Paraguin

Compiler for Distributed Systems

Paraguin Compiler Version 2.1

User Manual

Clayton S. Ferner

31 July 2013

Contents

1 Overview

The Paraguin compiler pass is a semi-automatic parallelizing compiler that will generate message-passing code using MPI that can run on any distributed-memory parallel system for which MPI has been installed. The goal of the Paraguin project is to create an open-source automatic parallelizing compiler that serves as an infrastructure to promote research in message-passing code generation. The user needs to direct the compiler for how it should parallelize the program through the use of **pragma** statements.

1.1 Getting Started

1.1.1 Installing Paraguin

Before installing the Paraguin compiler, MPI (http://www.mpi-forum.org/) and SUIF 1.3 (http://suif.stanford.edu) should also be installed and working correctly. The particular SUIF packages that are required by the Paraguin compiler are: *basesuif*, *baseparsuif*, and *suifbuilder*.

Installing the Paraguin compiler pass:

1. Download the tar/compressed file from http://people.uncw.edu/cferner/Paraguin/index.html
2. Copy the tar/compressed file to `$SUIFHOME/src`
3. cd to `$SUIFHOME/src` and untar/uncompress the file (`tar -zxvf paraguin-2_1_tar.gz`)
4. copy the file `paraguin/commands.def` to `$SUIFHOME/src/scc` (or add the PARAGUIN pass defined in `paraguin/commands.def` to the existing `scc/commands.def` file.)
5. cd to `$SUIFHOME/src/scc` and type `make install`
6. cd to `$SUIFHOME/src/paraguin` and type `make install`
7. To test, run the command `./doUnitTests` from within the `$SUIFHOME/src/paraguin` directory

1.1.2 Compiling and Running a Program

The Paraguin compiler pass will transform a sequential program into one that uses MPI to run on a distributed-memory system. MPI must first be installed before Paraguin can work. The user should refer to the MPI documentation for their particular machine. By modifying the `commands.def` file in the `scc` directory and rebuilding `scc`, Paraguin is registered as a compiler pass. The command to compile a program is:

```
$ scc -DPARAGUIN -cc mpicc <filename> -o <program>
```

Options can be passed through to the Paraguin pass by adding the option '-option PARAGUIN "<options>"', where <option> can be:

`-debug <n>`	Produce debugging info
`-help`	Print this info and quit

For example:

```
$ scc -option PARAGUIN "-debug 1" -DPARAGUIN -cc mpicc test1.c -o test1
```

The program can then be run using the `mpirun` command.

If the user wishes to see the transformations made by the Paraguin pass, the user only needs to provide the "-.out.c" option to scc. This will stop the compilation process just before using mpicc. For example:

```
$ scc -DPARAGUIN -.out.c test1.c
```

This will create a filed called `test1.out.c` that is the transformed program suitable for compilation through `mpicc`. The user is free to modify this code and compile directly with `mpicc` if they are not competely satisfied with the result.

Algorithm 1 Hello World Program

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

#include <stdio.h>

int __guin_rank = 0;

int main(int argc, char *argv[])
{
    char hostname[256];
    printf("Master process %d starting.\n", __guin_rank);

    #pragma paraguin begin_parallel

    gethostname(hostname, 255);
    printf("Hello world from process %3d on machine %s.\n", __guin_rank, hostname);

    #pragma paraguin end_parallel

    printf("Goodbye world from process %d.\n", __guin_rank);

    return 0;
}
```

1.1.3 Hello World

Algorithm 1 is a parallel hello world program using Paraguin. The parallel region (between the **pragma** statements, see section 2.1.1) will be executed by all processors. The statements outside of this region will be executed by the master processor only (processor zero). The variable `__guin_rank` is a reserved identifier (see section 4.1), which represents the id of each processor. The declaration of the `builtin_va_list` is to deal with a compatibility issue between `SUIF` and `gcc`.
This program would then be compiled and run as follows:

```
$ scc -DPARAGUIN -cc mpicc hello.c -o hello.out
$ mpirun -np 8 hello.out
Master process 0 starting.
Hello world from process 0 on machine compute-1-4.local.
Goodbye world from process 0.
Hello world from process 2 on machine compute-1-4.local.
Hello world from process 3 on machine compute-1-4.local.
Hello world from process 1 on machine compute-1-4.local.
Hello world from process 6 on machine compute-1-5.local.
Hello world from process 7 on machine compute-1-5.local.
Hello world from process 5 on machine compute-1-5.local.
Hello world from process 4 on machine compute-1-5.local.
$
```

2 Parallelization

2.1 User-directed Parallelization

The user provides instructions to the compiler through the use of **pragma** statements. Each **pragma** statement has the following syntax:

```
#pragma paraguin <name> <parameters>
```

The compiler will only look at **pragma**s with the Paraguin name. This means that the user can put other types of **pragma**s in the code which will be ignored. Likewise, other compilers (such as `gcc`) will ignore the Paraguin **pragma**s. The user is then able to compile the same program (without modification) using `gcc` to create a sequentially executed version of the program as well as Paraguin to create a parallel version of the program. Furthermore, the user can provide other **pragma** statements to create a hybrid program (such as using OpenMP **pragma**s).

Although **pragma**s can be placed through the program, occassionally they will be attached to nested statements. The Paraguin compiler will traverse the parse tree looking for the **pragma**s. However, it is possible that it can miss a **pragma** or the meaning is changed from the placement of the **pragma**. In this case, one can put a semicolon (;) in front of the **pragma** to force it to be attached to a `NOP` instruction that is not nested. For example, consider the following code:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = 0;
#pragma paraguin begin_parallel
```

The above **pragma** will be attached to the previous statement resulting in a parse tree that resembles:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        a[i][j] = 0;#pragma paraguin begin_parallel
    }
}
```

In the case that the **pragma** does not seem to be reconized or is in the wrong place, the programmer can solve this by placing a semicolon (;) in front of the **pragma** as follows:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = 0;
; #pragma paraguin begin_parallel
```

The semicolon will create a `NOP` instruction that follows the nested `for` loops and is therefore at the top level of nesting.

2.1.1 Parallel Region

The user can specify that all processors should execute parts of a program by putting that code inside a parallel region. Statements that are outside of a parallel region will only be executed by the master processor (processor zero).

```
... // Sequential Code
#pragma paraguin begin_parallel
... // Code to be executed by all processors
#pragma paraguin end_parallel
... // Sequential Code
```

The remaining processes still exists. Instead they simply ignore the code outside of the parallel region. In other words, parallel regions and sequential regions are implemented by using a simple `if` statement to check whether the rank of the process is zero. Because of this, it is incorrect to nest parallel regions or to have a parallel region span across a block of code. For example, the following is *INCORRECT*:

```
#pragma paraguin begin_parallel
if (x > 10) {

    #pragma paraguin end_parallel // INCORRECT

}
```

In fact, parallel regions are only defined at the topmost level within a function body. The user may create a sequential region within a parallel region simple by surrounding that code with an `if` statement such as:

```
int __guin_rank = 0;
...
    #pragma paraguin begin_parallel
    ...
    if (__guin_rank == 0) {
       // Sequential region within a parallel region
       ...
    }
    ...
    #pragma paraguin end_parallel
...
```

Parallel regions exists only within the scope of a single function. The parallel region will be closed at the end of a function whether or not there is an explicit `end_parallel` **pragma**. For any function which is to have any portion of its body executed in parallel, a *call* to that function should be within a parallel region. If not, then the entire function will executed by the master process only. For example:

```
void f()
{

    ... // Sequential Code
    #pragma paraguin begin_parallel
    ... // Code to be executed by all processors
    #pragma paraguin end_parallel
    ... // Sequential Code

}
int main()
{

    #pragma paraguin begin_parallel
    f(); // f() executed in parallel
    #pragma paraguin end_parallel
    f(); // f() executed sequentially regardless of its own parallel regions

}
```

Any initializations that are done as part of a declaration will be within a sequential region and therefore will only be initialized on the master process. If an initialization is to be done by all processes, the initialization should be done in an executable statement within a parallel region *AFTER* the declaration. For example:

```
int main()
{
    int x = 10, y; // x will be initialized on master only
    #pragma paraguin begin_parallel
    y = 20; // y will be initialized on all processes
    #pragma paraguin end_parallel
}
```

The reason is because the declaration is not an executable statement, but the initialization is. Therefore, the initialization of `x` is an executable statement the *PRECEDES* the begin parallel.

THERE IS NO IMPLIED BARRIER AT THE BEGINNING OR END OF A PARALLEL REGION. The user needs to use an explicit barrier (see section 2.1.2) to do that.

2.1.2 Barrier

A barrier is a point in the program where all processors must synchronize before any processors may proceed. The Paraguin barrier will perform a barrier on **MPI_COMM_WORLD**, which means that it involves *all* processors, not a subset. The syntax is:

```
#pragma paraguin barrier
```

The barrier ***must*** be within a parallel region, otherwise it will be ignored since a barrier outside a parallel region would result in a deadlock.

2.1.3 Parallel For

To execute a **for** loop in parallel, the user should first place the **for** loop inside a parallel section. Then the loop needs to be declared as a parallel **for** by placing a **forall pragma** in front of it. The **forall pragma** applies only to the next **for** loop nest (lexicographically). To encode this in a **pragma** statement, the syntax is as follows:

```
#pragma paraguin forall [chunksize]
```

The optional `chunksize` parameter allows the user to specify the number of chunks (or blocks) of iterations assigned to each processor. The `chunksize` must be an integer literal or an integer variable. ***THE*** `chunksize` ***CANNOT BE A PRE-PROCESSOR CONSTANT (such as*** `#define N 100`***) BECAUSE THE PRAGMA IS A PREPROCESSOR DIRECTIVE AS WELL AND THE CONSTANT WILL NOT BE PROPERLY RESOLVED.*** There is no dynamic or guided scheduling available. The reason is because there would be significant interprocessor communication in order to implement those scheduling options.

The next **for** loop that follows the **forall** directive will be tiled in chunk size iterations across all available processors. If the user does not specify a chunk size, then the default chunk size will be computed as:

$$\text{chunksize} = \left\lceil \frac{ub - lb + 1}{NP} \right\rceil$$

where lb is the lower bound of the loop, ub is the upper bound, and NP is the number of physical processors. This is *block* scheduling. The user can implement *cyclic* scheduling by simply making the chunk size equal to 1. If the user does provide a chunk size, then the each processor will iterate though cycles of chunk size iterations where the number of cycles is:

$$\text{number of cycles} = \left\lceil \frac{ub - lb + 1}{\text{chunksize} * NP} \right\rceil$$

For example, to tile a simple for loop with block scheduling:

```
#pragma paraguin begin_parallel
#pragma paraguin forall
for (i = 0; i < n; i++) {

    for (j = 0; j < n; j++) {
        a[i][j] = i * j;

#pragma paraguin end_parallel
```

In the above example, each processors would exectute $\lceil n/NP \rceil$ iterations of the outermost loop, but no processor would go beyond n. ***Forall pragmas** cannot be nested*, although they can be applied to any loop within a loop nest. For example:

```
#pragma paraguin begin_parallel
for (i = 0; i < n; i++) {

    #pragma paraguin forall 1
    for (j = 0; j < n; j++) {
        a[i][j] = i * j;

#pragma paraguin end_parallel
```

In the above example, each processor will execute all iterations of the outermost loop, but the inner loop will be divide up using a cyclic schedule (because the chunksize is specified as 1).

The programmer may provide integer literals or integer variables for the chunk size. The user may *NOT* provide floating-point literals, pre-processor constants, or expressions as the chunksize. If the user wishes to use an expression for the chunk size, then they should assign that expression to a variable and use the variable as the chunk size. For example:

```
int chunk = 2 * n + 3;
...
#pragma paraguin forall chunk
```

2.1.4 Broadcast

The data of a program is likely to originally be located on the master processor, since reading from input is likely to be done outside a parallel region. Therefore data may need to be sent to the all of the other processors. The most effecient way to send data to *ALL* processors is to use a broadcast. This is accomplished using the **broadcast pragma**. The syntax is as follows:

```
#pragma paraguin bcast <list of variables>
```

The root processor will be the processor with rank 0 (master). The list of variables is a whitespace separated list. Variables may be arrays or scalars of the following types: **byte**, **char**, **unsiged char**, **short**, **int**, **long**, **float**, **double**, and **long double**. There will be a separate broadcast performed for each variable in the list. For example:

```
int i, A[100][1000];
float x;
...
#pragma paraguin bcast i A x
```

If a variable is a pointer type, then only one element, to which the pointer points, of the base type will be broadcast. If the pointer is used to point to an array or a portion of an array, the user must provide the number of elements of the base type to send. This includes strings. There is no way at compile time to know how many elements to send using a pointer. For example, consider the following code:

```
char *s = “hello world”;
#pragma paraguin bcast s // INCORRECT
```

The above is incorrect for two reasons:

1. it will not broadcast the entire string; and
2. it may cause a segmentation fault if `s` does not point to any space with which to store the data on the various processors.

In other words, even if the pointer points to space to store the string on each processors, only the character 'h' (as a character) would be broadcast. There is no information in the symbol table that can be used to determine how many elements are in the array; especially since the pointer can be moved at runtime to point to something else.

The user specifies the number of elements by following the pointer variable with a parenthesized size, which can be a literal or variable. For example:

```
char *s = “hello world”;
int n = strlen(s) + 1;
#pragma paraguin begin_parallel
#pragma paraguin bcast n s( n )
```

In this example, `n` will be broadcast first, then `n` characters will be broadcast starting at the address of `s`, which is the correct result. The size can be either a symbol or a literal. Notice that, since `n` is used to determine how much will be broadcast using the pointer `s`, `n` must be broadcast first. ***NOTE: there should be a space between the parentheses and the symbol. This has to do with how SUIF processes the variable n.***

The same concept applies to arrays passed as parameters. The reason is because the **C** language implements array parameters as pointers. For example, consider the following code:

```
void f(int A[], int B[][N], int n)
{
    int C[N][N];
    #pragma paraguin begin_parallel
    #pragma paraguin scatter A( n ) B( n ) C
    ...
    #pragma paraguin end_parallel
}
void main(...)
{
    int A[N], B[N][N];
    #pragma paraguin begin_parallel
    f(A, B, N);
    #pragma paraguin end_parallel
}
```

Since `A` is in the parameter list of the function `f()`, `A` is a pointer to an integer (same as if it were declared as `int *A`). So when we broadcast `A`, we need to specify the number of elements. We do not need to worry about creating space for `A` since it was actually declared as an array in `main`, and therefore there are already `N` elements in `A` on each processor.

The `B` array is a two-dimensional array. The **C** language requires the size of the 2nd dimension (so that it can compute the row-major order mapping). Therefore, `B` is actually a pointer to a single-dimensional array of size `N`. We only need to specify the number of elements in the 1st dimention, since each element is an array of `N` integers instead of a single integer.

The `C` array is declared locally, which means it is a two-dimensional array and not a pointer. The size is not required because the symbol table entry for `C` indicates that its size is `N*N` integers.

2.1.5 Scatter

In order to scatter input on the various processors, one can use the **scatter** pragma. The syntax is:

```
#pragma paraguin scatter <list of variables>
```

The data is assumed to be on the master processor. The data will be divided up among the processor in chunk size portions. The chunk size is computed as:

$$\text{chunksize} = \left\lceil \frac{N}{NP} \right\rceil$$

where N is size of the data and NP is the number of physical processors.

User-specified chunksizes is not yet implemented. This will be coming in a future release.

Two very important distinctions between the Paraguin scatter and the MPI scatter:

1. The Paraguin scatter will scatter the data into the same name variable on the other processors; and
2. The Paraguin scatter will offset the chunks so that they stay in the same relative position in the variable.

To demonstrate the above differences, consider the following code:

```
int A[100];
for (i = 0; i < 100; i++) A[i] = i; // Initialize the values
for (i = 0; i < 100; i++)

    A[i] = A[i] + f(i);
```

If one were to implement this in MPI, one might do something like this:

```
int A[100], B[100];
if (rank == 0) for (i = 0; i < 100; i++) A[i] = i;// Initialize the values

MPI_Scatter(A, 100, MPI_INT, B, 100, MPI_INT, 0, MPI_COMM_WORLD);
for (i = 0; i < chunksize && rank * chunksize + i < n; i++)

    A[rank * chunksize + i] = B[i] + f(rank * chunksize + i);
```

The scatter will distribute the data as shown in Figure 1, assuming a chunk size of 2. Notice that the MPI_Scatter allows the programmer to distribute the data in the `A` array into another temporary array. This may be useful in MPI; however, in order to allow the programmer to parallelize code that is designed by thinking sequentially, we want to keep the data in the same array.

Even if we choose to scatter the `A` array into an array of the same name, the offsets of the values in the segments do not match the offsets of the values in the original array on the master. This is why one needs a separate index expression for the two references to the arrays on either side of the assignment. We want to allow the user to parallelize the original loop that does not have any reference to an array called `A`. That parallelized loop will have the iterations partitioned to processors, which will work on different parts of the array, where those parts are in their original location.

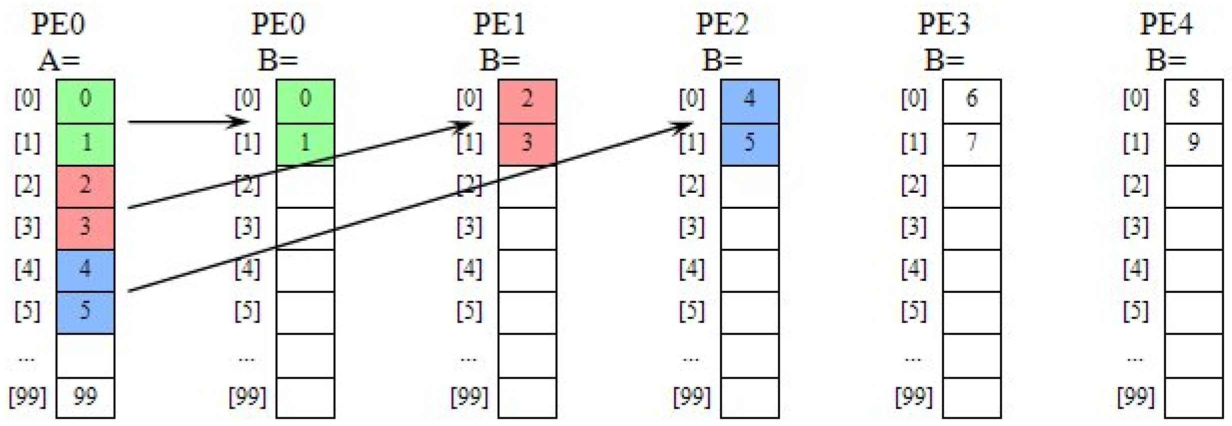

Figure 1: MPI_Scatter

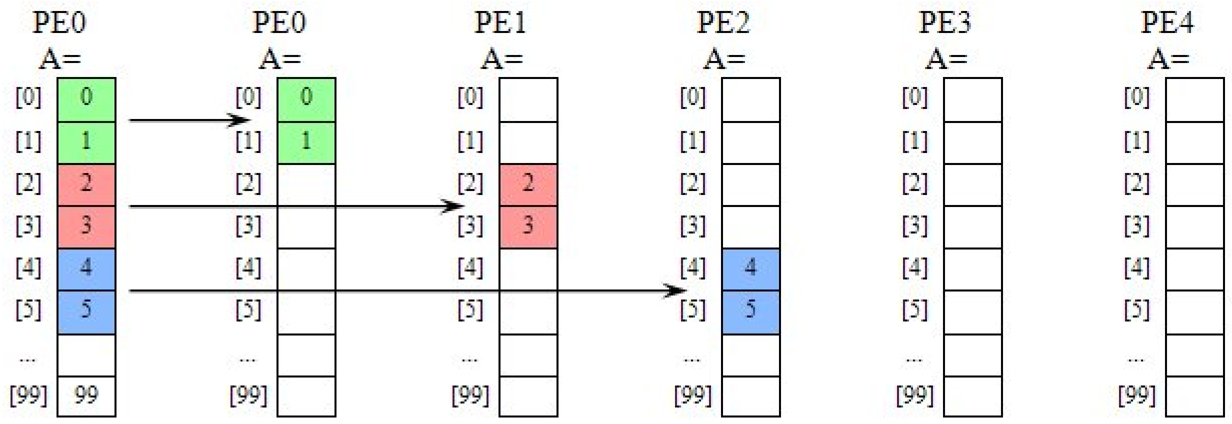

Figure 2: Paraguin Scatter

What we want the user ultimately to have is the following with the parallelized code to maintain the original array references:

```
int A[100];
for (i = 0; i < 100; i++) A[i] = i;
#pragma paraguin begin_parallel
#pragma paraguin scatter A
#pragma paraguin forall
for (i = 0; i < 100; i++)

    A[i] = A[i] + f(i);

#pragma paraguin end_parallel
```

When this gets parallelized by the compiler, the two array references on either side of the assignment will have the same expression for their indeces. That means we need the data to be scattered with the same offsets as in the original array as shown in Figure 2. The above code would be parallelized by the Paraguin compiler as:

```
chunksize = (int) ceil(((float) 100 - 0) / NP);
MPI_Scatter(&A[rank*chunksize], chunksize, MPI_INT, &A[rank*chunksize], chunksize,

    MPI_INT, 0, MPI_COMM_WORLD);
```

```
for (i = 0 + chunksiz * rank; i < min(100, 0 + chunksize * (rank + 1)); i++)
    A[i] = A[i] + f(i);
```

Notice how the array references are the same as they are in the original program. (Paraguin actually uses the MPI_Scatterv instead of MPI_Scatter to have greater control over the distribution of data.)

The same issue with broadcasting pointers to data exists with Scatter (as well as Gather). If the pointer points to an array (as is the case with array parameters), then the user needs to specify a size within parentheses following the pointer variable. For example, if `A` were passed as a parameter in the above code, the scatter would then have to be:

```
void f(int A[], int n)
{
    #pragma paraguin begin_parallel
    #pragma paraguin scatter A( n )
    #pragma paraguin forall
    for (i = 0; i < n; i++)
        A[i] = A[i] + f(i);
    #pragma paraguin end_parallel
    ...
```

Multi-dimensional arrays are scatter exactly as single-dimentsional array. The following code demonstrates two 3 dimensional arrays (one a parameter and one a local variable).

```
void f(int A[][M][L], int n)
{
    int B[N][M][L];
    ... // Initialize B
    #pragma paraguin begin_parallel
    #pragma paraguin scatter A( n ) B
    ...
```

Notice that the size of the 1st dimension of the `A` array needs to be provided because `A` is a pointer to a 2-dimensional array size `M*L`. However, the size of `B` does not need to be provided because it is a 3-dimensional array and not a pointer to a 2-dimensional array like `A`. Both arrays will be scattered such that each processor receives a partition of the first dimension of size $\lceil N/NP \rceil$.

The Paraguin compiler only scatters arrays on the first dimension. The user would have to rearrange the array (such as transpose) to scatter along other dimensions.

User-specified chunksizes is not yet implemented. This will be coming in a future release.

2.1.6 Gather

In order to gather the partial results from the various processors back to the master processor, one can use the **gather** `pragma`. The syntax is:

```
#pragma paraguin gather <list of variables>
```

The data is assumed to be divided up among the processor in chunk size portions. The chunk size is computed as:

$$\text{chunksize} = \left\lceil \frac{N}{NP} \right\rceil$$

where N is size of the data and NP is the number of physical processors. The data is also assumed to be offset chunksize $\times$ rank as it is in Figure 2. Similar to the scatter, the Paraguin gather differs from the MPI gather in two ways:

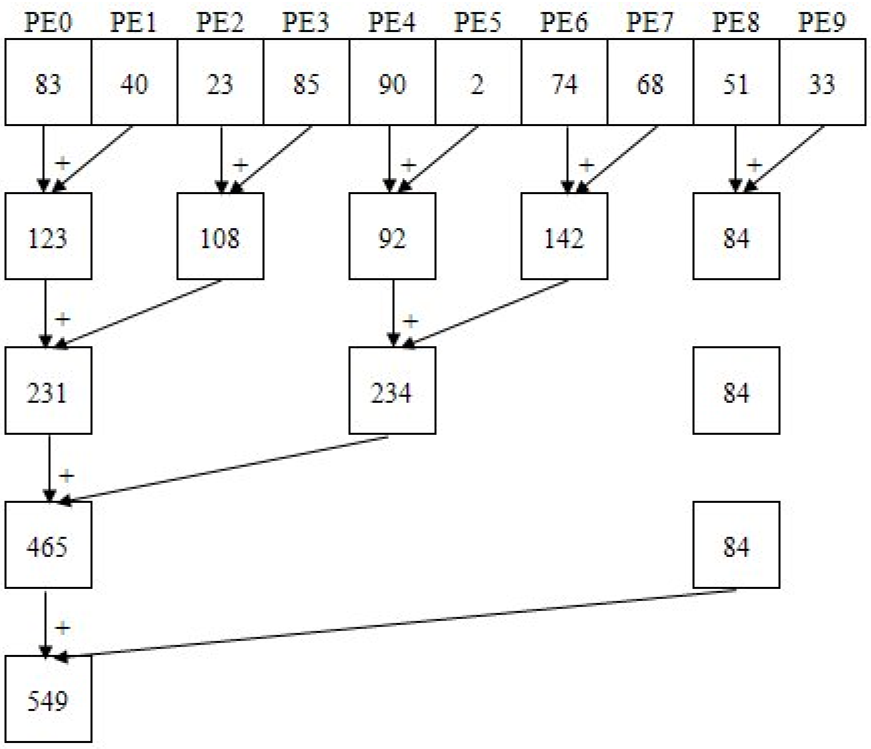

Figure 3: Example of a Reduction Using Summation

1. The Paraguin gather will gather the data from the same name variable on the other processors; and
2. The Paraguin gather will offset the chunks so that they stay in the same relative position.

All the same concepts of the scatter apply to the gather, except that the data is moved in the reverse direction. For example, the following code gathers rows of the first dimensions of both array A (pointer to a 2-dimensional array) and B (3-dimensional array):

```
void f(int A[][M][L], int n)
{
    int B[N][M][L];
    #pragma paraguin begin_parallel
    ... // Processors manipulate chunksize rows of both arrays
    ... // starting at row rank * chunksize.
    #pragma paraguin gather A( n ) B
    ...
}
```

2.1.7 Reduction

A reduction is where a collection of values is reduced to a single value by applying a commutative binary operator to the values. For example, summing the values: 83, 40, 23, 85, 90, 2, and 74 is 397. Typically operations that fit this requirement are: **summation**, **product**, **minimum**, **maximum**, **logical and**, **bitwise and**, **logical or**, **bitwise or**, **logical exclusive or**, and **bitwise exclusive or**. When the collection of values is spread across processors, then this reduction can be performed in a tree fashion, such as shown in Figure 3. The syntax for a reduction is:

```
#pragma paraguin reduction <op> <source data> <result data>
```

Table 1: Reduction Operators

Operator	Description	Allowed Datatypes
`max`	maximum	Integer[1] and floating point[2]
`min`	minimum	Integer and floating point
`sum`	summation	Integer and floating point
`prod`	produce	Integer and floating point
`land`	logical and	Integer
`band`	bitwise and	Integer
`lor`	logical or	Integer
`bor`	bitwise or	Integer
`lxor`	logical exclusive or	Integer
`bxor`	bitwise exclusive or	Integer
`maxloc`	maximum and location	Structure[3]
`minloc`	minimum and location	Structure

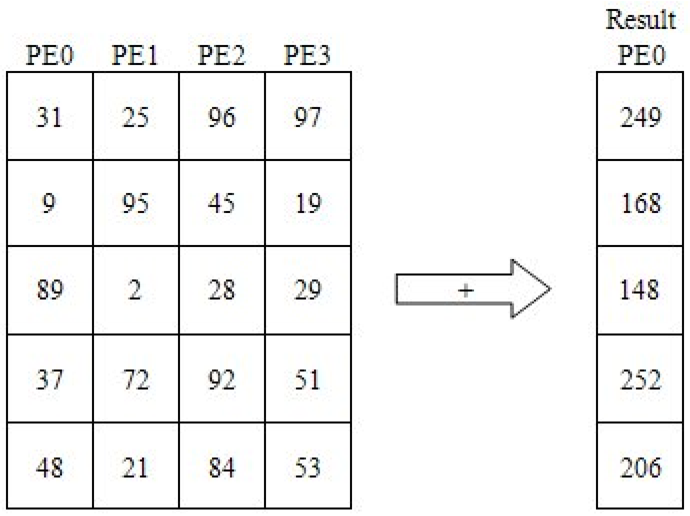

Figure 4: Reduction Applied to an Array

The `<op>` is the operation applied to the data to reduce it. The choices of operators are given in Table 1.

The user is required to provide space for the resulting data. Although MPI provided the ability to reduce the data in place (within the same space as the original data), the Paraguin reduction does not provide this. The resulting data needs to be in a separate space from the original data. The use of the reduction might look like the following:

```
double c, result_c;
...
#pragma paraguin begin_parallel
...
// Each processor assigns some value to the variable c
...
#pragma paraguin reduce sum c result_c
// The variable result_c on the master processor now holds the result of summing all the values in
```

The data may be arrays or pointers. In the case of an array, the reduction is applied to the values in the same relative position of the array (i.e. the same subscript value). For example, Figure 4 shows the result of applying a reduction to an array of size five. The code to perform the reduction shown in Figure 4 might look like the following:

```
double c[N], result_c[N];
...
#pragma paraguin begin_parallel
...
// Each processor assigns N values to the array c
...
#pragma paraguin reduce sum c result_c
// The array result_c on the master processor now holds the result of summing all the values in the
```

If the data is a pointer, it is assumed to point to a scalar. Providing the size is not yet implemented. This will be coming in a future release.

3 Patterns

Not all patterns are not yet implemented. These will be coming in a future release.

Many algorithms fall into classes of algorithms that follow patterns of dependencies. For example, some common patterns (shown in Figure 5)are scatter/gather, workpool, pipeline, stencil, divide-and-conquer, all-to-all, map-reduce, etc.

In the scatter/gather pattern, the master process prepares the data, scatters or broadcast it among the worker processes, then gathers or reduces the partial results back into the final result. The preparation of data may mean initializing the data or reading it from a file. Scattering the data means dividing up the data into peices where each worker receives some of the data but not all of it. Broadcast means that all workers receive the entire set of input data. The gather is where all of the partial data is collected again on the master process, whereas a reduction is where the partial results are reduced to a single value or a smaller set of data by applying an operation to the partial results such as summation.

In a workpool pattern, worker processes request work from the master, perform that work, report the results back to the master, and repeat until there is no work left to be done. A workpool has the nice feature of automatically load balancing. Embarrasingly parallel algorithms are usually well suited for a workpool pattern. Examples of problems that can use a workpool pattern are Monte Carlo solutions, graphic rendering (where each frame can be done independently), Traveling Salesman Problem, genetic algorithms, etc.

A pipeline patterns is one where the results of one stage are the inputs to the next stage. An assembly line is an example of a pipeline. Examples of algorithms that are pipelines are: summing, sorting, Sieve of Eratosthenes, Gaussian Elimination, etc.

A stencil pattern is one where a array or grid of processors iterate through some number of stages of processing and communication with their neighbors after each iteration. For example, each processor performs a computation, then exchange the result of that computation with the neighbors above, below, left, and right. Each processor uses the results of the computation from its neighbors in the next iteration of that computation. They they exchange results with their neighbors and continue. The stops after a some time. That time may be a fixed number of iterations or it may be after the data converges to a solution.

3.1 Scatter/Gather

The matrix addition example shown in section 4.2.2 is an example of a scatter/gather pattern. This pattern is not done with a single `pragma`, so it would best be described as a template than a pattern. Nonetheless, the code shown of matrix addition shows how one implements this pattern. The major sections of the code are:

1. Preparing the input data (in this case reading it from a file)
2. Scattering the data to the worker processes
3. Computing the partial results (in this case rows of the C matrix)
4. Gathering the partial results back to the master

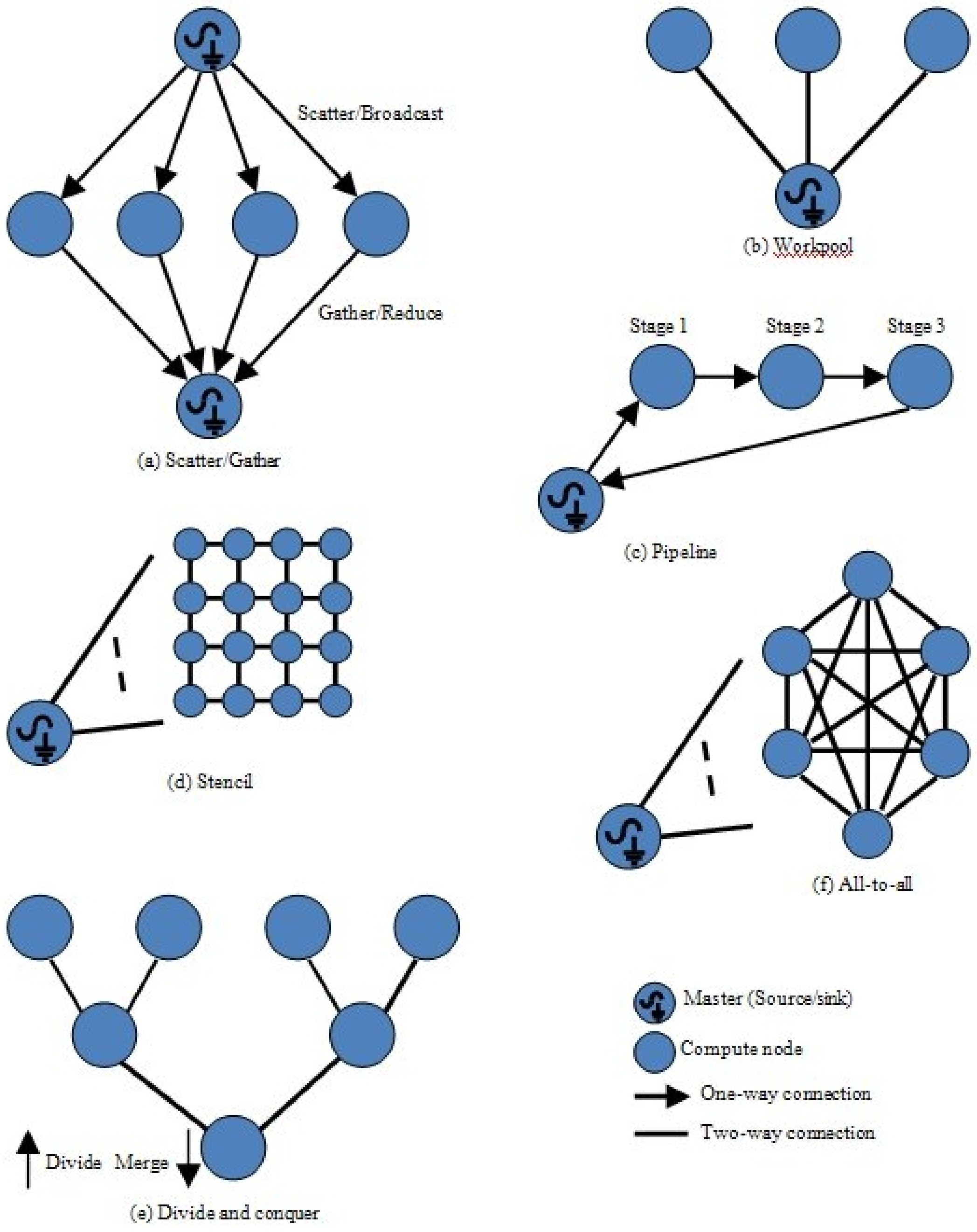

Figure 5: Examples of Patterns

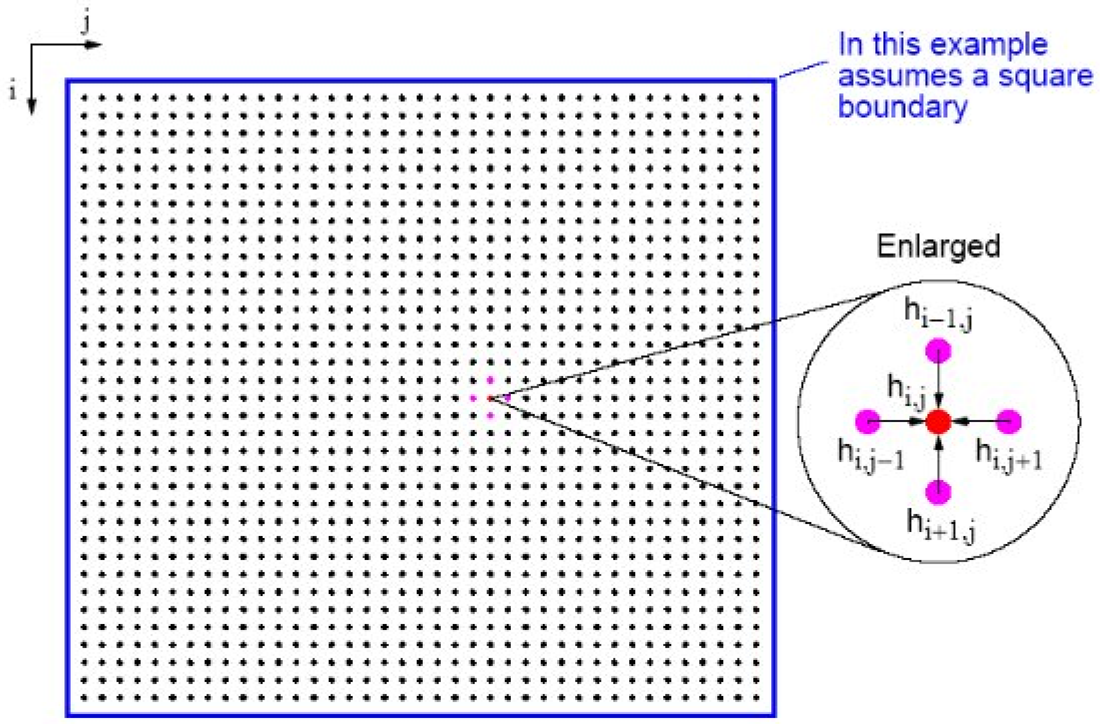

Figure 6: Jacobi Iteration

The Scatter/Gather pattern does not require only the use of scatter and gather. One can also use broadcast or reduction. An example the integration program shown in section 4.2.3. In this example, a function is being integrated between points a and b on the x-axis. The input are the values for a and b plus the number of rectangles to use for the approximation. These inputs are broadcast to the worker processors because they are scalars. Scatter does not make sense for scalars. Since each processor is computing a sum of areas of the rectangles, these partial sums need to be reduce to a final sum on the master. So the integration program uses broadcast and reduction instead of scatter and gather; yet, the patter is the same.

3.2 Workpool Pattern

3.3 Pipeline Pattern

3.4 Stencil Pattern

Only the 2-D stencil pattern is implemented. The 3-D stencil pattern will be coming in a future release.

The stencil pattern is one where there is a 2-D grid and each value is computed as some function of its current value and the values of its neighbors. A Jacobi iteration (shown in Figure 6) is an example of a stencil pattern. The sequential version of a Jacobi Iteration pattern could be the code shown in Algorithm 2:

Notice that the loops skip the border values. These values are fixed. Also notice that the new values are put into a new matrix. They are not copied back into the original matrix until all values have been computed.

If the new values are put into the A matrix as they are computed, the values will be different. For example, when computing the value `A[i][j]`, the values at `A[i+1][j]` and `A[i][j+1]` would be the previous iteration values whereas the values at `A[i-1][j]` and `A[i][j-1]` would be the newly computed values. The advantage of storing the newly computing values into the A matrix instead of a temporary matrix means that one does not need to double the amount of space required and and one does not need the second loop nest to copy

Algorithm 2 Jacobi Iteration Algorithm

```
int main()
{
     int i, j;
     double A[N][M], B[N][M];

     // A is intialized with data somehow

     for (time = 0; time < MAX_ITERATION; time++) {
         for (i = 1; i < N-1; i++)
             for (j = 1; j < M-1; j++)
               // Multiplying by 0.25 is faster than dividing by 4.0
               B[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) * 0.25;
         for (i = 1; i < N-1; i++)
             for (j = 1; j < M-1; j++)
               A[i][j] = B[i][j];
     }
     ...
}
```

the values into the original. This version of a Jacobi iteration is called Gauss-Seidel Relaxation. In addition to the advantages of less required space and time, the Gauss-Seidel Relaxation can converge faster; however, it may not converge at all. See [??] for more information on Gauss-Seidel Relaxation.

The main disadvantage of the Gauss-Seidel Relaxation is that it cannot be parallelized. It would required interprocessor communication for each iteration of the innermost loop causing it to execute sequentially. The algorithm shown above can be parallelized with communication occuring only after each iteration of the time loop.

The extra time required to copy the newly computing values in the B matrix back into the A matrix can be avoided by toggling between the two copies of the matrix. This is easily accomplished by making A a 3-dimensional array, where the first dimension is of size 2. This improvement is shown in Algorithm 3.

The Paraguin Stencil Pattern

The stencil pattern is specified to Paraguin by providing the stencil pragma. The syntax of the stencil pragma is:

```
#pragma paraguin stencil <data> <#rows> <#cols> <max iterations> <fname>
```

where `<data>` is a 3-dimensional array of size `[2][#rows][#cols]`. The outermost loop will iterate `<max_iterations>`. `<fname>` is the name of the function that will compute individual values.

Algorithm 4 shows how this would be setup using Paraguin.

Explanation of Code

The data should be declared as a 3-dimensional array, where the size of the 1st dimension is 2. The 1st dimension is used for the two copies of the data as the values are modify with each iteration of the `time` loop. The array can be of any type (not necessariy `double`).

The declaration of the variable `__guin_current` is declared globally because it is a Paraguin predefined variable. The only reason for declaring it is so that it can be used to access the correct copy of the matrix (either 0 or 1). Alternatively, if the `<max_iterations>` is even then the final results will be in `<data>[0]`, otherwise the final results will be in `<data>[1]`.

Algorithm 3 Improved Jacobi Iteration

```
int main()
{

    int i, j, current, next;
    double A[2][N][M];

    // A[0] is intialized with data somehow and duplicated into A[1]

    current = 0;
    next = (current + 1) % 2;

    for (time = 0; time < MAX_ITERATION; time++) {
       for (i = 1; i < N-1; i++)
          for (j = 1; j < M-1; j++)
            // Multiplying by 0.25 is faster than dividing by 4.0
            A[next][i][j] = (A[current][i-1][j] + A[current][i+1][j] +
              A[current][i][j-1] + A[current][i][j+1]) * 0.25;

       current = next;
       next = (current + 1) % 2;
    }

    // Final result is in A[current]
    ...
}
```

Algorithm 4 Paraguin Stencil Pattern Example

```
int __guin_current; // This is needed to access the last copy of the data

// Function to compute each value
double computeValue (double A[][M], int i, int j)
{


    // This function must be in a parallel region so that all processors will compute
    // The correct values.
    #pragma paraguin begin_parallel

    return (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) * 0.25;
    // Multiplying by 0.25 is faster than dividing by 4.0

    #pragma paraguin end_parallel

}

int main()
{

    int i, j, n, m, max_iterations;
    double A[2][N][M];

    // A[0] is intialized with data somehow and duplicated into A[1]

    #pragma paraguin begin_parallel
    n = N;
    m = M;
    max_iterations = TOTAL_TIME;

    #pragma paraguin stencil A n m max_iterations computeValue

    #pragma paraguin end_parallel
    // Final result is in A[__guin_current] or A[max_iterations % 2]

}
```

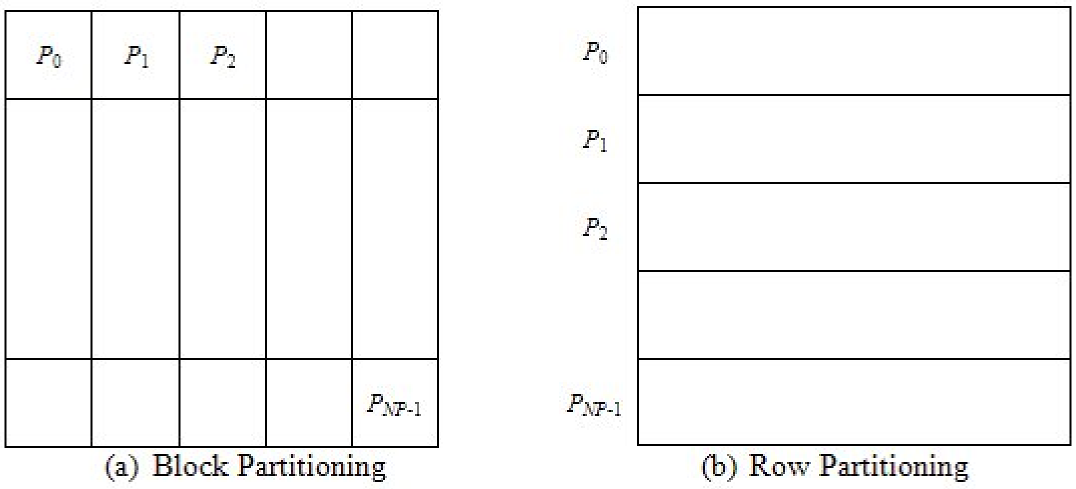

Figure 7: Block Partitioning vs. Row Partitioning

The `computeValue()` function is used to perform the individual calculations. This is a user-defined function so that this pattern can be applied to many algorithms. For example, the calculation may differ depending on the values of `i` and `j`. The protocol for this function is:

```
<type> <fname> (<type> <data>[][], int i, int j)
```

The base type can be any type as long as it is consistent throughout. The data will be a pointer to a 2-dimensional array (which toggles between copies). `i` and `j` are the row and column of the value being calculated. This function should return the new value to be put into location `<data>[__guin_next][i][j]`. The name of the function is user-defined. Whatever the user chooses for the name will be given in the **stencil pragma**.

The variables `n`, `m`, and `max_iterations` are used simply because pre-processor constants cannot be used in the **pragma** statements. For example, `#define N 100` cannot be then put into `#pragma paraguin stencil A N ...` . The reason is because the **pragma** is also a pre-precessors construct and the value of `N` will not get resolved correctly. The user can put literals or variables in the **pragma** statement, but not pre-processor constants.

Although it might be convenient to have the iterations stop when the change in values is less than a tolerance, the problem with this is that there needs to be a mechanism by which all processors can agree whether to terminate or continue. If they all don't agree, then there is a deadlock, because some processors will be waiting for data that won't arrive.

Paraguin will divide $\#rows - 2$ iterations among the available processors. The reason that it divides $\#rows - 2$ rows is because the first and last rows are considered to be boundary values and constant. Even though the stencil pattern as depicted in Figure5 implies the individual values or blocks of values will be divided up among the processors as shown in 7(a), Paraguin will only divide the rows as shown in 7(b). The reasons for this are:

1. Dividing the the matrix across both row and column will produce a partitioning that has too fine granularity. There will be too much interprocessors communication.

2. Dividing the matrix only across rows requires only one row of data to be sent with each message and only two messages are sent between any pair of processors.

3. Dividing the matrix only across rows requires communication between iterations to and from nearest neighbors. In other words, processor P_i will send its last row to processor P_{i+1} and its first row to processor P_{i-1}. This communication pattern is depicted in Figure8.

The stencil **pragma** will insert code into the resulting program to perform the following steps:

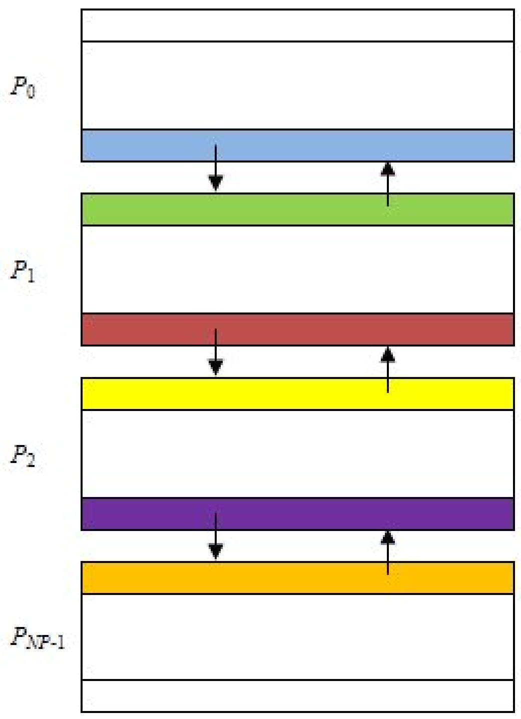

Figure 8: Stencil Communication Pattern

1. The 3-dimensional array given as an argument to the stencil **pragma** is broadcast to all available processors. Although all processors do not need the entire array, they need the rows above and below the rows for which they are responsible. Broadcasting the data ensure that all processors start with the correct values. Broadcasting is not quite as fast as scattering the data, but is it still relatively quick.

2. `__guin_current` is set to zero and `__guin_next` is set to one.

3. A loop is created to iterate `max_iteration` number of times. Within that loop, code is inserted to perform the following steps:

   (a) Each processor (except the last one) will send its last row to the processor with rank one more than its own rank

   (b) Each processor (except the first one) will receive the last row from the processor with rank one less than its own rank

   (c) Each processor (except the first one) will send its first row to the processor with rank one less than its own rank

   (d) Each processor (except the last one) will receive the first row from the processor with rank one more than its own rank

   (e) Each processor will iterate through the values of the rows for which it is responsible and use the function provided as an argument to the stencil **pragma** to compute the next value. This newly computed value is computed from the values in `<data>[__guin_current]` and is stored in `<data>[__guin_next]`.

   (f) `__guin_current` and `__guin_next` toggle

4. The data is gathered back to the root processor (rank 0).

3.4.1 Using a test to terminate stencil pattern

The Paraguin stencil pattern will execute a fixed number of iterations. The user may wish to execute the pattern for an unknown number of iterations until a test passes (or fails) based on the data. For example, the user may wish to run the stencil pattern until the data converges to a solution or the changes to the data with each iteration is smaller than some tolerance. This is achievable using the above pattern. Basically, the user places the stencil pattern within another (logical) loop to test the termination condition. The reason for doing this is because the test will require some sort of gather or reduction. It is not practical to perform such a test requiring significant communication after *each* iteration of the pattern. It is much more efficient to perform the pattern for a fixed and large number of iterations, then test the termination condition to determine if the pattern should run for another large number of iterations.

There are two reasons the testing for a termination condition requires communication:

1. the data is scattered among the processors; and

2. all processors need to agree whether to continue or terminate.

The 1st point requires all processors to test their partition of the data. The 2nd point means that the termination condition needs to be shared among the processors somehow. If the processors do not agree whether to continue or terminate, they will *deadlock*. Agreeing will probably require a reduction and broadcast. Algorithm 5 and 6 demonstrates how to achieve this.

Explanation of Code

New variables have been added to the code: `done` (`int`) which is used to signal completion, `diff` (`double`), `max_diff` (`double`), and `tol` (`double`). The variable `tol` is used to determine if the change in values is close enough to be consider to have converged. This variable needs to be initialized within a parallel region so that it is initialized on all processors. The variable `diff` is used to calculate the absolute difference between the last iteration value and the current value. The variable `max_diff` is the maximum difference the the current processors partition of data. After each processor determines the maximum absolute difference in the values from the current iteration and the previous iteration, these values need to be reduce. The reason is because some partitions of the data may have converged already, while others have not. In order for the processors to agree, the `max_values` need to be reduce to determine the overall maximum difference. Once that maximum difference is determine, it then needs to be broadcast to all processors so they all agree whether to continue or terminate.

3.5 Divide-and-Conquer Pattern

3.6 All-to-all Pattern

4 Miscellaneous

4.1 Reserved Identiers

The following names are used by the Paraguin compiler:

Algorithm 5 Stencil Pattern using a Termination Condition

```
int main()
{

    int i, j, n, m, max_iterations, done;
    double A[2][N][M], diff, max_diff, tol;

    // A[0] is intialized with data somehow and duplicated into A[1]

    #pragma paraguin begin_parallel

    // tol is used to determine if the termination condition is met
    // When the change in values are all less than tol, the values
    // have converged sufficiently.
    tol = 0.0001;
    n = N;
    m = M;
    max_iterations = TOTAL_TIME;

    done = 0; // false
    while (!done) {

        ; // This is the make sure the following pragma is inside the while loop
        #pragma paraguin stencil A n m max_iterations computeValue

        max_diff = 0.0;

        // Each processor determines the maximum change in values of the
        // partition for which it is responsible. The loop bounds need to be 1
        // and n-1 to match the bounds of the stencil. Otherwise, the
        // paritioning will be incorrect.
        #pragma paraguin forall
        for (i = 1; i < n - 1; i++) {
           for (j = 1; j < n - 1; j++) {

             diff = fabs(room[__guin_current][i][j] - room[__guin_next][i][j]);
             if (diff > max_diff) max_diff = diff;
           }
        }

        ; // This is needed to prevent the pragma from being located in the above loop nest
        // Reduce the max_diff's from all processors
        #pragma paraguin reduce max max_diff diff

        ; // This is needed to prevent the pragma from being located in the above if statement
        // Broadcast the diff so that all processors will agree to continue or terminate
        #pragma paraguin bcast diff

        // Terminattion condition if the maximum change in values is less than the tolorance.
        if (diff <= tol) done = 1;
    }
```

Algorithm 6 Stencil Pattern using a Termination Condition (continued)

```
        #pragma paraguin end_parallel
        // Final result is in A[__guin_current]
        // Cannot use max_iterations % 2

    }
```

Identier	type	Description
`__guin_NP`	`int`	Number of physical processors
`__guin_rank`	`int`	Current processor's processor id
`__guin_chunksz`	`int`	Chunk size (number of partitions per processor)
`__guin_cycle`	`int`	Cycle iteration (used when user provides a chunk size)
`__guin_nCycles`	`int`	Number of cycles each processors much execute chunksize iterations of the loop (used when user provides a chunk size)
`__guin_sendsizes`	`int (*)[1]`	A pointer to an array (dynamically allocated of size NP) of ints used in the MPI_Scatterv() and MPI_Gatherv() fuctions.
`__guin_sendoffsets`	`int (*)[1]`	A pointer to an array (dynamically allocated of size NP) of ints used in the MPI_Scatterv() and MPI_Gatherv() fuctions.
`__guin_status`	`MPI_Status`	MPI status variable used in MPI_Recv()
`__guin_iteration`	`int`	The current iteration number of the stencil pattern. This is the loop index.
`__guin_current`	`int`	The current iteration of the stencil pattern data. In the stencil pattern, the computation is made into a new copy of the stencil. Then the copy becomes the original in the next iteration. This toggles with each iteration: `__guin_current = __guin_next;` `__guin_next = (__guin_current + 1) % 2;`
`__guin_next`	`int`	The next iteration of the stencil patter data. This toggles opposite with the __guin_current: `__guin_current = __guin_next;` `__guin_next = (__guin_current + 1) % 2;`
`__guin_dataPtr`	`void *`	A pointer to the current stencil pattern data
`__guin_size`	`int`	Size of the stencil data minus 2. This is used to calculate the chunk size of the loop to do the stencil compution. It is assumed that the values of the outermost edges of the stencil will not change. Therefore the number of rows in the computation should be $n - 2$ where n is the number of rows in the stencil data.

These identifiers are inserted into the `<file>.out.c` program. It may be the case that the user would like to reference these variables for debugging purposes. The compiler will look for each variable in the global symbol table and, if they are found, will use the existing variable. If they are not found, the compiler will then add definitions for them. Therefore, the user may declare any of these variables in their program as *global variables* and then be able to reference them. Initialization is unnecessary, although the user may want to initialize them so that the program will be correct when using another compiler (such as `gcc`). However, *THE USER SHOULD NOT ATTEMPT TO MODIFY THESE VARIABLES AFTER THE DECLARATION.* The hello world example from sections 1 and 4.2.1 shows how to use the processor id in print statements, making it easier to know which processor produces which line of output. The user may also want to use the environmental variable `PARAGUIN` to specify that some statements should only appear in the source code to the Paraguin compiler and not to others. For example, using the PARAGUIN environmental variable in `#ifdef PARAGUIN` allows the users to make thier program compilable with `gcc` as well as Paraguin.

4.2 Examples

4.2.1 Hello World

File: $SUIFHOME/src/paraguin/examples/hello/helloWorld.c

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

#include <stdio.h>

int __guin_rank = 0;

int main(int argc, char *argv[])
{
    char hostname[256];
    printf("Master process %d starting.\n", __guin_rank);

    #pragma paraguin begin_parallel

    gethostname(hostname, 255);
    printf("Hello world from process %3d on machine %s.\n", __guin_rank, hostname);

    #pragma paraguin end_parallel

    printf("Goodbye world from process %d.\n", __guin_rank);

    return 0;
}
```

Explanation of Code

The variable `__guin_rank` is a variable used by the Paraguin compiler. In order to use it in the print statements, we need to declare it as a global variable and initialize it. The Paraguin compiler will use this variable instead of creating a new one. The advantage of doing this is so that this same program can be compiled with `gcc` to create a sequential version of the program without modifying the source code. Although we are free to reference (read) the `__guin_rank` variable, it could cause undesirable results if we tried to modify it.

The first `printf` statement ("Master process ...") is outside of a parallel region; therefore, only the master processor will execute it. Although the other processors have been created and are running, they simply do not execute that first `printf` statement. All processors (including the master) will execute the second `printf` statement. Then, again, only the master will execute the last `printf` statement.

4.2.2 Matrix Addition

```
#define N 512

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

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

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

int main(int argc, char *argv[])
{
    int i, j, error = 0;
    double a[N][N], b[N][N], c[N][N];
    char *usage = "Usage: %s file\n";
    FILE *fd;

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

    if (!error && (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]);
        error = -1;
    }

    // The error code is broadcast to all processors so that they know
    // to exit. If we just had a “return -1” in the above two if statements
    // then the master only would exit and the workers would not, causing
    // an MPI runtime error. Handling the error this way allows the master
    // to indicate to the workers that they need to exit immediately.

    #pragma paraguin begin_parallel
    #pragma paraguin bcast error
    if (error) return error;
    #pragma paraguin end_parallel

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

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

    fclose(fd);

    #pragma paraguin begin_parallel

    // Scatter the input to all processors.
    #pragma paraguin scatter a b

    // Parallelize the following loop nest assigning iterations
    // of the outermost loop (i) to different partitions.

    #pragma paraguin forall
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }

    // This semicolon is here to prevent the gather pragma from being
    // attached to the statement nested within the above loop nest.
    // A semicolon creates a NOOP instruction AFTER the loop nest to which
    // the gather pragma is attached.
    ;
    #pragma paraguin gather c
    #pragma paraguin end_parallel

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

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

    #pragma paraguin begin_parallel

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

    #pragma paraguin end_parallel
}
```

Explanation of Code

The major sections of the code are:

1. Preparing the input data (in this case reading it from a file).
    (a) Get the name of the file off of the command line if available.
    (b) Broadcast the error. The reason for this is that if there is an error (specifically the user didn't provide a valid filename) then the master needs to let the workers know to exit immediately. Without this, there will be an MPI runtime error as the master exits but the workers do not.

2. Scattering the data to the worker processes. The data will be scattered by rows. In other words, each processor (including the master) will receive $\lceil \frac{N}{NP} \rceil$ number of rows of the input matrices.

3. Computing the partial results (in this case rows of the C matrix). Since the outermost for loop is executed as a forall, the iterations of the outermost loop will be executed by the processors where each is responsible for $\lceil \frac{N}{NP} \rceil$ iterations of the interation space. Since the outermost loop represents the rows of the matrices, each processor will compute $\lceil \frac{N}{NP} \rceil$ rows of the resulting matrix.

4. Gathering the partial results back to the master

5. Print the results to stdout

4.2.3 Numerical Integration

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

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double f(double x)
{

    #pragma paraguin begin_parallel
    return 4.0 * sin(1.5*x) + 5;
    #pragma paraguin end_parallel

}

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

    char *usage = "Usage: %s a b N\n";
    int i, error = 0, N;
    double a, b, x, y, size, area, overal_area;
    struct timeval tv;

    if (argc < 4) {
        fprintf (stderr, usage, argv[0]);
        error = -1;
    }

    a = atof(argv[1]);
    b = atof(argv[2]);
    N = atoi(argv[3]);

    if (b <= a) {
        fprintf (stderr, "a should be smaller than b\n");
        error = -1;
    }

    #pragma paraguin begin_parallel
    #pragma paraguin bcast error
    if (error) return error;
```

```
        // This semicolon is here to prevent the bcast pragma from being
        // attached to the statement nested within the above if statement.
        ;
        #pragma paraguin bcast a b N

        size = (b - a) / N;
        area = 0.0;

        #pragma paraguin forall
        for (i = 1; i < N; i++) {
            x = a + i * size;
            y = f(x);
            area += y * size;
        }

        ;
        #pragma paraguin reduce sum area overal_area

        #pragma paraguin end_parallel

        printf ("area = %lf\n", overal_area);
    }
```

Explanation of Code

The numerical integration program determins the integral of a function for x values between a and b. The function used in the above function is $f(x) = 4cos(1.5x) + 5$. Although this function can be integrated mathematically, it is shown here simply as an example. This technique would be used on functions that are much more difficult to integrate mathematically. The program works by creating N rectangles between a and b. The height of each rectangle is considered to be $f(x')$, where x' is the left side of the rectangle. So the area of each rectangle is $\frac{b-a}{N}f(x')$. The sum of the areas of these rectangles is an approximation of the area under the function between a and b. The more rectangles, the better the approximation. In this example, the major sections of the code are:

1. Prepare the input
    (a) Get a, b, and N off the command line if available
    (b) Broadcast the error so all processors know whether or not to continue
2. Broadcast the inputs.
3. Computing the partial results. The parital results will be the sum of the areas of a partition of rectangles
4. Reduce the partial results back to the master. The result is the sum of all the partial sums.
5. Print the results to stdout

The function `f()` is the function being integrated. The body of this function ***MUST*** be within a parallel region because all processors need to be able to compute $y = f(x)$ for various values of $x \in [a..b]$. If the body is not within a parallel region, then the function `f()` will not return a value for the worker processors.