High-Level Programming of PIM Lite*

Jay B. Brockman
Department of Computer Science and Engineering, University of Notre Dame
Center for Advanced Computing Research, Caltech
jbb@cse.nd.edu

July 24, 2003

Contents

1 Introduction ........................................ 1

2 Basic Programming Constructs ......................... 2
  2.1 Motivating Example and Sequential C Implementation ................ 2
  2.2 Allocating PIM Memory ................................ 3
  2.3 Moving Threads to the Neighborhood of an Address ..................... 4
  2.4 Programming with Multiple Threads ............................... 5
  2.5 Profiling ........................................... 9

3 Data Structures ......................................... 13
  3.1 Distributed Linked List ................................ 13
  3.2 Distributed Tree ....................................... 15

4 Data Synchronization ..................................... 18
  4.1 Mutexes .............................................. 18
  4.2 Condition Variables ................................... 18

5 The pim_vector Data Type and Wide-Word Operations ................. 18

6 pimlib Implementation .................................. 18

1 Introduction

With PIM, independent processors are distributed throughout the memory system. This provides not only
the opportunity to perform computations in parallel, but also to schedule computation “close” to the site of
the data. The technique of moving the computation to the data, rather than the reverse, can be especially
effective when data sets have a high degree of spatial locality.

To properly orchestrate PIM programs, the programmer should have control over the following:

• ability to control data placement, either to collocate on the same processing node or to distribute it
  across different nodes,

• ability to flexibly specify where a given computation should take place,

• ability to package up the execution state of a thread, ship it over a network, and restart it at another
  location.

*PIM Lite has been funded by JPL under the HTMT project, by DARPA under the DIVA project, and by DARPA through
Cray Inc. as part of the HPCS program and the Cascade project.
2 Basic Programming Constructs

2.1 Motivating Example and Sequential C Implementation

The following simple example illustrates these mechanisms for PIM Lite. Suppose that we want to evaluate the statement

\[ c \leftarrow a^2 + b^2 \]

where the values of variables \( a, b, \) and \( c \) are stored somewhere in main memory. Assuming that \( a, b, \) and \( c \) are physically located on different PIM nodes, there are several different ways of performing the computation. A few possibilities are listed below. These examples are not necessarily designed to be efficient, but rather to demonstrate different methods for organizing the computation.

- perform all arithmetic operation serially on the node containing \( c \)'s value (in the neighborhood of \( c \)), copying the values of \( a \) and \( b \) to the neighborhood of \( c \),
- evaluate \( a^2 \) and \( b^2 \) in parallel in their own neighborhoods, then copy the results to \( c \)'s neighborhood to calculate their sum,
- start a thread in \( a \)'s neighborhood and calculate \( a^2 \), then move the thread to \( b \)'s neighborhood, carrying the value of \( a^2 \), and calculate \( b^2 \), then finally carry the values of \( a^2 \) and \( b^2 \) to \( c \)'s neighborhood, calculate the sum, and store the result.

We begin with a sequential C program that performs the computation using three functions: \textbf{main}, \textbf{square}, and \textbf{sum}, shown in Listing 1.

Listing 1: Single-threaded C example

```c
square(int *x, int *y)
{
    *y = *x * *x;
}

sum(int *x1, int *x2, int *y)
{
    *y = *x1 + *x2;
}

main()
{
    int *a, *b, *c, tmp1, tmp2;
    a = (int*) malloc(sizeof(int));
    b = (int*) malloc(sizeof(int));
    c = (int*) malloc(sizeof(int));
    *a = 3;
    *b = 4;
    square(a, &tmp1);
    square(b, &tmp2);
    sum(&tmp1, &tmp2, c);
    printf("c = %d\n", *c);
}
```

In this program, \( a, b, c, \) \textbf{tmp1}, and \textbf{tmp2} are local variables of function \textbf{main}. Variables \( a, b, \) and \( c, \) are pointers that contain addresses in main memory, while \textbf{tmp1} and \textbf{tmp2} hold integer values. These 5 variables,
together with an instruction pointer, comprise the state of an activation of function `main`. Physically, these 5 variables would be part of `main`'s activation record or frame. Similarly, the pointer variables `x` and `y` would be part of function `square`'s activation frame and the pointer variables `x1`, `x2`, and `y` would be part of function `sum`'s frame.

### 2.2 Allocating PIM Memory

The program fragment in Listing 2 shows a PIM version of the same program with some of the basic PIM controls, which are defined in the PIM library “`pimlib`”.

Listing 2: Using `pim_malloc` to allocate data across a PIM fabric.

```c
#include "pimlib.h"

square(int *x, int *y)
{
    *y = *x * *x;
}

sum(int *x1, int *x2, int *y)
{
    *y = *x1 + *x2;
}

main()
{
    int *a, *b, *c, tmp1, tmp2;

    pim_systemConfigure(16);

    a = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
    b = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
    c = (int*) pim_malloc(sizeof(int), PIM_HOME);

    *a = 3;
    *b = 4;

    pim_systemPrintWhere(a,"a");
    square(a, &tmp1);
    pim_systemPrintWhere(b,"b");
    square(b, &tmp2);
    pim_systemPrintWhere(c,"c");
    sum(&tmp1, &tmp2, c);
    printf("\nc = %d\n", *c);
}
```

The program output of the program in Listing 2 is given in Listing 3.

Listing 3: Output of program in Listing 2

accessed by thread T0 on node N0 b: address on node N15
accessed by thread T0 on node N0 c: address on node N0
accessed by thread T0 on node N0

```text
  c = 25
```
Here’s an explanation of the constructs used in the example:

**pim_systemConfigure(int n)** defines a PIM system with n nodes, where the address space is effectively distributed evenly across the nodes. This procedure also creates the tables needed by the pthreads emulation of the PIM virtual machine that map memory addresses and threads to PIM nodes.

**pim_addr pim_malloc(int nbytes, pim_addr neighborhood)** allocates and returns a pointer to a block of nbytes bytes of PIM memory, in the neighborhood of (on the same node as) address neighborhood. The value for neighborhood can be any address generated by a previous call to pim_malloc, or one of several macros.

- **PIM_HOME** on the home node, node 0
- **PIM_HERE** on the same node where the current thread is running,
- **PIM_RANDOM** on a randomly chosen (uniform distribution across all nodes) PIM node
- **PIM_NODE(n)** on PIM node n. Since the current implementation does not take network topology into consideration, it is preferred to use **PIM_RANDOM** rather than **PIM_NODE(n)** to distribute data structures evenly across the address space.

**pim_systemPrintWhere(pim_addr addr, char *msg)**, print a diagnostic messages stating the whereabouts of addr and the thread that accessed it. Message is of the form “msg: address on node Na accessed by thread T on node Nt”.

### 2.3 Moving Threads to the Neighborhood of an Address

The example in Listing 4 uses the **pim_MoveTo(pim_addr addr)** to move a thread to the neighborhood of (same node as) a given address.

Listing 4: Moving a thread

```c
main()
{
    int *a, *b, *c, tmp1, tmp2;

    pim_systemConfigure(16);

    a = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
    b = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
    c = (int*) pim_malloc(sizeof(int), PIM_HOME);

    *a = 3;
    *b = 4;

    pim_systemPrintWhere(a,"a");
    pim_MoveTo(a);
    pim_systemPrintWhere(a,"a");
    square(a, &tmp1);
    pim_MoveTo(b);
    pim_systemPrintWhere(b,"b");
    square(b, &tmp2);
    pim_MoveTo(c);
    pim_systemPrintWhere(c,"c");
    sum(&tmp1, &tmp2, c);
    printf("\n\n\nc = %d\n", *c);
}
```
The printed output of the program in Listing 4 as given in Listing 5. Here, the current (only) thread, T0, is moved to the neighborhood of a given memory address before attempting to read that address. The three calls to pim_alloc were all made by the thread running in the neighborhood of the home node, PIM_HOME. The first call to pim_systemPrintWhere was after the calls to pim_alloc but before the current thread was moved. The call to pim_systemPrintWhere was after moving the current thread to the neighborhood of a, and the remaining calls were after moving the thread to the neighborhoods of b and , respectively. Since c was allocated on PIM_HOME, thread T0 ends up on node N0.

2.4 Programming with Multiple Threads

In this section, we'll look at different ways to use multiple threads to evaluate \( c \leftarrow a^2 + b^2 \). In the first case, we'll run three threads sequentially, first one to calculate \( a^2 \), then one to calculate \( b^2 \) and then a third to add these together, just as was done in the program in Listing 2. In the next case, we'll calculate \( a^2 \) and \( b^2 \) concurrently with separate threads in the neighborhoods of \( a \) and \( b \), respectively, and use a third thread running in the neighborhood of \( c \) to fork the prior two threads and to join these threads and perform the final sum.

Forking and joining of threads is controlled by the functions pim_fork and pim_join. The function pim_fork activates a function call with a given argument, as part of “thread group” in the neighborhood of a given address. The syntax of pim_fork is given below.

\[
pim_fork(void*(*f)(void*), void *arg, pim_threadGroup *g, pim_addr nbrhd),
\]

where

\[
\begin{align*}
void*(*f)(void*) & \text{ is a pointer to a function named } f \text{ that return a void* and that takes a single argument also of type void*} \\
void *arg & \text{ is the argument of } f, \text{ which must be of type void*}. \text{ Typically, this argument is a struct} \\
& \text{ that is cast as a void* when the function is called, and then re-cast to it’s original type in the beginning of } f. \text{ This rather cumbersome syntax is needed to preserve the syntax of C for passing pointers to functions as arguments.} \\
pim_threadGroup *g & \text{ registers the thread being created as part of pim_threadGroup named } g, \text{ for the purpose of synchronizing a group of threads that will eventually be joined.} \\
pim_addr nbrhd & \text{ specifies the neighborhood in which the new thread should be created.}
\end{align*}
\]

The syntax of pim_join is much simpler:

\[
pim_join(pim_threadGroup *g) \text{ Wait until all threads in thread group } g \text{ have finished before continuing at the next instruction.}
\]

The first step in using pim_fork and pim_join to calculate \( c \leftarrow a^2 + b^2 \) is to rewrite the functions square and sum so that they take on a single void* as an argument. In order to do this, we define struct types squareArgs_t and sumArgs_t to hold the “actual” arguments to these functions, and then pass a pointer to one of these structures into the function as a void*. Once inside the function, we can then recast the argument structure to its original type and extract the “actual” argument values. The code for the new struct types and the new function definitions is shown in Listing 6.

In order to evaluate \( c \leftarrow a^2 + b^2 \) sequentially from a single PIM node, in much the same manner as in Listing 2, we replace the function calls to the old versions of square and sum with pim_fork and pim_join combinations to the new versions of these functions. This new version of the main functions is given in Listing 7.

The printed output of this program is given in Listing 8. From this we see that the program actually involved 4 threads, T0, T1, T2, and T3, running sequentially on PIM node N0. T0 is the original thread created when main is invoked, T1 and T2 are the new threads forked for the two calls to square, and T3 is the thread forked for the call to sum.

In this next example, we calculate \( a^2 \) and \( b^2 \) concurrently using two different threads on two different nodes. The main function of this program is shown in Listing 9 and the output of the program is shown in Listing 10. Note that this version of the program still uses four different threads, but threads T1 and T1...
Listing 5: Output of program in Listing 4

<table>
<thead>
<tr>
<th>Description</th>
<th>Node</th>
<th>Thread</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>N13</td>
<td>T0</td>
<td>N0</td>
</tr>
<tr>
<td>a</td>
<td>N13</td>
<td>T0</td>
<td>N13</td>
</tr>
<tr>
<td>b</td>
<td>N15</td>
<td>T0</td>
<td>N15</td>
</tr>
<tr>
<td>c</td>
<td>N0</td>
<td>T0</td>
<td>N0</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td>25</td>
</tr>
</tbody>
</table>

Listing 6: Redefining functions \texttt{square} and \texttt{sum} to take a single argument of type \texttt{void*}

```c
#include "pimlib.h"

typedef struct {
    int *x;
    int *y;
} squareArgs_t;

typedef struct {
    int *x1;
    int *x2;
    int *y;
} sumArgs_t;

void square(void *squareArgsVoid)
{
    squareArgs_t *squareArgs = (squareArgs_t*) squareArgsVoid;
    int* x = squareArgs->x;
    int* y = squareArgs->y;

    pim_systemPrintWhere(x,"x in square");
    *y = *x * *x;
}

void sum(void *sumArgsVoid)
{
    sumArgs_t *sumArgs = (sumArgs_t*) sumArgsVoid;
    int *x1 = sumArgs->x1;
    int *x2 = sumArgs->x2;
    int *y = sumArgs->y;

    pim_systemPrintWhere(y,"y in sum");
    *y = *x1 + *x2;
}
```
main()
{
    sumArgs_t sumArgs;
squareArgs_t squareArgs1, squareArgs2;
pim_threadGroup g1, g2, g3;
    int *a, *b, *c, tmp1, tmp2;

    pim_systemConfigure(16);
a = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
b = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
c = (int*) pim_malloc(sizeof(int), PIM_HOME);
    *a = 3;
    *b = 4;
pim_threadGroupInit(&g1);
pim_threadGroupInit(&g2);
pim_threadGroupInit(&g3);

    squareArgs1.x = a;
squareArgs1.y = &tmp1;
pim_systemPrintWhere(a, "a");
pim_fork(square, (void*) &squareArgs1, &g1, PIM_HERE);
pim_join(&g1);

    squareArgs2.x = b;
squareArgs2.y = &tmp2;
pim_systemPrintWhere(b, "b");
pim_fork(square, (void*) &squareArgs2, &g2, PIM_HERE);
pim_join(&g2);

    sumArgs.x1 = &tmp1;
    sumArgs.x2 = &tmp2;
    sumArgs.y = c;
pim_systemPrintWhere(c, "c");
pim_fork(sum, (void*) &sumArgs, &g3, PIM_HERE);
pim_join(&g3);

    printf("%d\n", *c);
pim_stop();
}

Listing 7: Sequential, multithreaded version of the program in 2

Listing 8: The printed output of Listing 7

a: address on node N13 accessed by thread T0 on node N0
x in square: address on node N13 accessed by thread T1 on node N0
b: address on node N15 accessed by thread T0 on node N0
x in square: address on node N15 accessed by thread T2 on node N0
c: address on node N0 accessed by thread T0 on node N0
y in sum: address on node N0 accessed by thread T3 on node N0
25
Listing 9: Calculating \( c \leftarrow a^2 + b^2 \), with threads to calculate \( a^2 \) and \( b^2 \) running concurrently on different nodes. A separate thread is forked to calculate the sum.

```c
main()
{
    sumArgs_t sumArgs;
    squareArgs_t squareArgs1, squareArgs2;
    pim_threadGroup g1, g2;
    int *a, *b, *c, tmp1, tmp2;

    pim_systemConfigure(16);
    a = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
    b = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
    c = (int*) pim_malloc(sizeof(int), PIM_HOME);
    *a = 3;
    *b = 4;
    pim_threadGroupInit(&g1);
    pim_threadGroupInit(&g2);

    squareArgs1.x = a;
    squareArgs1.y = &tmp1;
    squareArgs2.x = b;
    squareArgs2.y = &tmp2;

    pim_moveTo(c);
    pim_systemPrintWhere(c,"c in main");
    pim_fork(square, (void*) &squareArgs1, &g1, a);
    pim_fork(square, (void*) &squareArgs2, &g1, b);
    pim_join(&g1);

    sumArgs.x1 = &tmp1;
    sumArgs.x2 = &tmp2;
    sumArgs.y = c;
    pim_fork(sum, (void*) &sumArgs, &g2, PIM_HERE);
    pim_join(&g2);
    pim_systemPrintWhere(c,"c in main");

    printf("%d\n", *c);
    pim_stop();
}
```

Listing 10: Output of program in Listing 9

```
c in main: address on node N0 accessed by thread T0 on node N0
x in square: address on node N13 accessed by thread T1 on node N13
x in square: address on node N15 accessed by thread T2 on node N15
y in sum: address on node N0 accessed by thread T3 on node N0
25
```
are running concurrently in the neighborhoods of $a$ and $b$, respectively, as dictated by the last arguments in `pim_malloc`.

In a final version of the $c \leftarrow a^2 + b^2$ program, we use only a total of three threads, an original thread that forks two separate threads to calculate $a^2$ and $b^2$ in their respective neighborhoods, which in turn report back to the original thread, which then calculates the sum. Listing 11 gives the source code for the `main` function and Listing 12 shows the output of the program.

Listing 11: Program that calculates $c \leftarrow a^2 + b^2$ using three threads

```
main()
{
    sumArgs_t sumArgs;
    squareArgs_t squareArgs1, squareArgs2;
    pim_threadGroup g1;
    int *a, *b, *c, tmp1, tmp2;

    pim_systemConfigure(16);
    a = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
    b = (int*) pim_malloc(sizeof(int), PIM_RANDOM);
    c = (int*) pim_malloc(sizeof(int), PIM_HOME);
    *a = 3;
    *b = 4;
    pim_threadGroupInit(&g1);

    squareArgs1.x = a;
    squareArgs1.y = &tmp1;
    squareArgs2.x = b;
    squareArgs2.y = &tmp2;

    pim_moveTo(c);
    pim_systemPrintWhere(c,"c in main");
    pim_fork(square, (void*) &squareArgs1, &g1, a);
    pim_fork(square, (void*) &squareArgs2, &g1, b);
    pim_join(&g1);

    sumArgs.x1 = &tmp1;
    sumArgs.x2 = &tmp2;
    sumArgs.y = c;
    sum(&sumArgs);
    pim_systemPrintWhere(c,"c in main");

    printf("%d\n", *c);
    pim_stop();
}
```

2.5 Profiling

The function `pim_profilePrint()` creates a file named `profile.dat` that logs the number of threads per node over the execution of the program. A new entry is created in the profile with every call to `pim_fork`, `pim_join` or `pim_moveTo`.

To demonstrate this feature, Listing 13 shows a program that finds a Fibonacci number recursively. Each call to function `fib` forks two new threads on random PIM nodes and then joins them.

Figure 1 shows a plot of the thread profile generated by the Fibonacci program in Listing 13. As expected, the threads are evenly distributed across the nodes. The Matlab program used to generate the plot is shown in Listing 14.
Listing 12: Output of program in Listing 11

c in main: address on node N0 accessed by thread T0 on node N0
x in square: address on node N13 accessed by thread T1 on node N13
x in square: address on node N15 accessed by thread T2 on node N15
y in sum: address on node N0 accessed by thread T0 on node N0
c in main: address on node N0 accessed by thread T0 on node N0
25

Listing 13: Program to calculate a Fibonacci number recursively

```c
#include "pimlib.h"

typedef struct {
    int x;
    int y;
} fibArgs;

void* fib(void *args)
{
    fibArgs *myArgs, child1Args, child2Args;
    pim_threadGroup g;
    pim_threadGroupInit(&g);

    myArgs = (fibArgs*) args;

    if (myArgs->x == 0) myArgs->y = 0;
    else if (myArgs->x == 1) myArgs->y = 1;
    else {
        child1Args.x = myArgs->x - 1;
        pim_fork(fib, (void*)&child1Args, &g, PIM_RANDOM);

        child2Args.x = myArgs->x - 2;
        pim_fork(fib, (void*)&child2Args, &g, PIM_RANDOM);

        pim_join(&g);
        myArgs->y = child1Args.y + child2Args.y;
    }
    pim_stop();
}

int main(void)
{
    fibArgs args;
    pim_threadGroup g;

    pim_systemConfigure(4);
    pim_threadGroupInit(&g);

    args.x = 12;
    pim_fork(fib, (void*)&args, &g, PIM_RANDOM);
    pim_join(&g);
    printf("value:%d\n", args.y);
    pim_profilePrint();
    pim_stop();
}
```
Figure 1: Plot of thread profile generated by Fibonacci program in Listing 13
Listing 14: Matlab program for plotting the thread profile.

```matlab
% pimprofile.m
% Display graph of multithreaded PIM profile log

m = dlmread('profile.dat','
');
n = size(m,1);

for i = 1:n
    subplot(n+1,1,i);
    plot(m(i,:));
    ylim([0,max(max(m))]);
    ylabel(sprintf('node %d',i));
    if i==1
        title('Thread Count per Node');
    end
end
subplot(n+1,1,n+1);
plot(sum(m),'r');
xlabel('time (sample)');
ylabel('total');
```
3 Data Structures

3.1 Distributed Linked List

One of the key properties of mobile threads is their ability to efficiently access data structures that have spatial locality, making local memory references and only moving between nodes when necessary. The following example demonstrates creation and traversal of a linked whose elements are distributed in groups over a set of PIM nodes.

Listing 15 shows the function `stripeList` that builds a linked list with 16 elements distributed over 4 PIM nodes, with 4 consecutive elements on each node. It uses the function `pim_moveTo` to move to a new PIM node after allocating 4 list elements on the same node.

```c
typedef struct _elt {
    int val;
    struct _elt *next;
} elt;

typedef struct {
    elt *head;
} listArgs_t;

/*
 * Single thread to build up a linked list with 16 elements striped 
 * across the system with 4 elements per node.
 */
void* stripeList(void* listArgsVoid)
{
    int i, j;
    elt *head, *newElt, *curElt;
    listArgs_t* listArgs = (listArgs_t*) listArgsVoid;
    head = NULL;

    for (i = 0; i < 4; i++) {
        pim_moveTo(PIM_NODE(i));
        for (j = 0; j < 4; j++) {
            newElt = (elt*) pim_malloc(sizeof(elt), PIM_HERE);
            newElt->val = 4*i + j;
            newElt->next = head;
            head = newElt;
        }
    }
    listArgs->head = head;
    pim_stop();
}
```

Listing 16 shows the function `printListStaticThread`, which traverses a linked list, from a thread running on a single node, copying data to the node as needed. Listing 17 shows the function `printListTravellingThread`, which “follows” the linked list, moving to the node where the current list element is stored. Note that the only difference between the two functions is a single `pim_moveTo` function call in the latter.

Listing 18 shows a program to test the list construction and traversal functions. First, it uses one thread to build the list. Next it uses two additional threads to traverse the list, the first with a “static” thread and the second with a travelling thread. Listing 19 shows the printed output of the program. Observe that for the static thread, all accesses to the linked list are made by thread T2 from node N0, regardless of where the current list element is located. In the second case, thread T3 moves, so that all memory accesses are from the same node where the list element resides in memory.
Listing 16: Static thread traversing a linked-list

```c
/*
 * List traversal, copy data to thread
 */
void* printListStaticThread(void* listArgsVoid)
{
    elt *curElt;

    curElt = ((listArgs_t*) listArgsVoid)->head;
    while (curElt != NULL) {
        pim_systemPrintWhereValue(curElt, curElt->val);
        curElt = curElt->next;
    }
    pim_stop();
}
```

Listing 17: Travelling thread traversing a linked-list

```c
/*
 * List traversal, move thread to data
 */
void* printListTravellingThread(void* listArgsVoid)
{
    elt *curElt;

    curElt = ((listArgs_t*) listArgsVoid)->head;
    while (curElt != NULL) {
        pim_moveTo(curElt);
        pim_systemPrintWhereValue(curElt, curElt->val);
        curElt = curElt->next;
    }
    pim_stop();
}
### 3.2 Distributed Tree

Listing 20 shows a simple, recursive C function for building a binary tree of a given depth, breadth-first. The function assigns values to each node in the tree in the order visited. Listing 21 shows a multithreaded function `build` for building a binary tree of a given depth, in a breadth-first manner. The function recursively forks two new activations of `build` for the left and right subtrees. All of the nodes of the tree are allocated in the same PIM neighborhood (on the same PIM node).

Figure 2 illustrates the binary tree generated by `build` with a depth of 3.

![Figure 2: A binary tree, nodes numbered breadth-first.](image1)

Listing 22 shows a recursive function for a preorder traversal of a binary tree, using only a single thread. Listing 23 shows a program that builds a binary tree of depth 3 and then performs a preorder traversal of the tree. Referring to Figure 2, the order in which the nodes are visited is 1, 2, 4, 5, 3, 6, 7.

Things get more interesting if the tree is to be distributed across multiple neighborhoods, such as that illustrated in Figure 3. In this figure, each dotted-line region of the tree is stored in a (possibly) different neighborhood.

![Figure 3: A binary tree distributed across multiple PIM neighborhoods](image2)

Listing 24 shows a modification of the listing in Listing 21 that recursively build a distributed binary tree using multiple threads. The parameter `distDepth` specifies the depth at which the distribution of the tree begins (actually, `depth - distDepth`, if we think of the root of the tree as being depth 1). The significant difference between this function for building the tree, and the previous version which built a tree in a single neighborhood, is that at one stage in the recursion, a new tree element is forked in the neighborhood `PIM_RANDOM`, rather than `PIM_HERE`, and the thread that allocated the new element moves to that neighborhood.

Listing 25 shows a program that builds a distributed binary tree of depth 3, with the root of the tree in one neighborhood, and the left and right subtrees of the root in 2 different neighborhoods (`distDepth = 2`). Figure 4 illustrates the resulting tree. Listing 26 shows the output from the program. In this run of the program, the root of the tree was allocated in the neighborhood of node N0, the left subtree was allocated in the neighborhood of node N1, and the right subtree was allocated in the neighborhood of node N3. The single thread traversing the tree, T16, made all accesses to the tree from the neighborhood of node N0.

Changing the tree traversal algorithm so that the thread moves from neighborhood to neighborhood, following the tree, is simply a matter of adding `pim_moveto` commands. Listing 27. Listing 28 shows the
Listing 18: Test program for creating and traversing a linked-list

```c
int main(void)
{
    elt *head;
    listArgs_t listArgs;
    pim_threadGroup g;

    pim_systemConfigure(4);
    pim_threadGroupInit(&g);
    pim_fork(stripeList, (void*)&listArgs, &g, PIM_HOME);
    pim_join(&g);

    printf("Static thread: copy data to thread\n");
    pim_threadGroupInit(&g);
    pim_fork(printListStaticThread, (void*)&listArgs, &g, PIM_HOME);
    pim_join(&g);

    printf("Travelling thread: move thread to data\n");
    pim_threadGroupInit(&g);
    pim_fork(printListTravellingThread, (void*)&listArgs, &g, PIM_HOME);
    pim_join(&g);

    pim_stop();
}
```

Figure 4: A distributed binary tree with depth = 3 and distDepth = 2.
Listing 19: Output of test program from Listing 18

Static thread: copy data to thread
15: address on node N3 accessed by thread T2 on node N0
14: address on node N3 accessed by thread T2 on node N0
13: address on node N3 accessed by thread T2 on node N0
12: address on node N3 accessed by thread T2 on node N0
11: address on node N2 accessed by thread T2 on node N0
10: address on node N2 accessed by thread T2 on node N0
 9: address on node N2 accessed by thread T2 on node N0
 8: address on node N2 accessed by thread T2 on node N0
 7: address on node N1 accessed by thread T2 on node N0
 6: address on node N1 accessed by thread T2 on node N0
 5: address on node N1 accessed by thread T2 on node N0
 4: address on node N1 accessed by thread T2 on node N0
 3: address on node N0 accessed by thread T2 on node N0
 2: address on node N0 accessed by thread T2 on node N0
 1: address on node N0 accessed by thread T2 on node N0
 0: address on node N0 accessed by thread T2 on node N0

Travelling thread: move thread to data
15: address on node N3 accessed by thread T3 on node N3
14: address on node N3 accessed by thread T3 on node N3
13: address on node N3 accessed by thread T3 on node N3
12: address on node N3 accessed by thread T3 on node N3
11: address on node N2 accessed by thread T3 on node N2
10: address on node N2 accessed by thread T3 on node N2
 9: address on node N2 accessed by thread T3 on node N2
 8: address on node N2 accessed by thread T3 on node N2
 7: address on node N1 accessed by thread T3 on node N1
 6: address on node N1 accessed by thread T3 on node N1
 5: address on node N1 accessed by thread T3 on node N1
 4: address on node N1 accessed by thread T3 on node N1
 3: address on node N0 accessed by thread T3 on node N0
 2: address on node N0 accessed by thread T3 on node N0
 1: address on node N0 accessed by thread T3 on node N0
 0: address on node N0 accessed by thread T3 on node N0
output of a program that used this function to traverse a tree. Note that all accesses to tree elements are from the same PIM nodes on which the tree element resides, by the single thread T16, which is moving from node-to-node.

4 Data Synchronization
4.1 Mutexes
4.2 Condition Variables
5 The pim_vector Data Type and Wide-Word Operations
6 pimlib Implementation
Listing 20: Recursive, single-threaded C function for building a binary tree of a given depth, breadth-first.

```c
typedef struct _treeElt{
    int val;
    struct _treeElt *left;
    struct _treeElt *right;
} treeElt;

/*
 * Build tree and number nodes breadth-first
 */
void build_simple(treeElt **root, int val, int depth)
{
    if (depth == 0) *root = NULL;
    else {
        *root = (treeElt*) malloc(sizeof(treeElt));
        (*root)->val = val;
        build_simple(&(*(root)->left), 2*val, depth - 1);
        build_simple(&(*(root)->right), 2*val + 1, depth - 1);
    }
}
```
Listing 21: Multithreaded function to build a binary tree breadth-first, of a given depth

typedef struct {
    treeElt **root;
    int val;
    int depth;
} treeBuildArgs_t;

/*
 * Build tree and number treeElts breadth-first
 */
void build(void *treeBuildArgsVoid)
{
    treeBuildArgs_t *myArgs, argsLeft, argsRight;
    treeElt *newElt;
    pim_threadGroup g;

    myArgs = (treeBuildArgs_t*) treeBuildArgsVoid;
    if (myArgs->depth == 0) *(myArgs->root) = NULL;
    else {
        newElt = (treeElt*) pim_malloc(sizeof(treeElt), PIM_HERE);
        newElt->val = myArgs->val;
        *(myArgs->root) = newElt;

        argsLeft.root = &(newElt->left);
        argsLeft.val = 2*myArgs->val;
        argsLeft.depth = myArgs->depth - 1;
        argsRight.root = &(newElt->right);
        argsRight.val = 2*myArgs->val + 1;
        argsRight.depth = myArgs->depth - 1;
        pim_threadGroupInit(&g);
        pim_fork(build, (void*) &argsLeft, &g, PIM_HERE);
        pim_fork(build, (void*) &argsRight, &g, PIM_HERE);
        pim_join(&g);
    }
    pim_stop();
}
Listing 22: Recursive function for preorder traversal of a binary tree

```c
typedef struct {
    treeElt *root;
} treeWalkArgs_t;

treeElt *root = ((treeWalkArgs_t*) treeWalkArgsVoid)->root;
if (root != 0) {
    printf("%d", root->val);
    pim_systemPrintWhere(root," ");
    argsLeft.root = root->left;
    argsRight.root = root->right;
    preorder_recur(&argsLeft);
    preorder_recur(&argsRight);
}
```

Listing 23: Program that builds and performs a preorder traversal of a binary tree of depth 3.

```c
int main(void) {
    treeElt *tree = NULL;
    treeWalkArgs_t treeWalkArgs;
    treeBuildArgs_t treeBuildArgs;
    pim_threadGroup g1, g2;
    pim_systemConfigure(4);
    pim_threadGroupInit(&g1);
    pim_fork(build, (void*) &treeBuildArgs, &g1, PIM_HOME);
    pim_join(&g1);
    treeBuildArgs.root = &tree;
    treeBuildArgs.val = 1;
    treeBuildArgs.depth = 3;
    pim_threadGroupInit(&g2);
    pim_fork(preorder_recur, (void*) &treeWalkArgs, &g2, PIM_HOME);
    pim_join(&g2);
    pim_profilePrint();
    pim_stop();
    return 0;
}
```
Listing 24: Recursive function to build a distributed binary tree of a given depth.

typedef struct {
  treeElt **root;
  int val;
  int depth;
  int distDepth;
} treeBuildDistributedArgs_t;

/*
 * Build distributed tree and number treeElts breadth-first
 */
void buildDistributed(void *treeBuildDistributedArgsVoid)
{
  treeBuildDistributedArgs_t *myArgs, argsLeft, argsRight;
  treeElt *newElt;
  pim_threadGroup g;

  myArgs = (treeBuildDistributedArgs_t*) treeBuildDistributedArgsVoid;
  if (myArgs->depth == 0) *(myArgs->root) = NULL;
  else {
    if (myArgs->depth == myArgs->distDepth) {
      newElt = (treeElt*) pim_malloc(sizeof(treeElt), PIM_RANDOM);
      pim_moveTo(newElt);
    }
    else
      newElt = (treeElt*) pim_malloc(sizeof(treeElt), PIM_HERE);
    newElt->val = myArgs->val;
    *(myArgs->root) = newElt;
    argsLeft.root = &(newElt->left);
    argsLeft.val = 2*myArgs->val;
    argsLeft.depth = myArgs->depth - 1;
    argsLeft.distDepth = myArgs->distDepth;
    argsRight.root = &(newElt->right);
    argsRight.val = 2*myArgs->val + 1;
    argsRight.depth = myArgs->depth - 1;
    argsRight.distDepth = myArgs->distDepth;
    pim_threadGroupInit(&g);
    pim_fork(buildDistributed, (void*) &argsLeft, &g, PIM_HERE);
    pim_fork(buildDistributed, (void*) &argsRight, &g, PIM_HERE);
    pim_join(&g);
  }
  pim_stop();
}
Listing 25: Program that builds and traverses a distributed tree.

```c
int main(void)
{
    treeElt *tree = NULL;
    treeWalkArgs_t treeWalkArgs;
    treeBuildDistributedArgs_t treeBuildDistributedArgs;
    pim_threadGroup g1, g2;

    treeBuildDistributedArgs.root = &tree;
    treeBuildDistributedArgs.val = 1;
    treeBuildDistributedArgs.depth = 3;
    treeBuildDistributedArgs.distDepth = 2;
    pim_systemConfigure(4);

    pim_threadGroupInit(&g1);
    pim_fork(buildDistributed, (void*) &treeBuildDistributedArgs, &g1, PIM_HOME);
    pim_join(&g1);

    treeWalkArgs.root = tree;

    pim_threadGroupInit(&g2);
    pim_fork(preorder_recur, (void*) &treeWalkArgs, &g2, PIM_HOME);
    pim_join(&g2);

    pim_profilePrint();
    pim_stop();
    return 0;
}
```

Listing 26: Output of the program in Listing 25, traversing a distributed binary tree with a static thread.

```
1 : address on node N0 accessed by thread T16 on node N0
2 : address on node N1 accessed by thread T16 on node N0
4 : address on node N1 accessed by thread T16 on node N0
5 : address on node N1 accessed by thread T16 on node N0
3 : address on node N3 accessed by thread T16 on node N0
6 : address on node N3 accessed by thread T16 on node N0
7 : address on node N3 accessed by thread T16 on node N0
```
Listing 27: Function that uses a single, travelling thread to traverse a distributed binary tree, accessing all tree nodes from within their neighborhoods.

```c
void preorder_recur_travel(void *treeWalkArgsVoid)
{
    treeElt *root = ((treeWalkArgs_t*) treeWalkArgsVoid)->root;
    treeWalkArgs_t argsLeft, argsRight;
    if (root != 0)
    {
        printf("%d", root->val);
        pim_systemPrintWhere(root," ");
        argsLeft.root = root->left;
        argsRight.root = root->right;
        if (root->left != NULL) pim_moveTo(root->left);
        preorder_recur_travel(&argsLeft);
        if (root->right != NULL) pim_moveTo(root->right);
        preorder_recur_travel(&argsRight);
    }
}
```

Listing 28: Output of a program that uses a travelling thread to traverse a distributed binary tree

```
1 : address on node N0 accessed by thread T16 on node N0
2 : address on node N1 accessed by thread T16 on node N1
4 : address on node N1 accessed by thread T16 on node N1
5 : address on node N1 accessed by thread T16 on node N1
3 : address on node N3 accessed by thread T16 on node N3
6 : address on node N3 accessed by thread T16 on node N3
7 : address on node N3 accessed by thread T16 on node N3
```