



Paraguin Compiler Version 1.0

User Manual

Clayton S. Ferner

22nd November 2002

1 Overview

The Paraguin compiler is a quasi-parallelizing compiler that will generate message-passing code using MPI that can run on any distributed-memory parallel system that has MPI installed. The current version is not fully automated. Instead, the user needs to direct the compiler for how it should parallelize the program. We plan to fully automate the compiler in future releases. We felt that it was important to allow the user to direct the compilation for two reasons: First so that the compiler could be useful before it is fully implemented, and second so that the user can always override the decisions made by the compiler. We feel that it is important to give the user control. The input to the compiler for instructions related to parallelization is provided through the use of **pragma** statements.

The user is likely to experience poor performance for programs that involve much communication. This is the nature of parallel execution on a distributed system. We are continually trying various techniques to improve performance as well as long range plans for building a compiler infrastructure to promote research in improving performance of message-passing code.

2 Getting Started

2.1 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. Copy the tar/compressed file to `$SUIFHOMESRC`
2. cd to `$SUIFHOMESRC` and untar/uncompress the file (`tar -zxvf paraguin-1_0_1_tar.gz`)
3. Edit the file `runparaguin` and make any necessary changes for the current system (such as the name of the MPI compiler)
4. cd to `paraguin` and type `make install`

2.2 Compiling and Running a Program

The Paraguin compiler will transform a sequential program into one that use 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.

A script called `runparaguin` has been provided to compile a program. The script will run `scc` (the SUIF front end), the paraguin compiler pass, `s2c` (which produces a C program will calls to MPI), a script to clean up a few messy details, and finally `mpicc` (which compiles to an executable linked with the MPI library). The command to compile a program is:

```
$ runparaguin [<options>] <filename>
```

Options are:

<code>-Send2Self</code>	Send messages to self (default = don't send to self)
<code>-combDepth [n]</code>	Combine loops to depth n (default = -1, all levels)
<code>-map [n]</code>	set mapping to n (0 = Cyclic, 1 (default) = Block)
<code>-aggregate [n]</code>	Aggregate messages to same partition 0 = don't, 1 (default) = indep. loops, 2 = push p in
<code>-debug</code>	Produce debugging info
<code>-help</code>	Print this info and quit

If the program compiles successfully, it will produce a file called `<filenamebase>.out.c`, which is a C program with calls to MPI, and an executable with the extension `<filenamebase>.out`. The user is free to modify the `<filenamebase>.out.c` file to suit their needs and recompile with `mpicc`. The program can then be run using the `mpirun` command.

2.3 Hello World

The following is a parallel hello world program. The parallel region (between the `pragma` statements, see section ?? below) will be executed by all threads. The statements outside of this region will be executed by the master thread only (thread zero). The variable `__guin_mypid` is a reserved identifier (see section ?? below), which represents the id of each thread.

```
#include <stdio.h>
int __guin_mypid;
int main(int argc, char *argv[])
{
    printf("Master thread %d starting.\n", __guin_mypid);

    ;
    #pragma paraguin begin_parallel

    printf("Hello world from thread %d.\n", __guin_mypid);

    ;
    #pragma paraguin end_parallel

    printf("Goodbye world from thread %d.\n", __guin_mypid);
    return 0;
}
```

This program would then be compiled and run as follows:

```
$ runparaguin hello.c
Processing file hello.spd
Parallelizing procedure: "main"
$ mpirun -np 8 hello.out
Hello world from thread 3.
Hello world from thread 1.
Hello world from thread 7.
Hello world from thread 5.
Hello world from thread 4.
Hello world from thread 2.
Hello world from thread 6.
Master thread 0 starting.
Hello world from thread 0.
Goodbye world from thread 0.
$
```

3 Automatic Parallelization

3.1 Partitioning

Automatic partitioning of a loop nest is not implemented in this version. The user is required to specify the partitioning via the **for pragma** (see section ??).

3.2 Data Dependence

Automatic determination of data dependence is not implemented in this version. The user is required to specify the data dependence via the **dep pragma** (see section ??).

3.3 Parallelization of a For Loop

Given the partitioning of a **for** loop, the compiler will transform the loop nest such that each partition can potentially run on a separate processor. The mapping of partitions to processors is done through the mapping and scheduling (see section ??).

3.4 Message-passing Generation

Given the partitioning of a **for** loop and the data dependencies, the compiler will generate loop nests that will pack and send data to processors that need the data as well as generate loop nests that will receive and unpack data. These loop nests will be inserted before and after the main parallel loop. Then the loops will be merged to attempt to overlap communication with computation as much as possible.

3.5 Distributing or Broadcasting Data

Automatic distribution of data to other processors is not implemented in this version. The user is required to specify the distribution via the **bcst pragma** (see section ??).

3.6 Gathering Data

Automatic gather of data from other processors is not implemented in this version. The user is required to specify the gathering via the **gather pragma** (see section ??).

3.7 Mapping and Scheduling

Once a loop has been transformed to be parallel and the message-passing loops have been inserted, the partitions then need to be mapped to physical processors. At present, the compiler will support block and cyclic mapping of partitions to processors. This is specified by command line options to the compiler. In future releases, this information may be specified via **pragmas**, or the user may be able to specify a scheduling heuristic that should be used.

4 User-directed Parallelization

4.1 Affine Systems of Inequalities

Much of the information used by the compiler, such as loop bounds, partitioning, and data dependence is in the form of affine equations. Therefore, the **pragmas** are designed for the user to provide the necessary information to the compiler in the form that it needs. The user should first understand how to describe these affine equations in the form of a system of inequalities.

The bounds of any **for** loop are assumed to be affine expressions of containing **for** loop variables, literal constants, and symbolic constants. However, the current version does not support symbolic constants. For example:

```
for (i = 0; i <= N; i++)
  for (j = i + 1; j <= N; j++) {
    X[j][i] = X[j][i] / X[i][i];           // Statement S1
    for (k = i + 1; k <= N; k++)
      X[j][k] = X[j][k] - X[j][i] * X[i][k]; // Statement S2
  }
```

The bounds of the loops can be represented as the system of inequalities:

$$\begin{array}{rcl} 0 & \leq & i \leq N \\ i + 1 & \leq & j \leq N \\ i + 1 & \leq & k \leq N \end{array}$$

We will transform these inequalities so that all variables appear on one side:

$$\begin{array}{rcl} i & \geq & 0 \\ N - i & \geq & 0 \\ j - i - 1 & \geq & 0 \\ N - j & \geq & 0 \\ k - i - 1 & \geq & 0 \\ N - k & \geq & 0 \end{array}$$

This system can be represented in matrix form as $A\vec{x} + \vec{c} \geq \vec{0}$. For example:

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ N \\ -1 \\ N \\ -1 \\ N \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The zero matrix does not need to be stored because it is always a zero matrix. The constant vector \vec{c} can be rolled into the matrix of coefficients by making the first member of the vector \vec{x} be a 1. This results in the following system:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ N & -1 & 0 & 0 \\ -1 & -1 & 1 & 0 \\ N & 0 & -1 & 0 \\ -1 & -1 & 0 & 1 \\ N & 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ i \\ j \\ k \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Now we only need to store \vec{x} and the matrix A .

4.2 Pragma

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 **pragmas** with the paraguin name. This means that you can put other types of **pragmas** in the code and they will be ignored. Likewise, other compilers (such as gcc) will ignore the paraguin **pragmas**. The user should be able to compile the same program using gcc (sequental) as well as paraguin (parallel).

A set of **pragma** statements should first be preceded by a semicolon on a line by itself. This caveat exists because of the way SUIF stores **pragmas** in the intermediate representation. A **pragma** statement will be attached to the most recent statement, even if that statement is nested deep inside many blocks. This makes it more difficult to find them. By placing a semicolon on a line by itself before a set of **pragmas**, a **NOOP** statement is created at the top level (within a function) to which the **pragmas** are attached.

4.2.1 Parallel Region

The user can specify that all threads 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 thread (thread zero). The remaining threads still exists. There are not killed, but rather they simply ignore the code outside of the parallel region

```

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

```

4.2.2 Parallel For (Partitioning)

To execute a **for** loop nest in parallel, the user should first place the **for** loop inside a parallel section. Then the loop nest needs to be declared as a parallel **for** by placing a **forall pragma** in front of it and providing the partitioning. The **forall pragma** applies only to the next **for** loop nest (lexicographically). The partitioning describes how iterations of the loop will be assigned to *virtual processors* or *partitions*. The number of partitions can be assumed to be infinite (actually it will be bounded by the total number of iterations). The partitions will later be mapped to physical processors whose quantity is determined at load-time.

The user needs to create an assignment of iterations to partitions. For example, $p = j$ means that each iteration of the j loop will be executed by a different partitions. Another example is $p = i - j - 1$, which means that the partitions cut diagonally through the iteration space. Once the assignment of iterations to partitions is determined, this is then represented as a system of inequalities. For example, $p = i - j - 1$ can be represented as:

$$p \leq i - j - 1 \text{ AND } p \geq i - j - 1$$

which can be transformed to:

$$-p + i - j - 1 \geq 0 \text{ AND } p - i + j + 1 \geq 0$$

This produces the following system (assuming that the loop nest has indices i , j , and k):

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

To encode this in a **pragma** statement, the syntax is as follows:

```
#pragma paraguin forall <X> <A>
```

Each variable of \vec{x} is listed in order, although “C” (for “constant”) or some other symbol should be used instead of 1, followed by the values of each row of A . The reason that a “C” should be used instead of a 1 for \vec{x} is because the compiler is looking for the first numeric, which is assumed as the beginning of A . In fact, the compiler actually ignores the names of the variables, but instead only uses the number of variables to determine the number of columns. Also, zero should be written in hexadecimal (“0x0”), because a simple “0” seems to be lost by SUIF. The above system of inequalities, used as the partitioning, can be described in a **pragma** as follows:

```

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

```

4.2.3 Data Dependence

The data dependence also needs to be provided to the compiler. This is a very tedious step and will be automated in future releases. The data dependence should be described as a mapping from a *read access* of an array to the *write access* that produce the value used. A read or write *access* not only includes the particular *array reference* within a statement but also includes the *iteration instance*. The array reference is specified by a simple integer, enumerated starting at zero lexicographically. For example:

```

for (i = 0; i <= N; i++)
  for (j = i + 1; j <= N; j++) {
    X[j][i] = X[j][i] / X[i][i];           // Statement S1
    for (k = i + 1; k <= N; k++)
      X[j][k] = X[j][k] - X[j][i] * X[i][k]; // Statement S2
  }

```

The array references would be as follows:

Statement	Array Mention	Reference Number
S1	X[j][i] (write)	0
S1	X[j][i] (read)	1
S1	X[i][i]	2
S2	X[j][k] (write)	3
S2	X[j][k] (read)	4
S2	X[j][i]	5
S2	X[i][k]	6

The iteration instance is a vector of the loop variables. For example $\vec{i} = [i, j, k]^T$, where i, j, k are integer values withing the loop bounds, is an iteration instance for the above loop nest.

An array access is both the array reference and an iteration instance. A data dependence is a mapping from an array access that reads a value to the array access that produced it. For example, in the above loop nest, there is a data dependence from array reference 3 and iteration instance $\vec{i}_w = [i_w, j_w, k_w]^T$ to array reference 6 and iteration instance $\vec{i}_r = [i_r, j_r, k_r]^T$, such that

$$\begin{aligned}
 i_w &= i_r - 1 \\
 j_w &= i_r \\
 k_w &= k_r
 \end{aligned}$$

This can be rewritten as the following system:

$$\begin{bmatrix}
 1 & 1 & 0 & 0 & -1 & 0 & 0 \\
 -1 & -1 & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 1 & 0 & -1 & 0 & 0 \\
 0 & 0 & -1 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & -1 \\
 0 & 0 & 0 & -1 & 0 & 0 & 1
 \end{bmatrix} \cdot \begin{bmatrix}
 1 \\
 i_w \\
 j_w \\
 k_w \\
 i_r \\
 j_r \\
 k_r
 \end{bmatrix} \geq \begin{bmatrix}
 0 \\
 0 \\
 0 \\
 0 \\
 0 \\
 0
 \end{bmatrix}$$

All dependence **pragmas** (there may be several) should also appear before the loop nest, and they only apply to the next loop nest. The syntax for each data dependence is:

```
#pragma paraguin dep <write reference> <read reference> <X> <A>
```

where $\langle X \rangle$ is a vector of the write and read loop variable names and $\langle A \rangle$ is the matrix of coefficients that describes the mapping from the read iteration instance to the write iteration instance. For example, the above data dependence would be described in the following **pragma** statement.

```
#pragma paraguin dep 3 6   C   iw   jw   kw   ir   jr   kr \
                          -1  -1  0x0  0x0   1  0x0  0x0 \
                          1   1  0x0  0x0  -1  0x0  0x0 \
                          0x0 0x0   1  0x0  -1  0x0  0x0 \
                          0x0 0x0  -1  0x0   1  0x0  0x0 \
                          0x0 0x0  0x0   1  0x0  0x0  -1 \
                          0x0 0x0  0x0  -1  0x0  0x0   1
```

4.2.4 Broadcast

The data of a program is likely to originally be located on the master thread, since reading from input is likely to be done outside a parallel region. Therefore data may need to be sent to the other processors. This would be done by *broadcasting* the data. Although future releases may allow for distributing only those parts of the array that each processor actually needs, the broadcast is a simply way to accomplish the distribution in a reasonably quick manner. It is not clear which is faster: broadcasting too much information in $\log NP$ time or transmitting less information in NP time (NP being the number of processors). Any data that needs to be available to all processors should be broadcast using the **broadcast pragma**. The **pragma** also needs to appear before the loop nest. The syntax is as follows:

```
#pragma paraguin bcast <X>
```

where $\langle X \rangle$ is the name of a variable.

4.2.5 Gather

Once the loop nest has finished, then the results may be spread among the processors. They then need to be brought back to the master thread (thread zero). This is done by using a **gather pragma**, which should also appear before the loop nest. The syntax is as follows:

```
#pragma paraguin gather <write reference> <X> <A>
```

where $\langle X \rangle$ is a vector of the write loop variable names and $\langle A \rangle$ is the matrix of coefficients that gives the iteration instances where a value is written and never modified again. For example, in the loop nest above, array reference 3 will write final values to the array whenever $i = j - 1$. This would be written as the following **pragma**:

```
#pragma paraguin gather 3   C   i   j   k \
                          1   1  -1  0x0 \
                          -1  -1   1  0x0
```

All instances of the array reference 0 will also produce final values. In this case, the system of inequalities is empty and produces the following **gather pragma**:

```
#pragma paraguin gather 0   C   i   j \
                          0x0 0x0 0x0
```

This system will always be true, therefore all iteration instances will transmit their array writes back to the master thread.

5 Miscellaneous

5.1 Reserved Identifiers

The following names are used by the Paraguin compiler:

Identifier	type	Description
<code>__guin_NP</code>	<code>int</code>	Number of physical processors
<code>__guin_blksz</code>	<code>int</code>	Block size (number of partitions per processor)
<code>__guin_mypid</code>	<code>int</code>	This threads processor id
<code>__guin_pidr</code>	<code>int</code>	Receiving threads processor id
<code>__guin_pidw</code>	<code>int</code>	Sending threads processor id
<code>__guin_buffer</code>	<code>char []</code>	Buffer of data to be transmitted
<code>__guin_position</code>	<code>int</code>	Number of bytes in the buffer
<code>__guin_status</code>	<code>MPI_Status</code>	Status of the message
<code>__guin_p</code>	<code>int</code>	Partition number
<code>__guin_pr</code>	<code>int</code>	Receiving partition number
<code>__guin_pw</code>	<code>int</code>	Sending partition number

These identifies are inserted into the `file.out.c` program. However, 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 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 and then be able to reference them. However, **THE USER SHOULD NOT ATTEMPT TO MODIFY THEM**. The hello world example above shows how to use the processor id in print statements, making it easier to know which thread produces which line of output.

6 Examples