If two processes are trying to send information to each other they can get locked waiting for the sends to finish before they call their receives. The model will be locked up or “deadlocked.” The wave technique of sending information one direction at a time takes care of this issue for the most part. But there are cases in which problems can still occur. For example, if periodic boundaries are chosen and there are only two processes in one direction with respect to the grid topology, then the two processes will be trying to send and receive from each other. Because of the periodicity, the process to the right of a process is also the process to the left of a process.

The MPI_SENDRECV() function takes care of this issue. It can be used to replace consecutive MPI_SEND() and MPI_RECV() calls. Figure 1.7 shows an example of MPI_SENDRECV().

```

DO L=1,numghost
  DO J=1,J_arraylen
    pass(J+E_base+(L-1)*J_arraylen)=E(IDMX+1-L,JDMN-1+J)
    pass(J+DNbase+(L-1)*J_arraylen)=DN(IDMX+1-L,JDMN-1+J)
    pass(J+UXbase+(L-1)*J_arraylen)=UX(IDMX+1-L,JDMN-1+J)
    pass(J+UYbase+(L-1)*J_arraylen)=UY(IDMX+1-L,JDMN-1+J)
    pass(J+MXbase+(L-1)*J_arraylen)=MX(IDMX+1-L,JDMN-1+J)
    pass(J+MYbase+(L-1)*J_arraylen)=MY(IDMX+1-L,JDMN-1+J)
    pass(J+P_base+(L-1)*J_arraylen)=P(IDMX+1-L,JDMN-1+J)
  ENDDO
ENDDO

! send info right

Call MPI_SENDRECV(Pass,messageSize,MPI_DOUBLE_PRECISION,right,
$ tag,catch,messageSize,MPI_DOUBLE_PRECISION,left,
$ tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE,ierr)

if((xtcoord.NE.0).or.(xperiodic)) then
  DO L=1,numghost
    DO J=1,J_arraylen
      E(IDMN-L,JDMN-1+J)=catch(J+E_base+(L-1)*J_arraylen)
      DN(IDMN-L,JDMN-1+J)=catch(J+DNbase+(L-1)*J_arraylen)
      UX(IDMN-L,JDMN-1+J)=catch(J+UXbase+(L-1)*J_arraylen)
      UY(IDMN-L,JDMN-1+J)=catch(J+UYbase+(L-1)*J_arraylen)
      MX(IDMN-L,JDMN-1+J)=catch(J+MXbase+(L-1)*J_arraylen)
      MY(IDMN-L,JDMN-1+J)=catch(J+MYbase+(L-1)*J_arraylen)
      P(IDMN-L,JDMN-1+J)=catch(J+P_base+(L-1)*J_arraylen)
    ENDDO
  ENDDO
endif

```

Figure 1.7 Example of Send and Receive MPI Function

MPI_SENDRECV() is the same as launching separate threads—one for the send and one for the receive. This circumvents the deadlock issues mentioned. Figure 1.7 also depicts how multiple boundary values can be loaded into one buffer and then loaded back into the proper locations. It is advantageous to do this instead of having multiple MPI_SENDRECV() calls because of the extra communication overhead of initiating a data transfer.

In some cases there can be performance gains by using non-blocking sends and receives. In this case the send (or receive) call is initiated and the thread does not wait for completion. A

separate call is made later to verify completion. The implementation can be tricky and in the case of a distributed grid finite-difference model, the next work a process can do is usually dependent on the data to be received, so blocking calls make sense.

Reductions

There are operations that occur in models that cannot be done on portions of the grid independently. Many of these operations are common enough for MPI implementations to have available functions to handle them. Reductions are a set of these instances. `MPI_REDUCE()` is an MPI function that will allow operations such as *max*, *min*, *sum*, and *product* to be calculated across the entire grid. The function call is equivalent to performing the chosen operation on each local grid, sharing the result with one or all the MPI processes and then completing the operation for one final result.

1.3.4 Data Output

Data output techniques affect the performance of a distributed finite-difference parallel model. Techniques chosen and their effects on performance are dependent on cluster design and configuration. Data from each process can be output to a separate file. In this case each process outputs its grid and the resultant data on that grid in its own file. The filenames are chosen (or header information within the file) in such a way that which part of the total grid the data represents can be determined. These files could be analyzed separately or a post-processing script could be used to combine the files back together [17]. The advantage of this method is that if local scratch space is available and accessible by the user, then the data output can remain local to each process and not be dependent on the network of the cluster. The performance gains are obviously network speed and interconnect type dependent. The disadvantage is that the user may not have access to this temporary space. If the user is trying to check-point the code in case of failure, there may not be any way for the user to get at this data if the job fails (or runs out

of allocation time).

If the individual processes already output to a global file system, then the model should include an option for all the data to be output to one file (or set of files). The MPI I/O libraries allow a way for each process to allocate spaces in the same file. When the process calls that file for read or write operations, the allocation map is used to determine the proper location. This provides a nice way for the parallelization of a model to be somewhat transparent to the user. The data can be output in a way that results in the same data files a serial model would have produced. Clusters with parallel file systems and high speed interconnects will handle parallel output in this method extremely well.

Another option to consider is a binary data file format such as the Hierarchical Data Format (HDF). These formats save space over a text format. HDF also has built-in support for parallel file output that utilizes MPI I/O libraries. The other advantage of HDF is that it will have long-term support over any number of platform types, so the data will always be accessible.

1.4 Literature Review

Many parallelization techniques have been employed to finite-difference models. Zhang, Shao, and Wang [3] parallelized an ADI Finite-Difference Time Domain (FDTD) algorithm with OpenMP. The authors argued that with the cost of data exchange involved in solving the system of equations in the implicit portion of an ADI method, domain decomposition parallelism was not efficient. OpenMP proves to provide very efficient parallelization, but the size of the domain is limited to what can fit on available shared memory machines. Stefanski and Drysdale [14] also parallelized an ADI-FDTD algorithm, but used domain decomposition. The authors divided their grid in such a way as to minimize the division across implicit directions and showed a 60-80% scaling efficiency factor. Neither [3] or [14] employed pipelining for solving the system of equations generated by an implicit algorithm. The WENO model is not based on implicit finite-differences and it is desired to be able to run it for large problem sizes. Hence, domain decomposition is

used.

Wang et al. [17] implemented a parallel FDTD method for electromagnetic analysis in support of the design of MRI systems. They used a load balancing algorithm to optimize the parallel solution. In the parallel WENO model the number of calculations is relatively constant across the domain. Hence, dividing the domain evenly ensures load balancing.

Ray, Kondayya, and Menon developed an FDTD code in parallel for nuclear electromagnetic field simulation [12]. In the authors' model the domain is divided into strips in either of the two directions, and processes determine the rank of other processes containing adjacent grids as variables *above* and *below*. The same concept is employed in the parallel WENO model, but the domain can be divided in two directions and the processes with adjacent grids are stored as *up*, *down*, *left*, and *right*.

Araujo et al. [18] employed a parallel FDTD method for the analysis of antennas. They too divided the domain into sub-domains along only one direction. The results of this study showed increased computational times as the number of processes increased past a certain point, reinforcing the concept that parallel finite-difference models should be evaluated based on scaling to larger problem sizes versus speed up comparisons. Omar et al. [6] analyzed the efficiency of a parallel FDTD method on a local area network utilizing MPI and concluded that efficiency reaches a limit on small domains.

De Angeli et al. [10] used parallel distributed computing to solve the Navier-Stokes equations utilizing finite-differences. The authors employed the implicit Successive Over-Relaxation (SOR) method to solve for pressure values and then used first order finite-differences in time to solve for velocities. Although the SOR is an implicit method, it is also a relaxation problem. The model solves the systems of equations with respect to the local sub-domains only and not across the entire physical domain and then updates pressure values at the boundaries every few iterations. They can get by with this since at any given iteration the pressure values calculated in the SOR method are not physically representative. The values must reach convergence before they are

used to calculate velocities in the FDTD scheme.

Anders, Magi, and Abraham [9] employed domain decomposition parallelism utilizing MPI for large-eddy simulation of a multi-component gas jet with varying density in three dimensions. In their analysis, they investigated the effect of breaking up the domain into different numbers of sub-domains in different directions on the computational time. They found that minimizing the surface area of shared boundaries is most efficient. For relatively square domains, this is achieved by keeping the number of divisions in each direction as close as possible. For example, $4 \times 4 \times 4$ is more efficient than $2 \times 4 \times 8$ or $1 \times 1 \times 64$. However, for domains with large aspect ratios, it may be advantageous to divide along the direction with the most grid points (keeping the sub-domains as square as possible). The authors in [9] also mention that DNS solvers may provide the best data for certain applications, but argue that applying DNS to practical problems is “unrealistic with the computational capabilities today.” However, computational resources for use of an MPI parallel program have vastly increased in the last three years—in both size and availability.

Sermeus, Laurendeau, and Parpia [11] parallelized a multi-block structured Navier-Stokes solver called the FANCS (Full Aircraft Navier-Stokes Code). Not only did the authors employ parallelism utilizing MPI and hybrid MPI-OpenMP, but they also performed in-depth optimization of the model. They eliminated invariant calculations within loops, restructured arrays and loops to take advantage of data locality, and stored values used frequently to decrease calculation.

CHAPTER 2

METHODOLOGY

Although the focus of this chapter is on parallelization—specifically the parallelization of the WENO model—some of the model components will be discussed to help understand where parallelization concerns are present within the model. The parallelization implementation has components in the initialization, the solver, and data file creation. These techniques are not necessarily specific to the WENO model and are used in a variety of models.

2.1 WENO Model Structure

A simplified view of how the WENO model works is shown in figure 2.1. The model starts by setting up the initial conditions, boundary conditions, general grid information, and variables needed for calculation.

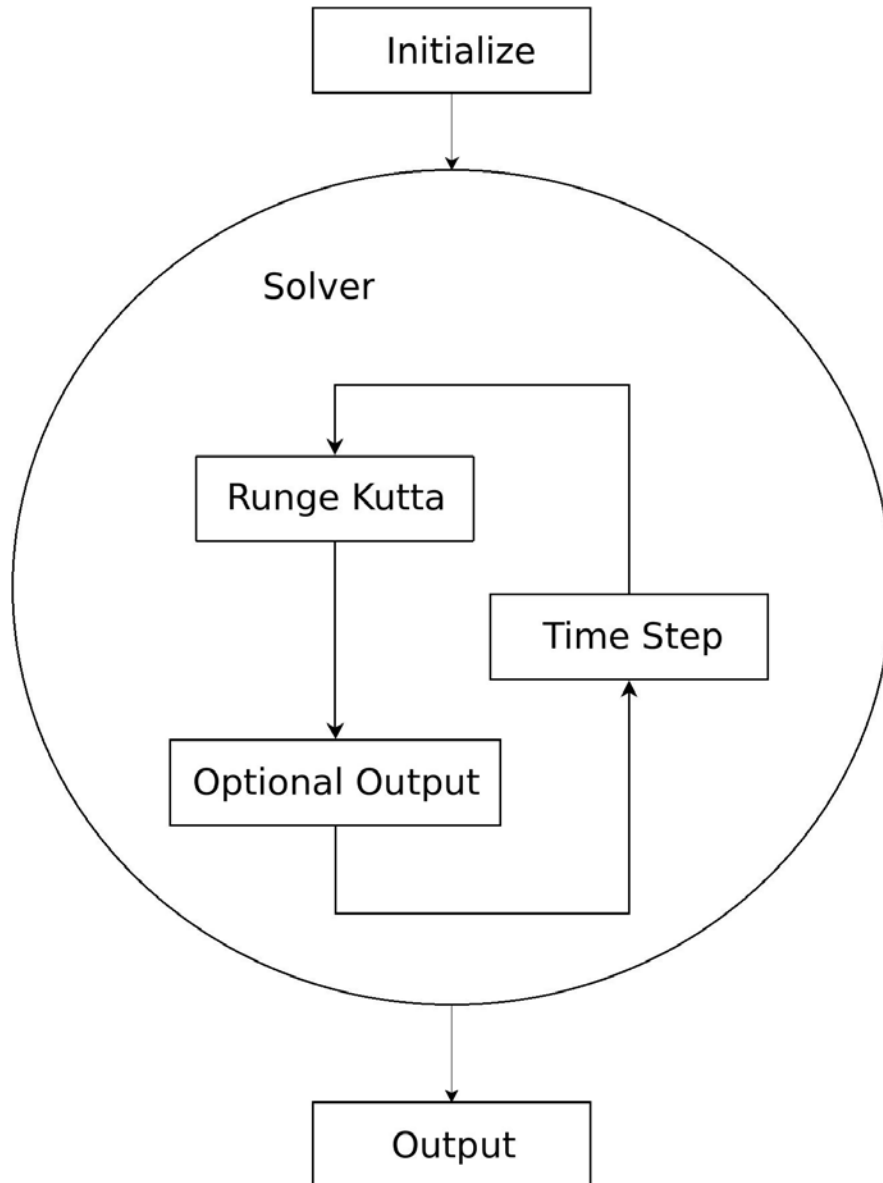


Figure 2.1 Flow Chart: Model Overview

After initialization, the model enters the solver. Within the solver, iteration over the Runge Kutta time differentiation methods occurs. Figure 2.2 depicts what happens within the Runge Kutta routine. The core algorithms are employed by a function called DFDT which is called at the beginning of each of the three stages of the Runge Kutta time step integration. This is where the fifth-order finite-differences are employed. After careful analysis of this routine, it is determined

that the DFDT routine can be performed independently on each process without communication stages. During the solver iterations, the results can be outputted at a chosen iteration interval. The time step is based on satisfying a chosen Courant number and is dynamically updated each iteration based on maximum values of velocity within the grid. Program termination happens after outputting the final results to file.

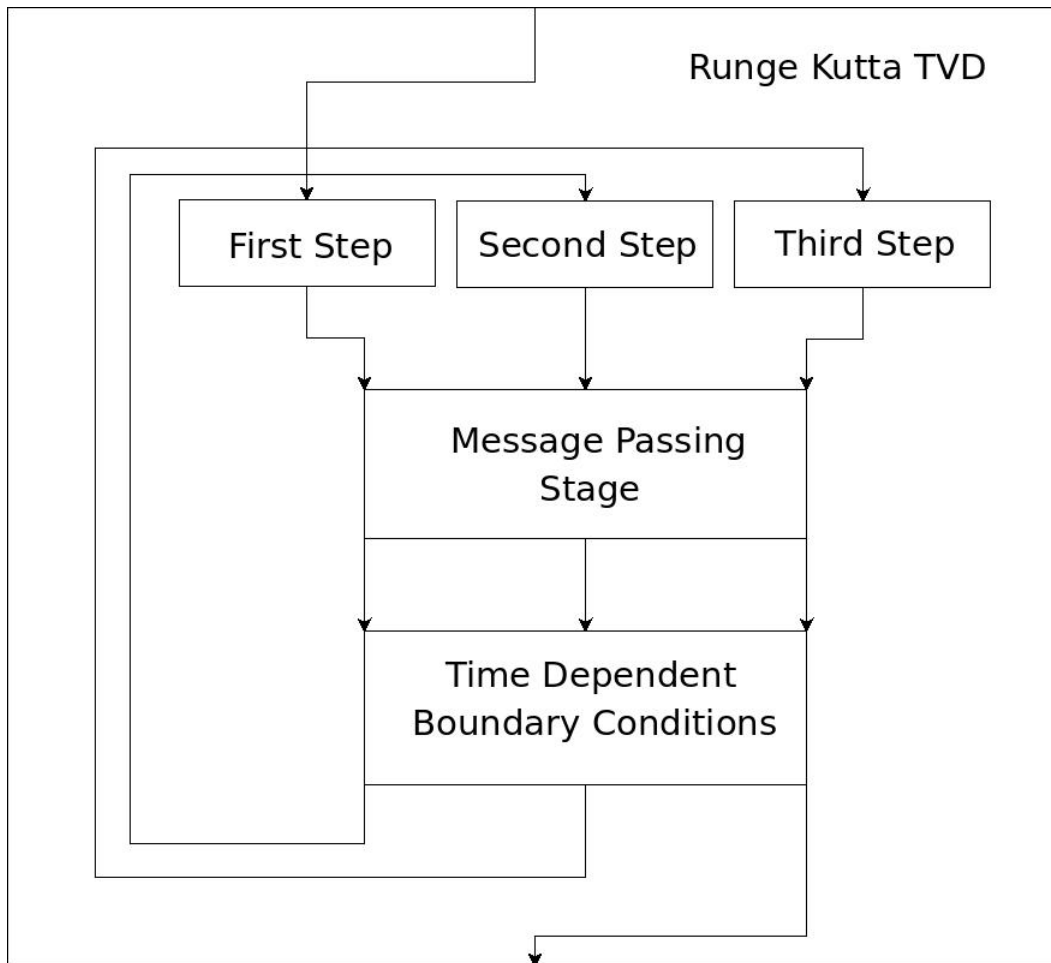


Figure 2.2 Flow Chart: Runge Kutta

In many ways the serial model remains unchanged to run in parallel. There are several sections that must be modified or added to. The changes and the complexity of the parallelization are dependent on the structure of the serial model. In the case of an explicit finite-difference model,

calculations can be done on separate portions of the grid, relatively independently. Message passing stages are needed to update values on the local boundaries between every dependent set of calculations.

2.2 Initialization

During initialization the grid must be divided and variables defining the topology are initialized. This information is used during run time for logic in message passing stages as well as logic to determine which processes must evaluate boundary conditions on physical boundaries. Some processes will have a grid that is against a physical boundary, whereas other processes may contain grids that are completely interior in the process topology.

2.2.1 Topology

The grid must be divided in a logical manner and each process must have information providing the location of its local grid with respect to the processes that contain adjacent portions of the total grid. Throughout discussion local grid or sub-domain will refer to each process's portion of the grid and total grid or domain will refer to the entire problem space. The grid topology shown in figure 2.3 is an example of what is used in the WENO model parallelization.

8 (0,2)	9 (1,2)	10 (2,2)	11 (3,2)
4 (0,1)	5 (1,1)	6 (2,1)	7 (3,1)
0 (0,0)	1 (1,0)	2 (2,0)	3 (3,0)

Figure 2.3 Grid Division Topology

The top number in each cell of figure 2.3 represents the process rank. Each process has a unique rank assigned to it ranging from zero to one less than the number of processes chosen. The bottom number represents the topology coordinates of that process's portion of the total grid. In the parallel model local variables on each process store this information. There are also variables corresponding to the ranks of the processes around them in the process topology. For example, the process with a rank of 5 has variables specifying its rank, its topology coordinates, and also has variables *up*, *down*, *right*, and *left* which correspond to the ranks of the processes around it. The *right* variable for process 5 would store the value 6. This information is useful for identifying which processes share boundary information.

The geometry of the process topology is based on the set of factors of the number of processes chosen closest to each other. This means the grid topology is chosen to be as square as possible. For example, the total number of processes in figure 2.3 is twelve and the the local grids are

arranged in a three by four topology instead of two by six or one by twelve. The larger of the total grid size in one direction is divided by the larger of the factors. MPI implementations have routines available for this, but it is up the implementor to decide to use these routines or build ones. In the parallel WENO model, the MPI topology functions are called to determine the process topology. Figure 2.4 shows how the MPI functions are used to set up this process topology.

```

call MPI_DIMS_CREATE(num_nodes,ndims,dims,ierr)
call MPI_CART_CREATE(MPI_COMM_WORLD,ndims, &
                    dims,periods,.false.,NEW_COMM,ierr)
call MPI_CART_GET(NEW_COMM,ndims,dims,periods,coords,ierr)

xnumnodes=dims(1)
ynumnodes=dims(2)
xtcoord=coords(1)
ytcoord=coords(2)

! DEFINE LEFT
if(xtcoord.NE.0) then
  coords(1)=xtcoord-1
  coords(2)=ytcoord
  call MPI_CART_RANK(NEW_COMM,coords,left,ierr)
elseif (xperiodic) then
  coords(1)=xnumnodes-1
  coords(2)=ytcoord
  call MPI_CART_RANK(NEW_COMM,coords,left,ierr)
else
  left=MPI_PROC_NULL
endif

```

Figure 2.4 Example of MPI Topology Functions

The MPI_DIMS_CREATE() function comes up with the number of divisions of the domain in each direction based on the dimensions of the domain and the number of MPI processes. The MPI_CART_CREATE() function creates an MPI topology of the dimensions found in MPI_DIMS_CREATE() and MPI_CART_GET() provides the coordinates of the MPI process which called it. Figure 2.4 also depicts how the MPI process left (or one with sub-domain containing the next smaller x values) is defined utilizing MPI_CART_RANK() and is stored in the variable *left*. The other directional process topology variables are found in a similar manner.

2.2.2 Grid Division

The number of grid points in each direction for the total grid is divided up by the number of processes in each direction in the process topology. Since the calculations in the WENO model across the grid are mostly uniform, a division in this manner achieves load balancing. However, the number of grid points may not necessarily be a multiple of the number of processes. In this case extra grid points are added to some of the MPI processes. The logic for this is based on the process's topology coordinates and the modulus or remainder of the grid points and number of processes in the corresponding direction. For example, if the process topology is as shown in figure 2.3, then the grid will be divided among three processes in the y direction. If the total number of points in the y direction is twenty, then integer division will provide the number of points in the y direction on each process as six—accounting for eighteen of the twenty points. If the process's coordinate is less than the modulus of the number of points and number of processes in a given direction (two in this case), then one point is added to the number of grid points in that direction. This results in seven points in the y direction on processes with ranks 0–7 and six on 8–11, which is twenty total with respect to the y direction. This and similar division in the x direction accounts for the division of the total physical grid. In addition to this, four grid points are added on each end of the local grids for ghost points. These ghost points will be used to emulate the presence of the adjacent grid to the local process's grid or for implementation of the physical boundary conditions depending on the location of the local grid on each process with respect to the total grid.

2.2.3 Initial Conditions

The initial values of dependent and independent variables used in calculations must be considered as a parallel aspect since they will need to be initialized on each sub-domain. There are two options for handling the initial conditions. They can be programmed in parallel into the model

or be read into the program. Reading the initial conditions from a file specifying the values on the entire domain is advantageous because future users can solve various problems by building the initial conditions separately. Reading initial conditions on large problem spaces may be time consuming, but it only happens once during program execution. In order to achieve this a general file reader must be developed for the parallel model. Each process will skip the lines that do not pertain to its sub-domain and read in the lines that correspond to portions of its domain.

2.2.4 General Computational Domain

The parallel WENO model is designed to run for many physical domains by performing calculations on a general rectangular computational domain. To perform these calculations correctly, mapping information known as metrics between the two domains is necessary. As with initial conditions this could be calculated in parallel, but would change with each problem geometry, so a reader is also used to read in the grid point locations and the metrics information.

2.3 Parallel Modifications

The serial model must be modified in several ways to achieve distributed parallelization. There are several instances where communication between MPI processes is necessary. The values on the ghost points must be updated between calculation steps and communication is required for calculating the time step. Also, physical boundary conditions only exist on edges of certain process sub-domains.

2.3.1 Minimizing Code Modification

As mentioned in section 2.1, many of the core components of the serial code can be left unchanged. To achieve this, careful examination of the algorithms employed is necessary and the meaning of some of the variables used in the serial code must be changed. For example, in the serial version of the WENO model all the maximum loop indices within routines such as DFDT

corresponded to the total grid size. It is beneficial to leave the variables for these indices in place and to change their meanings to correspond to the local grid. It may not always be possible to “wrap” large components of the serial code with the parallelization aspects, but it is ideal for a few reasons. It minimizes the amount of code changes, reducing the chance for bugs to be introduced as well as simplifies the comparison between the serial and parallel models. Model components may be tested in either model in some cases.

2.3.2 Boundary Update

Between calculation stages values on the ghost points must be updated. In the simplest of finite-difference algorithms, the boundary ghost points are updated once each iteration. In more complex models updates of the ghost point values are needed more frequently for calculations to remain the same as in serial. Figure 2.2 shows a simplified flow chart for the Runge Kutta TVD time step differentiation used in the WENO model. For the parallelization of the WENO model, the boundary ghost points must be updated between each stage of the Runge Kutta method.

Since the WENO model is a fifth-order finite-difference model, four points in any primary direction around the calculation point are utilized. This can be seen by the stencil in figure 1.3. Thus, the message passing stage updates four ghost points adjacent to any interior grid point on a local process. A representation for this is shown in figure 1.4. The intersection of the dotted lines represent the ghost points required on a local process for proper calculation of the real grid points which are represented by the intersection of solid lines. Each ghost point on an interior boundary is a copy of a real grid point and the dependent variables: density, x and y velocities, x and y momentums, pressure, temperature, and energy need to be updated from its source grid point.

The number of double precision values sent to update ghost points along one side is equal to the number of points along that side times the number of variables times the four points needed by the calculation stencil. For example if the number of points along the x direction is N_x , then

to update the ghost points along the local y minimum or maximum boundary, the number of ghost points passed is found by:

$$N_g = N_x * 8 * 4 \quad (2.1)$$

In this case, it is virtually impossible for memory locations for the information needed in the passing stage to be contiguous or all next to each other in the computer's memory. In order to pass the required info, multiple send and receive calls could be invoked—one for every contiguous chunk of memory. One would assume this to likely be one message passing stage for each variable and each ghost point number. However, the variables needed are part of much larger matrices and matrices are stored as one-dimensional arrays within the computer's physical memory. Therefore, it is unlikely to find large contiguous chunks of memory for passing consistently on all sides of the local grid. The other option is to load the memory into a single array of size N_g . This also minimizes the number of times MPI send and receive calls are invoked. Generally, fewer larger messages is more efficient than many smaller ones due to the overhead involved in initializing communication.

Figure 2.5 is an excerpt from the WENO code showing one part of the passing stage. In this segment values along the right side of the physical local grid (not the ghost points) are loaded into an array called *pass*. That information is then sent to the process containing the grid to the right (meaning the process with the next larger x location values) and received from the process containing the section of the grid to the left. The right and left variables used in the MPI_SENDRECV call are set up according to the grid topology discussed in section 2.2. Also, in this segment the variable *arraynum* has a value of thirty-two corresponding to the number of dependent variables times the number of ghost points—the four times eight in equation 2.1. The variables such as *E_base* and *MXbase* are used to order the eight variables and four sections of the arrays into one large array called *pass* and unload from the array called *catch*.

```

messageSize=arraynum*J_arraylen

E_base=0
DNbase=1*numghost*J_arraylen
UXbase=2*numghost*J_arraylen
UYbase=3*numghost*J_arraylen
MXbase=4*numghost*J_arraylen
MYbase=5*numghost*J_arraylen
P_base=6*numghost*J_arraylen
T_base=7*numghost*J_arraylen

ALLOCATE (pass(messageSize),catch(messageSize))

!=====
! send info right

DO L=1,numghost
  DO J=1,J_arraylen
    pass(J+E_base+(L-1)*J_arraylen)= E(IDMX+1-L,JDMN-1+J)
    pass(J+DNbase+(L-1)*J_arraylen)=DN(IDMX+1-L,JDMN-1+J)
    pass(J+UXbase+(L-1)*J_arraylen)=UX(IDMX+1-L,JDMN-1+J)
    pass(J+UYbase+(L-1)*J_arraylen)=UY(IDMX+1-L,JDMN-1+J)
    pass(J+MXbase+(L-1)*J_arraylen)=MX(IDMX+1-L,JDMN-1+J)
    pass(J+MYbase+(L-1)*J_arraylen)=MY(IDMX+1-L,JDMN-1+J)
    pass(J+P_base+(L-1)*J_arraylen)= P(IDMX+1-L,JDMN-1+J)
    pass(J+T_base+(L-1)*J_arraylen)= T(IDMX+1-L,JDMN-1+J)
  ENDDO
ENDDO

Call MPI_SENDRECV(Pass,messageSize,MPI_DOUBLE_PRECISION,right,
$ tag,catch,messageSize,MPI_DOUBLE_PRECISION,left,
$ tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE,ierr)

if((xtcoord.NE.0).or.(xperiodic)) then
  DO L=1,numghost
    DO J=1,J_arraylen
      E(IDMN-L,JDMN-1+J)=catch(J+E_base+(L-1)*J_arraylen)
      DN(IDMN-L,JDMN-1+J)=catch(J+DNbase+(L-1)*J_arraylen)
      UX(IDMN-L,JDMN-1+J)=catch(J+UXbase+(L-1)*J_arraylen)
      UY(IDMN-L,JDMN-1+J)=catch(J+UYbase+(L-1)*J_arraylen)
      MX(IDMN-L,JDMN-1+J)=catch(J+MXbase+(L-1)*J_arraylen)
      MY(IDMN-L,JDMN-1+J)=catch(J+MYbase+(L-1)*J_arraylen)
      P(IDMN-L,JDMN-1+J)=catch(J+P_base+(L-1)*J_arraylen)
      T(IDMN-L,JDMN-1+J)=catch(J+T_base+(L-1)*J_arraylen)
    ENDDO
  ENDDO
endif

!=====

```

Figure 2.5 Example of Message Passing Stage

There are three more parts to a passing stage in the WENO model similar to the one depicted in figure 2.5. The other stages correspond to passing information left, up, and down. Again, passing information up corresponds to sending physical grid information along the $y = y_{max}$ side of the local grid of a process to the process containing the next larger y values with respect to

the total grid to update the receiving process's ghost points along the local $y = y_{min}$ boundary.

2.3.3 Physical Boundary Conditions

Ghost points on exterior (physical) boundaries are not necessarily copies of real grid points and are updated based on the boundary conditions. For periodic boundary conditions they are treated as copies the same way the interior ghost points are handled, but the location of the process with the adjacent portion of the grid is on the other side of the total grid. For example, in the topology shown in figure 2.3 if a periodic boundary condition in the x direction exists, then the variable *left* for the process of rank 4 would be seven and the rank 7 process's *right* variable would contain four. This enforces the periodic boundary condition.

2.3.4 Reduction

The WENO model dynamically adjusts the time step based on the CFL number and the maximum velocities throughout the entire grid as shown in equation 1.1. Since the *max* operation is over the entire domain and each process only contains a sub-domain, communication in some form is necessary. One way to do this is to calculate the maximum velocities on each process, use MPI_GATHER() to bring them to one process, and then use MPI_BCAST() to give them or the calculated time step to the rest of the processes. Another way to achieve this process is MPI_REDUCE() which utilizes a binary tree communication algorithm, minimizing the communication required to provide every process with the maximum velocities across the entire domain.

2.3.5 Domain Reconstruction in Data Files

For visualization and analysis of the results computed by the computational model it is more often than not desirable to recombine the sub-domains in the output files. This is achieved in the WENO model by utilizing a post-processing module that recombines a data file produced by

each MPI process into one file. The post-processing script relies on information provided in the file name about the topology coordinates and the size of the grid from each MPI process.

2.3.6 Generality of Parallel Functions

Many of the parallelization functions built are general in design. If different algorithms are adopted, the parallel components would be adaptive or they could easily be migrated to a different model. The topology functions are not specific to the WENO model and could be used for initialization of process topology in other models. Also, the WENO model utilizes fifth-order spatial finite-differences and in turn requires four points in each direction for calculations. Higher or lower order calculations would require more or fewer ghost points, respectively. The message passing stage is dependent on a variable passed to it called *numghost*. If more or fewer boundary points were to be updated the message passing stage could be called specifying the desired value. The message passing stage will always update the dependent variables on all indices *numghost* lower than the local minimum index *IDMN* or *JDMN* and higher than the local maximum index *IDMX* or *JDMX*.

2.3.7 Communication Visualization

Just as it is important to profile a serial model to analyze it for efficiency and other concerns, it is beneficial to visualize the communication within a parallel model. Jumpshot-4 is a program produced by Argonne National Laboratory to be used in conjunction with the MPE logging features of some MPI distributions. Not only does this provide valuable information about efficiency with respect to communication stages, but also provides a useful mechanism in detecting errors within the parallel implementation.

For example, during each stage of the Runge Kutta iterations, a first-order derivative was calculated and subsequently the arrays used in the calculation were deallocated to save memory. To complete this derivative in parallel, a message passing stage was used. This one-sided derivative

was calculated inside of a loop iterating over the grid in one direction. Figure 2.6 depicts a snapshot from Jumpshot-4. If an MPI model is compiled and linked with some MPE utilities, a log file of the MPI functions is produced. Jumpshot provides visualization to these log files. The white arrows represent messages being passed between processes. The communication overload was observed to be high and the source was tracked back to the message passing stage required for these derivatives housed within the grid iteration loops.

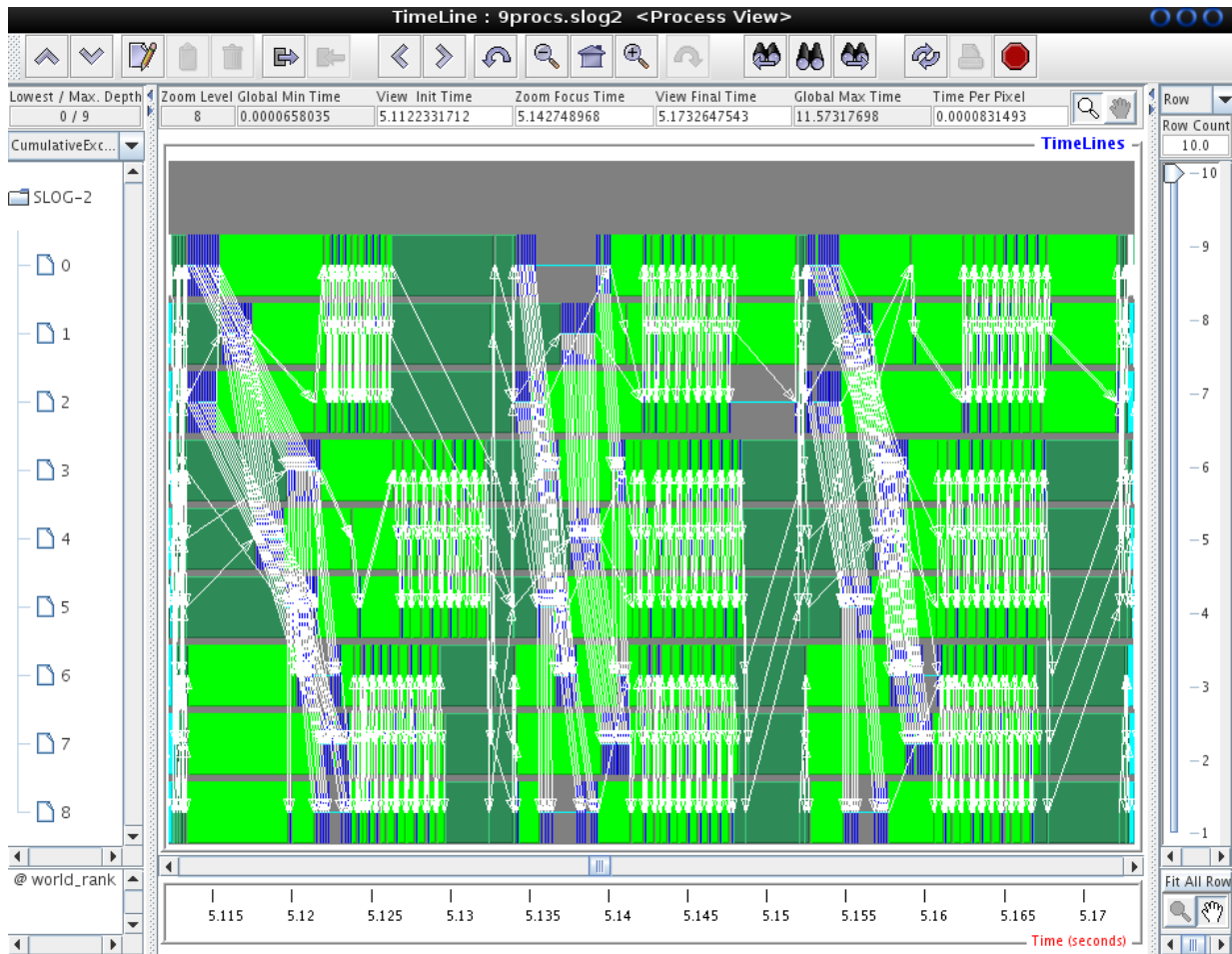


Figure 2.6 Jumpshot Before Communication Revision

Since the results from these derivatives were not required within the loop, space was allocated to store the dependent values until after the loop's completion. This allowed one message passing

stage to update the values required for every derivative along a process's grid boundary at once. The communication visualization with Jumpshot after this change is shown in figure 2.7. As can be seen, the communication is much more structured. There are definite message passing stages in figure 2.7, separated by computational stages.

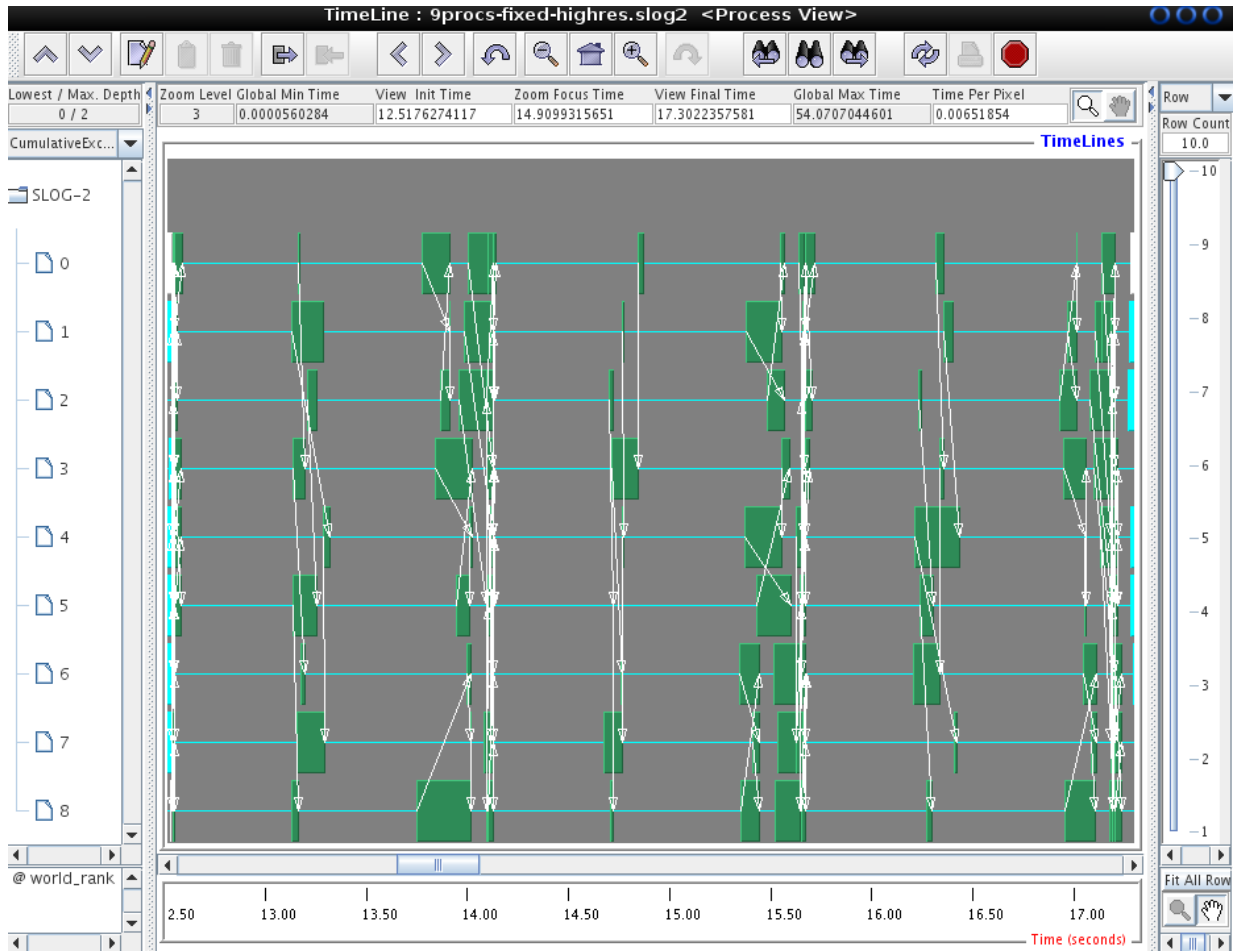


Figure 2.7 Jumpshot Communication Profile

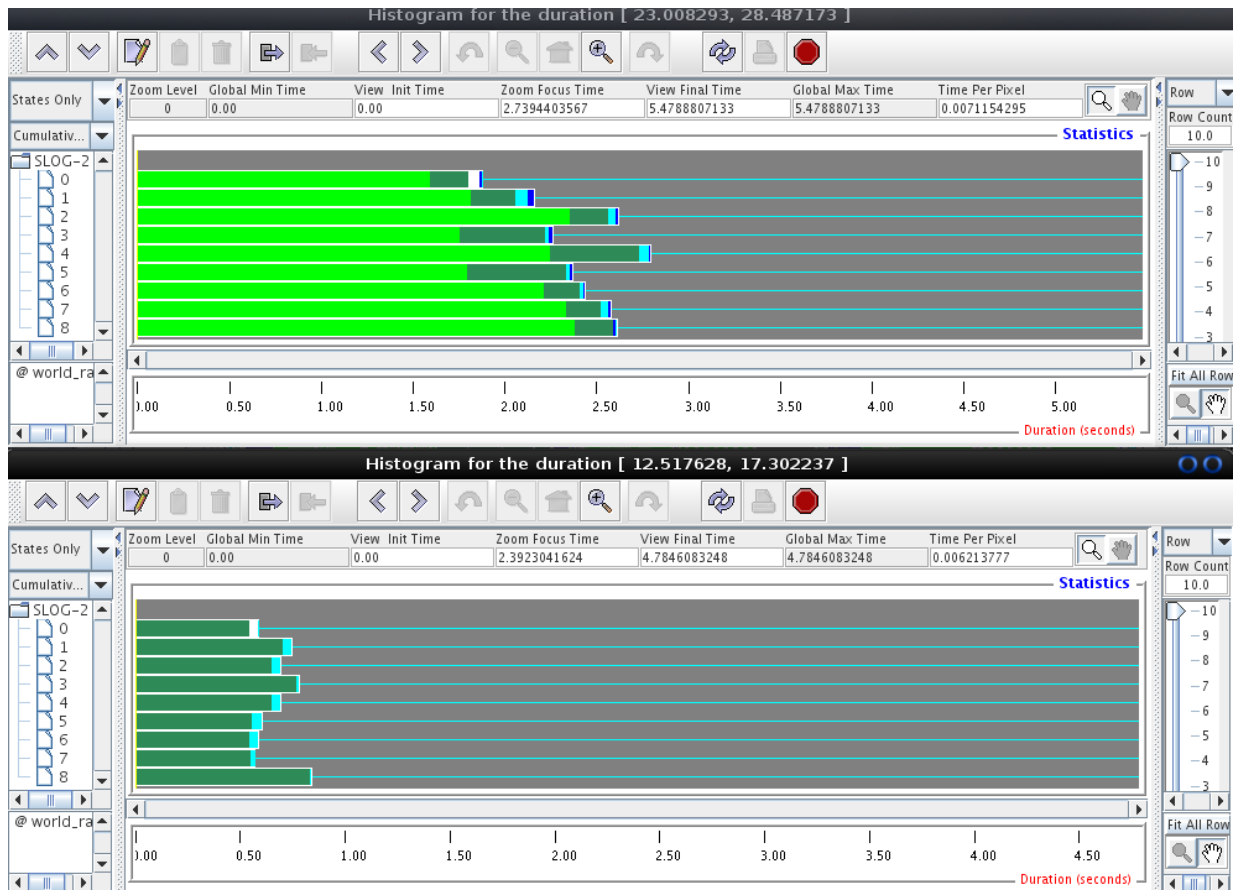


Figure 2.8 Communication Time Histogram Comparison

Figure 2.8 shows a histogram comparing the time spent in communication before and after the revision of the communication. The lighter bars in the top histogram depict the amount of time the processes spent waiting for messages. Both histograms represent a full iteration of the code. The bottom half shows the histogram corresponding to the model after the communication revision. More of the processes' total iteration time is spent in computation, making the model more efficient overall.

CHAPTER 3

IMPROVEMENTS

Many improvements were made to the WENO model besides the parallelization efforts. Functionality was increased by combining boundary conditions as options within the code and by creating an interface for a configuration file provided at runtime. Efficiency and portability were increased by addressing memory access concerns and switching to dynamic memory allocation.

3.1 Efficiency

There are several factors that can impact model efficiency. Many involve making tough choices about storage versus computation and are problem dependent. There are, however, some tricks and things to look for to improve the efficiency of a computational model.

3.1.1 Loop Efficiency

Generally, heavy computation within a model occurs within loops. In light of this, the following techniques all involve efficiency concerns around loops [19].

Hoisting Invariant Code

Many times much of the math within a loop is in fact independent of the loop index. This can occur by itself and is easily moved outside of the loop to decrease the number of computations performed. Or, the independent math can be part of some loop-dependent calculations. A common example of this is squaring a number, say r . Instead of calculating the square of r every iteration of a loop, the variable r^2 could be set as $r^2 = r * r$ and be calculated prior to the loop. This is a simple example, and there are many other possibilities of loop-invariant math that can be removed from loops.

Unswitching

This is perhaps one of the most important optimization techniques. Unswitching means removing independent branch statements from loops. Branch statements are “if” and “switch” type coding techniques. Compilers do many things to optimize the evaluation of loops and branch statements stop this because they must be evaluated individually at runtime. It is a common mistake to have one loop with multiple possible courses, instead of having multiple courses (as some sort of branch statement) each with its own loop. Often the branch statement is dependent on the loop index, and cannot be removed. The following techniques concentrate on removing this dependency in order to unswitch the loop. In other cases, it may be unavoidable.

Iteration Peeling

It is common for the ends of arrays to be treated differently than the rest due to physical boundary conditions or the switching of algorithms to use one-sided stencils. Instead of iterating a loop over the entire domain and using a branch statement to handle the end points, it is more efficient to evaluate the ends separately, eliminating the need for the branch statement.

Index Set Splitting

If a portion of the array is evaluated differently than the rest by a branch statement, then it is best to split the loop into multiple parts to eliminate the need for the branch statement. This may be difficult if the index at which the differences is not easily calculated before the loop or changes at runtime.

3.1.2 Memory Locality

Data locality makes a large difference in how fast a processor can perform calculations. When a processor uses a value, an entire line of memory is loaded into cache. A processor can perform

calculations on data in cache much faster than it can load data from memory to cache. Each time the processor needs a piece of data, it checks the cache. If it is there, a cache hit occurs. If not, a cache miss occurs. An efficient code is one that minimizes the number of cache misses during runtime. How matrices are defined and accessed can greatly impact the ratio of cache hits to cache misses.

A multi-dimensional array in code is mapped to contiguous memory as a large one-dimensional array. In a two-dimensional matrix, all the components in a row may be next to each other in memory, whereas column components may be far apart. The mapping varies by language. For example, $B(I, J)$ and $B(I + 1, J)$ are next to each other in memory in a Fortran code, but $B(I, J)$ and $B(I, J + 1)$ are near each other in memory in a C code.

If looping over calculations on a multi-dimensional array, then it is best if the inner loop is the one that loops over the index that corresponds to data locality in memory. It is implausible to satisfy this condition all the time, but may prove worth the effort if possible. Another possible way to increase cache hits without evaluating the loops is to switch the indices of matrices and time the model.

If it is impossible to take advantage of locality with respect to data indices, then it may be possible to localize the data corresponding to certain locations within a set of data matrices by combining them into a single higher order matrix. For example, instead of making an energy matrix $E(I, J)$, pressure matrix $P(I, J)$, and density matrix $DN(I, J)$, make a variable matrix $VAR(V, I, J)$. Here $V = E = 1$ would correspond to energy values (I, J) , $V = P = 2$ would correspond to pressure values at (I, J) , and so forth. For a Fortran program this would put all the parameters at a certain grid point together in memory, possibly decreasing cache misses during calculations. The best choice of storage implementation will vary with application.

3.1.3 Dynamic Allocation

Memory can be allocated for arrays at compile time or at runtime. If it is allocated at compile time it is called static allocation, and at runtime it is called dynamic allocation. Memory allocated at compile time is placed on the stack in memory storage terms and memory allocated during runtime is placed on the heap. This is significant because many computers and operating systems place a limit on how much memory is available in the stack. To utilize all of a system's memory, it is best to use dynamic memory allocation within a model. Also, this allows the size of arrays to be decided at runtime, based on either changing parameters or parameters provided by the user.

3.2 Configuration

For a model to be highly functional to all users, the model must be adaptable. Furthermore, the ability to configure the model must be straightforward and efficient. Branches within the code must exist to increase the model's overall functionality, and run-time configuration makes adjusting parameters for problem analysis more practical.

3.2.1 Configuration Branches

A general computational model is one that can be used for multiple problems. To achieve a high level of generality, it is necessary to execute different sections or branches of code based on the problem being investigated. The sections should be available through the use of configuration options, so that the same model can be tested and validated for multiple problems. One example of this is boundary conditions. All the boundary conditions are made available within the model and variables control whether the functions applying them are called. In the WENO model the variables are *BCXMN* for the boundary along the minimum *x* axis, *BCYMX* for the boundary along the maximum *y* axis, and so forth. The values and corresponding boundary conditions of these variables are outlined in the configuration file.

3.2.2 Configuration File

Large computational codes such as those employed for CFD analysis normally have many adjustable parameters and options. For testing and development purposes, many times these are just hardcoded within the initial portions of the model, and to change parameters the code must be recompiled. It is advantageous to have a configuration file that is read at runtime that specifies the options and parameters. If the number of options and initialization parameters is large enough, several configuration files may be necessary for organizational purposes. By initializing parameters at run time, one executable can be used for a wide variety of applications. It allows one to quickly edit a configuration file and rerun the model (or even automate the process) rather than edit the source code and recompile. Editing the source code increases the chance of introducing error and recompiling can be quite time consuming with large models.

The parallel WENO model employs the use of a key-value configuration reader. A key-value reader is best because the formatting and order of parameters and the corresponding values does not affect how they are read into the model. Figure 3.1 is an excerpt from a configuration file used in the parallel WENO model.

```

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!   EXAMPLE CONFIGURATION FILE
!
!   NOTE THAT
!   * INPUT PARAMETERS ARE CASE SENSITIVE
!   * LINES THAT BEGIN WITH ' ' or '!' are ignored
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Settable parameters displayed as
! parameter=type(default) : description

! NX=INT(100),NY=INT(100)  Number of physical grid points
NX=97
NY=65

! T=DOUBLE(0.2D0) : Time of run
T=0.2D0

! OPTION=INTEGER(1) : Flag for different run type
!                   : 1,2,3,4 is WENOJS,WENOM,WENOA,WENOMA
OPTION=4

! NG=INT(4) : number of boundary "ghost" points for passing with MPI

! BCXMN,BCXMX=INTEGER(0) : Flag for X Boundary Conditions
!                       : 0,1,2,3 is periodic,solid wall,fixed,extrapolated
!                       : either as 0 overrides the other

BCXMN=2
BCXMX=2

! BCYMN,BCYMX=INTEGER(0) : Flag for Y Boundary Conditions
!                       : 0,1,2,3 is periodic,solid wall,fixed,extrapolated
!                       : either as 0 overrides the other

! XMN,YMN=DOUBLE(0.0) : Minimum coordinate value
XMN=-1.5D0
YMN=-1.0D0
! XMX,YMX=DOUBLE(1.0) : Maximum coordinate value
XMX=1.5D0
YMX=1.0D0

! CFL=DOUBLE(0.6D0)

```

Figure 3.1 Example of Configuration File

The configuration file contains description and information about what the parameters do to help the user adjust the model. An excerpt of the code used to read the configuration file is shown in figure 3.2. The configuration file is read in a line at a time. All lines that start with

a space or an exclamation point are skipped as well as lines that do not contain an equal sign. If a line contains an equal sign, the characters before the equal sign are compared against the parameters listed for adjustment in the “select case” statement. If a match occurs, then the value after the equal sign is read into the corresponding parameter. If there is not a match, then a warning is printed to standard output.

```

5 Continue
  read(901,'(A)',end=10) the_line
  if ((the_line(1:1).eq.'!').or.(the_line(1:1).eq.' ')) then
    GOTO 5
  endif

  seperator='='
  pos=scan(the_line, seperator)
  if(pos.eq.0) GOTO 5

  my_parameter=the_line(1:pos-1)
  my_value=the_line(pos+1:)

  select case (my_parameter)
  case ("NX")
    read(my_value, *) NX
  case ("NY")
    read(my_value, *) NY
    .
    .
  case ("CFL")
    read(my_value, *) CFL
  case ("T")
    read(my_value, *) TTIME
  case ("OPTION")
    read(my_value, *) OPTION
  case default
    print*, "Warning: ", &
      my_parameter, " is not a valid input parameter."
  end select

  GOTO 5

```

Figure 3.2 Excerpt of Configuration File Reader

To add adjustable values to the configuration file read in at runtime, a corresponding case must be added to the “select case” statement shown in figure 3.2.

CHAPTER 4

RESULTS

The parallel WENO model was validated against the serial version during development. Many aspects were changed and improved in the parallel version, so validation was important in finding and fixing bugs introduced in the model. The scalability of the WENO and ADI models were analyzed and the results were very promising in support of parallelization feasibility.

4.1 Validation

The first validation case is based on an example problem originally presented by Shu and Osher [20] in which a shock wave moving at Mach eight interacts with a vorticity wave. The boundary conditions are periodic in the y direction and fixed in the x direction. The domain ranges from $-1.5 < x < 1.5$ and $-1 < y < 1$ with 97 and 65 points, respectively.

Figures 4.1 through 4.8 compare the results published by Ghulam and Hoffmann [1] for shock turbulence interaction produced with the serial model against those produced with the parallel model. Figures 4.5 through 4.8 compare results utilizing modification for critical points and Yang's slope modification and are denoted with an "MA". The others are denoted with a "JS". Figures 4.1, 4.3, 4.5, and 4.7 are the published serial results and figures 4.2, 4.4, 4.6, and 4.8 are the results produced in parallel. The meaningfulness of the data produced was less important for validation than the correspondence to the published serial results.

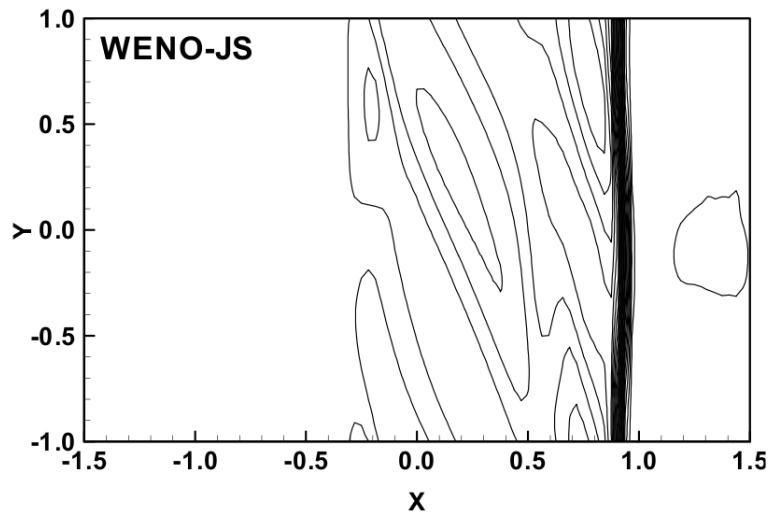


Figure 4.1 Serial Shock Turbulence Density - JS

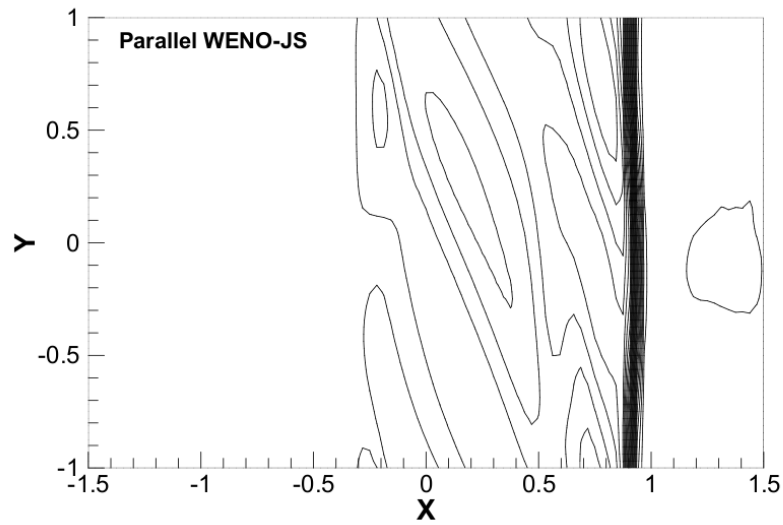


Figure 4.2 Parallel Shock Turbulence Density - JS

Figures 4.1 and 4.2 are contour plots produced in Tecplot depicting density for the shock turbulence case. There are 30 contours ranging from 0.99 to 6.16 [1].

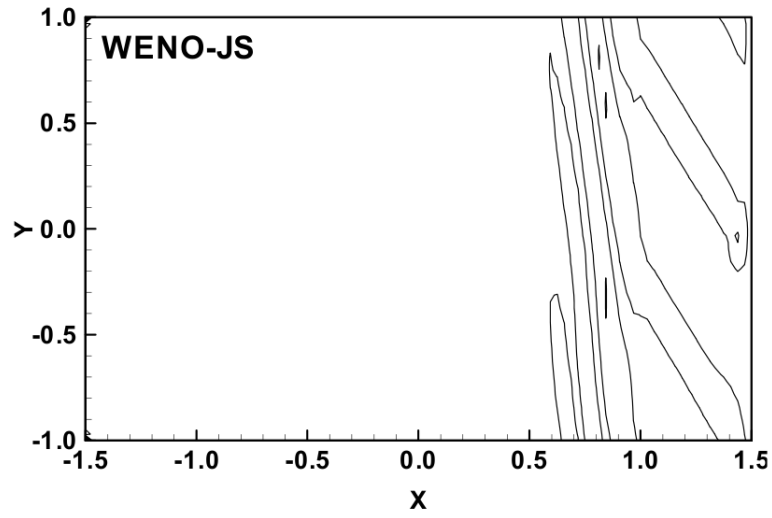


Figure 4.3 Serial Shock Turbulence Vorticity - JS

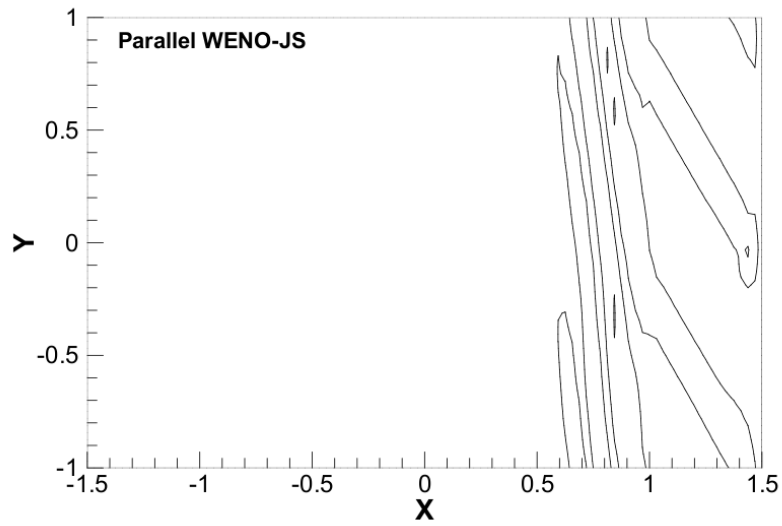


Figure 4.4 Parallel Shock Turbulence Vorticity - JS

Figures 4.3 and 4.4 depict vorticity as contour plots. There are 60 contours ranging from -100 to 100.

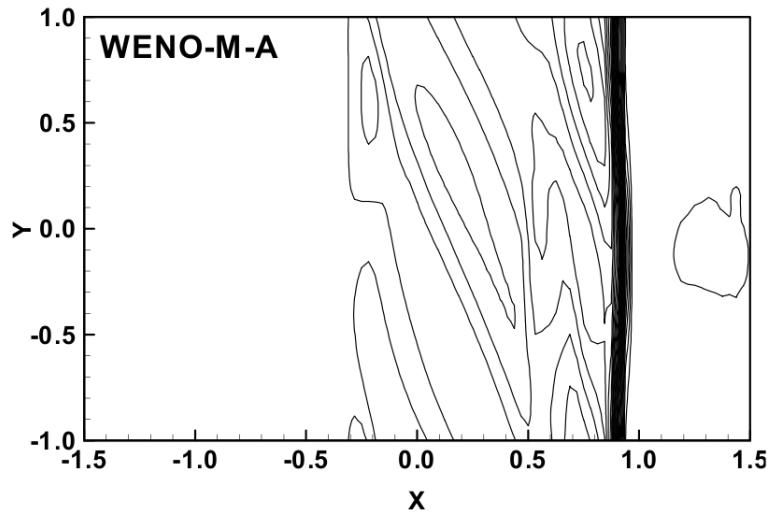


Figure 4.5 Serial Shock Turbulence Density - MA

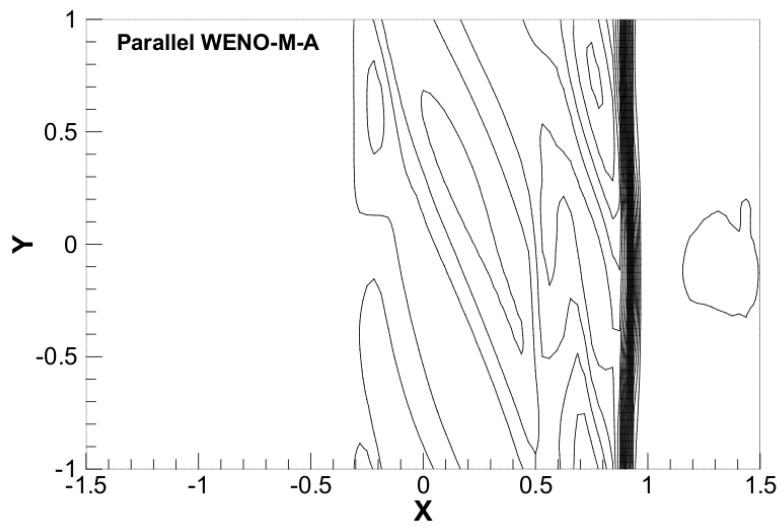


Figure 4.6 Parallel Shock Turbulence Density - MA

Similarly, figures 4.5 and 4.6 are contour plots depicting density for the shock turbulence case. However, both models were configured to use modification at critical points and slope modification. There are also 30 contours ranging from 0.99 to 6.16.

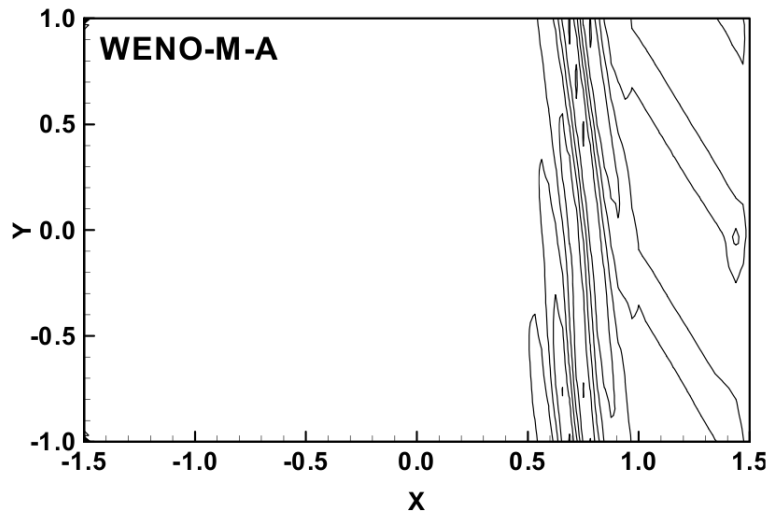


Figure 4.7 Serial Shock Turbulence Vorticity - MA

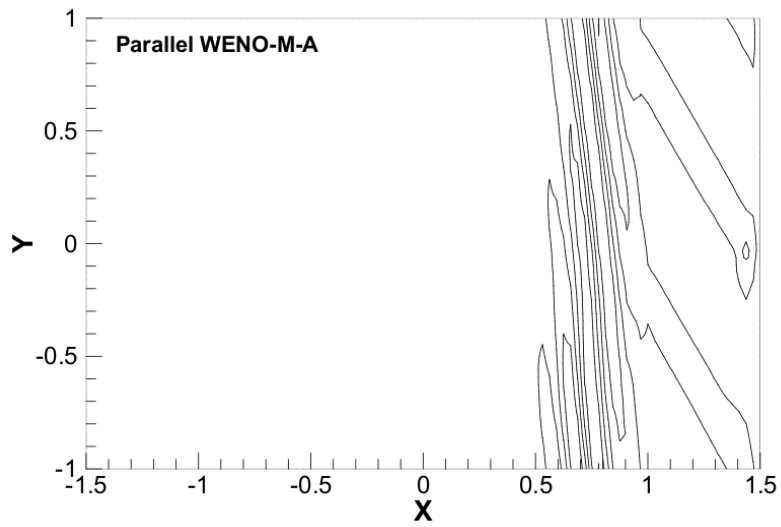


Figure 4.8 Parallel Shock Turbulence Vorticity - MA

Figures 4.7 and 4.8 depict vorticity as contour plots for the “MA” case. There are 60 contours ranging from -100 to 100.

The second validation case is a planar shock moving at $M = 2.3$ interacting with a cylinder [21]. In this case the grid information and initial conditions, shown as a density contour plot in

figure 4.9, are created using the serial model and read into the parallel model. The boundary condition along the cylinder surface is reflective; along the domain seam after the cylinder, the boundary condition is periodic. On the outer surface, an extrapolation boundary condition is used. There are 101x401 grid points.

As with the previous validation case, the results of the parallel model are compared against the published results produced by the serial model [21]. Although all of the time values from the published work were compared, two times each for both the standard model and the model employing the modifications discussed and denoted “MA” are shown below. Figures 4.10, 4.12, 4.14, and 4.16 depict the results produced in serial, whereas figures 4.11, 4.13, 4.15, and 4.17 depict the results produced in parallel.

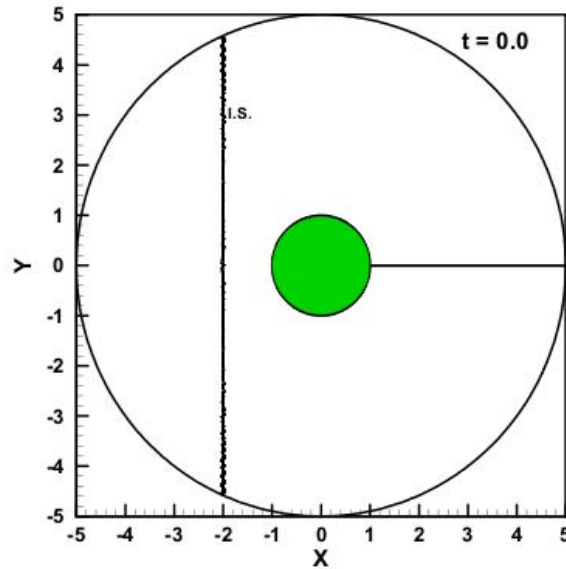


Figure 4.9 Initial Shock for Inviscid Flow Over Cylinder

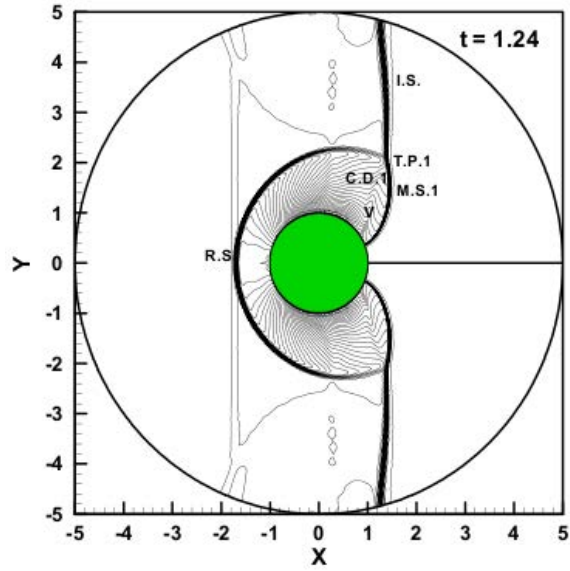


Figure 4.10 Serial Inviscid Shock Over Cylinder - JS: $t=1.24$

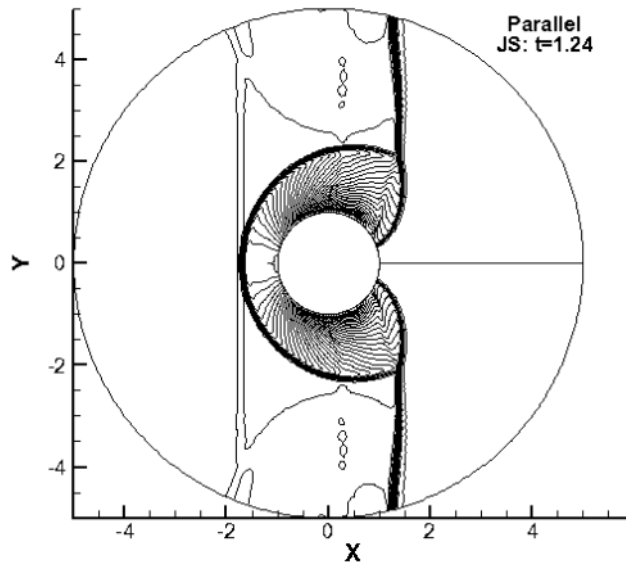


Figure 4.11 Parallel Inviscid Shock Over Cylinder - JS: $t=1.24$

Figures 4.10 and 4.11 depict the time $t = 1.24$ density contours chosen for comparison. There are 60 contours ranging from .76 to 7.9.

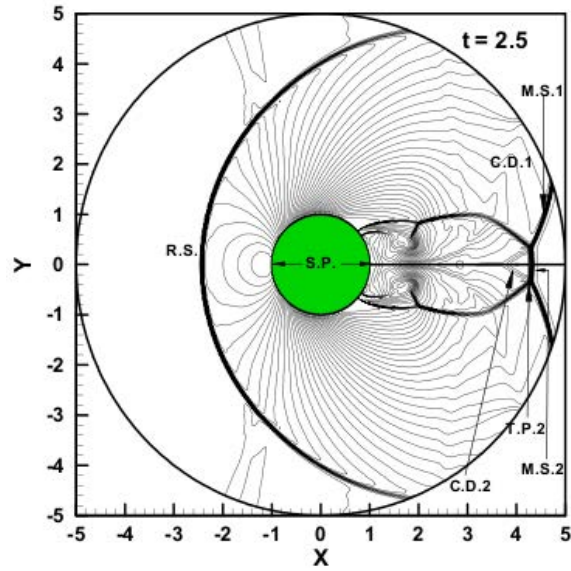


Figure 4.12 Serial Inviscid Shock Over Cylinder - JS: $t=2.5$

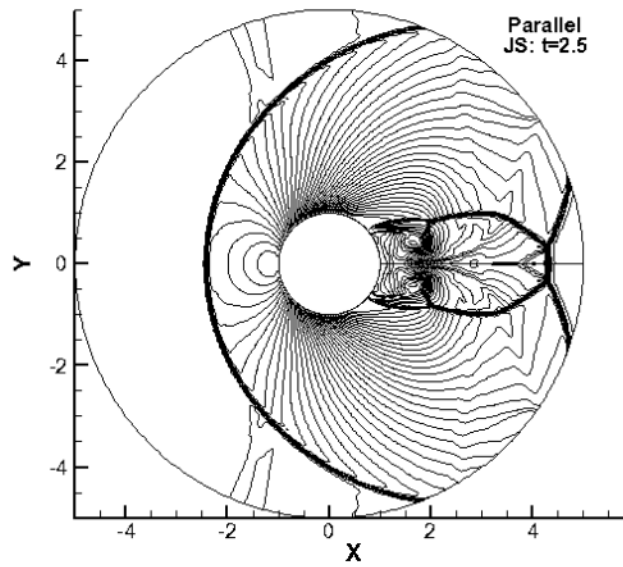


Figure 4.13 Parallel Inviscid Shock Over Cylinder - JS: $t=2.5$

Figures 4.12 and 4.13 depict the final time $t = 2.5$ published for the serial model. Again, there are 60 density contours ranging from .76 to 7.9.

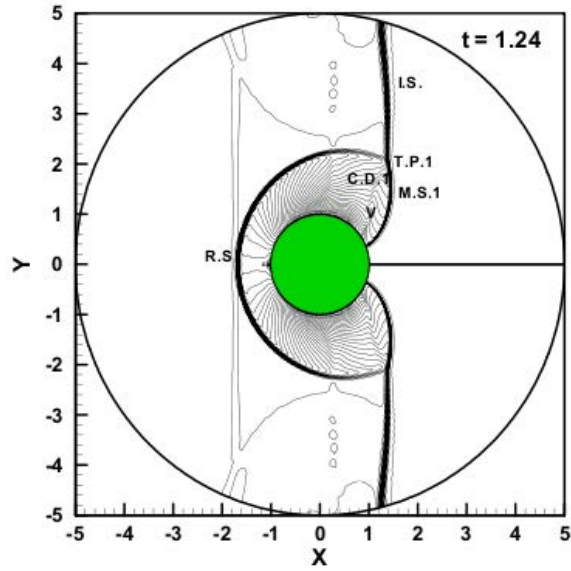


Figure 4.14 Serial Inviscid Shock Over Cylinder - MA: $t=1.24$

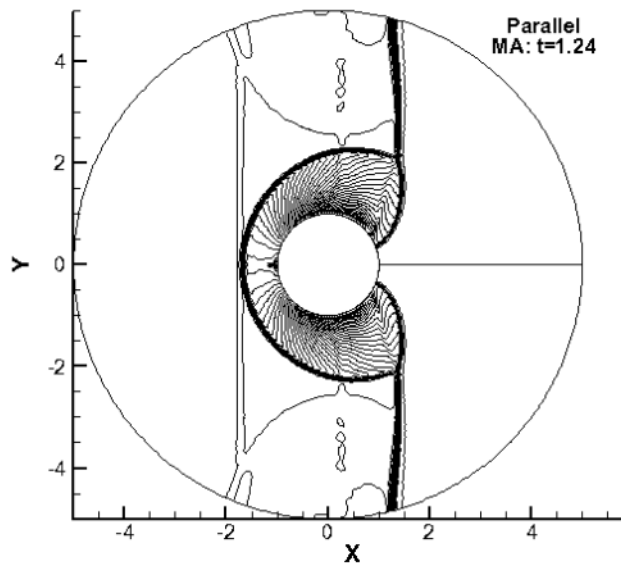


Figure 4.15 Parallel Inviscid Shock Over Cylinder - MA: $t=1.24$

Figures 4.14 and 4.15 depict the time $t = 1.24$ density contours chosen for comparison. For this case both models were configured to use modification at critical points and slope modification. There are 60 contours ranging from .76 to 7.9.

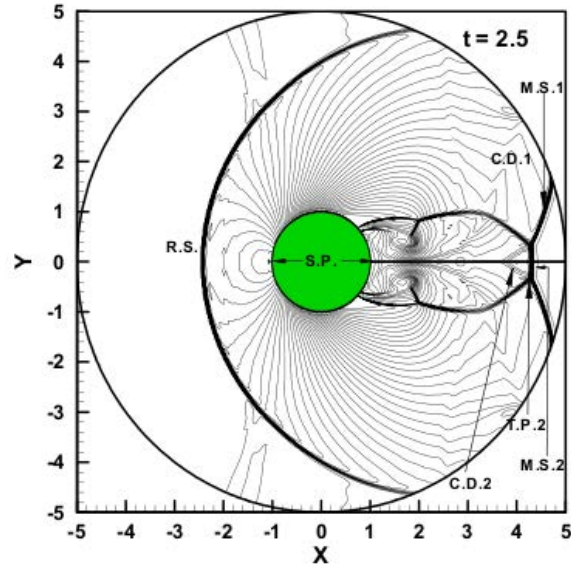


Figure 4.16 Serial Inviscid Shock Over Cylinder - MA: $t=2.5$

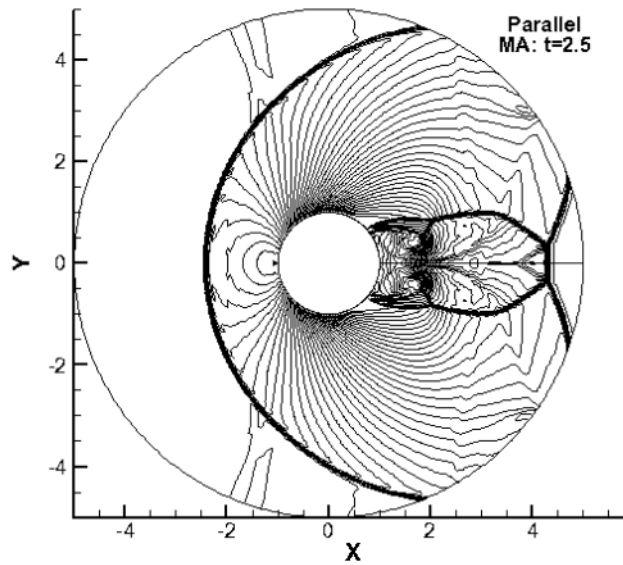


Figure 4.17 Parallel Inviscid Shock Over Cylinder - MA: $t=2.5$

Figures 4.16 and 4.17 depict contours at $t = 2.5$ for the “MA” case. Again, there are 60 density contours ranging from .76 to 7.9.

As can be seen, the results show excellent correspondence. This helps to validate the implementation of the parallelization. Slight variations exist, but are caused by differences in rounding error from one machine to the next. This has been confirmed by running the serial model on the same hardware as the parallel results were produced. The results produced by the serial model were compared against the parallel results with a differencing program and were exactly the same—confirming the variations to be from differences in rounding error.

These two validation cases test each type of boundary condition currently employed in the WENO model: periodic, fixed, solid wall (or reflective), and extrapolation. Both cases were run with the parallel model's functionality for a general computational grid, but since the first case is a square physical grid, the second case is more revealing in the validation of the general grid implementation. A file reader was used in the second case for both the initial conditions and the grid metrics—validating the functionality of the parallel model to be used on a wide variety of cases if an initial condition file and metrics file are provided.

In all of the previous cases the flow was inviscid, therefore the model was simplifying to a solution of Euler's equations. The following case is similar to the previous one but the full Navier-Stokes solver is used because the effects of viscosity are included. In this case the fluid is initialized at a speed $M = .9$ and a shock forms on the cylinder and moves behind it. Von Karman vortices are formed in the wake. To capture these effects a much larger domain is used than in the inviscid case. The vorticity at $t = 70.0$ units shown in figure 4.18 were produced with the serial model and the results in 4.19 were produced with the parallel model utilizing six cores on an AMD Phenom 2. The resolution of this case was increased threefold in each direction, and the Ranger cluster from Texas Advanced Computing Center (TACC) was utilized to produce the results in figures 4.20 and 4.21. This case was ran on 192 processing cores.

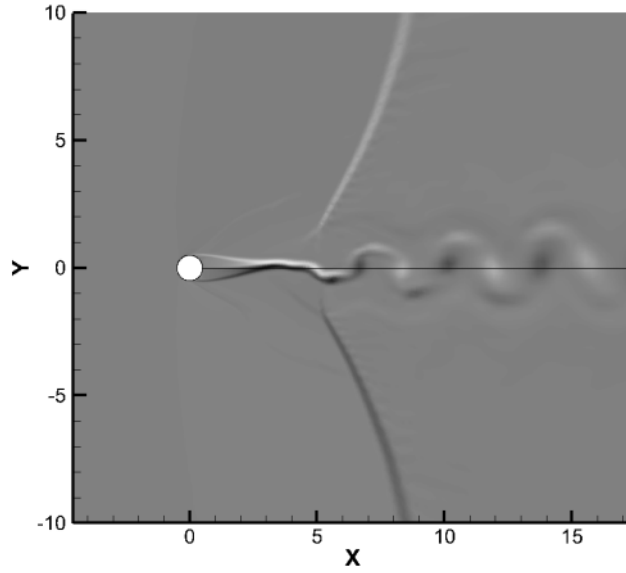


Figure 4.18 Serial Viscous High Speed Flow Over Cylinder

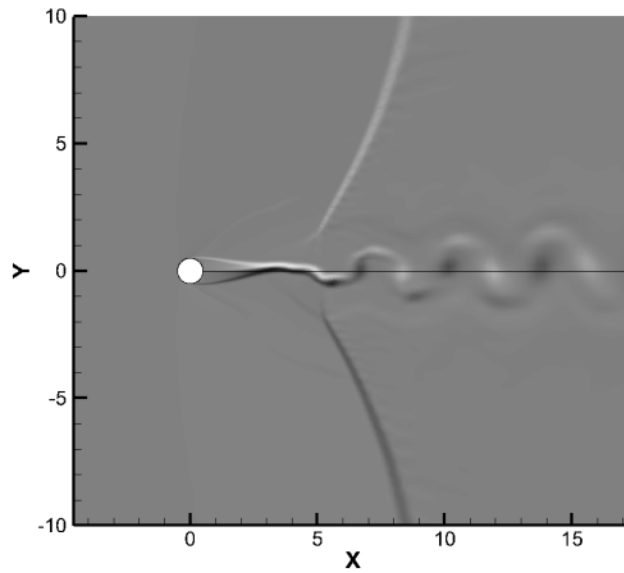


Figure 4.19 Parallel Viscous High Speed Flow Over Cylinder

As with the two other validation cases, the parallel model matches the serial model well. However, investigation of the dependence of grid resolution on flow pattern is ongoing.

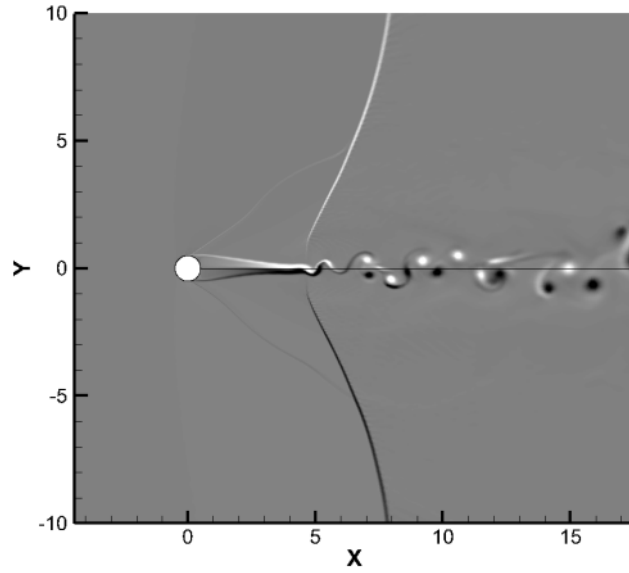


Figure 4.20 Viscous High Speed Flow Over Cylinder - High Resolution

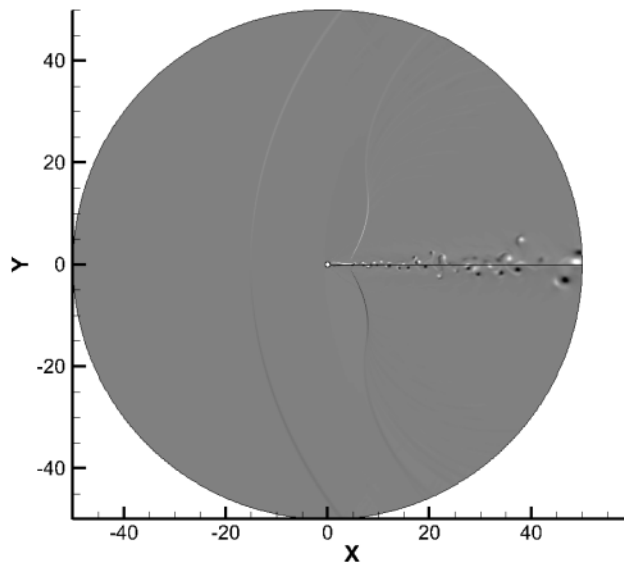


Figure 4.21 Viscous High Speed Flow Over Cylinder - High Resolution, Full Domain

The von Karman vortex shedding is much more refined in the higher resolution case. This case would have taken a serial processor close to a month and was done in around four hours in parallel.

4.2 Scalability

As mentioned in section 1.2.2, there are two types of scaling in parallel models: strong and weak. Strong scaling means that the model continues to scale well no matter how many processors are chosen for a specific problem. There are sufficient independent calculations available such that no matter how many processors are chosen improvement in the model can be seen. The WENO model is not considered a strong scaling model. Figure 4.22 depicts a strong scaling comparison for the WENO model. For a certain sized grid, the number of processors is increased. As the grid on each process becomes smaller, eventually the communication overhead is too large compared to the amount of calculations performed on each process.

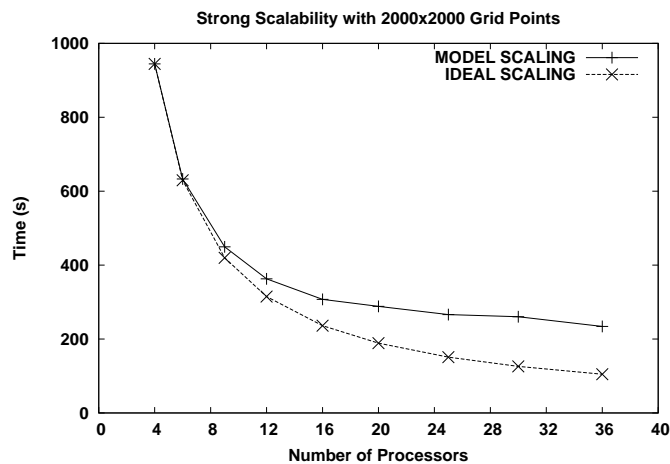


Figure 4.22 Notus WENO Strong Scaling

Finite-difference models such as the WENO are considered weak scaling models. Weak scaling refers to how a model scales by increasing the number of processors and the domain size for a problem proportionately. In practice, this is very useful. If a model scales well in this manner, the amount of time can remain the same for a problem with a much denser or larger grid. Figure 4.23 is an example of how the WENO model performs in a weak scaling comparison. In general scalability is defined by weak scaling [4].

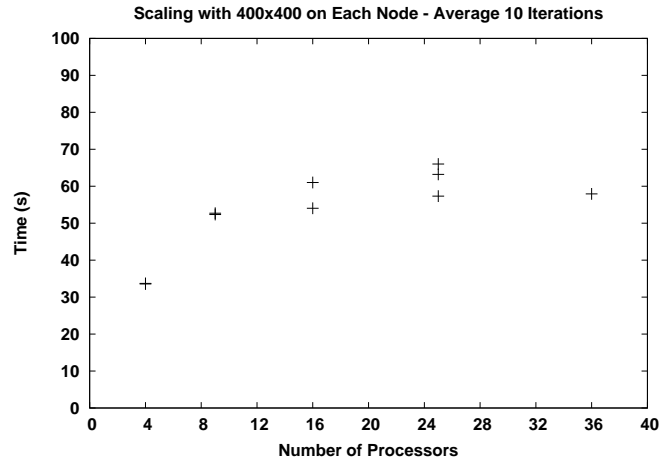


Figure 4.23 Notus WENO Weak Scaling

The scalability results in figures 4.22 and 4.23 were produced on the cluster called Notus purchased by Dr. Hoffmann's research group and managed by the High Performance Computing Center at Wichita State University. Notus is comprised of 45 dual processor compute nodes, each containing 4 GB of RAM.

File output choices can affect the parallel model's performance depending on the network interface of the computer cluster. If the destination of the output files is allocated on a networked file system, then the job's performance is dependent on the network load. Figure 4.24 compares run times with the model outputting data to a network file system to those in which the model has each node output its data to its own local scratch space. As can be seen, the network output in this case can be almost as good as the local file output, but varies from run to run because of the network load.

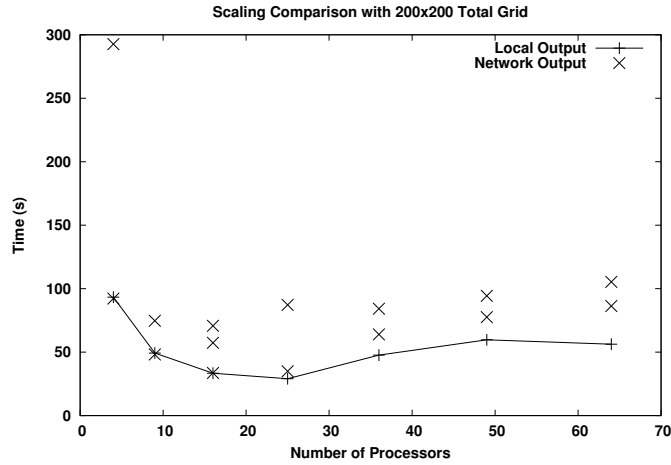


Figure 4.24 Network and Local File Output Comparison

The developed parallel WENO code was also tested on resources gained through an allocation with TeraGrid. The two resources tested on TeraGrid are Texas Advanced Computing Center (TACC)'s Ranger and Lonestar clusters. Ranger has approximately 63,000 computing cores and 123 terabytes of total memory. Lonestar has 5,200 computing cores and about 10 terabytes of total memory. These computer clusters also incorporate Infiniband network interconnects. Scaling results are shown in figures 4.25 and 4.26.

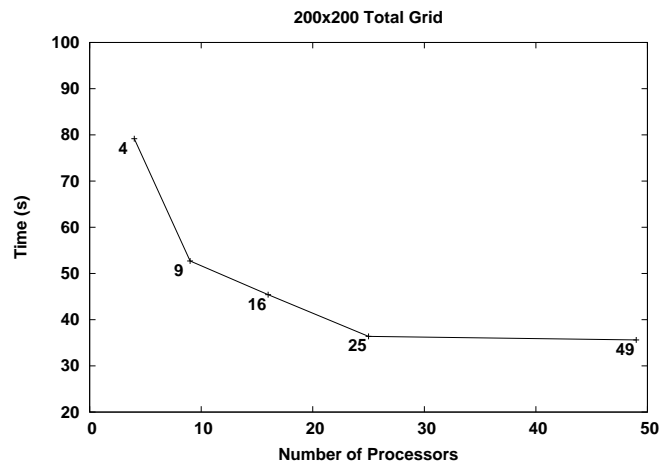


Figure 4.25 Lonestar WENO Strong Scaling

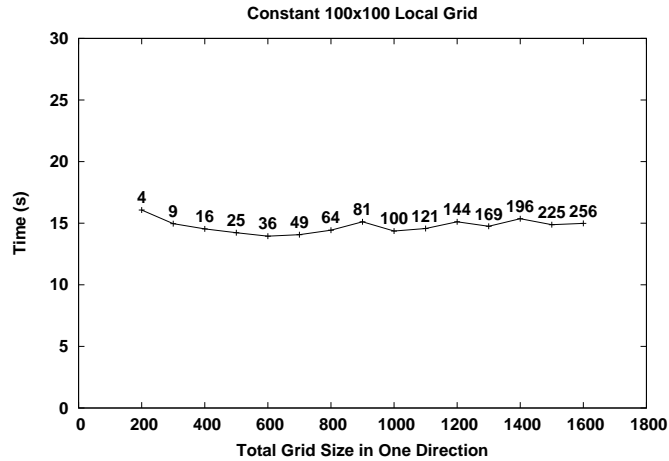


Figure 4.26 Lonestar WENO Weak Scaling Normalized

Figure 4.25 represents a strong scaling comparison. As seen in the other strong scaling comparison, there is a limit to the gains for a certain problem size. However, figure 4.26 shows an extremely promising weak scaling comparison. The same number of iterations on a grid 64 times as large can be completed in approximately the same amount of time if the computational resources are available. Although fluctuations along the scaling line exist most likely due to network load, this could be considered perfect weak scaling.

It is worth noting that the weak scaling comparisons are normalized to a constant number of iterations. This is the best way to analyze the performance of the parallelization. However, in actuality the model is still subject to the CFL condition. To achieve a runtime with more grid points in the same amount of time, the increase in grid points would have to correspond to an increase in the size of the physical domain and not an increase in the density of the grid. Figure 4.27 shows the actual time to completion of a run when the increase in total grid size corresponds only to increasing the density of the grid in an equally sized domain. The slope is linear, representing the linear decrease in time step due to the CFL condition. In serial the time increase would have been cubic due to the quadratic increase in number of gridpoints and corresponding calculations.

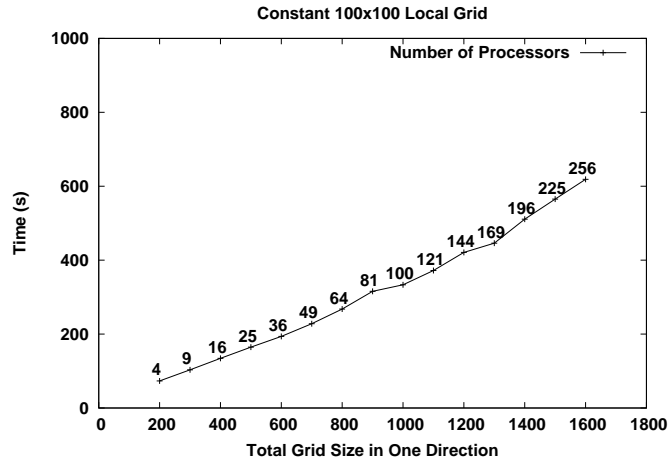


Figure 4.27 Lonestar WENO Weak Scaling

The parallel ADI model is a good example of a model exhibiting less than perfect scalability. Both figures 4.28 and 4.29 are strong scaling comparisons produced with the ADI model. In contrast to the WENO strong scaling comparisons, notice the slope of the line. There is some loss in time for the larger grids (and corresponding number of processes) due to the depth of the pipeline in the pipelining technique employed to solve the tridiagonal system of equations in parallel. The number of grid points per MPI process is held to four million in figure 4.28 and to 64 million in figure 4.29

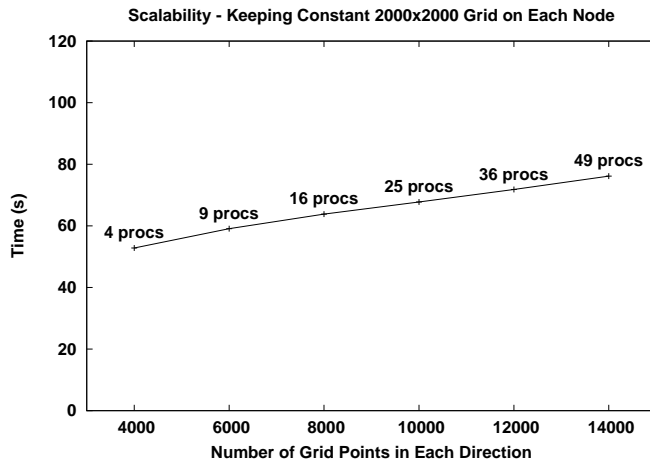


Figure 4.28 ADI Weak Scaling

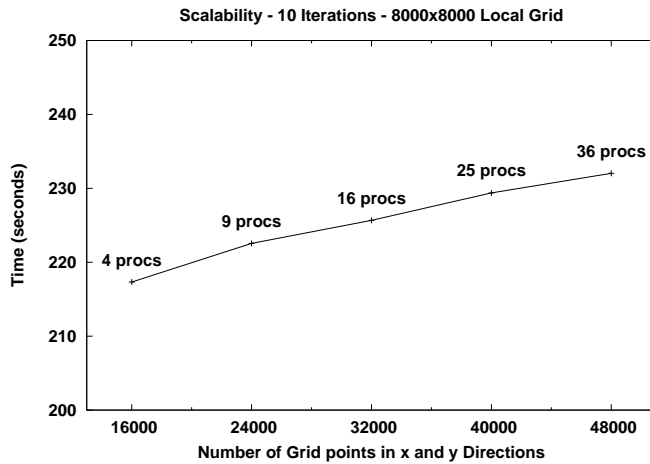


Figure 4.29 ADI Weak Scaling High Grid Density

Despite the shortcomings in scalability, the benefit of parallelization in the ADI model is still significant. The loss in parallel performance is less pronounced for the high grid density cases since the ratio of calculation to communication is much higher. It only takes 1.05 times as long

to perform ten iterations on a domain nine times larger in the second figure versus 1.4 times as long in the first. Furthermore, the ADI is an unconditionally stable algorithm, so much denser grids can be analyzed without decreasing the time step due to the CFL condition as in the WENO model.

The WENO model scales extremely very well to large problem sizes on a per iteration basis. The results shown in figure 4.23 from Notus are not as convincing as the results obtained on Lonestar in figure 4.26. The variance from one run to the next can be attributed in network load on the much more latent Gigabyte Ethernet than the Infiniband interconnects used on the Lonestar cluster. Also, an interesting data point is the one for four processors in figure 4.23. The lower time for this case versus the others can be attributed to the fact that non-periodic physical boundaries do not require communication, and with four MPI processes every sub-domain contains a physical boundary, thus reducing the amount of communication required on every MPI process. This effect does not show up on Lonestar due to the faster network and because all four MPI processes were likely on the same compute node, removing the need for communication across the network.

The compute times for the viscous cylinder case are of interest. Reportedly, it took around five days to perform the serial results on a Core 2 duo processor. In parallel the same case took 3.2 hours on an AMD Phenom 2 six-core processor. Granted, while the hardware is faster per core, this is still considerable speedup. Furthermore, the same case was ran on the new Aries cluster that Dr. Hoffmann's group purchased. The serial version took around 28 hours and the parallel version took around five. This is a speedup factor of 5.996 utilizing six cores for a scaling efficiency of 99.9%. This may be somewhat skewed as the parallel compiler may have been more efficient than the one used for the serial model. Also, the six cores existed on the same node, so the communication would have been much faster than if it was communicating across its Infiniband network. At the time of this writing restrictions prohibit the use of more than one node at a time on Aries to test the scaling efficiency using the network. Ranger was used to

run cases at both 3 times and 4 times the resolution. This would have increased the number of calculations by at least 27 and 64 times respectively on the serial model—squared for the spatial domain and at least linear due to the CFL condition. The results were produced in 4.43 hours on 192 processing cores and 4.97 hours on 384 processing cores respectively. These results would have taken months in serial.

4.3 Conclusions

Parallelization offers worthwhile advantages. Although the WENO model is still subject to the increase in iterations temporally mandated by the CFL condition, the increase in calculation time due to the spatial domain increase can effectively be nullified by utilizing more processing cores. This advantage holds true in the extension to three dimensions, as the increase of spatial calculations due to domain size increase will be cubic instead of quadratic and the increase in time steps will remain linear when compared to the two-dimensional model.

Today's available high performance computational resources allow the implementation of algorithms and models once deemed too computationally intensive. Although Amdahl made strong arguments against the need for parallelization, his viewpoint was short-sighted. He did not consider the need for larger problems to be as important as the need for computing problems faster. Direct Numerical Simulation for the solution of the Navier-Stokes is widely considered to be the state of the art method for computational fluid dynamics. Although the larger shared memory workstations today may handle most two-dimensional problems, the parallel techniques developed in two dimensions are needed in the extension to three-dimensional solvers. Without parallelization practical problems could not be computed in a timely manner, and depending on the availability of systems with large enough shared memory, perhaps not at all. In light of this and the scalability results presented here, the benefit of parallelization is clear.

4.4 Recommendations

The three-dimensional WENO model is under development in serial. The parallelization techniques used in the two-dimensional model could be extended to work in three dimensions. The size of the arrays loaded for message passing would need to be increased to store faces of a computational cubic domain instead of edges of a square domain. The initialization routines would be changed to break up the domain into three directions instead of two. The process topology would be defined based on three dimensions and the information about processes with neighboring sub-domains would be extended to match.

Reading from a single file for grid information and initial conditions was implemented for the ease of testing and using the model for multiple cases without having to program them in parallel. However, for very large domains, it may prove difficult to transfer or even create these initial condition files. In this case the initial conditions and grid information may need to be programmed into the model. Another option is to create a separate parallel model to produce a different initial condition file for each MPI process in the WENO model to read.

As domain size increases the size of the data files increase as well as the time it takes to write and transfer such files. Binary file formats require less space to store double precision values and many are designed to work efficiently with MPI and the underlying high performance network hardware to write the sub-domains into one file as it would be in serial. ASCII file format was chosen in the parallel model because of the familiarity in the research group with the Tecplot ASCII data format. However, for runs involving very large domains a binary file format such as NetCDF or HDF5 may prove to be more practical.

REFERENCES

REFERENCES

- [1] Ghulam M. Arshed and Klaus A. Hoffmann. Implementation of Improved WENO Scheme to Higher Dimensions in Relation to Shock-Turbulence Interaction. In *19th AIAA Computational Fluid Dynamics*, AIAA 3654. AIAA, 2009.
- [2] Lloyd D. Fosdick, Elizabeth R. Jessup, Carolyn J.C. Schauble, and Gitta Domik. *An Introduction to High-Performance Scientific Computing*. Scientific and Engineering Computation. The Massachusetts Institute of Technology, Cambridge, Massachusetts and London, England, 1996.
- [3] Ke-Di Zhange, Wei Shao, and Bing-Zhong Wang. Parallelized ADI-FDTD Algorithm for Attenuation Constant Extraction by Using OpenMP Library. In *ICMMT 2010 Proceedings*, pages 786–788. IEEE, 2010.
- [4] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White. *Sourcebook of Parallel Computing*. Morgan Kaufmann, San Francisco, CA, 2003.
- [5] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Spring Joint Computer Conference*. AFIPS, 1967.
- [6] Omar Ramadan, Oyku Akaydin, Muhammed Salamah, and Abduallah Y. Oztoprak. Parallel Implementation of the Finite-Difference Time-Domain Method on Local Area Network Using the Message Passing Interface. In *Information and Communication Technologies: from Theory and Applications*, pages 651+. IEEE, 2004.
- [7] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532+, 1988.
- [8] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison Wesley, 1994.
- [9] J.W. Anders, V. Magi, and J. Abraham. Large-Eddy Simulation in the Near-Field of a Transient Multi-Component Gas Jet with Density Gradients. *Computers and Fluids*, 36:1609–1620, 2007.
- [10] Joao Paulo De Angeli, Andrea M. P. Valli, Neyval C. Reis Jr., and Alberto F. De Souza. Finite Difference Simulations of the Navier-Stokes Equations using Parallel Distributed Computing. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*. IEEE, 2003.
- [11] Kurt Sermeus, Eric Laurendeau, and Farid Parpia. Parallelization and Performance Optimization of Bombardier Multiblock Structured Navier-Stokes Solver on IBM eserver Cluster 1600. In *45th AIAA Aerospace Sciences Meeting and Exhibit*, pages 1–24, 2007.
- [12] Aditi Ray, G. Kondayya, and S. V. G. Menon. Developing a Finite Difference Time Domain Parallel Code for Nuclear Electromagnetic Field Simulation. *IEEE Transactions on Antennas and Propagation*, 54(4):1192–1199, 2006.

REFERENCES (continued)

- [13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C++, The Art of Scientific Computing*. Cambridge University Press, New York, second edition, 2002.
- [14] Tomasz P. Stefanski and Timothy D. Drysdale. Large Scale ADI-FDTD Parallel Computations. In *Microwave Conference*, pages 1–4, 2008.
- [15] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference*, volume 1. Massachusetts Institute of Technology, second edition, 1998.
- [16] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard: Version 2.2. September 2009.
- [17] Hua Wang, Adnan Trakic, Ling Xia, Suart Crozier, Feng Liu, and Marek Bialkowski. A Parallel FDTD Scheme for Electromagnetic Analysis and Design of MRI System. In *Proceedings of the 28th IEEE EMBS Annual International Conference*, pages 264–267, 2006.
- [18] Josivaldo de S. Araujo, Ronaoldo O. Santos, Carlos L. da S. S. Sobrinho, Johnny M. Rocha, Luiz A. Guedes, and Regiane Y. Kawasaki. Analysis of Antennas by FDTD Method Using Parallel Processing with MPI. In *Proceedings SBMO/IEEE MTT-S IMOC*, pages 1049–1054, 2003.
- [19] Henry Neeman. Stupid Compiler Tricks. In *Supercomputing in Plain English*. Oklahoma University Supercomputing Center for Education and Research, 2009.
- [20] C.W. Shu and S. Osher. Efficient Implementation of Essentially Non-Oscillatory Shock Capturing Schemes, II. *Journal of Computational Physics*, 83:32–78, 1989.
- [21] Ghulam M. Arshed and Klaus A. Hoffmann. Implementation of Improved WENO Scheme for a Generalized Coordinate System. In *40th Fluid Dynamics Conference and Exhibit*, AIAA 5033. AIAA, 2010.

APPENDIX

APPENDIX A

MPI FUNCTIONS

A complete listing of the MPI functions used follows. The descriptions were taken from the MPI Forum standard [16] and is allowed under the following copyright notice.

©1993, 1994, 1995, 1996, 1997, 2008, 2009 University of Tennessee, Knoxville, Tennessee. Permission to copy without fee all or part of this material is granted, provided the University of Tennessee copyright notice and the title of this document appear, and notice is given that copying is by permission of the University of Tennessee.

APPENDIX A (continued)

MPI_INIT()

```
int MPI_Init(int *argc, char ***argv)
```

```
MPI_INIT(IERROR)  
    INTEGER IERROR
```

```
{void MPI::Init(int& argc, char**& argv) (binding deprecated, see Section 15.2) }
```

```
{void MPI::Init() (binding deprecated, see Section 15.2) }
```

All MPI programs must contain exactly one call to an MPI initialization routine: `MPI_INIT` or `MPI_INIT_THREAD`. Subsequent calls to any initialization routines are erroneous. The only MPI functions that may be invoked before the MPI initialization routines are called are `MPI_GET_VERSION`, `MPI_INITIALIZED`, and `MPI_FINALIZED`. The version for ISO C accepts the `argc` and `argv` that are provided by the arguments to `main` or `NULL`:

```
int main(int argc, char **argv)  
{  
    MPI_Init(&argc, &argv);  
  
    /* parse arguments */  
    /* main program    */  
  
    MPI_Finalize();    /* see below */  
}
```

The Fortran version takes only `IERROR`.

Conforming implementations of MPI are required to allow applications to pass `NULL` for both the `argc` and `argv` arguments of `main` in C and C++. In C++, there is an alternative binding for `MPI::Init` that does not have these arguments at all.

Figure A.1 MPI Function: `MPI_INIT`

MPI_FINALIZE()

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERROR)  
    INTEGER IERROR
```

```
{void MPI::Finalize() (binding deprecated, see Section 15.2) }
```

This routine cleans up all MPI state. Each process must call `MPI_FINALIZE` before it exits. Unless there has been a call to `MPI_ABORT`, each process must ensure that all pending nonblocking communications are (locally) complete before calling `MPI_FINALIZE`. Further, at the instant at which the last process calls `MPI_FINALIZE`, all pending sends must be matched by a receive, and all pending receives must be matched by a send.

Figure A.2 MPI Function: `MPI_FINALIZE`

APPENDIX A (continued)

```

MPI_SEND(buf, count, datatype, dest, tag, comm)
  IN      buf          initial address of send buffer (choice)
  IN      count        number of elements in send buffer (non-negative integer)
  IN      datatype     datatype of each send buffer element (handle)
  IN      dest         rank of destination (integer)
  IN      tag          message tag (integer)
  IN      comm         communicator (handle)

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

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
  <type> BUF(*)
  INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
{void MPI::Comm::Send(const void* buf, int count, const
                     MPI::Datatype& datatype, int dest, int tag) const (binding
                     deprecated, see Section 15.2) }

```

The blocking semantics of this call are described in Section 3.4.

Figure A.3 MPI Function: MPI_SEND

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Figure A.4 MPI & Fortran Data Type Comparison

APPENDIX A (continued)

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of receive buffer (choice)
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element (handle)
IN	source	rank of source or MPI_ANY_SOURCE (integer)
IN	tag	message tag or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (Status)

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

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE),
    IERROR
```

```
{void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
                    int source, int tag, MPI::Status& status) const (binding
                    deprecated, see Section 15.2) }
```

```
{void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& datatype,
                    int source, int tag) const (binding deprecated, see Section 15.2) }
```

The blocking semantics of this call are described in Section 3.4.

The receive buffer consists of the storage containing `count` consecutive elements of the type specified by `datatype`, starting at address `buf`. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

Figure A.5 MPI Function: MPI_RECV

APPENDIX A (continued)

`MPI_SENDRECV`(`sendbuf`, `sendcount`, `sendtype`, `dest`, `sendtag`, `recvbuf`, `recvcount`, `recvtype`, `source`, `recvtag`, `comm`, `status`)

IN	<code>sendbuf</code>	initial address of send buffer (choice)
IN	<code>sendcount</code>	number of elements in send buffer (non-negative integer)
IN	<code>sendtype</code>	type of elements in send buffer (handle)
IN	<code>dest</code>	rank of destination (integer)
IN	<code>sendtag</code>	send tag (integer)
OUT	<code>recvbuf</code>	initial address of receive buffer (choice)
IN	<code>recvcount</code>	number of elements in receive buffer (non-negative integer)
IN	<code>recvtype</code>	type of elements in receive buffer (handle)
IN	<code>source</code>	rank of source or <code>MPI_ANY_SOURCE</code> (integer)
IN	<code>recvtag</code>	receive tag or <code>MPI_ANY_TAG</code> (integer)
IN	<code>comm</code>	communicator (handle)
OUT	<code>status</code>	status object (Status)

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                int dest, int sendtag, void *recvbuf, int recvcount,
                MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
                MPI_Status *status)
```

```
MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
             RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

```
{void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const
                        MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,
                        int recvcount, const MPI::Datatype& recvtype, int source,
                        int recvtag, MPI::Status& status) const (binding deprecated, see
                        Section 15.2) }
```

```
{void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount, const
                        MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf,
                        int recvcount, const MPI::Datatype& recvtype, int source,
                        int recvtag) const (binding deprecated, see Section 15.2) }
```

Execute a blocking send and receive operation. Both send and receive use the same communicator, but possibly different tags. The send buffer and receive buffers must be disjoint, and may have different lengths and datatypes.

The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads.

Figure A.6 MPI Function: `MPI_SENDRECV`

APPENDIX A (continued)

`MPI_DIMS_CREATE(nnodes, ndims, dims)`

IN	<code>nnodes</code>	number of nodes in a grid (integer)
IN	<code>ndims</code>	number of Cartesian dimensions (integer)
INOUT	<code>dims</code>	integer array of size <code>ndims</code> specifying the number of nodes in each dimension

`int MPI_Dims_create(int nnodes, int ndims, int *dims)`

`MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)`

INTEGER NNODES, NDIMS, DIMS(*), IERROR

{void MPI::Compute_dims(int nnodes, int ndims, int dims[]) (*binding deprecated, see Section 15.2*) }

The entries in the array `dims` are set to describe a Cartesian grid with `ndims` dimensions and a total of `nnodes` nodes. The dimensions are set to be as close to each other as possible, using an appropriate divisibility algorithm. The caller may further constrain the operation of this routine by specifying elements of array `dims`. If `dims[i]` is set to a positive number, the routine will not modify the number of nodes in dimension `i`; only those entries where `dims[i] = 0` are modified by the call.

Negative input values of `dims[i]` are erroneous. An error will occur if `nnodes` is not a multiple of $\prod_{i, \text{dims}[i] \neq 0} \text{dims}[i]$.

For `dims[i]` set by the call, `dims[i]` will be ordered in non-increasing order. Array `dims` is suitable for use as input to routine `MPI_CART_CREATE`. `MPI_DIMS_CREATE` is local.

	dims before call	function call	dims on return
Example 7.1	(0,0)	<code>MPI_DIMS_CREATE(6, 2, dims)</code>	(3,2)
	(0,0)	<code>MPI_DIMS_CREATE(7, 2, dims)</code>	(7,1)
	(0,3,0)	<code>MPI_DIMS_CREATE(6, 3, dims)</code>	(2,3,1)
	(0,3,0)	<code>MPI_DIMS_CREATE(7, 3, dims)</code>	erroneous call

Figure A.7 MPI Function: `MPI_DIMS_CREATE`

APPENDIX A (continued)

```

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)
    IN      comm_old      input communicator (handle)
    IN      ndims         number of dimensions of Cartesian grid (integer)
    IN      dims          integer array of size ndims specifying the number of
                        processes in each dimension
    IN      periods       logical array of size ndims specifying whether the grid
                        is periodic (true) or not (false) in each dimension
    IN      reorder       ranking may be reordered (true) or not (false) (logical)
    OUT     comm_cart     communicator with new Cartesian topology (handle)

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
                   int reorder, MPI_Comm *comm_cart)

MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER

{MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[],
     const bool periods[], bool reorder) const (binding deprecated, see
     Section 15.2) }

```

MPI_CART_CREATE returns a handle to a new communicator to which the Cartesian topology information is attached. If `reorder = false` then the rank of each process in the new group is identical to its rank in the old group. Otherwise, the function may reorder the processes (possibly so as to choose a good embedding of the virtual topology onto the physical machine). If the total size of the Cartesian grid is smaller than the size of the group of `comm`, then some processes are returned `MPI_COMM_NULL`, in analogy to `MPI_COMM_SPLIT`. If `ndims` is zero then a zero-dimensional Cartesian topology is created. The call is erroneous if it specifies a grid that is larger than the group size or if `ndims` is negative.

Figure A.8 MPI Function: MPI_CART_CREATE

APPENDIX A (continued)

```
MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN      comm      communicator with Cartesian structure (handle)
IN      maxdims   length of vectors dims, periods, and coords in the
              calling program (integer)
OUT     dims      number of processes for each Cartesian dimension (ar-
              ray of integer)
OUT     periods   periodicity (true/false) for each Cartesian dimension
              (array of logical)
OUT     coords    coordinates of calling process in Cartesian structure
              (array of integer)

int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods,
                int *coords)

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
LOGICAL PERIODS(*)

{void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
                             int coords[]) const (binding deprecated, see Section 15.2) }
```

Figure A.9 MPI Function: MPI_CART_GET

APPENDIX A (continued)

```
MPI_CART_RANK(comm, coords, rank)
  IN      comm      communicator with Cartesian structure (handle)
  IN      coords    integer array (of size ndims) specifying the Cartesian
                   coordinates of a process
  OUT     rank      rank of specified process (integer)

int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
  INTEGER COMM, COORDS(*), RANK, IERROR

{int MPI::Cartcomm::Get_cart_rank(const int coords[]) const (binding
  deprecated, see Section 15.2) }
```

For a process group with Cartesian structure, the function `MPI_CART_RANK` translates the logical process coordinates to process ranks as they are used by the point-to-point routines.

For dimension `i` with `periods(i) = true`, if the coordinate, `coords(i)`, is out of range, that is, `coords(i) < 0` or `coords(i) ≥ dims(i)`, it is shifted back to the interval $0 \leq \text{coords}(i) < \text{dims}(i)$ automatically. Out-of-range coordinates are erroneous for non-periodic dimensions.

If `comm` is associated with a zero-dimensional Cartesian topology, `coords` is not significant and 0 is returned in `rank`.

Figure A.10 MPI Function: `MPI_CART_RANK`

```
MPI_COMM_RANK(comm, rank)
  IN      comm      communicator (handle)
  OUT     rank      rank of the calling process in group of comm (integer)

int MPI_Comm_rank(MPI_Comm comm, int *rank)
MPI_COMM_RANK(COMM, RANK, IERROR)
  INTEGER COMM, RANK, IERROR

{int MPI::Comm::Get_rank() const (binding deprecated, see Section 15.2) }
```

Figure A.11 MPI Function: `MPI_COMM_RANK`

APPENDIX A (continued)

```

MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm)
    IN      sendbuf      address of send buffer (choice)
    OUT     recvbuf      address of receive buffer (choice, significant only at
                        root)
    IN      count        number of elements in send buffer (non-negative inte-
                        ger)
    IN      datatype     data type of elements of send buffer (handle)
    IN      op           reduce operation (handle)
    IN      root         rank of root process (integer)
    IN      comm         communicator (handle)

int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
{void MPI::Comm::Reduce(const void* sendbuf, void* recvbuf, int count,
                      const MPI::Datatype& datatype, const MPI::Op& op, int root)
  const = 0 (binding deprecated, see Section 15.2) }

```

Figure A.12 MPI Function: MPI_REDUCE

```

MPI_BCAST( buffer, count, datatype, root, comm )
    INOUT   buffer      starting address of buffer (choice)
    IN      count       number of entries in buffer (non-negative integer)
    IN      datatype     data type of buffer (handle)
    IN      root        rank of broadcast root (integer)
    IN      comm        communicator (handle)

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
             MPI_Comm comm )
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
{void MPI::Comm::Bcast(void* buffer, int count,
                     const MPI::Datatype& datatype, int root) const = 0 (binding
                     deprecated, see Section 15.2) }

```

Figure A.13 MPI Function: MPI_BCAST

APPENDIX A (continued)

```

MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
    IN      sendbuf      starting address of send buffer (choice)
    IN      sendcount    number of elements in send buffer (non-negative integer)
    IN      sendtype     data type of send buffer elements (handle)
    OUT     recvbuf      address of receive buffer (choice, significant only at root)
    IN      recvcount    number of elements for any single receive (non-negative integer, significant only at root)
    IN      recvtype     data type of recv buffer elements (significant only at root) (handle)
    IN      root         rank of receiving process (integer)
    IN      comm         communicator (handle)

int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
              void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
              MPI_Comm comm)

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
          ROOT, COMM, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR

{void MPI::Comm::Gather(const void* sendbuf, int sendcount, const
    MPI::Datatype& sendtype, void* recvbuf, int recvcount,
    const MPI::Datatype& recvtype, int root) const = 0 (binding
    deprecated, see Section 15.2) }

```

Figure A.14 MPI Function: MPI_GATHER