A Heterogeneous Parallel Programming Language and Compiler Architecture

Nicholas Tomlinson

14th June 2013

We present Polycute: an imperative-like programming language with block structures that have parallel semantics, no requirement for a global shared memory, and data structures and types that support the resultant programming paradigm. We set out the goals for such a programming language that we argue are necessary for such a language to be intuitive to programmers. We show that Polycute's block structures allow programmers to compose code fragments that are parallel in nature in much the same way as they would sequential code fragments.

Today's computers are becoming increasingly parallel in nature, owing to limitations imposed by limits of instruction-level parallelism and design team size preventing the creation of ever higher performance superscalar processors. If processors become more parallel, then programmers are required to write parallel programs. Unfortunately, today's programming languages do not allow programmers to write parallel programs in an intuitive way while permitting the compiler to optimise what parallelism is exploited, and how it is exploited.

This project aims to answer the following questions: How might a programming language be designed that allows the programmer to use their intuition about how programs should be written to write parallel programs? How might we design a compiler to compile and optimise that language to several different targets that interact with each other? We show that:

- Parallelisation primitives can be expressed syntactically as control-flow-like statements that are similar in vein to an if statement or while statement.
- A static type system can be leveraged to express how parallel access to a variable should take place – in particular, to facilitate associating a mutex with a variable.
- Many of the thread-synchronisation operations that often appear in parallel code can be expressed as control-flow-like statements.
- Polycute's type system, and parallelising primitives allow code fragments to be written that can be composed even when part or all of them are to be executed in parallel. In this context, we define composition to be the act of using two separate pieces of code to produce the effect that the programmer expects would result from combining them.
- The Polycute language permits implementation of a compiler that can produce programs for a NUMA (Non-Uniform Memory Access) architecture such as that found on a computing cluster, or a CPU and GPGPU (General Purpose Graphics Processing Unit) combination.
- By treating parallelisation structures as indications that code *can* be parallelised rather than that it *must* be parallelised, we are able to perform optimisations that we could not perform had we chosen to make parallelisation mandatory.

We observe that an important aid to programmers' intuition when writing sequential programs is that code fragments can be composed. A programmer expects, for example, that if two statements are executed on after another, then the first statement will transform the program's state in some fashion, and the second statement will transform the resulting state in some way consistent with the result of the first statement. We can reason about such sequences of statements with Hoare-logic[5], and programmers often have a strong intuition of the properties that this type of reasoning leads to even if they have not studied it theoretically. The programmer would expect, for example, that if the first statement ensured that the program's state met the second statement's precondition, then the second statement's postcondition would be met immediately upon completion of the second statement. These properties are often used by the programmer to enable them to decompose a program such that only a small part of it need be examined at a given time in order to understand its operation. This is an important aid the programmer's ability to understand the program.

Unfortunately, many of these intuitions break down when writing parallel programs. This problem is compounded by parallel programming paradigms that do not attempt to modify this reasoning to suit parallel programming, and thus the programmer is forced to abandon their intuition entirely. Rely-guarantee reasoning extends Hoare-logic to account for parallelism – adding a condition that will be satisfied by the statement during its execution, and a condition that must be satisfied by all other statements executing at the same time[4]. Such a condition might be that no other statement may modify a variable while the statement is executing.

Unfortunately, most imperative programming languages do not implement parallelism in a way that is easily composed. Examples of existing imperative programming languages that implement parallelism are given in

Section 6, along with a critique of their support for parallelism. We have therefore designed Polycute to provide a set of statements that instruct the compiler that it has the option to parallelise its child statement. This approach, along with a suitable type system, allows code fragments to be composed.

Modern commodity computers contain powerful graphics cards that are capable of general purpose computation. These graphics cards support a style of computation known as GPGPU computation that is intended for highly parallel applications. We drop the language assumption of a global shared memory in order to produce a language that allows a compiler to easily schedule appropriate code to run on a GPGPU. A GPGPU often has a memory that is disjoint from the computer's main memory. We choose a generalised approach to partitioning work between processors with disjoint memories. This makes Polycute a candidate for extension to write programs that can run on a cluster of computers. Polycute's expression of parallelism allows a compiler to produce code that is well suited to run on either a GPGPU or a multi-threaded CPU.

1 The Polycute Language

The Polycute programming language is strongly influenced by the C programming language, but with concepts that are amenable to parallel programming. We have prioritised improving the language over dogmatically reproducing C. A detailed comparison of the parallel programming languages is presented in Section 6.

1.1 Notation and Language Concepts

To understand our description of Polycute's novel features, we have provided a very quick primer to Polycute's syntax. We will also introduce the notation we use throughout the remainder of this report.

We use **boldface** when referring to Polycute keywords. Where a word is to be used in a particular way in Polycute's grammar, we specify the non-terminal in *CAPITAL-TELETYPED-ITALICS*. Table 1 describes the non-terminals that we use. Table 2 describes some of Polycute's keywords and constructions, and their usage.

1.2 For

Polycute's **for** loop is provided for the programmer's convenience, but also as a way for the programmer to express a loop that is guaranteed to have an iteration count that is known at entry to the loop. When used with **spawn**, the compiler can be sure that parallelising the loop will not affect the control flow of the loop itself.

```
1 int 32 i = 0;
2 int 32 sum = 0;
3 while (i < 1024){
4     sum += i;
5     i = i + 1;
6 }
```

Non-terminal	Description
EXPRESSION	A Polycute expression. An expression (for example, a function call) may have side effects.
STATEMENT	Statements such as those that would be found in a C program. Additionally, this includes blocks such as those used by C, and statements such as **while** statements. An *EXPRESSION* may also be used as a *STATEMENT*.
TYPE	A Polycute type. Polycute types should be read left to right, with each subsequent type modifier wrapping the already specified type. For example: **bool lock pointer** is a pointer to a locked boolean, whereas **bool pointer lock** is a locked pointer to a boolean.

Table 1: *Non-terminals used in our description of Polycute*

Listing 1: *A while loop with a fixed iteration count*

The code examples given in Listing 1 and Listing 2 are equivalent. The **for** loop is a good candidate to optimise code to. An example of such code is given in Listing 1. The optimised version is given in Listing 2. Once the loop is expressed as a **for** loop, the compiler may perform subsequent optimisations as if the **while** loop had been tagged as suitable for such optimisations. Further, the programmer may notice that such an optimisation is possible and can easily perform this optimisation by hand where the compiler cannot (for example, if a reference to the loop counter is passed, but known never to be updated).

```
1 int 32 sum = 0;
2 for (int 32 i to 1024){
3     sum += i;
4 }
```

Listing 2: *A for loop equivalent to Listing 1*

1.3 Spawn and Sync

Like the Cilk programming language (see Section 6.4), Polycute has the keywords **spawn** and **sync**. In Polycute, both the **spawn** and **sync** constructs have a child statement, as opposed to Cilk, where **spawn** may only be used as a modifier for a function call, and **sync** is used to wait for all previously spawned functions without any further structure. This is an important part of Polycute's design. Just as an if statement or while loop in C has a statement or block associated with it, so too do Polycute **spawn** and **sync** statements.

Constructions	Description
if *STATEMENT*, **while** *STATEMENT*	Analogous to C.
for (*TYPE IDENTIFIER* **to** *EXPRESSION*) *STATEMENT*	Executes *STATEMENT* n times, where n is calculated at entry to the loop to be the *EXPRESSION*. The loop variable is named by *IDENTIFIER* and takes a value between 0 and $n-1$, in ascending order. Assignments to the loop variable do not affect subsequent iterations.
spawn *STATEMENT*, **sync** *STATEMENT*	Used for parallelism, and is described in Section 1.3.
TYPE **lock**, **atomic**, **atomic reeval**, **release**	These are used to allow operations on data structures to be parallelised correctly, and are described in Section 1.4.
TYPE `[]`	A "dynamic vector" type – so called because its size and the memory it is located in (for example, CPU or GPU) are determined at run time. Dynamic vectors are described in Section 1.5.
TYPE **ref**	A reference type. Similar to C++'s reference type, except that a reference may refer to a variable that is not in the local memory.

Table 2: *Polycute keywords and constructions*

This allows **spawn** to operate on any statement, rather than just function calls. As a result, it is possible to add the **spawn** keyword to suitable sequential code (such as a block statement) without significant restructuring of the code.

Unlike in Cilk, Polycute's **sync** statement waits only for spawns that are nested within the **sync** statement, or a function that is consequently called. This allows code with **spawn** and **sync** statements to be composed without losing the ability for child **sync** statements to wait only for their child **spawn** statements. Consider the example given in Listing 3. All four iterations of the loop may execute in parallel, despite the **sync** statements in the body of the loop. In Cilk, using **sync** in the second iteration of the loop would wait for the entire of the first iteration of the loop to complete.

```
1  sync for (int 32 i to 4) spawn {
2      sync {
3          spawn a();
4          spawn b();
5      }
6      sync {
7          spawn c();
8          spawn d();
9      }
10 }
```

Listing 3: *An example to demonstrate the utility of structured* **sync** *statements*

Consider implementing merge sort recursively. On very large data sets, we may wish to partition the data set to be processed in different threads. We would not, however, wish to process small partitions in parallel, as this would be inefficient. The programmer might therefore wish to write code such as that given in Listing 4. This is cumbersome – the code must be specified twice! Although Listing 4 is not too obnoxious to duplicate, a more complex example than a function call might be unacceptable to use this way.

```
1  void MergeSort
2      (int 32 n, int 32 off, int 32 [] ref array)
3  {
4      /* Code to sort if length(array) <= 2 */
5
6      bool should_spawn = n > 10000;
7      if (should_spawn){
8          //The array is large enough to be worth
9          //processing in parallel
10         sync {
11             spawn MergeSort(n/2, off, array);
12             spawn MergeSort
13                 (n - n/2, off + n/2, array);
14         }
15     }
16     else {
17         spawn MergeSort(n/2, offs, array);
18         spawn MergeSort
19             (n - n/2, offs + n/2, array);
20     }
21
22     /* Code to merge the two halves */
23 }
```

Listing 4: *Merge sort with conditional parallelisation using an* **if** *statement*

Both the **spawn** and **sync** statements have two forms: conditional, and unconditional. In their conditional form, their parallel semantics apply if and only if the condition is true. This relieves the code duplication problem that occurs in Listing 4. For **spawn**, the child statement may execute in parallel if the condition is true, but must execute sequentially if the condition is false. For **sync**, execution may not proceed beyond the end of the **sync** statement until its child spawns are completed if the condition is true, but it may proceed if the condition is false. Listing 5 demonstrates the conditional variants of **spawn** and **sync**.

```
1  void MergeSort
2      (int 32 n, int 32 off, int 32 [] ref array)
3  {
4      /* Code to sort if length(array) <= 2 */
5
6      bool should_spawn = n > 10000;
7      //The array is large enough to be worth
8      //processing in parallel
9      sync (should_spawn) {
10         spawn (should_spawn) MergeSort
11             (n/2, off, array);
12         spawn (should_spawn) MergeSort
13             (n - n/2, off + n/2, array);
14     }
15
16     /* Code to merge the two halves */
17 }
```

Listing 5: *Merge sort with conditional parallelisation using the conditional variants of* **sync** *and* **spawn**

We have chosen to allow a function to use a **spawn** statement without being nested in a **sync** statement. This has two advantages: Firstly, the programmer may divide their code arbitrarily rather than being compelled to ensure the **sync** statement is in the same function as the **spawn** statement. The programmer may wish, for example, to have a set of functions that have several **spawn** statements, with the intention that these functions be used with a **sync** statement in the caller (or indeed the caller's caller). Secondly, there may be no instructions outside of the **spawn** statement that cannot begin until the **spawn** statement has ended. Consider Listing 6. Here, we have a server program that spawns upon receipt of a client connection. The rest of the program flow is then unconcerned with when the spawned statement completes.

```
1  void ServerProgram()
2  {
3      while (true){
4          struct Connection connection =
5              WaitForConnection();
6          spawn ProcessConnection(connection);
7      }
8  }
```

Listing 6: *Server program demonstrating a* **spawn** *without a* **sync**

We have chosen not to implement function annotations to indicate that a function may spawn without syncing, similar to Java's **throws** annotation for uncaught exceptions. This would be cumbersome, and is an arbitrary side effect to choose to annotate – particularly in light of programs such as the one in Listing 6. It is not necessary to be concerned about **sync** statements that do not contain any **spawn** statements, as such a **sync** statement would have no effect on the semantics of the program.

1.4 Lock and Atomic

In order to allow the programmer to express the rely and guarantee conditions from rely-guarantee reasoning[4], Polycute includes a set of features to allow the programmer to place limits on how two statements may mutate the program's state concurrently.

In Polycute, **lock** is a type qualifier. The type *TYPE* **lock** is a *TYPE* that has an associated lock. This lock is semantically taken whenever a variable of this type is read from, or written to, and freed afterwards. It is up to the compiler to determine exactly when the lock is taken and freed, however the behaviour of the program should be consistent with that which would result if the lock operations were performed immediately before and after the variable access. This ensures the entire variable is read from or written to in a consistent way – with no other thread modifying the value during the read or write operation.

The programmer may wish to carry out operations that involve many read and write accesses in a consistent way; **atomic** *STATEMENT* ensures that all variable accesses in *STATEMENT* are consistent with having been performed atomically. Variables that are read from in *STATEMENT* are not written to at any time while *STATEMENT* is executed. Variables that are written to in *STATEMENT* are neither read from nor written to at any time while *STATEMENT* is executed.

Listing 7 demonstrates a parallelised increment operation. The locks for both `a` and `b` are acquired at the beginning of the **atomic** statement – before any read or write operation occurs. The lock for `b` is taken even if `a < 42`, as the decision to lock `b` is based on syntactic analysis of the code at compile time. The locks are not unlocked until the end of **atomic** statement – after all read and write operations have taken place – including those that might be made by function calls if present.

```
1  int 32 lock a = 0;
2  int 32 lock b = 1;
3  for (int 32 i to 1000) spawn atomic {
4      a = a + 1;
5      if (a > 42){
6          b = a * b;
7      }
8  }
```

Listing 7: *Atomic operation on* `a` *and* `b`

An **atomic** statement also acquires the lock for each *TYPE* **lock ref** type variable that could (and is by syntactic analysis) be dereferenced. As with non-reference variables, the set of references is that which could be taken given syntactic analysis of the code. The value of each reference is taken to be that at the beginning of the atomic statement. Null references, and references that are declared within the atomic statement are not taken, as neither refer to locks that are available at entry to the **atomic** statement. This rule is necessary to allow references to variables of a **lock** type to be used correctly. If a function returns from within an **atomic** statement, the locks it has taken are unlocked.

Consider the problem of inserting into a linked list. It is not possible to be certain, ahead of time, which elements of the list will be accessed, and thus must be locked. Recursively locking all references that could be taken would at best allow most of the program's state to be locked, and at worst be undecidable. Polycute supports two companions to **atomic**: the **release** *STATEMENT* and the **reeval** modifier. The **release** statement and **atomic reeval** statement are designed to be used together to enable the programmer to traverse a structure of locks without having to lock the entire structure but always maintaining a lock on some part of it in a similar manner to that suggested by Moir and Shavit[7]. Such an algorithm written in Polycute is presented in Listing 8.

Once the first lock has been taken, at least one lock is held until the end of the **release** *STATEMENT* is reached, at which time all locks that are still locked by *STATEMENT* are unlocked. The set of locks held is reevaluated at the end of the **atomic reeval** *STATEMENT* by iterating through the list of acquired locks, and determining whether they are still required. For each lock that is locked by the *STATEMENT* and that is syntactically accessed in a hypothetical subsequent iteration, the lock remains locked. This ensures no other thread can take the lock between iterations of the loop. If the variable is no longer syntactically accessible (for example, because the value of `head` is updated in the *STATEMENT*), the lock is released. Any new locks that have become accessible are also locked at the end of the **atomic reeval** statement – in a predetermined order to avoid deadlock. In this context, a lock is accessible if it would be taken at the beginning of the **atomic reeval** statement. To make this determination, it is assumed that the **atomic reeval** statement will execute again, and that the values of the references do not change. It is important that new locks are acquired before old locks are unlocked, otherwise another thread may acquire an old lock and a new lock while no locks are held by the **atomic reeval** statement. The set of held locks is reevaluated in the same way the next time the **atomic reeval** *STATEMENT* is executed so that the correct locks are held upon entry to the *STATEMENT*.

```
1  struct ListElement
2  {
3      int 32 element;
4      struct ListElement lock ref next;
5  }
6
7  /*
8      It is safe to call this function with the
9      same head (or subsequent element) from
10     different threads at the same time. For
11     simplicity, we assume head is not null.
12 */
13 void Insert(
14     struct ListElement lock ref head,
15     int 32 element)
16 {
17     release while (head != null) atomic reeval
18     {
19         if (head->next == null){
20             head->next = NewElement(element);
21             head = null;
22         }
23         else if ((head->element < element) &
24         (head->next->element) >= element){
25             struct ListElement lock ref new =
26                 NewElement(element);
27             new->next = head->next;
28             head->next = new;
29             head = null;
30         }
31         else {
32             head = head->next;
33         }
34     }
35 }
```

Listing 8: *Thread-safe linked list insertion*

As with **spawn** and **sync**, Polycute places no requirement that the **release** statement should appear in the same function as the **atomic reeval** statement as the programmer may wish to place the **release** in a caller function.

The design of Polycute aims to prevent the creation of programs that can deadlock, however this is not always possible. Allowing a statement to exist between the **release** and the **atomic reeval** introduces the possibility of deadlock as an **atomic** statement could attempt to obtain locks that have not been released by a previous **atomic reeval** statement. Such a bug is exhibited by Listing 9. The use of **reeval** should therefore be a reminder to the programmer that they should be careful to design their code such that it will not deadlock.

```
1  void Deadlock(int 32 lock ref i)
2  {
3      release
4      {
5          atomic reeval {
6              deref (i) = deref (i) + 1;
7          }
8
9          /* The lock associated with i is not
10            released until the end of the
11            release statement, but an attempt
12            is made to reacquire the lock in the
13            next atomic statement */
14
```

```
        atomic {
            deref (i) = deref (i) + 1;
        }
    }
}
```

Listing 9: *A program that deadlocks*

1.5 Dynamic Vector Type

Many embarrassingly parallel algorithms operate on large arrays of data. In C, such an array may be allocated with `malloc()`, which returns a pointer to memory that it has allocated in local memory. Polycute programs, however, may execute in a NUMA environment.

Polycute requires a data type that can reference a large array of data that may be in local memory, remote memory, or partitioned over several memories. The data type used for this purpose is the **dynamic vector** type – named dynamic as their size and the memory in which they are located is determined at run time. The dynamic vector type has syntax: *TYPE* `[]`. Listing 10 demonstrates the usage of the dynamic vector type.

```
/* A function that returns a dynamic vector of
   int 32 elements */
int 32 [] GetSquareNumbers(int 32 n)
{
    /* Create a dynamic vector of n int 32
       elements */
    int 32 [] squares = int 32 [n];

    /* i ranges from 0 to n - 1 */
    for (int 32 i to length(squares)){
        /* Assignment to an element of the
           dynamic vector */
        squares[i] = i*i;
    }

    return squares;
}
```

Listing 10: *Example usage of the dynamic vector type*

Unlike in C, where assignment to a pointer to the first element of an array results in a second reference to the same array, dynamic vectors in Polycute have copy semantics; they are considered to be values that are copied in the same way as any other, such as an integer. A detailed justification for this is given in Section 3.2.

2 Conceptual Implementation

2.1 Spawn and Sync

Polycute's **spawn** keyword instructs the compiler that the child statement may be executed in parallel. In a heterogeneous environment, there may be many different types of parallelism that can be enabled by the **spawn** statement, for example:

- Threads running on a multicore CPU
- Kernels executing on an OpenCL device (see Section 6.2)
- Softcores running on an FPGA
- Processes running on remote processors

Selecting which of these forms of parallelism should be selected is partly an optimisation challenge based on where data is located (see Section 3.2) and establishing whether such parallelism can be exploited efficiently (see Section 3.1).

The simplest implementation of **spawn** on a multicore CPU is that of a thread pool and a job queue. Each time a spawn statement is executed, a job is created with the spawn statement's child statement. In order to support the **sync** statement, each job must be added to a list for its parent **sync** statement. The **sync** statement must also be added to the list of its parent **sync** statement in order to avoid having to add jobs to all parent **sync** statements' lists.

An OpenCL kernel is a piece of code that is executes many times with the same set of arguments, save for a small number of indexes. Polycute's **for** loop is a good conceptual match for this. Subject to several restrictions on the code to be executed, the **for**/**spawn** structure can be converted to an OpenCL kernel, in addition to the usual CPU code. When the loop is to be executed, the runtime determines whether to execute the CPU loop, or the OpenCL kernel.

As with OpenCL kernels, FPGA softcores and remote processes have restrictions on the code that may be executed on them. The trade-offs in each case are different, and are discussed in Section 3.1.

2.2 Lock and Atomic

It is desirable to minimise the opportunity for a program to enter deadlock, and in any case provide a clear set of rules to avoid it. Therefore, an **atomic** *STATEMENT* acquires its locks in a well defined order. This is a standard solution to the Dining Philosophers Problem[6].

In order to permit nested **atomic** *STATEMENT*s such as in Listing 11, the compiler may have to create second locks for `a` and `b` at the beginning of the outer block in order for the child **spawn atomic** statements to be able to lock the shared `a`, and `b` while preserving the atomicity of the their child **atomic** statements. The child **atomic** statement then acquires this second lock, rather than attempting to acquire the original lock which has already been acquired by the outer **atomic** statement.

```
spawn atomic sync {
    spawn atomic {
        a = b;
        c = d;
    }
    spawn atomic {
        a = e;
        b = f;
    }
```

```
}
```

Listing 11: *Nested* **spawn/atomic** *statements*

In order to allow the **atomic** statement to have the correct effect for dereferences that occur within function calls, the compiler must handle locks acquired by the callee as it would a child statement. An example of this is given in Listing 12.

```
void Callee(int 32 lock ref n)
{
    deref n = 42;
}

void Caller(int 32 ref lock n)
{
    /*
        Observe that b is not dereferenced in
        the atomic statement, unless the body
        of Callee() is considered
    */
    atomic Callee(n);
}
```

Listing 12: *Callee()'s use of* `deref n` *requires that* `n` *be locked, even though it is not dereferenced by Caller()*

This would require helper functions to be generated by the compiler if the callee is imported or exported. These helper functions would give the caller information about locks that must be acquired if the callee is called from within an **atomic** statement. In order to allow Polycute's function pointer to be represented as a single pointer, a Polycute function pointer is a pointer to a structure describing the Polycute function. An example of this structure is given in Figure 1. Any of the function pointers except the body function pointer may be null.

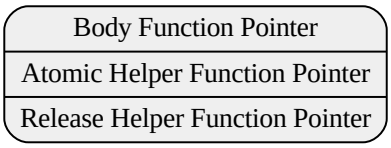

Figure 1: *The structure pointed to by a Polycute function pointer*

To support **atomic reeval** statements and **release** statements, we must maintain a list of acquired locks for each **atomic reeval** statement. A **release** statement is able to access the list of locks for each child **atomic reeval** statement not nested closer to another **release** statement. At the end of the **release** statement, the generated program traverses each list of locks, and unlocks them. Upon entry to an **atomic reeval** statement, the set of locks required is reevaluated, and any locks that are required and have not already been taken are acquired in a predetermined order, and added to the list of acquired locks. At the end of the **atomic reeval** statement, the set of required locks is reevaluated, and any locks not already held are acquired, before those no longer required are unlocked. The list of locks is kept for the entire duration of the **release** statement. Any data structure that is able to store a set of mutexes may be used to track the acquired locks. A linked list is suitable, as there is no requirement for random access; traversal need only occur from the beginning of the list to the end.

2.3 Dynamic Vector Type

The dynamic vector type is implemented as a structure that may have different forms. We shall call this structure "`dynvec`", and variants of it "`dynvec_form`" where "form" is a specific form. Each `dynvec` object contains standard information: element size and element count, and information that specifies where the vector is stored. This latter information takes a different form depending on what memories the vector is stored on. Figure 2 shows the general form of a `dynvec` object.

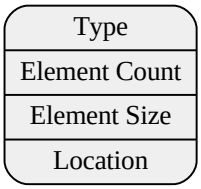

Figure 2: *General form of a* `dynvec` *object*

An important concept to enable vectors to work with a NUMA architecture is that it must be possible to represent different types of memory locations – from a local memory pointer to a split memory layout where different parts of the vector are held in different memories. This is why a `dynvec` may be one of several different forms.

The simplest case – a vector held in local memory – is shown in Figure 3. The location field is a pointer to local memory. The structure of that memory is that of a C array.

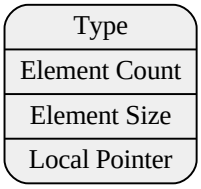

Figure 3: *A* `dynvec_local` *object referencing local memory*

Consider a vector that is stored partly on a GPU, and partly in main memory. This might happen, for example, if processing a vector with a length that is not a multiple of the SIMD width, or if using both the CPU and the GPU to process the vector. In this case, it is necessary to store a pointer to the CPU data, the location on the GPU of GPU data, and the location of the split. If the CPU always processes the end of the array when its length is not a multiple of the GPU's SIMD width, the structure looks like Figure 4. In principle, one could write to any element of a vector such as this by determining whether the element is in CPU memory or GPU memory and issuing the appropriate command to the GPU in the latter case. Unfortunately, writing a single element to the GPU in this fashion is inefficient. Efficient implementation of dynamic vectors is discussed in Section 3.2.

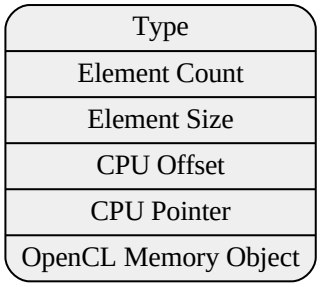

Figure 4: *A `dynvec_gpu_cpu` object referencing both CPU and GPU memory*

In a more general case, it may be necessary to store a list of vector fragment locations. An example of this is given in Figure 5.

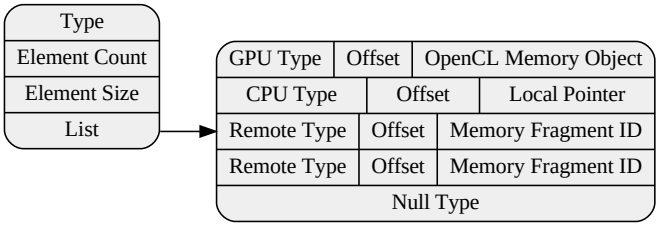

Figure 5: *A `dynvec_many` object referencing several memories*

A sophisticated compiler might use other `dynvec` forms to efficiently handle different processing patterns and computing environments. As a result, the compiler and runtime library should be written such that it is possible for optimisers to add forms of `dynvec`.

In a NUMA architecture, there is a trade-off between processing data where the data is already located, and moving the data to where the data can be most efficiently processed.

2.4 Compiler Architecture

The Polycute compiler is designed to be sufficiently extensible to allow for the easy addition of optimisations and target languages. Our implementation of the Polycute compiler consists of six stages:

Tokenization We use Flex for tokenization.

Parsing We use Bison for tokenization. This stage produces an AST (abstract syntax tree) with default code generator objects for each node, but no type checking, or name dereferencing. The default code generator objects generate LLVM Intermediate Representation (LLVM IR).

Name Dereferencing Nodes in the AST which reference other nodes by a string (such as an identifier referring to a variable declaration or function declaration, or struct type referring to a struct declaration) have pointers to the nodes to which they refer. After the parsing stage, these are null pointer. The name dereferencing stage sets these pointers to point to the correct nodes.

Normalization Produces an AST that has been type checked, and where implied type cast operations have been inserted. Some types, such as the lock type, have implicit operators attached to each access of variables of that type. These implicit operators are similar to Polycute's **deref**, except that they are implied from the variable type rather than explicitly stated in the Polycute language. The normalization stage is responsible for ensuring these implied operators are present where and only where they are required.

Optimisation Performs optimisation from Polycute AST representation to Polycute AST representation. Some optimisations require the name dereferencing and normalization stages to be rerun for a sub-tree of the AST. It is this stage that would replace the default code generator objects with ones to generate different target code, such as OpenCL. Such an optimisation might copy a sub-tree of the AST, and replace the code generation objects in the copy. This would produce a program that can execute that piece of code both on the CPU, or the GPU.

Code Generation This stage generates the LLVM IR code. If it had been implemented, it would also generate OpenCL code, or code for any other implemented language. The compiler can also produce a Polycute-like pseudo-code to aid the debugging of optimisers.

3 Efficient Implementation and Optimisation

3.1 Spawn and Sync

Consider a kernel-like loop such as that in Listing 13 to be executed on a GPGPU. Such a loop is ideal for execution as a kernel: it consists of a large and predictable number of iterations that are independent of each other. Unfortunately, if the same piece of code is to be run on a CPU, the resulting code would be extremely inefficient if a job were created and added to the job queue for each iteration. If there are no statements that must be executed sequentially between issuing different iterations of the loop, Listing 13 could be optimised to Listing 14. This optimisation is made possible by the fact that Polycute's **spawn** statement does not *mandate* the compiler to run code in parallel. This is also an example of an optimisation that relies on structured parallelism, and could not easily be performed had the parallelism been expressed using pthreads primitives. It is also an optimisation that is tedious to perform by hand.

```
1 /* Each element of array can be calculated in
```

```
    parallel. Execution does not proceed beyond
    the end of the sync statement until all
    elements have been calculated. This is
    equivalent to a kernel in OpenCL. */
sync for (int 32 i to 1024){
    for (int 32 j to 1024) spawn {
        array[i * 1024 + j] =
            exp(array[i*1024 + j])/(i*i + j*j);
    }
}
```

Listing 13: *A kernel-like piece of code to calculate* $\frac{e^x}{i^2+j^2}$ *for each element,* $x = A_{ij}$*, of an array*

```
sync {
    /* Execute most of the iterations in
       different threads */
    //Spawn some number of times
    spawn for (int 32 thread to job_count()) {
        //Iterate over part of the array
        //a is an identifier of the compiler's
        //choice
        for (int 32 a to 1024/job_count()){
            //Determine i from thread and a
            int 32 i =
                thread*(1024/job_count()) + a;
            //iterate over j as normal
            for (int 32 j to 1024) {
                //Calculate the new array
                //element; not how this code
                //does not need to change
                array[i * 1024 + j] =
                    exp(array[i*1024 + j])/
                    (i*i + j*j);
            }
        }
    }

    /* We generated efficient code above, but
       it only works in multiples of
       job_count(). So, we need to finish off
       the remainder of the iterations */
    spawn for (int 32 a to 1024 -
    (1024/job_count())){
        //determine i from a
        int 32 i = a + (1024/job_count());
        for (int 32 j to 1024) {
            array[i * 1024 + j] =
                exp(array[i*1024 + j])/
                (i*i + j*j);
        }
    }
}
```

Listing 14: *CPU optimised code to calculate* $\frac{e^x}{i^2+j^2}$ *for each element,* $x = A_{ij}$*, of an array*

3.2 Dynamic Vector Type

When a dynamic vector variable is assigned to, the vector is copied. An example of this is when a dynamic vector is used as an argument to a function, where its semantics are that of pass by value. This is much simpler to implement than a reference-counting approach that one might be tempted to employ in order to avoid unnecessary copying. Such reference counting would also require mutex operations to be performed whenever an an assignment is performed – not only to a dynamic vector variable, but also when assigning to a single element of a dynamic vector. Instead, the compiler should optimise copies of dynamic a vector to be references to the dynamic vector where possible.

```
int 32 [] Fn(int 32 [] a, int 32 [] b)
{
    int 32 [] r = int 32 [length(a)];
    for (int 32 i to length(a)){
        r[i] = a[i] + b[i];
        if (i + 1 < length(a)){
            b[i + 1] = r[i];
        }
    }
    return r;
}

void Process(int 32 len)
{
    int 32 [] a = int 32 [len];
    int 32 [] b = int 32 [len];
    //Initialise a and b

    int 32 [] c = Fn(a, b);
    //do something with c
}
```

Listing 15: *Unoptimised function call that copies dynamic vectors unnecessarily*

Consider Listing 15. We may optimise the return value to a reference to avoid the naïve copying that otherwise occurs (`return r` results in a copy). We may also optimise `a` to a reference, as it is never written to by `Fn`. We must copy `b`, however, as it is written to by `Fn`, so optimising it to a reference would change the semantics of the program. The optimised version is given in Listing 16. We could perform the return value optimisation across translation units, provided the compiler performs the optimisation for all translation units. The parameter optimisation requires a change to the caller that is dependent on the behaviour of the callee, and thus cannot be optimised in the same way across translation units without whole-program optimisation.

```
void Fn(int 32 [] ref r, int 32 [] ref a,
    int 32 [] b)
{
    deref(r) = int 32 [length(a)];
    for (int 32 i to length(a)){
        deref(r)[i] = deref(a)[i] + b[i];
        if (i + 1 < length(a)){
            b[i + 1] = deref(r)[i];
        }
    }
}

void Process(int 32 len)
{
    int 32 [] a = int 32 [len];
    int 32 [] b = int 32 [len];
    //Initialise a and b

    int 32 [] c;
    Fn(ref c, ref a, b);
    //do something with c
}
```

Listing 16: *Optimised version of Listing 15*

4 Evaluation

One of the aims of Polycute is to enable programmers to write efficient parallel code without the need to hand optimise it. In this section, we compare the execution times of the code generated by Polycute, GCC, Clang, and GCC with OpenMP for a selection of example programs. The machine on which these tests are performed has an Intel i3770k processor. This processor has four cores, each capable of executing two threads.

For each example, equivalent C (or C++) and Polycute programs have been written, generally by direct translation of a program written in one language to the other. In some cases, we have tried minor variations of the programs, such as to move a Polycute **spawn** or an OpenMP `#pragma omp parallel for` from an inner loop to an outer loop. This is an example of a hand optimisation that should not be required, but that we find improves the performance of OpenMP programs.

4.1 Mandelbrot Set

Of the three examples presented here, calculation of the Mandelbrot set is the most amenable to parallelisation on a CPU. To make this example more computationally expensive, this function uses oversampling – calculating 256 samples for every pixel. The problem can be split into many large sub-problems that can execute in parallel. This contrasts with the RBM example given in Section 4.3.

Figure 6 compares the execution time of the Polycute program given in Listing 17 with the equivalent C and OpenMP programs compiled with GCC and Clang. The edited versions correspond to the hand optimisation of moving the **spawn** or `#pragma omp parallel for` from an inner loop to an outer loop.

The Polycute implementation of this program performs significantly better than the OpenMP version: at least 9.0 s at the 95% confidence level. GCC's OpenMP compiler is unable to move the parallelisation pragma; when moved by the programmer, the execution time of the program reduces by at least 4.3 s at the 95% confidence level.

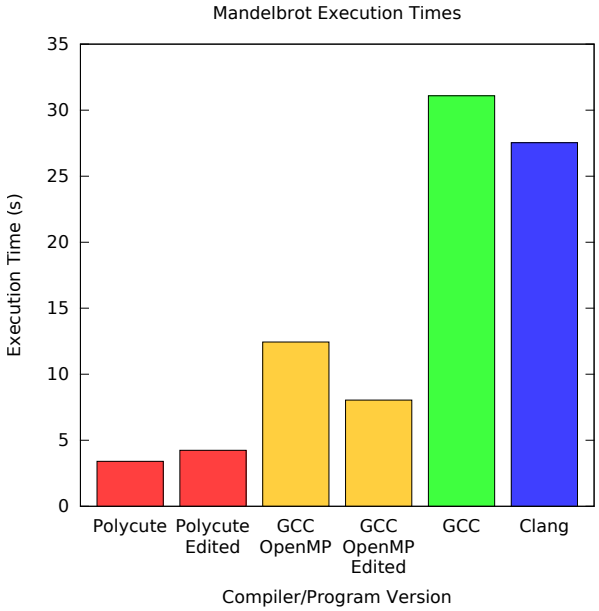

Figure 6: *Execution times for the Mandelbrot set example*

```
1  int 8 [] Mandelbrot
2      (int 32 width, int 32 height)
3  {
4      int 8 [] image = int 8 [width * height];
5      int 32 factor = 16;
6
7      sync for (int 32 imgy to height) {
8          for (int 32 imgx to width) spawn {
9              int 32 total = 0;
10
11             for (int 32 suby to factor){
12                 for (int 32 subx to factor){
13                     /* Use the escape time
14                        algorithm to calculate
15                        the escape time, i, at
16                        point (imgx, imgy) */
17                     total = total + i;
18                 }
19             }
20
21             total = total / (factor*factor);
22             if (total > 255){
23                 total = 255;
24             }
25
26             image[(imgy * width) + imgx] =
27                 total;
28         }
29     }
30
31     return image;
32 }
```

Listing 17: *A function to calculate the Mandelbrot set*

4.2 Merge Sort

Figure 7 demonstrates how Polycute, Clang, GCC, and GCC with OpenMP compare for the merge sort example. Polycute outperforms its nearest contender (sequential code from GCC) by a factor of 2.1, executing in at least 5.6 s less time at the 95% confidence level than GCC. GCC's OpenMP fares badly – taking 1.7 times as long the sequential equivalent.

Listing 18 shows the algorithm we used as for this example. We use `minspawn` to prevent the code from creating jobs that are too small to be efficiently parallelised (the Polycute compiler does not perform this optimisation). This is a good example of where the conditional semantics of **spawn** are useful. We implemented this behaviour in the OpenMP version of this example, but could not improve upon the sequential code, despite trying to optimise the value of `minspawn`.

```
1  void MergeSort(int 64 n, int 32 pointer a1,
2      int 32 pointer a2)
```

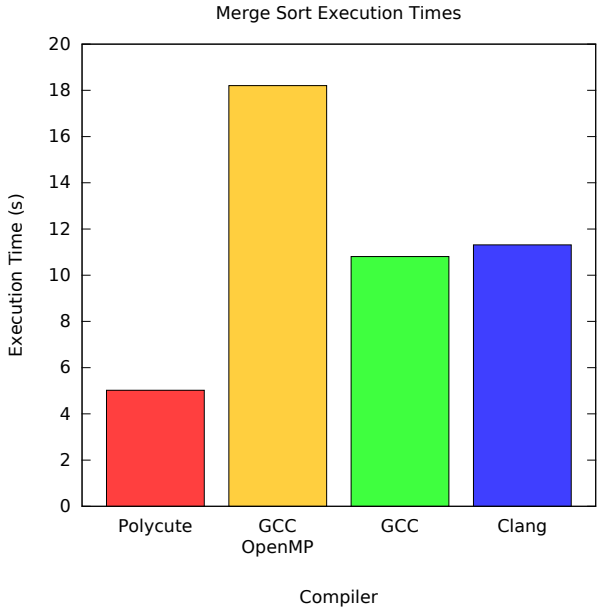

Figure 7: *Execution times for the merge sort example*

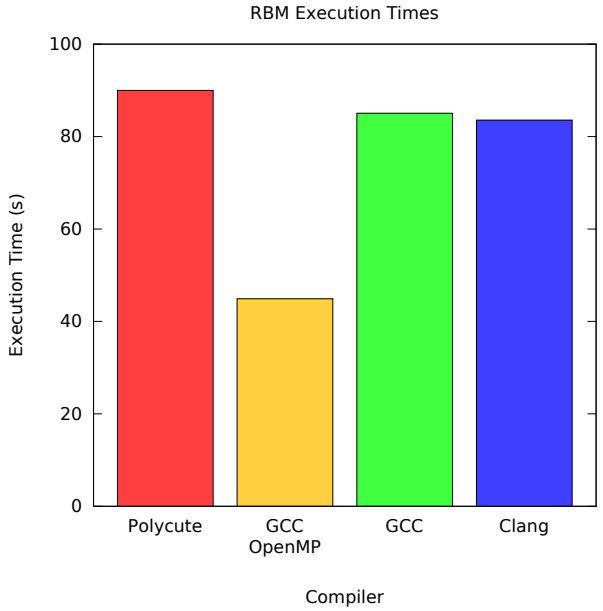

Figure 8: *Execution times for the RBM example*

```
{
    if (n < 8){
        BubbleSort(n, a1);
    }
    if (n >= 8) {
        int 64 partition = n/2;
        int 64 minspawn = 10000;

        sync (n > minspawn) {
            spawn (n > minspawn)
                Sort(partition, a1, a2);
            spawn (n > minspawn)
                Sort(n - partition,
                    AddPointer(a1, partition),
                    AddPointer(a2, partition));
        }

        //Merge the two sorted halve
    }
}
```

Listing 18: *The sorting algorithm we used for this example*

4.3 RBM

Figure 7 demonstrates how Polycute, Clang, GCC, and GCC with OpenMP compare for the RBM example. The RBM example is a memory intensive one – where an iteration performs a small amount of work on each element of a large array, compared to the Mandelbrot set example where calculation of each element of the array is much more CPU intensive. We found this example to be very sensitive to the placement of parallelisation instructions, so there are minor differences between the OpenMP and Polycute programs that we feel correspond to reasonable (not tedious to perform) programmer attempts at hand optimisation. In this example, the OpenMP version of the program executes in half the time of the other versions.

The computationally expensive part of the RBM example is primarily floating point manipulation. An outline of the algorithm for this example is given in Listing 19.

```
void CalculateVisible(struct RBM ref rbm,
    float 32 [] ref visible,
    float 32 [] ref hidden)
{
    /* Initialise visible to zero, and if we
       should, apply the function to hidden to
       get values of either 0.0 or 1.0 */

    sync for (int 32 v to rbm->visible_size)
    spawn {
        for (int 32 h to rbm->hidden_size){
            (deref visible)[v] =
                (deref visible)[v] +
                (rbm->weights
                    [v * rbm->hidden_size + h]
                * (deref hidden)[h]);
        }
    }

    for (int 32 i to rbm->visible_size){
        (deref visible)[i] =
            Function((deref visible)[i] +
            rbm->visible_biases[i]);
    }

    /* If we should, apply the function to
       visible to get values of either 0.0 or
       1.0 */
}

/* CalculateHidden works the same way as
   CalculateVisible */

void Train(struct RBM ref rbm, float 32 [] v0,
    float 32 rate)
{
    /* If we should, apply the function to
       visible to get values of either 0.0 or
       1.0 */
```

```
41      float 32 [] h0 =
42          float 32 [rbm->hidden_size];
43      float 32 [] v1 =
44          float 32 [rbm->visible_size];
45      float 32 [] h1 =
46          float 32 [rbm->hidden_size];
47
48      CalculateHidden(rbm, ref v0, ref h0);
49      CalculateVisible(rbm, ref v1, ref h0);
50      CalculateHidden(rbm, ref v1, ref h1);
51
52      sync {
53          for (int 32 v to rbm->visible_size)
54          spawn {
55              for (int 32 h to rbm->hidden_size){
56                  rbm->weights
57                      [v * rbm->hidden_size + h]=
58                      rbm->weights
59                      [v * rbm->hidden_size + h]+
60                      (rate * ((v0[v] * h0[h]) -
61                      (v1[v] * h1[h])));
62              }
63          }
64
65          for (int 32 i to rbm->hidden_size){
66              rbm->hidden_biases[i] =
67                  rbm->hidden_biases[i] +
68                  (rate * (h0[i] - h1[i]));
69          }
70
71          for (int 32 i to rbm->visible_size){
72              rbm->visible_biases[i] =
73                  rbm->visible_biases[i] +
74                  (rate * (v0[i] - v1[i]));
75          }
76      }
77  }
```

Listing 19: *The computationally expensive part of the RBM algorithm*

Upon disassembly of the resulting binary, we discovered that GCC's OpenMP code uses many fewer floating point instructions compared to its sequential code. The x86 instruction set has several SIMD floating point extensions. Today, the most powerful of these are SSE and AVX. SSE uses 128-bit registers, while AVX uses 256-bit registers. GCC uses AVX and SSE registers, whereas LLVM uses only SSE registers. We suspect that the performance of the Polycute code might improve if we improved our code generator to take account of vectorisable loops. We also suspect that both the Clang code and the Polycute code might improve if LLVM were to generate instructions using AVX's registers. Table 3 summarises our findings for the floating point instructions generated for each version. Note that both Polycute and Clang use LLVM as a back end.

Compiler	**Instruction Sets**	**Instruction Count**
Polycute	SSE	257
GCC OpenMP	SSE and AVX	356
GCC	SSE and AVX	770
Clang	SSE	264

Table 3: *Floating point instructions generated by each compiler*

5 Further Work

This report has been informed by our implementation of a compiler for the Polycute language we describe. It represents our current thinking for how the Polycute language should be designed, however the compiler itself lags behind this, with some features not yet implemented. From a practical perspective, completing this would improve our understanding of the trade-offs that must be taken in Polycute's design. We could then expand on the Polycute language, and explore how concepts such as object oriented programming might best be handled in a language like Polycute without leading to the problems associate with "spaghetti data"[9].

The current implementation of the Polycute compiler leaves some questions unresolved – such as how might we best allocate jobs created by Polycute's spawn statement between different machines in a heterogeneous environment? Could we use an architecture mapping script[3] to help the compiler do this?

We designed Polycute so that it could produce target code in several different languages – even from the same source file. We envisaged, for example, producing LLVM IR for code to be executed on the CPU, and OpenCL for code to be executed on the GPU. We have not yet implemented the OpenCL target code generator, or optimisers to make use of it. Once this work is done, we could consider how we might change the Polycute language to improve the compiler's ability to make job allocation decisions.

We have given a flavour of the kinds of optimisations that can be performed on Polycute programs beyond the standard optimisations one would expect of any optimising compiler, however one could devote entire books to exploring all the imaginable optimisations. The Polycute compiler itself implements only a handful of optimisations, so there is much room for further research in this area.

The authors observe that the Polycute language is a convenient language to express transformations of parallelism in – not only *from* Polycute source, but *to* Polycute source (in much the same way as one could express code motion as a transformation from a C program to another C program). Typically, however, it is useful to have an intermediate representation that is independent of the programming language (such as LLVM's intermediate representation). How might an analogue be developed for parallel concepts, and how far would this improve on what the Polycute language offers?

Much of the work we have presented has centred around generating efficient parallel code for computationally expensive problems. We could likely improve some of our results by improving the code generator, such as to exploit the fact that Polycute's dynamic vectors are always

aligned to suit the processor's SIMD instructions – particularly for tight parallelised loops.

Although we have designed Polycute with other parallel applications in mind, such as server applications, we would like to expand our research to include them more directly. For example we would like to implement the run time library support that would be required to allow the programmer to use blocking IO while the library transforms this into asynchronous IO, and explore the implementation of a high performance server application in Polycute.

6 Related Work

6.1 C

C provides primitives for working with threads by making use of libraries – such as pthreads[8] – that provide methods for creating threads, mutexes, and so on. The sequential part of the code is written in a way that is intuitive to the programmer. Unfortunately, the parallelism must be micro-managed and there is no opportunity for the compiler to optimise. In this way, these languages are akin to assembly in that there is very little abstraction of the parallel primitives exposed by the operating system. There are several consequences of hard-coding parallelism in this way. Firstly, the implementation of parallelism is error prone. Secondly, it is not possible for the compiler to perform optimisation on the parallelism. Thirdly, this approach leads to code that cannot be ported – it is not possible to adapt a program written for a multi-core CPU to run on a GPGPU without significant work.

6.2 OpenCL

OpenCL is a programming language designed for hetrogeneous computing environments such as those offered by desktop PCs with GPGPUs[10]. OpenCL requires the programmer to separate the parallel code from the sequential code. It supports data types that are appropriate for SIMD operation, and presents a computing model that is intuitive to the programmer. In OpenCL, the parallel code is represented as a kernel that is executed many times, differing only by a global ID (which may be used for example, to calculate an index into a large array). This is a much higher-level representation of parallelism, and permits portability between different systems. Unfortunately, this separates out the parallel part of the code from the sequential to too great an extent. It becomes difficult to write programs that share code and ideas between the sequential and parallel component. Arguments to kernels are represented in the sequential code as library calls that add a single argument to the end of a list of arguments. This makes it very difficult for the compiler to check this for correctness in any case, and impossible in the general case. It is hard for the compiler to perform optimisations that require changes of both the parallel code and the sequential code. For example, AMD's compiler does not optimise executions of a kernel on a single data item into an equivalent SIMD or loop based kernel, as this would require changing the sequential code to create fewer work items, and changing the kernel to operate on more work items. Hence the programmer is forced to hand-optimise this aspect of the code in order to prevent the performance from being inferior to that of a sequential implementation.

6.3 OpenMP

Languages such as OpenMP work by extending an existing language, in this case C, C++, or Fortran. OpenMP allows the programmer to annotate the code with **pragma**s that alter the semantics of some of the constructs to include parallelism and information about shared memory access. Like OpenCL, OpenMP fits a programmer's intuition well, but compilers find it hard to perform optimisations on the parallelism. Additionally, since OpenMP is an extension of an existing language, it is limited to semantics compatible with the original. OpenMP also lacks a more powerful set of concepts that would allow for easy use of GPGPUs where there is no global shared memory.

6.4 Cilk

The Cilk programming language has the keywords **spawn** and **sync**[1]. A function call may be prefixed with the **spawn** keyword. This instructs the compiler that the function call may be run in parallel with other executing code. The **sync** keyword instructs the compiler to wait until all functions that are currently executing in parallel finish executing.

6.5 X10

X10 is a language that is designed for a similar computing environment to that envisaged by us – a cluster-like environment where processors may or may not have access to shared memory [2]. It has the concept of places. These can be thought of as a single machine in a cluster of machines – with its own memory, and its own processor cores. Items of data belong to a single place, where this data may be accessed in sequential code. Items in a place other than that on which the current thread is executing may only be accessed asynchronously. X10 supports spawning and management of new threads with **asynch** and **finish** concepts that are similar in nature to Cilk's **spawn** and **sync**. X10 also supports **atomic**, which prevents other threads from executing at the same time as the atomic thread. These concepts are powerful, however X10's concept of places forces the programmer to make decisions about the location of data within the computer cluster, rather than allowing the compiler to optimise this.

7 Conclusions

We have designed an imperative programming language that allows programmers to represent the paralellism of their programs using concepts that extend familiar ones, such as control flow and data types. We explored some of the problems that result from such a language, and present solutions to them. We have produced a compiler that implements much of our language, though this is incomplete The language design offers scope for features such as NUMA targets that are currently not supported by the compiler. We feel that the language we have designed fits in well with programmers' existing intuitions, and that efficient parallel code can be generated from it. The language presented in this report would be a good candidate to improve upon in future work.

Acknowledgements

I would like to thank the following people:

- Alan Mycroft for supervising this project
- Jason Bell for the example given in Listing 11
- Chris Kitching for proof reading

References

[1] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, 1995.

[2] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[3] Robert Ennals, Richard Sharp, and Alan Mycroft. Task partitioning for multi-core network processors. *Compiler Construction, Joint European Conferences on Theory and Practice of Software*, pages 76–90, 2005.

[4] Xinyu Feng. Local rely-guarantee reasoning. *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–327, 2009.

[5] Charles Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, pages 576–580, 1969.

[6] Charles Hoare. Communicating sequential processes. *Communications of the ACM*, pages 666–677, 1978.

[7] Mark Moir and Nir Shavit. Concurrent data structures.

[8] Frank Mueller. Pthreads library interface. 1995.

[9] Alan Mycroft. Isolation types and multi-core architectures. *Proceedings of the 2011 international conference on Formal Verification of Object-Oriented Software*, pages 33–48, 2011.

[10] John Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for hetrogeneous computing systems. *IEEE Design and Test*, pages 66–73, 2010.