Annals of the

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

http://www.csb.uncw.edu/mscsis/

EXTENDING SHARED-MEMORY PARELLELISM TO A DISTRIBUTED-MEMORY PARALLELIZING COMPILER

Lyndon K. Holt

A Capstone Project Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2011

Approved by

Advisory Committee

Dr. Gur Adhar	Dr. Douglas Kline

Dr. Clayton Ferner, Chair

Accepted By

Dean, Graduate School

Abstract

Extending Shared-Memory Parallelism to a Distributed-Memory Parallelizing Compiler. Holt, Lyndon, 2011. Capstone Paper, University of North Carolina Wilmington.

A source-to-source parallelizing compiler which generates parallel code for distributed-memory systems is modified to support statements for expressing shared-memory parallelism. The multi-core push is changing commodity clusters and high-end computing systems to include denser nodes containing more processors with shared-memory. This has drawn increased interest in hybrid parallel programming of these systems. The result of this compiler modification allows valid hybrid parallel programs to be produced from high-level statements guiding two forms of parallelism from the sequential source. Further implications and a detailed description of the implementation of this optimization are discussed. The modified compiler was validated by compiling two sequential sources containing high-level distributed-memory and shared-memory statements which perform matrix multiplication and Sobel edge detection. Performance timings are collected on the generated hybrid parallel source and distributed-memory only parallel source.

Table of Contents

Figures

Chapter 1. Introduction

Within the past decade chip vendors have revolutionized hardware architectures by coordinating multiple processor cores on a single die. These changes, though currently effective at increasing transistor counts, have tremendous repercussions in terms of programmability to achieve wall-time speedups. Furthermore, as the number of cores increase, it will become increasingly vital for software to exploit shared-memory parallelism. High-performance computing (HPC) applications are also affected by a multi-core era since future high-end computing systems will certainly encase interconnections of multi-core chips. There will be a greater concern to utilize shared-memory parallelism available within the multi-core chips of a HPC or cluster system.

Currently, parallel programming for HPC systems and clusters, particularly at a level just above the operating system, is a painstaking task. This low-level or raw form of parallel programming is seen in models such as MPI or Pthreads. At this level programmers must manage every detail in developing parallel algorithms. Consequently, there is a greater burden in development and testing to ensure software reliability with raw models. Parallel programs expressed with raw models add additional complexity to sequential implementations which can hinder development and maintainability. Raw parallel programming models also are sometimes more difficult to apply to existing sequential programs without modifying the algorithm.

This research focuses on an optimization of a distributed-memory parallelizing compiler. Distributed-memory systems are networks of disparate nodes or processing elements where each processor has its own memory. The Message-Passing Interface (MPI) standard provides a means of programming on these systems. Data is shared by inter-process messages over a network interconnect. In contrast, multi-core processors have access to a global address space referred to as shared-memory. The OpenMP standard provides a way to program shared-memory systems with threads.

In this research, the *Paraguin* compiler, a distributed-memory parallelizing compiler pass was

modified to parse and redirect compiler hints for shared-memory parallelism to output MPI parallel source. The result of such a modification allows a sequential source expressed with Paraguin hints to also include hints for shared-memory parallelism and thereby producing *hybrid* parallel code from sequential code. Hybrid parallel programs are programs which use two or more parallel programming models such as MPI and OpenMP. The compiler hints specified for both Paraguin and OpenMP are provided through the use of pragma (`#pragma`) statements.

The modified Paraguin pass was tested for correctness along with a performance analysis by expressing Paraguin and OpenMP pragmas within two candidate sequential programs and compiling them through the source-to-source Paraguin compiler. Such a modification of the Paraguin experimental compiler is important because *hybrid* parallel programs can be expressed with no impact to an existing sequential source. Since the parallelism is guided by compiler-specific statements known as pragmas, compilers which do not recognize the pragma statement simply ignore them. Therefore, the sequential program with pragma statements can still be compiled using a native compiler without affecting the ability to produce a sequential executable.

Based upon current research, it is still an open problem if the hybrid model parallel programming produces parallel programs which perform better than the single model parallel programs prior to execution. This notion as well as the multi-core push by chip vendors gives purpose in developing tools to help conceive hybrid parallel programs quickly and noninvasively. Nonetheless, comparing the performance of hybrid and non-hybrid parallel programming models is beyond the scope of this research. See [3,5,9,15] for discussions of the debate between hybrid and non-hybrid models.

The subsequent chapters are organized as follows: Chapter 2 discusses literature and background surrounding hybrid parallel programming and the Paraguin compiler, Chapter 3 discusses Paraguin's current implementation and how it was modified, Chapter 4 describes the implementation,

Chapter 5 verifies correctness and shows performance for two input source programs, and Chapter 6 concludes this research and mentions further work.

Chapter 2. Literature and Background

2.1 Parallelizing Compiler Overview

Automatic parallelization is an ambitious high-level technique especially realized at the compiler-level for dealing with the development of parallel programs. It involves detecting a particular pattern of code which can be transformed to make use of parallelism and then carrying out these transformations. However, due to the nature of parallel execution and particularly difficult sub-problems in this area, some parallel programming models make use of hints specified by the programmer within the source which guide compiler parallelization for certain language constructs such as `for` loops. OpenMP compilers and the Paraguin compiler are examples of such which can generate parallel code for shared-memory systems and for distributed-memory systems respectively, from compiler hints.

Parallel programming models which use source hints to guide parallelization support the concept of incremental parallelism. Incremental parallelism is the application of stepwise changes to existing code which adds parallelism. Thus hybrid parallel programmers can reason about the decomposition of an algorithm for distributed-memory parallelism and shared-memory parallelism in isolation and also apply parallelism as a series of incremental changes to sequential code. Incremental parallelism has further implications in that a single source program can represent a sequential program, distributed-memory parallel program, shared-memory parallel program, or hybrid parallel program simply by changing compilers or compiler options.

2.1.1 OpenMP Compilers

OpenMP is a shared-memory parallel programming model with a mature specification and industry-wide support [18]. The OpenMP model differs greatly from existing parallel programming models by providing a higher level of abstraction using compiler directives, or pragmas, to express parallelism compared to a raw model like Pthreads or MPI. The model works by using an OpenMP-supported compiler to generate native parallel code from API-documented pragmas and function calls. OpenMP also includes a runtime system which launches and controls parallel execution. Thus OpenMP pragmas can be used directly in source code and compiled by an OpenMP-supported compiler which generates a shared-memory parallel, or multithreaded, executable.

2.1.2 Paraguin Compiler

Paraguin is a source-to-source distributed-memory parallelizing compiler developed originally by Dr. Clayton Ferner at UNC-Wilmington. Paraguin's conception was influenced by the OpenMP model where pragma hints from the sequential source are used to guide parallelization. These hints target parallelism in `for` loops where the bulk of computations can occur for many applications. The Paraguin compiler generates MPI code which adds distributed-memory parallelism to the sequential source resulting in a new source-level program suitable for a final compilation. However, since Paraguin uses the MPI model, processes must communicate the dependent data that would be necessary for computations via messages. Thus Paraguin also includes support for broadcasting as well as finer granularity by specifying data dependences that exist in `for` loop computations.

2.1.3 Definition of terms

pragma (hints) - Compiler-specific statements in source code; ignored by compiler if unknown

process – task unit in distributed-memory system; cannot share memory by default

thread – task unit in shared-memory system; shares access to memory

Tile/partition - A partition of an iteration space containing one or more iteration instances

OpenMP-only program – parallel program using OpenMP code only

MPI-only program – parallel program using MPI code only

hybrid program – program using more than one form of parallelism

intermediate representation (IR) – compiler-specific abstraction of a language in a multi-phase compiler

source-to-source compiler – a compiler which reads input at the source level and produces output at the source level

2.2 Hybrid Parallel Programming

Programs which execute using two or more forms of parallelism are known as hybrid parallel programs. Hybrid parallel programming currently involves integrating two or more parallel programming models such as MPI and OpenMP in a single source. The ideal result of programming using more than one means of parallelism is a decreased wall-time of computations or the ability to solve greater size problems in the same amount of time.

Most research in hybrid parallel programming is from the current decade and current research suggests it does not guarantee speedup for all applications. Reasoning about the performance of MPI programs relative to Hybrid MPI/OpenMP programs in general is beyond the scope of this research.

2.2.1 Motivation

Parallel programming has for some time been utilized in the area of scientific computing on forms of distributed-memory processors. However, due mainly to the power wall, chip vendors are now creating denser chipsets with increasingly more processor cores sharing memory and caches. Consequently, today's HPC systems and clusters contain more processing elements within the node. Many applications in this area use the MPI-only model for parallelism across a HPC system or cluster. More effective forms of parallelism need to be investigated and perhaps utilized within the node for certain applications to maximize efficiency. MPI implementation designers have recognized this

notion stating that MPI alone "does not make the most efficient use of the shared resources within the node of a HPC system [1]."

Although some current MPI implementations can make use of shared-memory, inter-process shared-memory communication with MPI is considered less efficient than synchronization of threads in a shared-memory model. As node resources become denser, slower shared-memory communication in MPI could hinder potential wall-clock time of executions. In effect, the MPI-only parallel program may not be the most scalable solution possible for certain problems.

Increased complexity of hybrid parallel programming over traditional single-level parallel programming such as MPI-only is also another issue. Research conducted at the University of Tokyo on three-level hybrid parallel programs running on a large SMP cluster determined that it was inconclusive whether the extra development effort outweighed the perceived benefit [3]. Tools which ease the development burden of expressing multiple levels of parallelism can help in this regard by quickly adapting parallel programs to suit the target computing environment.

2.2.2 Current Efforts

Many current efforts in hybrid parallel programming involve measuring performance. The popularity of clustering symmetric multi-processor (SMP) nodes drove the adaptation of the NAS Parallel Benchmarks known as Multi-Zone to utilize hybrid parallelism [4]. The Multi-Zone suite includes MPI/OpenMP hybrid parallel programs. Hybrid parallel programs using MPI and OpenMP have been developed mainly in the past decade with mixed results comparing performance relative to MPI-only parallel versions [3,5,9]. For instance, hybrid execution in sparse matrix-vector multiplication was outperformed by MPI-only with increasing problem sizes and nodes due to increased communication overhead in the hybrid model [9]. Further, research on a large scale supercomputer showed that hybrid performance of the NAS parallel benchmarks scaled equivalently to

MPI-Only up to 1024 nodes when MPI communication was used across nodes and sockets and OpenMP within sockets [5].

Improvements to existing parallel programming models such as MPI and OpenMP implementations and hardware support are affecting hybrid performance. Early research in hybrid performance using MPI and OpenMP was less promising than recent research due to limited hardware support with shared-memory locking and immature OpenMP implementations [2]. Also, significant effort has been placed in developing thread-safe MPI implementations. Thread-safe MPI is important in cases where multiple threads execute MPI calls concurrently [1,2]. The OpenMP standard has provided more constructs for expressing shared-memory parallelism in subsequent versions.

Little research has been done in the area of compilers for hybrid parallel programming using MPI. This is partly due to the undetermined perceived benefit of MPI-hybrid parallel programming and an effort to model performance [9,10]. The idea of using compiler hints to generate MPI code in a fashion similar to OpenMP compilers with shared-memory code is a relatively new idea and an even more challenging problem.

There are few examples of compilers which can currently make use of a higher-level statement to generate distributed-memory MPI programs. Intel's Cluster OpenMP is a similar concept which can generate distributed-memory parallel programs from standard OpenMP directives by using a distributed-shared memory (DSM) subsystem. However, DSM models add significant overhead compared to MPI due to maintaining global memory coherence and more importantly, does not combine two parallel programming models to produce hybrid programs [7].

A similar effort to Paraguin is the *llc* source-to-source compiler which transforms OpenMP-like hints to parallel programs using MPI [8]. However, the methodology to code generation in *llc* is different from Paraguin and no claim is made to produce valid hybrid code from the sequential source with *llc* and OpenMP statements. My research is novel in that OpenMP statements are directed to the

appropriate location in the Paraguin compiler pass to produce a valid hybrid parallel program source. Thus in addition to compiling a sequential source to generate MPI-only and OpenMP executables similar to *llc*, the source can also be compiled as a valid hybrid MPI/OpenMP executable.

2.3 Paraguin Parallelization Techniques

Paraguin applies transformations and generates additional code to the sequential source which enables distributed-memory parallelization of `for` loops. Parallel regions are defined by the programmer which specifies the sections of code to be executed by all processes. Additional code generated from Paraguin involves MPI communication. At a minimum for data parallel problems, this includes calls to broadcast data needed by each process and gather the results computed by each process. The Paraguin `forall` pragma can be specified to target the main computation `for` loop to be executed by multiple MPI processes. This loop is tiled based on the partitioning supplied by the programmer in the `forall` pragma so that each process can execute a separate partition of the loop's iteration space.

Additional communication code in send and receive blocks are generated as specified by the dependence (`dep`) pragma. The `dep` pragma is used when there exist dependencies between iteration instances, or loop-carried dependencies. If attempting to execute a `for` loop in parallel with loop-carried dependencies, the `dep` pragma must be defined in the sequential source for the execution to be correct. The focus of this research is in applying OpenMP statements from the sequential source containing the Paraguin `forall` pragma to algorithms with no data dependencies between partitions. Thus there is no communication/synchronization needed among MPI process partitions to ensure that each process has the correct data needed to do its work. Effectively the class of problems in this research to which hybrid parallelization is applied is known as *embarrassingly parallel.* Embarrassingly parallel problems where little to no communication is needed beyond the initial distribution of data each MPI process is to compute and the final gathering of partial results. This

scenario creates a clear distinction between the duties of MPI and OpenMP in computations. Furthermore, parallel algorithms which have significant communication between partitions are not well suited for distributed-memory systems as demonstrated in parallel Gaussian elimination and LU decomposition [19,20].

2.3.1 Loop Tiling

The significant parallel computation transformation involves tiling the target loop to be parallelized. The target loop is most often the outermost loop in a loop nest to which a `forall` pragma is applied. Tiling of the outermost loop in a loop nest is desirable because it usually provides the coarsest granularity. The programmer defined partitioning determines how iterations are partitioned. The tiling transformation carries out this process by splitting the target loop's iteration space into blocks known as partitions. These partitions are mapped to physical processors at runtime. Consider the naïve matrix multiplication algorithm show in *Figure 1*. There exists no data dependence relation between statements of the loop; therefore, no loop-carried data dependencies exist for this algorithm.

```
#pragma paraguin forall    C       p        i      j      k \
                          0x0     -1        1    0x0    0x0 \
                          0x0      1       -1    0x0    0x0
for (i = 0; i < N; i++) {
     for (j = 0; j < N; j++) {
         c[i][j] = 0.0;
         for (k = 0; k < N; k++) {
             c[i][j] = c[i][j] + a[i][k] * b[k][j];
         }
     }
}
```

Figure 1*: Naïve square matrix multiplication of matrices a and b*

As such, partitioning on the outermost *i* loop can be specified so that iterations of this loop will be executed at different partitions. The mapping of partitions to processors is currently done as a block mapping. Each block is to contain approximately an equal amount of work as determined by the block size. The block size is calculated by dividing the loop iteration count by the number of processors. Iterations which do not divide evenly will force the last processor(s) to execute a smaller partition as

determined by the minimum of the last iteration and each processor iteration. This ensures that the *i* loop does not execute beyond its original iteration space [11]. Figure 2 shows the automatically generated parallelized program from the source in Figure 1.

```
...
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &__guin_NP);
MPI_Comm_rank(MPI_COMM_WORLD, &__guin_mypid);
__guin_blksz = __suif_divceil((N - 1) - 0 + 1, __guin_NP);
if (0 <= __guin_mypid & __guin_mypid <= -1 + 1 * __guin_NP)
{
    for (__guin_p = 0 + __guin_mypid * __guin_blksz; __guin_p <=
__suif_min((N - 1), 0 + (__guin_mypid + 1) * __guin_blksz - 1); __guin_p++)
    {
          i = 1 * __guin_p;
          for (j = 0; j <= (N - 1); j++)
          {
               c[i][j] = 0.0F;
               for (k = 0; k <= (N - 1); k++)
               {
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
               }
           }
     }
}
...
```

***Figure 2**: Tiled matrix multiplication of matrices a and b including generated code showing partition mapping*

2.3.2 Determining Partition Loop

Determining the partition loop in the tiled loop is important for the purpose of redirecting OpenMP statements to target the partition loop. Because the partition loop defined with index *__guin_p* is generated as part of the Paraguin pass, OpenMP application to this loop must be integrated as part of the tiling process.

The Paraguin compiler constructs a system of linear inequalities in matrix form derived from the affine expressions of the loop variables and constants. In Paraguin's current implementation, the partitioning is specified as a matrix form of a system of linear equalities and creates the assignment of iterations to partitions known as the computation decomposition [12]. To partition on the *i* loop as above for matrix multiplication, the assignment of iterations to partitions would be $p = i$. This

assignment statement is synonymous with the two linear inequalities $p \leq i$ and $p \geq i$. Moving all variables to the left-hand side yields $-p + i \geq 0$ and $p - i \geq 0$. Now these inequalities can be expressed in matrix form with all loop indices and represent the assignment statement below:

$$\begin{pmatrix} 0 & -1 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ p \\ i \\ j \\ k \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

The system of inequalities above is encoded within the pragma using the form in Figure 3.

```
C     p     i     j     k \
0x0   -1    1     0x0   0x0 \
0x0   1     -1    0x0   0x0
```

***Figure 3**: Matrix system representing p = i as specified with a paraguin forall construct; 0x0 is zero in hexidecimal and backslash \ delimits a new row*

The effect of this partitioning as it applies to the iteration space in matrix multiplication is illustrated below in *Figure 4*. Blocks referencing the rows of matrix *a* are assigned to each partition. The size of the partition which is the number of rows in a block is computed and assigned to physical processors at runtime.

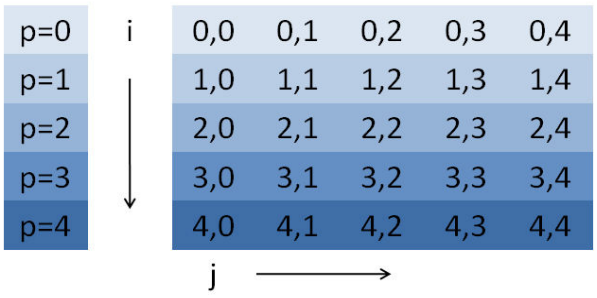

***Figure 4**: Partitioning of loop **i** on an iteration space*

2.4 SUIF Compiler Infrastructure

Paraguin is a compiler pass built from the open-source SUIF compiler infrastructure developed at Stanford University. The SUIF compiler is important because it provides a source-to-source compiler platform for research in compiler technology and thus facilitates the development of experimental passes without the investment required in developing a compiler. Paraguin leverages the SUIF kernel and toolkit within its pass to perform code generation and transformations for MPI parallelization. Consequently, it is important to understand how SUIF represents source code and how this representation can be traversed and manipulated to direct OpenMP directives to the output source.

The SUIF kernel defines the intermediate representation (IR) also known as SUIF code, functionality for accessing and manipulating the IR, and the compiler pass interface. The Paraguin compiler uses the ANSI C frontend and backend available as part of the SUIF toolkit to translate C source into high-level SUIF code and back.

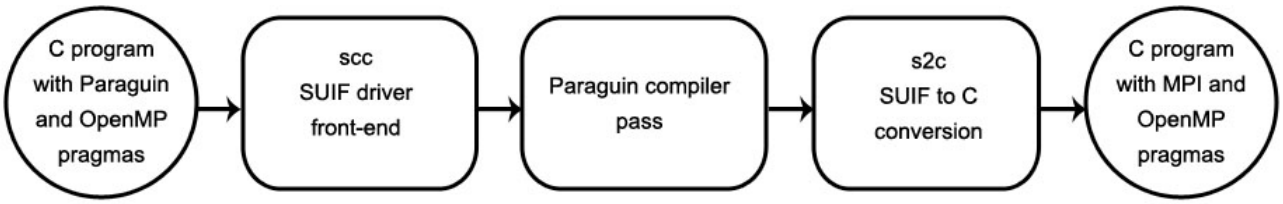

***Figure 5**: The Paraguin Compiler pass within the SUIF compiler architecture*

2.4.1 SUIF Code

SUIF code is the IR that the Paraguin pass interprets and manipulates to perform code transformations. SUIF code is a language-independent form of abstract syntax trees (AST), or high-level SUIF, and is implemented in a hierarchy of object-oriented tree data structures. Any point on the tree is referred to as a node. The leaves of all SUIF tree structures are instruction nodes which can

derive lists of assembly-like instructions known as low-level SUIF code. Paraguin's tiling method is performed on high-level SUIF.

A SUIF compiler pass typically works by accessing all procedure bodies of the input source. This is done by iterating the procedure symbols accessed through the file set entries at the root of the SUIF hierarchy as in the case of Paraguin. Procedure bodies are tree node lists, or doubly-linked lists of tree nodes. Tree nodes are derived from a base `suif_object` class and are the representation of a parse tree or abstract syntax trees (AST). The children of most tree nodes are accessed through tree node lists. The ability to search arbitrary depths of tree nodes is critical because the input source is not known until compile time. Recursion allows tree nodes to be searched at arbitrary depths. A loop nest is stored as a `tree for` node whose body is a tree node list containing another `tree for` node.

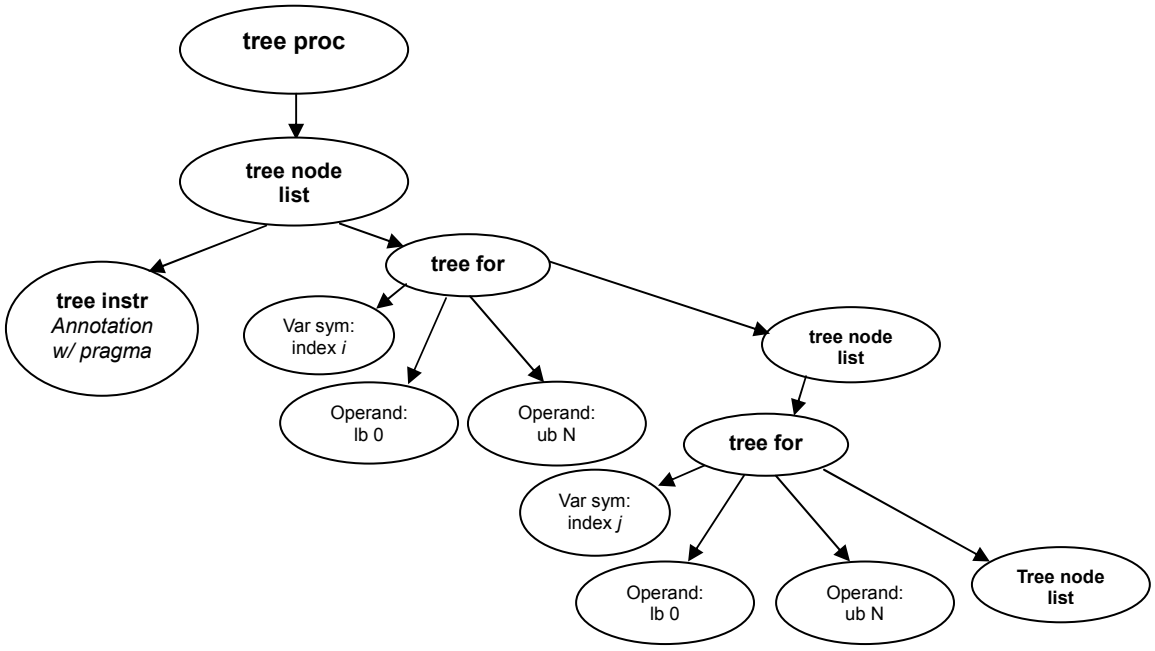

***Figure 6**: High-level SUIF code for a procedure containing a pragma statement followed by a nested loop; procedure bodies are represented by abstract syntax trees*

2.4.2 Annotations

SUIF annotations are containers for arbitrary data structures and the default object type for storing and manipulating pragma statements. Annotations are `annote` objects with a name and data pointer field. Annotations can be used to transfer data between SUIF passes, or as markers in the SUIF code. For Paraguin, annotations are the storage mechanism for parallelization pragma hints. All pragmas in the sequential source are parsed and read as an annotation by SUIF in the Paraguin pass. Annotations exist in the tree hierarchy through tree node objects inheriting from `suif_object`. Tree nodes have lists of associated annotations which are implemented as annotation lists. Pragma statements specified in the source by the programmer are attached to SUIF code as tree instruction nodes. Prior revisions of Paraguin simply ignore pragmas in the source input not specified as Paraguin-specific by `#pragma paraguin`. Pragmas which are to exist in the output source must be renamed as an `s2c_pragma` to be retained in the s2c translation. The `set_name` method is called to change the name of an annotation.

Chapter 3. Pre-Implementation Analysis and Scope

The main objective of this research is to produce valid hybrid parallel programs from a sequential source using pragma statements which guide the parallelization. Pre-implementation analysis includes defining the specific requirements necessary to produce hybrid source code and determining a baseline for what needs to be done to complete the requirements. Defining the scope is also necessary in determining the e*x*pressiveness of shared-memory parallelization the optimization will allow in context with Paraguin (MPI) parallelization.

3.1 Combining OpenMP w/ Paraguin

A study sequential program which can currently be parallelized by Paraguin is used in determining the most logical way to express hybrid parallelization by including OpenMP pragmas in a sequential source with Paraguin pragmas. In the current study programs, hybrid parallelization in `for` loops with loop carried-dependencies are disregarded. This presumption means that no MPI communication takes place within an OpenMP parallel region and makes this optimization much more tractable. In effect, there is a clear distinction between the duties of MPI processes and OpenMP threads. When data dependencies exist across loop iterations, MPI processes must send messages. If threads are executing within each process, these threads must synchronize to perform the MPI communication. Additionally, as mentioned in section *2.2.2 - Current Efforts,* MPI parallelization of loop nests with many loop-carried dependencies in a distributed-memory system yields little gain or even worse performance than its sequential version.

For this research, we have confined the support for hybrid programming to the `hybrid masteronly` model. In `hybrid masteronly`, no threads in an OpenMP parallel region use MPI communication [13]. This hybrid model also is commonly run using a single MPI process per node and OpenMP threads to compute on the node's cores. A typical structure of the hybrid code is shown in *Figure 7* with the outer loop as the target for the parallelization.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &NP);
MPI_Comm_rank(MPI_COMM_WORLD, &mypid);
blocksize = ceil(N/NP);
#pragma omp parallel for …
for ( partition = mypid*blocksize … min(N-1,(mypid+1)*blocksize-1) )
{
      I = f(partition);
      … /* No MPI communication in body */
      for (J = 1 .. N )
      {
            /* computations on array */
      }
}
```

***Figure 7**: Pseudocode of tiled loop with OpenMP where f( ) is an affine function on the partition*

The master thread within the MPI process encounters the outer `for` partition loop which is a tiling of the original sequential loop. The partition loop bounds are determined at runtime just before entering the OpenMP parallel region so the iterations for each MPI process are unique. The OpenMP `parallel for` construct produces parallel code that interacts with the OpenMP runtime library to determine a scheduling of each partition to threads at runtime. Data parallel problems such as matrix multiplication are good candidates for applying OpenMP to the generated MPI source with a `hybrid masteronly` decomposition.

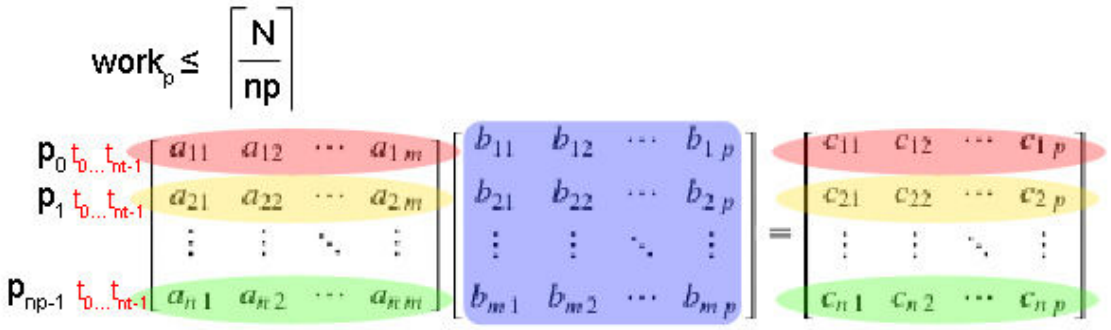

***Figure 8**: Conception of the work distribution of block-row decomposition of matrix multiplication in hybrid MPI/OpenMP. All processes receive the b matrix and each process is allocated a block and a team of threads further divides each block to compute independently.*

Paraguin parallel regions surround loop nests to be executed by all processes using `paraguin begin_parallel` and `paraguin end_parallel`. Within the loop nest are pragmas relevant

to performing MPI parallelization. The `forall` pragma is specified above the target loop nest and specifies the iteration space partitioning. A broadcast using `bcast` is necessary when data held by a master process must be distributed across all processes for computation. Gather communication using the `gather` pragma then sends process results to a single process by specifying which iterations of the loop variables produce the final values of the array. In the matrix multiplication example, we want to gather all values in c[i][j], thus the matrix representation of assignment k = 0 is sufficient. After the `paraguin` statements in the region, the programmer should be able to specify an OpenMP pragma just above the `for` loop nest. This location is important because the Paraguin pass is executed before the OpenMP pass and the OpenMP `parallel for` statement must immediately precede the target loop nest to be valid to an OpenMP compiler. Also by keeping with this rule, the sequential source can be compiled directly by an OpenMP compiler (ignoring paraguin pragma statements) to produce a multithreaded executable suitable for shared-memory systems.

```
...
 #pragma paraguin begin_parallel
 #pragma paraguin forall    C       p        i      j      k \
                          0x0      -1        1    0x0    0x0 \
                          0x0       1       -1    0x0    0x0

 #pragma paraguin bcast a b

 #pragma paraguin gather   1   C      i      j     k \
                             0x0    0x0    0x0    1 \
                             0x0    0x0    0x0   -1

  #pragma omp parallel for private(i,j,k) schedule(static) num_threads(NUM_THREADS)
  for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
          c[i][j] = 0.0;
          for (k = 0; k < N; k++) {
              c[i][j] = c[i][j] + a[i][k] * b[k][j];
          }
      }
  }

 ;
 #pragma paraguin end_parallel
...
```

Figure 9*: An example of sequential matrix multiplication combining Paraguin (MPI) and OpenMP pragma statements. This research effort consists of compiling this with Paraguin to produce valid hybrid parallelization of the loop nest. As a result using unique compilation which ignores certain pragmas makes this source sequential, OpenMP-only, MPI-only, and hybrid parallel source.*

An OpenMP `parallel for` pragma should be able to be applied to the same loop that a `paraguin forall` pragma is also applied. This case arises when it is desirable to split the iteration space, first among the set of nodes on a cluster with MPI and then again for each processor on the node with OpenMP threads. Paraguin creates a new loop which nests the target loop by mapping tiles (loop iterations) to partitions. This new tiled loop must now have the OpenMP `parallel for` pragma applied to it in the final source. The effect of this transformation is shown below by revisiting the output source of compiling matrix multiplication with Paraguin.

```
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &__guin_NP);
 MPI_Comm_rank(MPI_COMM_WORLD, &__guin_mypid);
 ...
 MPI_Bcast((float ((*)[N])[N])a, 1044484, MPI_BYTE, 0, MPI_COMM_WORLD);
 MPI_Bcast((float ((*)[N])[N])b, 1044484, MPI_BYTE, 0, MPI_COMM_WORLD);
 __guin_blksz = __suif_divceil(N - 0 + 1, __guin_NP);
 if (0 <= __guin_mypid & __guin_mypid <= -1 + 1 * __guin_NP)
 {
      #pragma omp parallel for private(i,j,k) schedule(static) num_threads(4)
      for (__guin_p = 0 + __guin_mypid * __guin_blksz; __guin_p <= __suif_min((N-1), 0 +
(__guin_mypid + 1) * __guin_blksz - 1); __guin_p++)
      {
            i = 1 * __guin_p;
            for (j = 0; j <= (N-1); j++)
            {
                  c[i][j] = 0.0F;
                  for (k = 0; k <= (N-1); k++)
                  {
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
                  }
            }
      }
 }
 ...
```

***Figure 10**: An example of valid hybrid parallel source produced from the source shown in Figure 9 - The original outermost i loop is now in the body of the partition loop. When an OpenMP parallel for (in red) is applied to the same loop as a paraguin forall statement, it must be attached to the partition loop in the output source to be valid.*

The programmer should also be able to apply an OpenMP `parallel for` pragma to a loop within the body of the partition loop. This will allow for finer granularity in shared-memory as threads compute less between synchronizations. For certain loop computations this may be necessary to guarantee correctness in shared-memory.

3.2 Paraguin Implementation

A brief understanding of how the Paraguin pass interacts with SUIF code and also modifies the computation loop targeted by a `paraguin forall` is essential to generating valid placement of OpenMP pragmas in the generated MPI output source. Additionally, awareness of few key concepts in how the SUIF frontend and backend handle pragmas to reach the output source is also vital. This section discusses how SUIF handles key structures, and this is used in Chapter 4 to describe the implementation details.

Pragma statements in SUIF code exist as annotation objects. All programmer-specified pragma statements which are passed through the SUIF `scc` front-end exist on the tree as `tree instruction (tree instr)` nodes within the Paraguin pass. Accessing a pragma statement means reaching the leaves of a particular tree node which are the `tree instr` objects of type `suif_object`. Annotation names are important to certain passes of the overall compilation. Pragmas, such as the OpenMP pragmas, which are to pass through the `s2c` backend must have its annotation object renamed as an `s2c_pragma` in the Paraguin pass. Pragma annotation objects processed in Paraguin are named `c_pragmas`.

The Paraguin pass starts near the top of the SUIF tree hierarchy checking sequentially each procedure in the SUIF code for pragmas. Pragmas specified directly in the scope of a procedure exist as `tree instr` node leaves of procedure bodies. Paraguin processes every annotation object containing pragmas at procedure-level scope and ignores non-paraguin type pragmas such as OpenMP pragmas. Pragmas which exist within structured control flow constructs such as `for` loops or `if` statements in the source are not parsed as SUIF code in the Paraguin pass. As such, pragmas which may exist inside `for` loop bodies or nested `for` loop bodies do not reach the output source. This process of handling pragmas by the Paraguin pass had to modified to be able to produce hybrid code.

When a `paraguin forall` pragma is detected, the immediately following loop in the procedure's tree is tiled. This loop is often a loop nest stored as a SUIF tree-for node object whose body (tree node list) contain a reference to another tree-for node. This loop nested `tree for` node is copied, tiled, and replaced with the newly tiled loop node in the containing procedure's tree. The tiled loop nest is generated from a copy of a `tree for` node representing the original loop nest. The result of the tiling transformation is a new loop which partitions the iteration space using the process id and replaces the original loop. The original loop index becomes dependent upon the partition loop index. This assignment of the original loop index to an affine expression of the partition loop index is controlled by the matrix specified in the `paraguin forall` pragma.

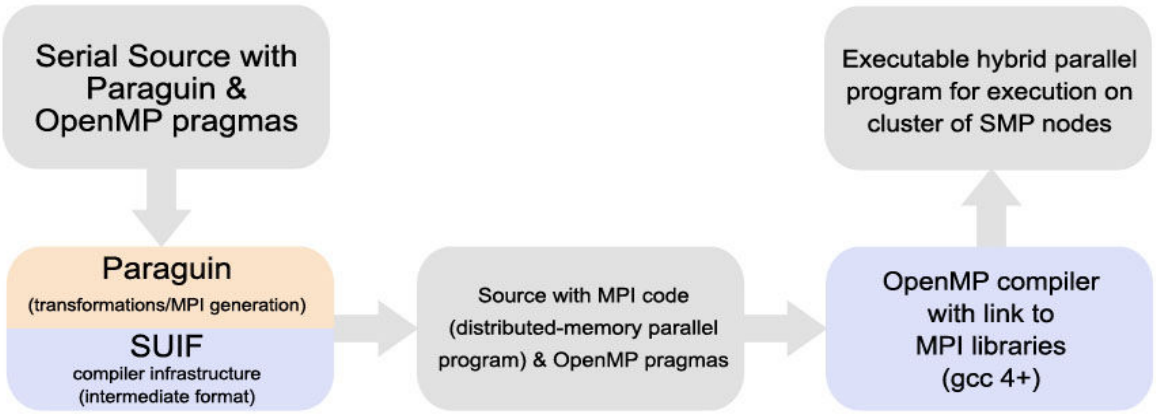

***Figure 11**: Overall compilation process with OpenMP support*

3.3 OpenMP support considerations and limitations

For preliminary efforts in producing valid hybrid parallel code from only pragmas at the sequential source, the concept of structured blocks defined in the OpenMP API standard have been excluded. OpenMP pragmas are similar to Paraguin pragmas except they can also include a body or structured block specified by an opening and closing brace. Structured blocks are an important concept in OpenMP as they define a single point of entry and exit for a parallel construct containing compound statements. All OpenMP constructs must include a structured block either implicitly or explicitly. This

Paraguin modification ignores OpenMP pragmas with explicit use of structured blocks. A simple but practical example of explicit structured blocks with an OpenMP pragma in a hybrid context is in printing the thread id's created by each MPI process as illustrated in *Figure 12*.

```
pragma omp parallel shared(numThreads) private(tID)
{ /* begin parallel region structured block */
  tID = omp_get_thread_num();
  if (tid == 0)
  {
       numThreads = omp_get_num_threads();
       printf("Number of threads in process %d= %d\n", pID, numThreads);
  }
  printf("Message from process %d, thread %d\n", pID, tID);
}/* end of parallel section, close structured block */
```

***Figure 12**: Example of OpenMP explicit structured block*

An OpenMP `parallel` construct is defined followed by a new-line and a block defined by an opening and closing brace. The parallel construct defines a block of code that will be executed in parallel by a team of threads. Support for OpenMP explicit structured blocks is left for future work.

OpenMP directives without explicit structured blocks can still prove useful for expressing shared-memory parallelism in `for` loop structures in many cases as is shown in two candidate hybrid parallel programs performing matrix multiplication and image edge detection. OpenMP runtime library function calls are parsed by Paraguin. By excluding support for explicit structured blocks in OpenMP constructs, additional mechanisms to bind blocks of code to OpenMP pragma statements do not need to be implemented. The optimization supports the passing of OpenMP constructs and directives with implicit structured blocks. This includes applying an OpenMP `parallel for` construct to any loop of a `for` loop nest in the Paraguin output source.

Chapter 4: Implementation

This chapter discusses the changes necessary to allow OpenMP pragmas to traverse the Paraguin compiler pass and reach the output MPI parallel source in the appropriate locations creating a correct hybrid parallel program. As discussed in the previous chapter, the changes include support for procedure and loop scoped pragmas. A special case exists when a Paraguin `forall` pragma is applied to the same loop as an OpenMP `parallel for` construct. Since this loop is tiled in the pass, it is desirable to redirect the OpenMP statement to the new partition loop which maps iterations onto processes at runtime.

4.1. Parsing Pragmas

The Paraguin pass reads all pragmas which are accessed as `annote` objects at the procedure scope. Thus any statement starting with `#pragma` located at the source directly inside a procedure body is read and passed to the `processPragma()` function. In the matrix multiplication and Sobel edge detection applications, this includes the `#pragma paraguin` and `#pragma omp` statements.

In the pseudocode of *Figure 13* there are calls to the functions `processPragma()` and `findOutermostFors()`. It was necessary to expand the details of these functions to determine how `pragmas` in nested scopes such as conditional and loop structures are parsed. The changes necessary to carry out OpenMP redirects to the output source are discussed and highlighted in red in the pseudocode of *Figure 14*.

```
while( next entry in file set )
{
      while( next procedure in file set)
      {
            while (tree_instr in procedure tree_node_list)
            {
                  …
                  annote = tree_instr->annotes()->peek(k_c_pragma);
                  processPragma(annote);
                  if(in parallel region)
                  {
                        findOutermostFors(outer tree_node_list);
                  }
            }
      }
}
```

***Figure 13**: Pseudocode of top-down parsing of procedure-scoped pragmas in the Paraguin pass*

The `processPragma()` function, shown in *Figure 14*, determines a pragma's name, reads any associated data with the pragma, and adjusts the global state of the pass. This function has a parameter to an annotation object which is attached to a `tree_instr` node. The function checks if the pragma is of paraguin-type as well as the directive of the pragma. A change to `processPragma()` was made to return a boolean to indicate if a `paraguin` pragma was read so that in the calling context it can be determined if the pragma was not of type `paraguin`. This was necessary because further processing is required to handle OpenMP pragmas. Another change in the *unknown* conditional detects if the pragma is of type `omp` and set its name as an `s2c pragma`. The result of this change allows procedure-scoped OpenMP pragmas to reach the C source produced by the `s2c` pass. The function `passOMPPragma()` was written to check if an annotation contains the name `omp` as required for all OpenMP pragma directives. Following this logic, all pragmas not named *omp* will simply be unrecognized to the s2c pass and are not included in the output source. The purpose of the variable PARSE_OMP will be discussed in the section *Usage refinements* of this chapter. No effort is made to validate the syntax of `omp` directives as this is done by the backend OpenMP compiler.

```
static boolean passOMPPragma(annote *an){
    boolean isOMP;
    if(isOMP = isOMPPragma(an)){
        an->set_name(k_s2c_pragma);
        …  // Additional statements discussed in section 4.3
           // and shown in Figure 18
    }
   return isOMP;
}

boolean processPragma(annote *an)
{
        knownPragma = true;
        if(paraguin begin_parallel)
        {
                …
        }
        else if(paraguin end_parallel)
        {
                …
        }
        else if(paraguin forall)
        {
                …
        }
        …
        else
        {
                …
                knownPragma = false;
                if(PARSE_OMP)
                {
                    passOMPPragma(an);
                }
        }
        return knownPragma;
`
```

***Figure 14**: Paraguin passOMPPragma() and processPragma() functions*

4.1.1 For-Loop Pragmas

Next, functionality is introduced to allow OpenMP pragmas to target any loop of a loop nest. This includes inner `for` loops which may be nested in a tiled `for` loop. Since the tree of a loop nest is not known to the compiler until compile-time, recursion is necessary. Recursion allows each nested structure's tree to be searched until the bottom of the tree is reached at the tree instructions (`tree_instr`). A recursive function, `passInnerForPragmas()`, takes a `for` loop tree and iterates the leaves of the tree as shown in *Figure 15*. At the leaves, a tree instruction is prepared for the `s2c` pass if detected as an OpenMP pragma.

The `passInnerForPragmas()` function is called once tiling has completed with the tiled loop nest tree being passed.

```
/*
 * Function: passInnerForPragmas
 * Parameters: tree node list pointer
 * Return: void
 * Description: side effect of setting all annotations attached to tree_nodes
in list pointer as k_s2c_pragmas
 * This allows the s2c conversion to retain pragmas in the original source
 */
static void passInnerForPragmas(tree_node_list *tnl){
    annote *an;
    annote_list *annotes;
    tree_node_list_iter iter(tnl);
    tree_node *tn;
    int annoteCnt = 0;
    while(!iter.is_empty()) {
        tn = iter.step();
        switch(tn->kind()){
            case TREE_FOR:
                tree_for *tnf = (tree_for *)tn;
                passInnerForPragmas(tnf->body());
                break;
            case TREE_IF:
                tree_if * tni = (tree_if *) tn;
                passInnerForPragmas(tni->then_part());
                passInnerForPragmas(tni->else_part());
                break;
            case TREE_LOOP:
                tree_loop * tnl = (tree_loop *) tn;
                passInnerForPragmas(tnl->body());
                break;
            case TREE_BLOCK:
                tree_block * tnb = (tree_block *) tn;
                passInnerForPragmas(tnb->body());
                break;
            case TREE_INSTR:
                tree_instr *tinstr = (tree_instr *) tn;
                if (tinstr->instr()->are_annotations()){
                    annotes = tinstr->instr()->annotes();
                    if ((an = annotes->peek_annote(k_c_pragma)) != NULL){
                        passOMPPragma(an);
                    }
                }
                                break;
            default:
                assert(0);
                break;
        }
    }
}
```

***Figure 15**: Paraguin passInnerForPragma() function*

4.2 Applying Pragmas to Partition Loop

The function `findOutermostFors`(), shown in Figure 16, searches a procedure's tree structure for an outermost `for` loop which is to be tiled. It is initially called when a parallel region is detected by a `#pragma paraguin begin_parallel` directive. Since the procedure's tree is not known to the compiler until compile time, this function is called recursively. Recursion allows each nested structure's tree to be searched until the bottom of the tree is reached at the tree instructions (`TREE_INSTR`). Additional code was needed at the instruction nodes where instructions containing OpenMP pragmas at the location on the tree where an outermost `for` loop is the target of tiling. In this code a copy of the instruction is stored in a global list and later referenced by the `tileForLoop()` function. The original instruction node is removed from the tree so that the user-specified OpenMP statement does not appear at the original location in the output source. The global list OMP_PRAGMA_LIST contains the OpenMP pragmas which are to be placed before the new tiling loop.

```
void findOutermostFors(tree_node_list, tree_node_list_iter, …)
{
        …
        while(tree_node_list_iter is not empty)
        {
                tree_node = tree_node_list_iter->step();
               annote = tree_node->annotes()->peek(k_c_pragma);
                switch(tree_node->kind())
                {
                        case TREE_FOR:
                                …
                                processPragma(annote);
                                if(tile_next_for)
                                        tileForLoop(tfor, …);
                                else
                                        findOutermostFors(tfor->body(), NULL, …);
                                break;
                        case TREE_IF:
                                …
                                processPragma(annote);
                                findOutermostFors(tif->body(), NULL, …);
                                break;
                        case TREE_LOOP:
                                …
                                processPragma(annote);
                                findOutermostFors(tloop->body(), NULL, …);
                                break;
                        case TREE_BLOCK:
                                …
                                processPragma(annote);
                                findOutermostFors(tblock->body(), NULL, …);
                                break;
                        case TREE_INSTR:
                                …
                                if(!processPragma (an))
                                {
                                   if(PARSE_OMP)
                                   {
                                     if(passOMPPragma(an))
                                     {
                                         OMP_PRAGMA_LIST->push(tnin);
                                      /* remove tree instruction
                                         containing the annotation from the tree
                                        list so that the pragma does not appear at
                                         the original location */
                                        tnlp->remove(tnin->list_e());
                                     }
                                   }
                                }
                                break;
        }
}
```

Figure 16: *Paraguin findOutermostFor() function*

The `tileForLoop()` function, shown in *Figure 17*, provides the bases for transforming the computation loop into a loop suitable for execution by many processes. The new loop nest is built from systems of linear inequalities representing the user-defined mapping of iterations to partitions specified in the `paraguin forall` pragma and the loop bounds in the iteration space.

The SUIF Builder library is used to construct a partition loop surrounding the original loop nest. Using the Builder library, SUIF tree structures are wrapped in `block` objects which represent code segments. Blocks can be combined and the SUIF tree structures reflecting these changes can be generated afterwards.

The loop in statement (1) in `tileForLoop()` of *Figure 17* iterates through the loop indices of the `for` loop nest of the input source code being tiled starting from the innermost loop index to the outermost loop indices. A condition using the variable symbol `pv` checks if the building loop iteration has constructed the partition index loop. Using the global list of OpenMP pragma tree instructions, we iterate the list creating a block from each instruction and prepending it to the current code block containing the loop nest. At the end of the list, all instructions containing OpenMP annotations have been added to the tree of the tiled loop nest immediately before the partition loop.

```
void tileForLoop(tree_for orig_tf, …)
{

    // Duplicate the computation loop nest and transform the duplicate
    block tf_blk(orig_tf);
    tree_for * tf = (tree_for *) tf_blk.make_tree_node();
    orig_tf->parent()->insert_after(tf, orig_tf->list_e());
    …
    /* Setup and generate communication code from data dependencies */
    …
    block code(tf->body());

    for(number of loops involved in tiling) (1)
    {
        …
        //prepends instruction nodes to suif code representing the partition loop
        if (pv == loop_var) {
            // Prepend the pragma to the current loop
            omp_pragma_list_iter ompItr(OMP_PRAGMA_LIST);
            while(!ompItr.is_empty()){
               block NoOpb(ompItr.step());
               code.set(block(NoOpb, code));
            }
        }
        …
    }
    …
}
```

Figure 17: *Pseudocode of the tiling function in the Paraguin pass which attaches OpenMP instructions to the partition loop*

4.3 Usage refinements

A few considerations were made to facilitate the support of OpenMP in the Paraguin pass. A compiler flag `-openmp` is added to control when it is desired to create hybrid parallel source from a sequential source with Paraguin and OpenMP statements. By omitting the flag, OpenMP statements will be ignored and the MPI-only source will be generated. The flag sets a global boolean `PARSE_OMP` in the compiler source and guards statements which would manipulate annotations with OpenMP statements. Another consideration is to include the OpenMP header file `omp.h` which is necessary when interacting with the OpenMP API. The pass will include the `omp.h` header only when an OpenMP pragma is detected, but the header is only included once by using the `IS_OPENMP` flag. The condition to perform this inclusion is shown in *Figure 18*. This is a modification to the `passOMPPragma()` function shown in *Figure 14*.

```
static boolean passOMPPragma(annote *an){
    boolean isOMP;
    if(isOMP = isOMPPragma(an)){
       an->set_name(k_s2c_pragma);
         if(!IS_OPENMP){
              setHeaderFile("include <omp.h>");
              IS_OPENMP = true; //set our global flag
         }
    }
   return isOMP;
}
```

Figure 18: *Modification to the code shown in Figure 14 to add the OpenMP header file when an annotation containing a* `#pragma omp` *is detected in the sequential source*

Chapter 5: Experimentation and Analysis

The purpose of this chapter is to verify that valid hybrid parallel code can be produced from a sequential source with pragma statements using the modified Paraguin compiler pass described in the previous chapter. Two programs are used as input for the modified Paraguin compiler and the generated parallel source is executed to verify correctness. A quick analysis of the computation loops in these two problems will give some insight into their parallel implementations.

There exists limited data of hybrid parallel performance particularly from compiler-generated parallel programs and thus in addition to testing for correctness, the executions will include wall clock timings which will be described in further detail. Measurements of the effectiveness of the generated hybrid parallel programs are of interest particularly to researchers of parallelizing compilers and hybrid executions of embarassingly-parallel programs on SMP-clusters.

5.1 System overview

The cluster at UNC-Wilmington consists of 8 Sunfire X4100 servers on 4-Gigabit network connections. Each machine has two Dual Core AMD Opteron 285 Processors at 2.6GHz with 8 GB of system memory. Storage includes two 72GB SCSI Drives in hardware RAID 1.

All programs where compiled using gcc version 4.1.2 with no compiler optimization flags. Each cluster machine runs on the Linux CentOS release 5.4 kernel 2.6.18 platform. MPI programs were run using the Open MPI version 1.3.3 implementation.

5.2 Program Analysis

Two sequential programs were chosen to pilot the modified Paraguin pass and verify that correct hybrid parallel programs can be generated. One program performs naïve square matrix multiplication and the other performs Sobel edge detection of a square grayscale image. These applications were chosen because their communication patterns are typical of embarrassingly parallel problems where little communication is required to divide the problem among non-communicating

parallel tasks. Once the input data is broadcast or scattered to the processors, each processor performs its work independently. There is communication required at the end to gather partial results back to the master process. This communication pattern is shown in Figure 19. Non-communicating parallel tasks are important for testing purposes so that the `hybrid masteronly` model can be applied to the generated parallel programs. In this model, threads executing in each process do not need to perform MPI communication to guarantee correctness.

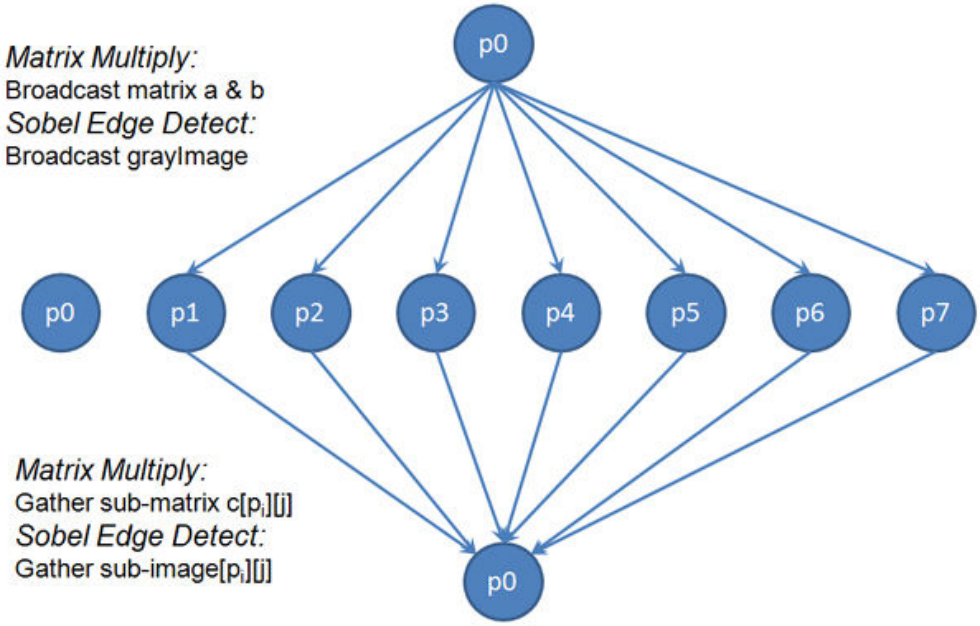

***Figure 19**: Communication pattern of two parallel programs*

The naïve square matrix multiplication example shown in Appendix C contains no loop-carried data dependencies. This means there is no array value written in one iteration that is subsequently read in another iteration. As such, the outermost loop *i* is tiled among a set of MPI processes mapping iteration *i* to partition *p* such that $p = i$.

Even though no loop-carried data dependencies exist, data locality in distributed memory is still critical so that all processes have the data it must read to perform the dot product on its portion of the arrays. A simple solution is to broadcast all values of array *a* and *b* to all processes. Though this may not be the most effective means of communicating the input data since redundant data on array *a* is

unnecessarily sent to all processes, it is not in the scope of this research.

A second sequential program, used as input to Paraguin to produce a hybrid parallel version, is show in Appendix E. This program performs Sobel edge detection on a grayscale 8-bit image. Sobel edge detection works by moving two convolution masks across every pixel of the image to determine the approximate derivative or gradient in both the horizontal and vertical directions of the image. Each pixel's gradient approximation is combined to determine the gradient magnitude which becomes the pixels of the edge -detected image. Formally, applying a convolution mask *G* to the input gray image pixels *gp* is given by the follow expression.

$$\sum_{i=-1}^{1}\sum_{j=-1}^{1} G(i+1, j+1)\, gp(x+i, y+j) \quad \forall x,y \in [1..N]$$

Figure 20: *Sobel Edge Detection algorithm*

This programs communication pattern is similar to that of matrix multiplication. No loop-carried dependencies exist in the edge detection loop. The gradient magnitude values are determined at each pixel by reading the pixel value from the input image and writing it at the corresponding image location in the output image. It is safe to tile the outermost loop *x* by mapping an iteration *x* to a partition *p* such that $p = x$. Data locality of the input image is once again important because each process needs to read its region of the input image to perform its independent work. In the generated parallel implementation, the input gray image is broadcast to all processes.

5.3 Application and Experimental Design

The execution plan outlines executing three versions of the two programs. These versions comprised the execution of sequential, OpenMP-only, generated MPI-only, and generated hybrid programs. The sequential executable was created from the same source code using `gcc`. Variations in input size, number of nodes, and number of processors for each version were also considered to further

verify correctness. The output of each parallel combination was compared, using the Linux `diff` and `cmp` command, to the output of the sequential execution for the same input size. In every case, the output of the MPI-Only and hybrid programs produced the exact same output as the corresponding sequential program. For every case the mean of ten execution times is taken to smooth variability.

Factor	**Levels (# of levels)**	**Description**
Types	Sequential, MPI-only, Hybrid, OpenMP-only	Variations in execution of the programs
Input size (N)	100, 200, 300,..., 1000	N x N is size of matrix computed
Number of processors (np)	4, 8, 12,..., 28, 32	Number of processors used in parallel programs; Hybrid program uses $^{np}/_{nt}$ nodes
Number of threads (nt)	4	Applies only to Hybrid program; Each node has 2 dual-core processors
Stream	1, 2,..., 10	Replicated runs of each level

Figure 21*: Execution plan of the matrix multiplication program*

The execution plan above defines the combinations of input size and processes for the parallel program runs. Hybrid programs executed with $^{np}/_{nt}$ nodes each with *nt* cores for every MPI-only program executing *np* processes [9]. The number of threads executing in each MPI process of the hybrid programs was chosen to be four with the rationale that four physical processor cores were available on each node.

It was originally planned that the Sobel edge detection program would follow the same execution plan above. However, variations in input sizes were excluded by using a fixed size of a 2500 x 2500 pixel image. This reasoning for this decision is discussed in the last section of this chapter *5.5, Sobel Edge Detection Results*.

Machine files were used when executing the parallel programs to effectively allocate processes across the cluster and not oversubscribe nodes. The machine file is used to allocate processes to certain machines. A machine file contains a list of machine addresses and the number of processors, or slots,

each machine has available. The Open-MPI implementation maps processes to slots using a block scheduling by default [14].

MPI-Only machine/host file	*Hybrid machine/host file*
compute-0-0.local slots=4	compute-0-0.local slots=1
compute-0-1.local slots=4	compute-0-1.local slots=1
compute-0-2.local slots=4	compute-0-2.local slots=1
compute-0-3.local slots=4	compute-0-3.local slots=1
compute-1-0.local slots=4	compute-1-0.local slots=1
compute-1-3.local slots=4	compute-1-3.local slots=1
compute-1-4.local slots=4	compute-1-4.local slots=1
compute-1-5.local slots=4	compute-1-5.local slots=1

Figure 22*: Machine file used for MPI-Only/Hybrid executions*

Wall clock timings of each program variant were also considered. These wall times did not include file I/O which was done sequentially by the master process and must be performed by all versions of the programs. Communication costs to distribute the input to machines and gather partial results into a final result are included in the MPI-only and hybrid timings. Although the compiler-generated MPI communication code is not guaranteed to be efficient, it is included in the timings because communication is necessary to determine the final result in a distributed-memory system. Furthermore, the sequential and OpenMP versions do not need to perform this communication.

5.4 Matrix Multiplication Results

The generated MPI-only and hybrid matrix multiplication program executed correctly for all combinations of processes and input sizes. Each combination wrote the resulting matrix to a file which was compared to the resulting matrix written by the sequential program for the same input size. The hybrid program would not execute completely when allocating 2 or 3 processes and the input size was greater than 800. This was due to an overflow of the default buffer size in the generated communication code during the gather phase.

Wall clock times surrounding communication and computation were also written to output along with the resulting matrix as shown in Appendix A. Figure 23 shows the runtimes of the various

programs on 1000x1000 floating point matrices. The hybrid program appears to scale down similarly to the MPI-Only program by using $^{np}/_{4}$ less cluster nodes (although the same number of processors/cores) for the hybrid runs. The reasoning behind the slightly greater wall-times for hybrid over MPI-Only is difficult to determine, but results are platform dependent as former research demonstrated [15]. Research by Rabenseifner, Hager, and Jost mentions that some compilers cannot make certain performance-increasing loop optimizations when compiling OpenMP programs [13]. The line graph below also includes an OpenMP-only timing for an input size of 1000x1000 for only 4 threads since OpenMP-only makes use of cores in shared-memory on a single cluster node. An OpenMP-only executable was created by compiling the sequential source using the `-fopenmp` option in `gcc`. The hybrid program with *np* = 1 and *nt* = 4 performed equally well as the OpenMP-only program. This is an expected result since the overhead of message-passing communication is not needed to move data to additional processes.

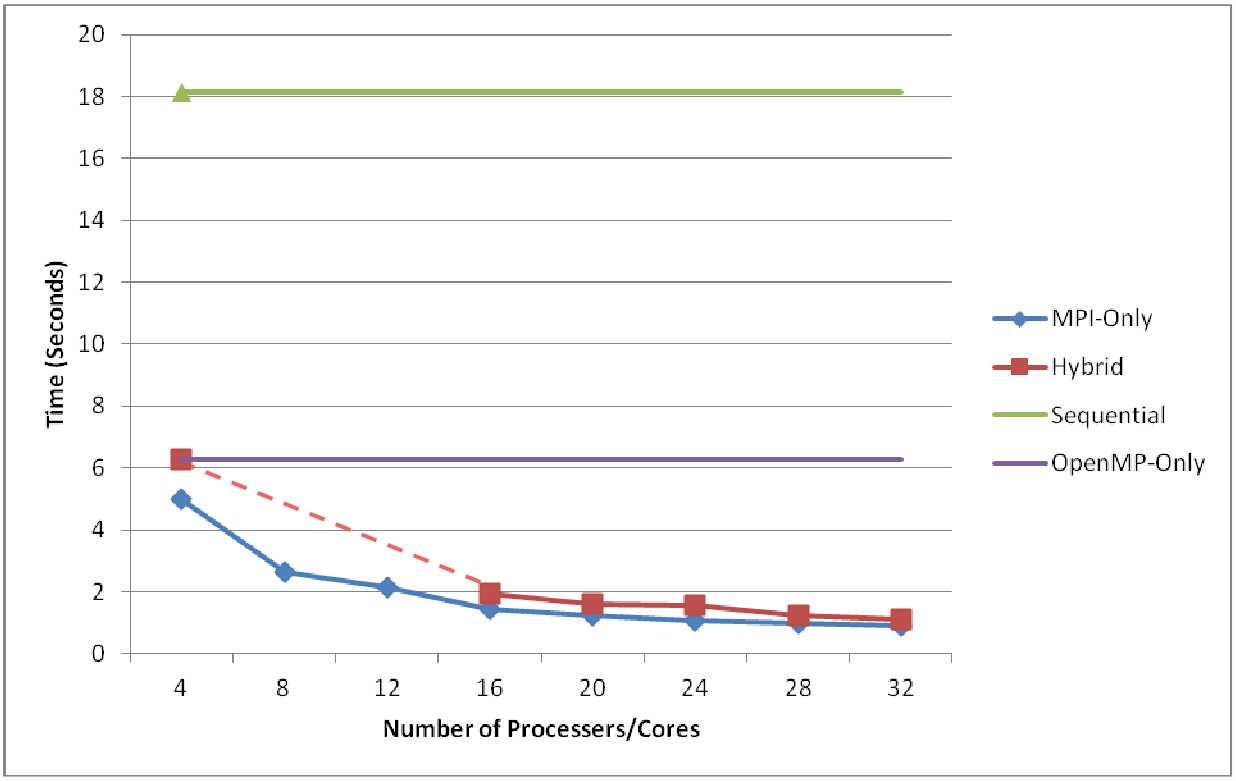

Figure 23: *Matrix Multiplication runtime of 1000x1000 floating-point matrices MPI-Only/Hybrid Code generated by Paraguin*

The line graph of *Figure 24* shows the runtime of the MPI-only and hybrid programs for various input sizes. Each series is either a hybrid or MPI-only run along with the number of processes used. Since the number of threads was set to four for the hybrid programs, the program mpi-*X*p uses the same number of processors as the program hyb-*Y*p where $X = 4Y$. In the figure, lines representing the same number of processors are of the same color. As the number of processes increase, scaling of the hybrid program approaches that of MPI-only. At one combination, *hybrid np=3* outperformed *MPI-only np=12* for a 500x500 float multiplication.

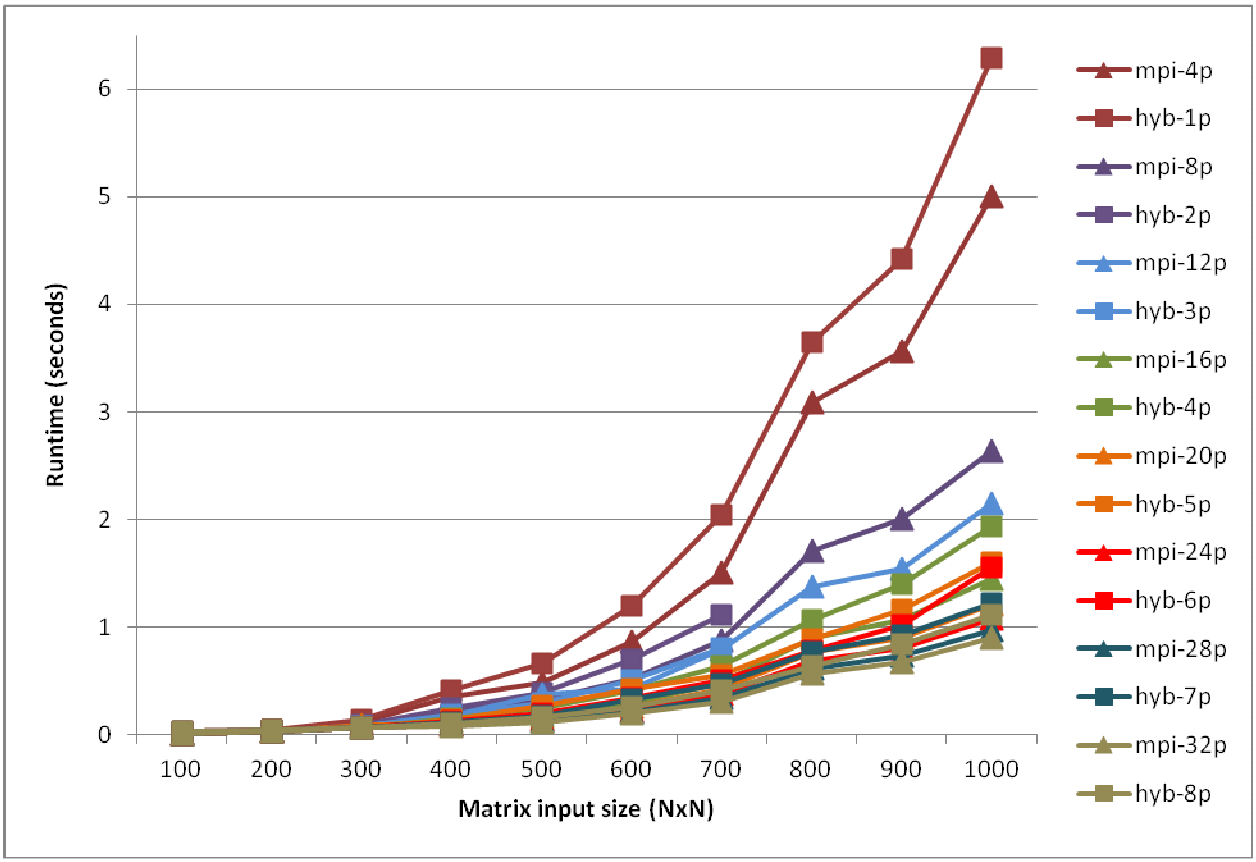

Figure 24: *Parallel Matrix Multiplication Scale-up of MPI-Only/Hybrid Paraguin generated code*

The compiler-generated hybrid program's performance was near that of MPI-only. In fact, the performance of the generated hybrid code is congruent with recent research at Northwest University in Xi'an, China measuring the performance of a hand-coded hybrid implementation of block-row matrix multiplication compiling with `gcc` on Linux [15]. Hence, not only does the compiler generate a correct hybrid program, performance is similar to hand-coding a hybrid program.

5.5 Sobel Edge Detection Results

The Paraguin pass was able to correctly generate MPI-Only and hybrid programs with two modifications to the output source. The first modification to the generated Sobel MPI-only and hybrid source was due to a bug found in the Paraguin pass. An addition modification was made to the hybrid source due to an optimization attempt made by the backend pass conversion to C.

An attempt was made to compile the program with Paraguin and generate all communication code. However, a bug in the pass hindered Paraguin's ability to generate gather MPI communication code from a `paraguin gather` pragma. In the `gather` pragma an array reference is specified which contains final values to be communicated. The pass seemed to have trouble finding this reference and as such the `gather` pragma was removed and the program was recompiled with Paraguin. A gather was hand-coded for the MPI-only and hybrid output source similar to what Paraguin would produce. Since the modifications to the compiler made for this research has nothing to do with gather communications, the gather phase of both programs is identical and does not affect the outcome of the results.

The SUIF `s2c` backend pass modified the partition loop so that the loop upper bound is assigned to a temporary variable because the upper bound is loop invariant. This optimization is performed *after* the Paraguin pass. This posed a problem because the OpenMP statement was pushed out of context with the partition loop. Fixing this bug is not straight forward and is left as future work. The solution for this instance was to move the OpenMP statement below the assignment statement produced by the subsequent backend pass as demonstrated below:

```
…
__guin_blksz = __suif_divceil(999 - 0 + 1, __guin_NP);
if (0 <= __guin_mypid & __guin_mypid <= -1 + 1 * __guin_NP)
{
  #pragma omp parallel for private ( __guin_p , x , y , i , j , sumx , sumy , sum )
  shared ( w , h ) num_threads ( 4 )
  //Local variable produced by s2c
  __s2c_tmp = __suif_min(999, __guin_blksz + -1 + __guin_blksz * __guin_mypid);
  //omp pragma moved here
  for (__guin_p = __guin_blksz * __guin_mypid; __guin_p <= __s2c_tmp; __guin_p++)
  {
…
```

The Sobel program output was an edge-detected binary image written to a file which was compared using the `cmp` command to the image produced by the sequential run. In all cases, the hybrid and MPI-only programs produce the same output as the sequential program. A sample edge-detected image produced from parallel Sobel edge detection is shown in *Figure 25*. Wall-clock times of the Sobel edge detection programs are supplied in Appendix B and illustrasted in *Figure 26*.

As the number of processes increase, there is a diminising return on speedup to the point that MPI-Only time exceeds sequential. The sequential run performs Sobel edge dection, an $O(N^2)$ complexity algorithm, reasonably well with a 2.5 MP image. This was not the case with the $O(N^3)$ complexity matrix multiplication algorithm when *N=1000*. The rising curves of the Sobel Edge detection MPI-Only/Hybrid programs reflects the nature of parallel programming on a distributed-memory system with smaller workloads. Amadahl's law, which defines that parallel execution time is boundeed by the time it takes to execute the serial portion of the program, is demonstrated here. It is expected that a larger input image would result in parallel wall-time curves similar to matrix multiplication at *N=1000*.

***Figure 25**: A sample grayscale image with parallel Sobel edge detection applied, Scaled to fit page*

Some additional profiling was also done to analyze the timings concluded. Individual timings measuring workload computation and gather communication time were also conducted and scaling is shown in *Figure 27.*

Communication overhead specifically in the gathering was determined to be the main performance bottleneck for both the MPI-only and hybrid programs. Gather communication quickly dominated the overall runtime of the MPI-only and hybrid programs as the number of processes increased. Since the hybrid program used only one MPI process per node, the gather communication in hybrid used a fourth of the MPI processes required by MPI-Only. This suggests that for this SMP cluster configuration it is faster to have contention on fewer messages with more data than more messages with less data in each message. When this is the case, communication in the hybrid model is more optimal since less processes are actually used in message-passing communication. Paraguin implements a gather by having each process send its results to the master process. This essentially saturates the network and, due to message contention, the gather is sequential. A future optimization that would improve Paraguin would be to have it make use of the more efficient `MPI_Gather` routine instead of point-to-point communication.

Workload time also decreased in the hybrid model for Sobel edge detection of the 2.5 MP image. Clearly for the small problem size, OpenMP-Only using just shared-memory is the most effective parallel program. The OpenMP-Only solution to the Sobel edge detection algorithm on a single 4-way node performed a 3.83x speedup over its sequential version as compared to a 2.88x speedup of the OpenMP-Only naïve matrix multiplication over its sequentialversion. The effectiveness of multithreading in a hybrid context can be measured by multithreading efficiency denoted as e_{mt} using the following formula [9]:

$$e_{mt} = {}^{T_{seq}}/_{(nt \cdot T_{hyb})} \text{ where } nt = 4$$

The hybrid wall-time for a single-node allocation (*np=1*) is used. Applying the multithreading efficiency formula yields for Sobel edge detection *N=2500* an approximately .90 efficiency. An approximate multithreaded efficiency of .72 in hybrid matrix multiplication *N=1000* was achieved. This metric suggests that the benefit of multithreading in hybrid sobel-edge detection was greater than in hybrid matrix multiplication. Although this may not be the case for increasing input sizes, it is further evidence as to the faster workload times in the hybrid version of Sobel edge detection compared to MPI-only.

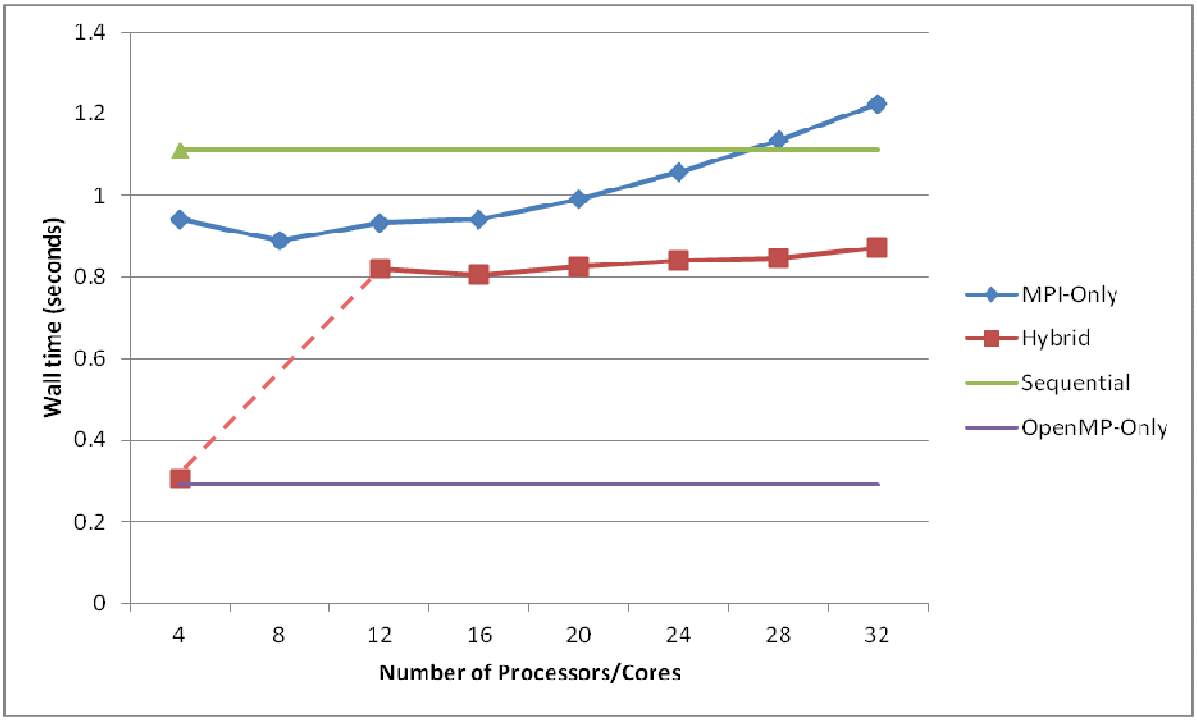

Figure 26: *Sobel edge detection runtime of 2.5 MP grayscale 8-bit image, MPI-Only/hybrid Code generated by Paraguin expect gather communication*

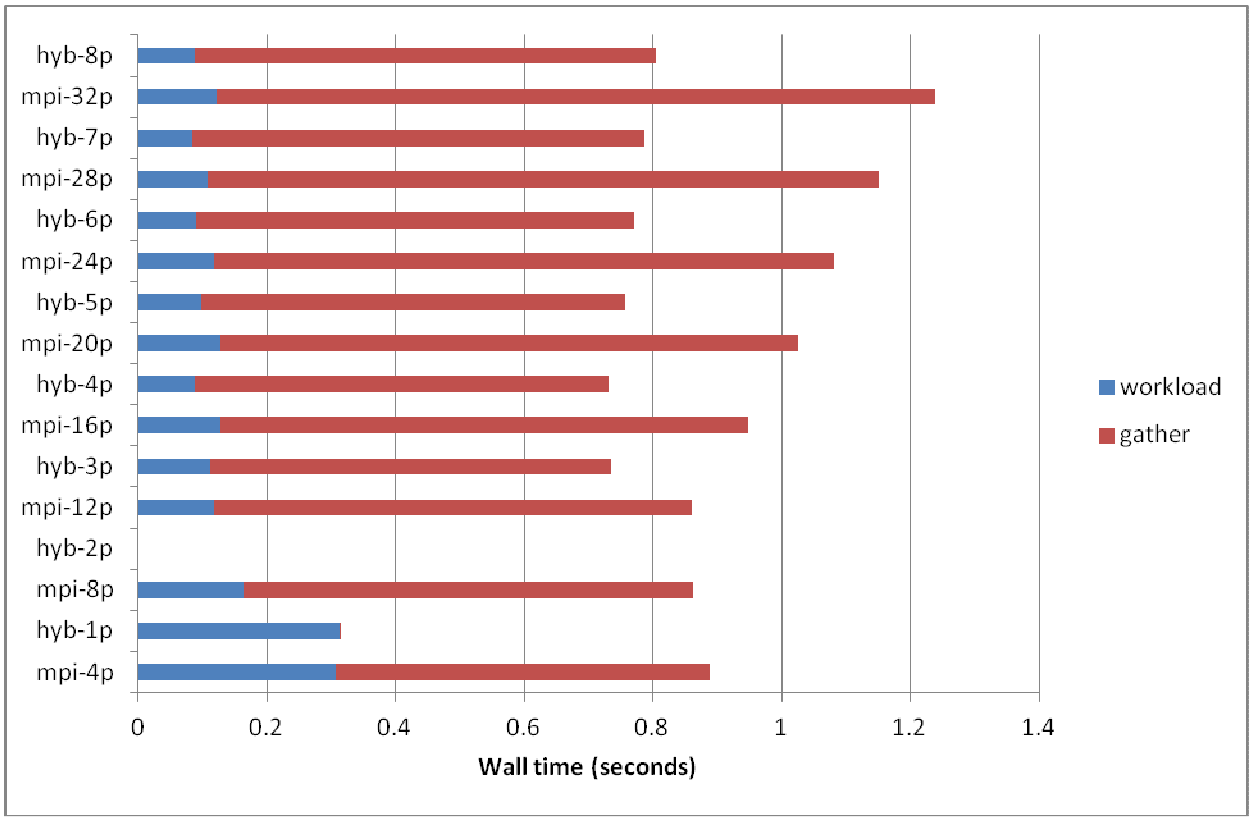

Figure 27: *Approx. workload and gather communication time in Sobel edge detection of 2.5 MP image*

Chapter 6: Conclusion

In this research, a distributed-memory parallelizing source-to-source compiler pass called Paraguin was modified to support the parsing and redirection of OpenMP statements for shared-memory parallelization in the output MPI parallel source. The result of such a modification allows the generation of hybrid parallel source from high-level guided parallelization at the sequential source using pragma statements. Expressing hybrid parallel programs from high-level statements at the sequential source is desirable since the cost of development is high and HPC architectures are consisting of denser multi-core nodes.

The success of high-level guided parallelism is shown in parallel programming models like OpenMP for shared-memory parallelism. Since the sequential source is unchanged and contains pragmas expressing parallelism only relevant to a particular compiler, using the modified Paraguin pass, the source code can represent a sequential, MPI-only, OpenMP-only, and hybrid MPI/OpenMP program.

The new modified compiler pass was verified by compiling two sequential programs of naïve square matrix multiplication and Sobel edge detection containing Paraguin and OpenMP pragmas. The generated hybrid parallel versions were executed and tested for correctness by comparing the output to the output of sequential execution. Basic profiling was also done comparing the performance of the generated hybrid and MPI-only versions of the compilation.

This effort should be considered a proof of concept for the Paraguin compiler pass developed by Dr. Clayton Ferner at UNC-Wilmington. It is a preliminary effort toward parsing and redirection of OpenMP statements to the appropriate location in the output source with the Paraguin pass. To reduce the implementation effort and verify the hybrid concept within Paraguin generated MPI code, this research supported OpenMP pragmas without the use of explicit structured blocks.

6.1 Final Analysis

The OpenMP Paraguin pass was successfully applied to two sequential programs to generate *correct* hybrid MPI/OpenMP programs. This stride alone is encouraging since current research has shown some success in hybrid parallelism and future chip roadmaps are suggesting the need for hybrid execution in a distributed-memory environment. Additionally, developing and debugging parallel programs is a difficult task using raw parallel programming models. Expressing multiple forms of parallelism simultaneously by applying higher level models on existing sequential code is clearly a step in the right direction toward making parallel programming easier.

In addition to testing for correctness, the preliminary performance measurements of the generated hybrid programs have some implications as well. Timings of the generated hybrid parallel matrix multiplication program suggest that it is possible to generate hybrid parallel programs which perform as well as hand-coded implementations [15]. The generated Sobel edge detection program suggests that hybrid performance can exceed MPI-only for embarrassingly parallel problems particularly when an increase in MPI processes causes communication to become the performance bottleneck. Based on the above findings, an effort to further support and integrate OpenMP statements into the output source of the Paraguin pass is a worthy endeavor.

6.2 Future Work

The Paraguin compiler project is an ambitious effort in aiding the development of parallel programs using MPI. A plethora of research is possible toward reducing complexity while striking a balance with performance and expressiveness. This research altered the Paraguin distributed-memory parallel programming model by allowing OpenMP statements to be specified in conjunction with Paraguin statements, thus merging two models for distributed-memory and shared-memory parallelism. Further research can be done in testing and validating that OpenMP statements specified in the sequential source are directed to the most logical and correct locations in the generated MPI source.

Increasing the expressiveness of OpenMP statements in conjunction with Paraguin statements is a

natural extension to this research. OpenMP statements can include explicit structured blocks, thus separating parallel regions from parallel constructs. This is currently not studied and determining how to place such statements in a generated hybrid context is unknown. Additionally, the `s2c` backend pass used by Paraguin posed a problem for the OpenMP statement targeting the partition loop, and the programmer was left to adjust this before compiling to an executable.

Paraguin's generated gather communication code can potentially be improved. The `MPI_Gather` routine does not send all messages to the master process causing the gather to execute sequentially. Complexity could be $O(\log_2 NP)$ as opposed to $O(NP)$. Hybrid programming could also improve current gathering by allowing the master process to receive messages concurrently using threads. Thus complexity would change by a factor of the number of threads/cores available on the master process.

Although this research can be further extended, it also exposes potential research efforts not necessarily related to generating hybrid parallel programs. The Paraguin pass was unable to detect the array access where final values are written for the Sobel edge detection program to generate gather code. More research can involve the automatic detection of the optimal partitioning of a loop nest which currently must be specified by the programmer in Paraguin.

References

[1] G. D´ozsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems", 2010.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. "Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming", 2008.

[3] K. Nakajima. "Parallel iterative solvers for finite-element methods using an OpenMP/MPI hybrid programming model on the Earth Simulator", Science Direct, Parallel Computing 31 (2005) 1048–1065. Bunkyo-ku, Tokyo, Japan, March 2005.

[4] H. Jin and R. Van der Wijngaart, "Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks", NASA Advanced Supercomputing Division, Moffett Field, CA, 2003.

[5] D. S. Henty, "Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling", Edinburgh Parallel Computing Centre, The University of Edinburgh, Scotland, U.K., 2000.

[6] M. Tsuji and M. Sato, "Performance evaluation of OpenMP and MPI hybrid programs on a large scale multi-core multi-socket cluster, T2K Open Supercomputer", 2009 International Conference on Parallel Processing Workshops, 2009.

[7] J. P. Hoeflinger, "Extending OpenMP to Clusters", White Paper, Intel Corporation, 2006.

[8] A. Lorenzo, "OpenMP on Clusters", Dpt. De Estadística, Computación e I. O. Universidad de La Laguna, Facultad de Matemáticas, Tenerife, Spain. 2004.

[9] E. Chow and D. Hysom. "Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters", 2001.

[10] L. Adhianto and B. Chapman, Performance modeling of communication and computation in hybrid MPI and OpenMP applications. Science Direct, Simulation Modelling Practice and Theory 15 (2007) 481–491, University of Houston, TX, Nov. 2006.

[11] M. Wolfe. "High Performance Compilers for Parallel Computing", Addison-Wesley Publishing Company, Redwood City, CA, 1996.

[12] C.S. Ferner, "The Paraguin compiler---Message-passing code generation using SUIF", in the *Proceedings of the IEEE SoutheastCon 2002*, 1—6, Columbia, SC, April 5—7, 2002.

[13] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes", 2009.

[14] "Open-MPI FAQ", http://www.open-mpi.org/faq/**,** The Open MPI Project, Bloomington, IN, Aug, 2011.

[15] L. He, Weichang Shen, Y. Li, A. Shi, D. Zhao, "MPI+OpenMP implementation and results analysis of matrix multiplication based on rowwise and columnwise block-striped decomposition of the matrices", *2010 Third International Joint Conference on Computational Science and Optimization*, 2010.

[16] R. Wilson, R. French, C. Wilson, S. Amarasignhe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy, "An Overview of the SUIF Compiler System", Computer Systems Lab, Stanford University, CA.

[17] "The SUIF Version 1.0 Library Documentation", http://suif.stanford.edu/suif/suif1/ , Stanford Compiler Group, Standford University, Palo Alto, CA, 1996.

[18] "OpenMP Application Program Interface Version 3.0", OpenMP Architecture Review Board. May 2008.

[19] S.F.McGinn and R.E.Shaw, "Parallel Gaussian Elimination Using OpenMP and MPI", in the *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'02)*, *2002.*

[20] C. Ferner, "Other Means of Executing Parallel Programs: OpenMP and Paraguin", presented to the *Grid Computing course (CSC 446/546)*, November 15, 2011.

Appendix A

Naïve Square Matrix Multiplication Timings

**Hybrid executions used np/4 processes and 4 threads per process*

Sequential Wall clock time

N	Seconds
100	0.013821
200	0.112498
300	0.366526
400	1.321117
500	1.7695
600	3.18919
700	5.767642
800	11.8482
900	12.50146
1000	18.13692

N = 1000 wall time (seconds)

	MPI-Only	Hybrid	Sequential	OpenMP
4	5.006365	6.284995	18.13692	6.289887
8	2.638909	-	18.13692	6.289887
12	2.145128	-	18.13692	6.289887
16	1.44953	1.929921	18.13692	6.289887
20	1.211733	1.595291	18.13692	6.289887
24	1.075128	1.554222	18.13692	6.289887
28	0.970137	1.223806	18.13692	6.289887
32	0.895734	1.116921	18.13692	6.289887

MPI-Only Wall clock time (seconds)

	NP							
N	**4**	**8**	**12**	**16**	**20**	**24**	**28**	**32**
100	0.006201	0.006786	0.007164	0.008133	0.007816	0.008089	0.008132	0.009039
200	0.036411	0.029831	0.029438	0.03018	0.029419	0.031002	0.030733	0.031292
300	0.109856	0.079771	0.072446	0.069998	0.087577	0.06927	0.066208	0.06498
400	0.35564	0.226338	0.177434	0.120716	0.105061	0.09383	0.090449	0.081355
500	0.484009	0.312052	0.373683	0.170392	0.146627	0.135679	0.125984	0.11914
600	0.868309	0.52247	0.425222	0.296283	0.257287	0.22416	0.209021	0.193469
700	1.507734	0.877366	0.794708	0.48221	0.41747	0.379406	0.340867	0.300074
800	3.083934	1.702195	1.373548	0.896807	0.764353	0.679876	0.614728	0.561775
900	3.565298	2.009324	1.54273	1.0658	0.901691	0.803257	0.731174	0.670317
1000	5.006365	2.638909	2.145128	1.44953	1.211733	1.075128	0.970137	0.895734

Hybrid MPI/OpenMP Wall clock time (seconds)

	NP							
N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
100	0.008381	0.009762	0.009588	0.009171	0.015663	0.02612	0.027282	0.028694
200	0.041265	0.032912	0.029854	0.026192	0.038707	0.03883	0.040179	0.039489
300	0.142972	0.093792	0.078347	0.067141	0.064639	0.066327	0.064762	0.06605
400	0.409443	0.241162	0.180856	0.149386	0.135347	0.124358	0.11375	0.104796
500	0.659626	0.387761	0.290165	0.239944	0.271974	0.194877	0.176936	0.160319
600	1.197064	0.693908	0.501442	0.408132	0.418116	0.34029	0.326397	0.26222
700	2.038315	1.118503	0.80688	0.642351	0.555614	0.503161	0.466509	0.417062
800	3.650771	-	-	1.06546	0.891521	0.785332	0.767585	0.634097
900	4.420645	-	-	1.395099	1.156201	1.01782	0.926384	0.841611
1000	6.284995	-	-	1.929921	1.595291	1.554222	1.223806	1.116921

Appendix B

Sobel Edge Detection (2500x2500 pixel image) Timings

**Hybrid executions used np/4 processes and 4 threads per process*

Sobel Edge Detection wall times (seconds)

np	MPI-Only	Hybrid	Sequential	OpenMP
4	0.9414041	0.3065436	1.1105564	0.2898374
8	0.8889763	-	1.1105564	0.2898374
12	0.9313033	0.8196769	1.1105564	0.2898374
16	0.940546889	0.8063131	1.1105564	0.2898374
20	0.990767556	0.824817	1.1105564	0.2898374
24	1.0566138	0.8404589	1.1105564	0.2898374
28	1.1343584	0.8454437	1.1105564	0.2898374
32	1.223049667	0.8715673	1.1105564	0.2898374

Compute times (seconds) for generated parallel programs

np	MPI-Only	Hybrid
4	0.3072306	0.3123132
8	0.164404	-
12	0.1177586	0.111716
16	0.1283676	0.0874599
20	0.1281333	0.0981019
24	0.1174586	0.0907923
28	0.1085537	0.0848004
32	0.1224682	0.0879252

Gather Communication times (seconds) for generated parallel programs

np	MPI-Only	Hybrid
4	0.5819117	0.000001
8	0.6987496	-
12	0.7432633	0.6227164
16	0.8199001	0.6430389
20	0.8979341	0.6583478
24	0.9642838	0.6807821
28	1.0434284	0.7029633
32	1.1154515	0.7171176

Appendix C

Input matrix multiplication source w/ Paraguin and OpenMP pragmas, N = 800

```
#ifdef PARAGUIN
typedef void* __builtin_va_list;
#endif

#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define N 800

#ifdef PARAGUIN
extern int MPI_COMM_WORLD;
int MPI_Barrier();
#endif

print_results(char *prompt, float a[N][N]);

int main(int argc, char *argv[])
{
    int i, j, k;
    float a[N][N], b[N][N], c[N][N];
    char *usage = "Usage: %s file\n";
    FILE *fd;
    double elapsed_time;
    struct timeval tv1, tv2;

    if (argc < 2) {
        fprintf (stderr, usage, argv[0]);
        return -1;
    }

    if ((fd = fopen (argv[1], "r")) == NULL) {
        fprintf (stderr, "%s: Cannot open file %s for reading.\n",
                            argv[0], argv[1]);
        fprintf (stderr, usage, argv[0]);
        return -1;
    }

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            fscanf (fd, "%f", &a[i][j]);

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            fscanf (fd, "%f", &b[i][j]);

#ifdef PARAGUIN
    ;
    #pragma paraguin begin_parallel
        MPI_Barrier((int)MPI_COMM_WORLD);
    #pragma paraguin end_parallel
#endif

    gettimeofday(&tv1, NULL);
```

```
    ;
    #pragma paraguin begin_parallel
    #pragma paraguin forall    C       p        i      j      k \
                             0x0      -1        1    0x0    0x0 \
                             0x0       1       -1    0x0    0x0

   #pragma paraguin bcast a b

   #pragma paraguin gather   1   C      i      j    k \
                               0x0    0x0    0x0    1 \
                               0x0    0x0    0x0   -1

   // We need to gather all c[i][j].  However, array reference
   // one is inside the k loop.  If we put in an empty gather
   // then we'll have N copies of each c[i][j] send to the
   // master.  To send just one, then we use k = 0.

   #pragma omp parallel for private(__guin_p,i,j,k) schedule(static) num_threads(4)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            c[i][j] = 0.0;
            for (k = 0; k < N; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
        }
    }

   ;
   #pragma paraguin end_parallel

    gettimeofday(&tv2, NULL);

    elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
                        ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);

    printf ("elapsed_time=\t%lf\n", elapsed_time);

    /*print result*/
    print_results("C = ", c);

}

print_results(char *prompt, float a[N][N])
{
    int i, j;

    printf ("\n\n%s\n", prompt);
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf(" %.2f", a[i][j]);
        }
        printf ("\n");
    }
    printf ("\n\n");
}
```

Appendix D

Hybrid matrix multiplication source generated by Paraguin, N = 800

```
#define __suif_min(x,y) ((x)<(y)?(x):(y))
#define __suif_divfloor(x,y) (((x)<0)^((y)<0) ? ((x) < 0 ? ((x)-(y)+1)/(y) : ((x)-(y)-
1)/(y)) : (x)/(y))
#define __suif_divceil(x,y) (((x)<0)^((y)<0) ? (x)/(y) : ((x) < 0 ? ((x)+(y)+1)/(y) :
((x)+(y)-1)/(y)))

…

#include <mpi.h>
#include <omp.h>
static union __tmp_union3 __huge_val =
  {
      {
        (unsigned char)'\000', (unsigned char)'\000', (unsigned char)'\000', (unsigned
char)'\000', (unsigned char)'\000', (unsigned char)'\000', (unsigned char)240, (unsigned
char)'\177'
      }
  };
static char __tmp_string_3[3] = "%f";
int __guin_NP;
int __guin_blksz;
int __guin_mypid;
int __guin_pidr;
int __guin_pidw;
int __guin_position;
char __guin_buffer[1048574];
MPI_Status __guin_status;
int __guin_p;
int __guin_pr;
int __guin_pw;

extern int print_results(char *prompt, float (*a)[800])
  {
    int i;
    int j;

    printf("\n\n%s\n", prompt);
    for (i = 0; i < 800; i++)
      {
        for (j = 0; j < 800; j++)
          {
            printf(" %.2f", a[i][j]);
          }
        printf("\n");
      }
    printf("\n\n");
    return 0;
  }

extern int main(int argc, char **argv)
  {
    int i;
    int j;
    int k;
    float (a[800])[800];
    float (b[800])[800];
    float (c[800])[800];
    char *usage;
    struct _IO_FILE *fd;
    double elapsed_time;
    struct timeval tv1;
```

```
    struct timeval tv2;
    int __guin_i_r;
    int __guin_i_w;
    int jr;
    int jw;
    int kr;
    int kw;
    int __s2c_tmp;
    int __s2c_tmp0;
    int __s2c_tmp1;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &__guin_NP);
    MPI_Comm_rank(MPI_COMM_WORLD, &__guin_mypid);
    if (__guin_mypid == 0)
      {
        usage = "Usage: %s file\n";
        if (argc < 2)
          {
            fprintf(stderr, usage, *argv);
            MPI_Finalize();
            return -1;
          }
        fd = fopen(argv[1l], "r");
        if (fd == (struct _IO_FILE *)0)
          {
            fprintf(stderr, "%s: Cannot open file %s for reading.\n", *argv, argv[1l]);
            fprintf(stderr, usage, *argv);
            MPI_Finalize();
            return -1;
          }
        for (i = 0; i < 800; i++)
          {
            for (j = 0; j < 800; j++)
              {
                fscanf(fd, __tmp_string_3, &a[i][j]);
              }
          }
        for (i = 0; i < 800; i++)
          {
            for (j = 0; j < 800; j++)
              {
                fscanf(fd, __tmp_string_3, &b[i][j]);
              }
          }
      }
    MPI_Barrier(MPI_COMM_WORLD);
    if (__guin_mypid == 0)
      {
        gettimeofday(&tv1, (struct timezone *)0);
      }
    MPI_Bcast((float ((*)[800])[800])a, 2560000, MPI_BYTE, 0, MPI_COMM_WORLD);
    MPI_Bcast((float ((*)[800])[800])b, 2560000, MPI_BYTE, 0, MPI_COMM_WORLD);
    __guin_blksz = __suif_divceil(799 - 0 + 1, __guin_NP);
    if (0 <= __guin_mypid & __guin_mypid <= -1 + 1 * __guin_NP)
      {
#pragma omp parallel for private ( __guin_p , i , j , k ) schedule ( static ) num_threads (
4 )
        for (__guin_p = __guin_blksz * __guin_mypid; __guin_p <= __suif_min(799,
__guin_blksz + -1 + __guin_blksz * __guin_mypid); __guin_p++)
          {
            i = 1 * __guin_p;
            for (j = 0; j <= 799; j++)
              {
                c[i][j] = 0.0F;
```

```
                for (k = 0; k <= 799; k++)
                  {
                    c[i][j] = c[i][j] + a[i][k] * b[k][j];
                  }
              }
          }
      }
    if (0 <= __guin_mypid & __guin_mypid <= 0)
      {
        __s2c_tmp0 = __suif_min(__suif_divfloor(799, __guin_blksz), -1 + 1 * __guin_NP);
        for (__guin_pidw = 1; __guin_pidw <= __s2c_tmp0; __guin_pidw++)
          {
            if (__guin_mypid != __guin_pidw)
              {
                MPI_Recv((char (*)[1048574])__guin_buffer, 1048574, MPI_PACKED, __guin_pidw,
1 * (__guin_NP - 0 + 1) + __guin_mypid, MPI_COMM_WORLD, &__guin_status);
                __guin_position = 0;
                __s2c_tmp = __suif_min(799, __guin_blksz + -1 + __guin_blksz * __guin_pidw);
                for (__guin_p = __guin_blksz * __guin_pidw; __guin_p <= __s2c_tmp;
__guin_p++)
                  {
                    i = 1 * __guin_p;
                    for (j = 0; j <= 799; j++)
                      {
                        k = 0;
                        MPI_Unpack((char (*)[1048574])__guin_buffer, 1048574,
&__guin_position, &c[i][j], 4, MPI_BYTE, MPI_COMM_WORLD);
                      }
                  }
              }
          }
      }
    __guin_pidr = 0;
    if (__guin_mypid != __guin_pidr)
      {
        __guin_position = 0;
        __s2c_tmp1 = __suif_min(799, __guin_blksz + -1 + __guin_blksz * __guin_mypid);
        for (__guin_p = __guin_blksz * __guin_mypid; __guin_p <= __s2c_tmp1; __guin_p++)
          {
            i = 1 * __guin_p;
            for (j = 0; j <= 799; j++)
              {
                k = 0;
                MPI_Pack(&c[i][j], 4, MPI_BYTE, (char (*)[1048574])__guin_buffer, 1048574,
&__guin_position, MPI_COMM_WORLD);
              }
          }
        MPI_Send((char (*)[1048574])__guin_buffer, __guin_position, MPI_PACKED, __guin_pidr,
1 * (__guin_NP - 0 + 1) + __guin_pidr, MPI_COMM_WORLD);
      }
    if (__guin_mypid == 0)
      {
        gettimeofday(&tv2, (struct timezone *)0);
        elapsed_time = (double)(int)(tv2.tv_sec - tv1.tv_sec) + (double)(int)(tv2.tv_usec -
tv1.tv_usec) / 1000000.0;
        printf("elapsed_time=\t%lf\n", elapsed_time);
        print_results("C = ", (float (*)[800])c[0l]);
        MPI_Finalize();
        return 0;
      }
    MPI_Finalize();
}
```

Appendix E

Input Sobel edge detection source w/ Paraguin and OpenMP pragmas

```
#ifdef PARAGUIN
typedef void* __builtin_va_list;
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <sys/time.h>

#define N 2500

#ifdef PARAGUIN
extern int MPI_COMM_WORLD;
int MPI_Barrier();
#endif

static int clamp(int val){
        if(val < 0)
                val = 0;
        else if(val > 255)
                val = 255;
        return val;
}

int main(int argc, char **argv) {

        FILE *inFile, *oFile;
        //int colorImage[N][N*3];
        unsigned char grayImage[N][N];
        unsigned char edgeImage[N][N];
        char type[2];
        int w, h, max;
        int r, g, b, y, x, i, j, sum, sumx, sumy;

        int GX[3][3], GY[3][3];
        double elapsed_time;
        struct timeval tv1, tv2;
        int tID;

        ;
        #pragma paraguin begin_parallel
        /* 3x3 Sobel masks. */
        GX[0][0] = -1; GX[0][1] = 0; GX[0][2] = 1;
        GX[1][0] = -2; GX[1][1] = 0; GX[1][2] = 2;
        GX[2][0] = -1; GX[2][1] = 0; GX[2][2] = 1;

     GY[0][0] =  1; GY[0][1] =  2; GY[0][2] =  1;
     GY[1][0] =  0; GY[1][1] =  0; GY[1][2] =  0;
     GY[2][0] = -1; GY[2][1] = -2; GY[2][2] = -1;
        ;
        #pragma paraguin end_parallel
        inFile = fopen(argv[1], "rb");
        if(inFile == 0) {
                fprintf(stderr, "Open failed\n");
                fprintf(stderr, "Usage: %s target\n", argv[1]);
                return EXIT_FAILURE;
        }else{
                fscanf(inFile, "%s", type);
                fscanf(inFile, "%d %d %d", &w, &h, &max);
        }

        if(type[1] != '5'){
                fprintf(stderr, "%s is not a valid pgm file", argv[1]);
                return EXIT_FAILURE;
        }
```

```
        if(fread(grayImage, h, w, inFile) == 0){
                fprintf(stderr, "Failed to read %s", argv[1]);
                return EXIT_FAILURE;
        }

        fclose(inFile);

        #ifdef PARAGUIN
        ;
        #pragma paraguin begin_parallel
        MPI_Barrier((int)MPI_COMM_WORLD);
        ;
        #pragma paraguin end_parallel
        #endif

        gettimeofday(&tv1, NULL);

        ;
        #pragma paraguin begin_parallel

        #pragma paraguin forall    C       p        x      y     i     j \
                                 0x0      -1        1    0x0   0x0   0x0 \
                                 0x0       1       -1    0x0   0x0   0x0

        #pragma paraguin bcast grayImage
        #pragma paraguin bcast w
        #pragma paraguin bcast h

        #pragma omp parallel for private(x,y,i,j,sumx,sumy,sum) shared(w,h) num_threads(4)
        for(x=0; x < N; ++x){
                for(y=0; y < N; ++y){
                        sumx = 0;
                        sumy = 0;
                        // handle image boundaries
                        if(x==0 || x==(h-1) || y==0 || y==(w-1))
                                sum = 0;
                        else{
                                //x gradient approx
                                for(i=-1; i<=1; i++)  {
                                        for(j=-1; j<=1; j++){
                                                sumx += (grayImage[x+i][y+j] * GX[i+1][j+1]);
                                        }
                                }
                                //y gradient approx
                                for(i=-1; i<=1; i++)  {
                                        for(j=-1; j<=1; j++){
                                                sumy += (grayImage[x+i][y+j] * GY[i+1][j+1]);
                                        }
                                }
                                //gradient magnitude approx
                                sum = (abs(sumx) + abs(sumy));
                        }
                        edgeImage[x][y] = clamp(sum);
                }
        }

        ;
        #pragma paraguin end_parallel

        gettimeofday(&tv2, NULL);

        elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
                      ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
        printf ("%lf\n", elapsed_time);

        oFile = fopen(argv[2], "wb");
        if(oFile != 0){
                fprintf(oFile, "P5\n%d %d\n%d\n", w,h,max);
                fwrite(edgeImage, h, w, oFile);
                fclose(oFile);
        }
        return EXIT_SUCCESS;
}
```

Appendix F

Hybrid Sobel edge detection source generated by Paraguin except gather code (hand written)

```
#define __suif_min(x,y) ((x)<(y)?(x):(y))
#define __suif_divfloor(x,y) (((x)<0)^((y)<0) ? ((x) < 0 ? ((x)-(y)+1)/(y) : ((x)-(y)-
1)/(y)) : (x)/(y))
#define __suif_divceil(x,y) (((x)<0)^((y)<0) ? (x)/(y) : ((x) < 0 ? ((x)+(y)+1)/(y) :
((x)+(y)-1)/(y)))

...

extern int __guin_NP;
extern int __guin_blksz;
extern int __guin_mypid;
extern int __guin_pidr;
extern int __guin_pidw;
extern int __guin_position;
//extern char __guin_buffer[1048574];
extern char __guin_buffer[2097152];

extern int __guin_p;
extern int __guin_pr;
extern int __guin_pw;

#include <mpi.h>
#include <omp.h>
static int clamp(int);

static union __tmp_union3 __huge_val =
  {
      {
        (unsigned char)'\000', (unsigned char)'\000', (unsigned char)'\000', (unsigned
char)'\000', (unsigned char)'\000', (unsigned char)'\000', (unsigned char)240, (unsigned
char)'\177'
      }
  };
int __guin_NP;
int __guin_blksz;
int __guin_mypid;
int __guin_pidr;
int __guin_pidw;
int __guin_position;
char __guin_buffer[2097152];
MPI_Status __guin_status;
int __guin_p;
int __guin_pr;
int __guin_pw;

static int clamp(int val)
  {
    if (val < 0)
      {
        val = 0;
      }
    else
      {
        if (255 < val)
          {
            val = 255;
          }
      }
    return val;
  }
```

```
extern int main(int argc, char **argv)
  {
    struct _IO_FILE *inFile;
    struct _IO_FILE *oFile;
    unsigned char (grayImage[2500])[2500];
    unsigned char (edgeImage[2500])[2500];
    char type[2];
    int w;
    int h;
    int max;
    int r;
    int g;
    int b;
    int y;
    int x;
    int i;
    int j;
    int sum;
    int sumx;
    int sumy;
    int (GX[3])[3];
    int (GY[3])[3];
    double elapsed_time;
    struct timeval tv1;
    struct timeval tv2;
    int tID;
    int __guin_x_r;
    int __guin_x_w;
    int yr;
    int yw;
    int ir;
    int iw;
    int jr;
    int jw;
    int __s2c_tmp, __s2c_tmp0;
    int nn = 2500;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &__guin_NP);
    MPI_Comm_rank(MPI_COMM_WORLD, &__guin_mypid);
    if (__guin_mypid == 0)
      {
      }
    GX[0l][0l] = -1;
    GX[0l][1l] = 0;
    GX[0l][2l] = 1;
    GX[1l][0l] = -2;
    GX[1l][1l] = 0;
    GX[1l][2l] = 2;
    GX[2l][0l] = -1;
    GX[2l][1l] = 0;
    GX[2l][2l] = 1;
    GY[0l][0l] = 1;
    GY[0l][1l] = 2;
    GY[0l][2l] = 1;
    GY[1l][0l] = 0;
    GY[1l][1l] = 0;
    GY[1l][2l] = 0;
    GY[2l][0l] = -1;
    GY[2l][1l] = -2;
    GY[2l][2l] = -1;
    if (__guin_mypid == 0)
      {
        inFile = fopen(argv[1l], "rb");
        if (inFile == (struct _IO_FILE *)0)
```

```
        {
          fprintf(stderr, "Open failed\n");
          fprintf(stderr, "Usage: %s target\n", argv[1l]);
          MPI_Finalize();
          return 1;
        }
      else
        {
          fscanf(inFile, "%s", type);
          fscanf(inFile, "%d %d %d", &w, &h, &max);
        }
      if ((int)type[1l] != 53)
        {
          fprintf(stderr, "%s is not a valid pgm file", argv[1l]);
          MPI_Finalize();
          return 1;
        }
      if (fread(grayImage, h, w, inFile) == 0ul)
        {
          fprintf(stderr, "Failed to read %s", argv[1l]);
          MPI_Finalize();
          return 1;
        }
      fclose(inFile);
    }
  MPI_Barrier(MPI_COMM_WORLD);
  if (__guin_mypid == 0)
    {
      gettimeofday(&tv1, (struct timezone *)0);
    }
  MPI_Bcast((unsigned char ((*)[2500])[2500])grayImage, 6250000, MPI_BYTE, 0,
MPI_COMM_WORLD);
  MPI_Bcast(&w, 4, MPI_BYTE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&h, 4, MPI_BYTE, 0, MPI_COMM_WORLD);
  __guin_blksz = __suif_divceil(2499 - 0 + 1, __guin_NP);
  if (0 <= __guin_mypid & __guin_mypid <= -1 + 1 * __guin_NP)
    {
      __s2c_tmp = __suif_min(2499, __guin_blksz + -1 + __guin_blksz * __guin_mypid);
#pragma omp parallel for private ( x , y , i , j , sumx , sumy , sum ) shared ( w , h )
num_threads ( 4 )
      for (__guin_p = __guin_blksz * __guin_mypid; __guin_p <= __s2c_tmp; __guin_p++)
        {
          x = 1 * __guin_p;
          for (y = 0; y <= 2499; y++)
            {
              int suif_tmp;
              int suif_tmp0;
              int suif_tmp1;
              int suif_tmp2;
              int suif_tmp3;
              int suif_tmp4;

              sumx = 0;
              sumy = 0;
              suif_tmp3 = x == 0;
              if (suif_tmp3)
                {
                  suif_tmp4 = suif_tmp3;
                }
              else
                {
                  suif_tmp1 = x == h - 1;
                  if (suif_tmp1)
                    {
                      suif_tmp2 = suif_tmp1;
```

```
                    }
                  else
                    {
                      suif_tmp = y == 0;
                      if (suif_tmp)
                        {
                          suif_tmp0 = suif_tmp;
                        }
                      else
                        {
                          suif_tmp0 = y == w - 1;
                        }
                      suif_tmp2 = suif_tmp0;
                    }
                  suif_tmp4 = suif_tmp2;
                }
              if (suif_tmp4)
                {
                  sum = 0;
                }
              else
                {
                  for (i = -1; i <= 1; i++)
                    {
                      for (j = -1; j <= 1; j++)
                        {
                          sumx = sumx + (int)grayImage[x + i][y + j] * GX[i + 1][j + 1];
                        }
                    }
                  for (i = -1; i <= 1; i++)
                    {
                      for (j = -1; j <= 1; j++)
                        {
                          sumy = sumy + (int)grayImage[x + i][y + j] * GY[i + 1][j + 1];
                        }
                    }
                  sum = abs(sumx) + abs(sumy);
                }
              edgeImage[x][y] = (unsigned int)clamp(sum);
            }
        }
    }

    /* ----- HAND-CODED GATHER ----- */
    if (0 <= __guin_mypid & __guin_mypid <= 0)
    {
            __s2c_tmp0 = __suif_min(__suif_divfloor(nn, __guin_blksz), -1 + 1 *
__guin_NP);
        for (__guin_pidw = 1; __guin_pidw <= __s2c_tmp0; __guin_pidw++)
        {

          if (__guin_mypid != __guin_pidw)
          {

            MPI_Recv((char (*)[2097152])__guin_buffer, 2097152, MPI_PACKED, __guin_pidw,
1 * (__guin_NP - 0 + 1) + __guin_mypid, MPI_COMM_WORLD, &__guin_status);
            __guin_position = 0;
            __s2c_tmp = __suif_min(nn, __guin_blksz + -1 + __guin_blksz * __guin_pidw);
            for (__guin_p = __guin_blksz * __guin_pidw; __guin_p <= __s2c_tmp;
__guin_p++)
            {

                  x = 1 * __guin_p;
                  for (y = 0; y <= nn; y++)
                  {
```

```
                        i = 0;
                        j = 0;
                        MPI_Unpack((char (*)[2097152])__guin_buffer, 2097152,
&__guin_position, &edgeImage[x][y], 1, MPI_BYTE, MPI_COMM_WORLD);

                    }
                }
            }

          }
      }

      __guin_pidr = 0;
        if (__guin_mypid != __guin_pidr)
         {
            __guin_position = 0;
            __s2c_tmp = __suif_min(nn, __guin_blksz + -1 + __guin_blksz * __guin_mypid);
            for (__guin_p = __guin_blksz * __guin_mypid; __guin_p <= __s2c_tmp;
__guin_p++)
            {
                  x = 1 * __guin_p;
                  for (y = 0; y <= nn; y++)
                  {
                              i= 0;
                              j = 0;
                        MPI_Pack(&edgeImage[x][y], 1, MPI_BYTE, (char
(*)[2097152])__guin_buffer, 2097152, &__guin_position, MPI_COMM_WORLD);
                  }
            }
            MPI_Send((char (*)[2097152])__guin_buffer, 2097152, MPI_PACKED, __guin_pidr,
1 * (__guin_NP - 0 + 1) + __guin_pidr, MPI_COMM_WORLD);
         }
   /* ------ END HAND CODED GATHER ----- */


    if (__guin_mypid == 0)
      {
        gettimeofday(&tv2, (struct timezone *)0);
        elapsed_time = (double)(int)(tv2.tv_sec - tv1.tv_sec) + (double)(int)(tv2.tv_usec -
tv1.tv_usec) / 1000000.0;
        printf("%lf\n", elapsed_time);
        oFile = fopen(argv[2l], "wb");
        if (oFile != (struct _IO_FILE *)0)
          {
            fprintf(oFile, "P5\n%d %d\n%d\n", w, h, max);
            fwrite(edgeImage, h, w, oFile);
            fclose(oFile);
          }
        MPI_Finalize();
        return 0;
      }
    MPI_Finalize();
  }
```