Expanding the Synthesis of Distributed Memory Implementations

by

Jeremy Edward Sharpe

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

© Massachusetts Institute of Technology 2015. All rights reserved.
Expanding the Synthesis of Distributed Memory Implementations

by

Jeremy Edward Sharpe

Submitted to the Department of Electrical Engineering and Computer Science on August 18, 2015, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I expanded the programming model implemented by the Sketch language to supplement its distributed memory parallelism with shared memory parallelism that uses the popular fork-join model. The primary contribution of this thesis is the means by which the code is assured to be free of race conditions. Sketch uses constraint satisfaction analysis to ensure it synthesizes code the functions properly for all inputs, and I demonstrate how assertions can be generated and inserted into the analysis to guarantee freedom from race conditions. This expanded programming model is then evaluated using test cases to ensure correct operation and benchmarks to examine overall performance.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor
Acknowledgments

I would like to thank Professor Solar-Lezama for providing guidance on the project and my friends and family for their support during my time at MIT.
Contents

1 Introduction .......................................................... 13
  1.1 Motivation .......................................................... 13
  1.2 Thesis Scope ....................................................... 13

2 Background ............................................................ 15
  2.1 Single Program Multiple Data (SPMD) .......................... 15
  2.2 Related Work ...................................................... 16
    2.2.1 Chapel ......................................................... 16
    2.2.2 X10 .......................................................... 17
    2.2.3 Habanero Java ................................................. 18
  2.3 Sketch ............................................................ 18

3 Virtual Multiprocessor Design ..................................... 19
  3.1 Parallel For Loop .................................................. 19
  3.2 Technical Approach to Generation ............................... 20

4 Leveraging Synthesis for Checking Correctness .................. 23
  4.1 Basic Algorithm ................................................... 23
  4.2 Proof of Correctness .............................................. 25
  4.3 Implementation ................................................... 27
    4.3.1 Basic State .................................................. 27
    4.3.2 Reaction to key statements ................................. 27
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1</td>
<td>Race detection state diagram.</td>
<td>24</td>
</tr>
<tr>
<td>5-1</td>
<td>Permissible scalar access.</td>
<td>34</td>
</tr>
<tr>
<td>5-2</td>
<td>Permissible global scalar write.</td>
<td>34</td>
</tr>
<tr>
<td>5-3</td>
<td>Local arrays are race free.</td>
<td>35</td>
</tr>
<tr>
<td>5-4</td>
<td>Indirection array that legally permutes the accesses.</td>
<td>36</td>
</tr>
<tr>
<td>5-5</td>
<td>Function call legally accessing global scalar.</td>
<td>37</td>
</tr>
<tr>
<td>5-6</td>
<td>Loop carried dependency.</td>
<td>38</td>
</tr>
<tr>
<td>5-7</td>
<td>Obvious race conditions.</td>
<td>39</td>
</tr>
<tr>
<td>5-8</td>
<td>Indirections that collide are detected.</td>
<td>40</td>
</tr>
<tr>
<td>5-9</td>
<td>Race detected in function call.</td>
<td>41</td>
</tr>
<tr>
<td>5-10</td>
<td>Reduce() runtime vs log₁₀(input length).</td>
<td>43</td>
</tr>
<tr>
<td>5-11</td>
<td>Reduce() speedup in a 10⁷ element list.</td>
<td>43</td>
</tr>
<tr>
<td>5-12</td>
<td>Image processing speedup scaling.</td>
<td>44</td>
</tr>
<tr>
<td>6-1</td>
<td>Performance of serial C++ and default Sketch.</td>
<td>46</td>
</tr>
<tr>
<td>6-2</td>
<td>Speedup from Sketch optimizations.</td>
<td>48</td>
</tr>
<tr>
<td>6-3</td>
<td>Performance of parallel implementations on a 5³ element mesh.</td>
<td>49</td>
</tr>
<tr>
<td>6-4</td>
<td>Performance of parallel implementations on a 15³ element mesh.</td>
<td>49</td>
</tr>
<tr>
<td>6-5</td>
<td>Performance of parallel implementations on a 25³ element mesh.</td>
<td>50</td>
</tr>
<tr>
<td>6-6</td>
<td>A comparison of the fastest serial and parallel implementations.</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Tables

6.1 Speedup achieved with parallel Sketch ..................... 51
Chapter 1

Introduction

1.1 Motivation

High performance computing (HPC) systems are highly complex and there are many potential areas that allow for parallelism to be leveraged, such as between compute nodes, coprocessors within a node, or multiple cores on a chip. Unified solutions to this programming challenge are lacking, and programmers must use a variety of tools at various levels of abstraction. This fractured approach can make it difficult to reason about correctness and efficiency.

1.2 Thesis Scope

For my Master of Engineering thesis, I worked under Professor Armando Solar-Lezama to contribute to the Sketch, or MSL, programming language. Sketch already incorporates distributed memory parallelism, and my goal was to expand upon this with shared memory parallelism. The primary contributions of this thesis were integrating the existing fork-join parallel programming model into Sketch, leveraging the Sketch analysis engine to avoid non-deterministic behavior, and investigating the interaction of this programming model with the Sketch feature-set.

Chapter 2 presents relevant background material. Chapter 3 discusses the design of a virtual multiprocessor. Chapter 4 covers the application of Sketch’s deep semantic
checking to race condition detection. Chapter 5 evaluates the correctness and small scale performance of the shared memory parallelism. Chapter 6 presents a case study regarding the implementation of a large benchmark using Sketch. Chapter 7 concludes the thesis while giving suggestions for future work.
Chapter 2

Background

This section gives a brief survey of trends in HPC programming and an introduction to relevant features of Sketch.

2.1 Single Program Multiple Data (SPMD)

A common method for achieving parallelism on HPC systems is to run a program that assigns different parts of the shared data to each process by checking the id of the process. This is referred to as SPMD. In the past, a cluster of single core computers would assign one process per compute node. This was a simple arrangement because each process that had its own individual memory and all processes were separated on the cluster’s network.

With the advent of multicore processors, SPMD has become more complex, as a single CPU can host a large number of processes. This means that some processes will be accessing the same local memory and that inter-process communication bandwidth is no longer uniform, or there could be unused multicore resources. Since SPMD does not allow for separation between parallelism and the locality of data, it is severely limited in terms of expressiveness. High performance is often achieved through handwritten low-level code that makes use of powerful parallelism tools. From a programming standpoint, marshaling these processes and reasoning about their performance and correctness can be extremely difficult.
2.2 Related Work

Since Sketch currently generates SPMD code, in order to improve the performance and flexibility of Sketch, it was important to understand the shortcomings of SPMD by studying burgeoning HPC languages. Many popular HPC languages use a model based on a partitioned global address space (PGAS).

2.2.1 Chapel

Chapel is a PGAS language that is designed to be a complete solution to all parallel programming challenges and uses several themes to support this aim. The salient themes are robust support for general parallel programming and explicit locality control [1]. There are multiple ways in which Chapel supports general parallel programming. Firstly, "Chapel was designed such that higher-level abstractions, such as those supporting data parallelism, would be built in terms of lower-level concepts in the language, permitting the user to select amongst differing levels of abstraction and control as required by their algorithm or performance requirements[1].” Besides fine grain control, there is also the added benefit of compatibility between user defined parallelism and higher-level abstractions [1]. These abstractions also support Chapel’s general parallel programming by making it possible to leverage all available parallel resources without needing to use a variety of different tools. The bases of this unified parallelism are locales, which "represent units of the target system architecture that are useful for reasoning about locality and affinity [1],” and tasks, which "represent units of computation that can, and should, execute in parallel [1].” Processes within Chapel will typically be multi-threaded to support having a single locale running multiple tasks, which can spawn additional local or remote tasks [1]. This construction of tasks is used to support global views. Chapel uses a global view of control, since the entire program is actually a single task that spawns additional sub-tasks[1]. Additionally, to support the PGAS model, Chapel allows users to leverage a global view of data structures [1], which allows for computation to be easily distributed between locales. However, as is consistent with the desire for Chapel to be an all-purpose
solution, users are not forced to use global data structures and can choose to use local data structures and manual sharing when necessary [1]. If the user leverages the PGAS to access a remote variable, generating the required communication is the responsibility of the compiler and runtime, which also allows users to dynamically query for the location of data [1]. Chapel also allows users to specify the locale in which a statement is to be executed using an on statement, and since locales are essentially virtualizations of multiprocessors, local accesses will be cheaper than remote ones [1]. Chapel provides multiple ways to synchronize tasks. ”Two ways in which a user can check for task completion are through the sync keyword or by synchronizing with it through shared variables [1].” Additionally, Chapel includes atomic variables, updates to which are only visible to other tasks once completed.

2.2.2 X10

X10 is another language targeted at productive HPC programming that uses a partitioned global address space. At a high level, its salient features are similar to Chapel. The X10 analog of Chapel’s locale, virtualized multiprocessor, is called a place, and within a place the programmer can spawn activities [2], which are analogous to tasks in Chapel. Activities are meant to be lightweight and run asynchronously in a multi-threaded environment. X10 provides statements for waiting for a process to terminate, and to interact with remote processes [2]. This explicit control of communication makes it clear to the programmer when communication is taking place and is also similar to Chapel. One major difference between the two languages seems to be their approach to atomicity. Chapel makes use of atomic variables and also provides helpful synchronization variables [1]. This seems consistent with the language’s approach to allowing the user to leverage low-level primitives. X10 appears to take a different, more unique approach. It allows the user to declare a segment of code as atomic and guarantees that segment will not be interrupted by other activities.
2.2.3 Habanero Java

Habanero Java is an effort to extend X10 [3]. Habanero Java builds upon the benefits of being a high level language and focuses on providing usability and safety. Two of the major additions to X10 include futures, which are return values from asynchronous activities, and phasers, which allow activities to register for signaling [3]. The language can actually be added to java programs quite easily, and be seen as a fairly simple extension. [3] outlines the conditions under which a Habanero Java program is guaranteed to be deadlock free, data race free, serializable, and deterministic.

2.3 Sketch

Professor Solar-Lezama’s Computer Assisted Programming Group has been developing a novel programming language, Sketch or MSL, which seeks to generate efficient parallel code from high-level abstractions. These abstractions make the complex underlying implementations invisible to the programmer, which makes the code cleaner and more reusable. Sketch achieves this through a combination of generative programming and software synthesis [4]. The Sketch synthesizer can simplify code and make it more reusable by being able to infer details about the implementation using constraint based synthesis. Additionally, the synthesizer can check correctness through deep semantic analysis that will “ensure that the resulting code avoids assertion failures for any input in the input space under consideration.” [4]. Currently, the parallel code generated by MSL uses the single program multiple data (SPMD) model.
Chapter 3

Virtual Multiprocessor Design

Although Sketch does not implement a PGAS, similar features would be beneficial additions. Primarily, the PGAS model requires the addition of a virtual multiprocessor that allows the programmer to leverage the parallel resources of a multiprocessor. Providing this functionality requires striking a balance between expressiveness and the problems with traditional parallel programming tools that were mentioned earlier. It was determined that the best implementation would be the addition of a parallel for loop, i.e. a new for loop statement that executes each iteration in parallel to the greatest extent possible in a fork-join manner without explicit location objects like those found in PGAS. Additionally, the Sketch solver will be used to guarantee that the parallel code is free from race conditions.

3.1 Parallel For Loop

The primary benefit of the parallel for loop, "pfor," is that its basis in a serial construct aids programmability. A programmer can use the pfor statement to designate where parallel processing ought to be leveraged without necessarily having to go through quite the same level of reasoning that is involved with traditional parallel programming, e.g. explicit multithreading. This, however, introduces a trade-off between expressiveness and efficiency vs simplicity. Designating an inherently serial loop as parallel could present several issues, including dependencies and poor locality. How-
ever, a benefit of the parallel for loop is that it can be used similarly to an spmd fork command that takes place only on one machine. This is not as simple but can be used for better locality.

Sketch does not include synchronization primitives because they present a variety of parallel programming issues. Mismanaging synchronization results in subtle data races or deadlocks. Properly managing synchronization can still make performance difficult to reason about and would drastically complicate our determinism analysis. This limits expressiveness but we can ensure correctness by checking that iterations are independent and force the programmer to avoid race conditions.

3.2 Technical Approach to Generation

The last step in compiling Sketch is the generation of parallel code. The process performs certain optimizations [1], but, for the purposes of my thesis, the most important action is the translation of Sketch’s high-level parallelism instructions into C++/MPI instructions. Currently, Sketch’s generation of MPI instructions is fairly straightforward. "The language includes two communication primitives: reduce and transfer [1].” The generation of instructions for reduce is simple because there is a nearly direct MPI analog [1]. The transfer function is more complicated to implement, but [1] details how it can be implemented using asynchronous MPI calls in conjunction with sequential message tags.

This thesis focuses on creating a virtual multiprocessor, which requires leveraging a multiprocessor level parallel tool and using the sketch constraint satisfaction to ensure the correctness of parallel code. We rely on an implementation that generates C++ with openmp ”#pragma for” loops. Since the final code generation is a distinct step, it would be a simple matter to replace the openmp statements with a different parallelism engine, which could include a thread pooling solution that is semantically identical to a parallel for loop.

The parallel for statement easily integrates with the sketch language. A new statement "pfor", which is written identically to a normal for, is turned into a for loop
in the sketch AST. However, an attribute, isParallel, was added to differentiate. From there, an AST visitor was written to apply sketch’s satisfiability solver to ensuring race free parallel loops. Assertions are injected into a copy of the program that is used by the sketch backend solely for analysis. These assertions allow the sketch solver to reason about the parallel region in a sequential manner, and their design and implementation of this assertion analysis is discussed later.
Chapter 4

Leveraging Synthesis for Checking Correctness

Sketch creates a copy of the input program that it uses solely for the purpose of synthesizing code and serially checking constraint satisfiability. Sketch will throw an error if it determines that not all constraints can be satisfied. Sketch also allows for assertions to be injected into the analysis copy, so assertions related to race detection can be used to prove correctness, although it must be shown that assertions injected into parallel code can be effective when reasoned about in a serial manner by the Sketch solver, which unrolls loops.

4.1 Basic Algorithm

Avoiding data races requires tracking states related to the accesses to each memory location. 4.1 displays the state transition diagram that is described here. The diagram shows only valid transitions and those not pictured cause assertion failures. In the Sketch race detection system, a variable can be unused, exclusively read, shared, or modified. "Unused" variables have not been read or written by any iteration. "Exclusively read" variables have only been read by one iteration. Upon entering this state, a variable can either be written only by the one iteration to have read or be read by a different iteration and promoted to "Shared." Shared variables are read by
multiple iterations and written by none. "Modified" variables are written by a single iteration and cannot be shared by others, since this would constitute a race condition. A location that has been written by an iteration can be said to be "owned" by that iteration.

![Race detection state diagram](image)

**Figure 4-1: Race detection state diagram.**

In order to track the necessary variable states, assertions and bookkeeping variables are injected into the program prior to loop unrolling. Each variable and array slot that was initialized outside of the parallel loop receives a "ghost_read" and "ghost_write" shadow variable that are used to track which iterations are reading and writing to that location. These shadow variables are integers that will be used to store iteration ids or special values outside the loop’s range to represent the shared or unused state. Variables declared inside of parallel loops are local to each iteration and therefore cannot cause race conditions and do not require shadow variables. For each read, we must assert that the location is either unwritten or owned by the reading iteration. This means checking ghost_write for a null value or the accessing iteration’s id. After the assertion, we update ghost_read. If ghost_read hasn’t been written, then the variable is unused. The iteration id is written to ghost_read and the
variable is promoted to exclusively read. If ghost_read already has another iterations id, then the variable is already exclusively read and the second read means a special value indicating that the location is shared must be written to ghost_read. Sharing must be specially recorded because reads by multiple iterations are permissible and loop unrolling cannot be allowed to clobber ghost_read. If the variable is already exclusively read by the accessing iteration, i.e. ghost_read contains the iteration id, then ghost_read need not change. Race conditions occur when a write occurs to a location another iteration has read or written. Therefore, for each write, we must assert that the location is not shared and that it is either unwritten or owned. This means that ghost_read and ghost_write can only contain either null or the iteration id. Then ghost_write is written with the iteration id to indicate ownership. Only one iteration may write, so only a single iteration id needs to be stored.

4.2 Proof of Correctness

It must be shown that these assertions, when reasoned about serially, will actually prevent race conditions from occurring while the accesses are executed in parallel. First we must develop the notion of a pfor instruction sequence. Within a parallel region, the instructions within an iteration will be executed in their program order, however, there are no guarantees about the order in which instructions in different iterations are executed with respect to each other. For an example, let’s define a pfor instruction sequence as a series of pairs containing the iteration number and the instruction’s place within an iteration respectively. The properties discussed so far allow for a parallel for loop with two instructions per each of its two iterations to be represented as $[(1,1)(1,2)(2,1)(2,2)], [(2,1)(2,2)(1,1)(1,2)], [(1,1)(2,1)(1,2)(2,2)]$, etc.

Not all pfor instruction sequences represent correct, race-free operation. We can define a correct, deterministic sequence as one in which exchanging two operations from different iterations, while preserving the intra-iteration ordering, does not affect the program output and leaves us with a correct sequence. Such sequences will give
the same output regardless of how parallel execution interleaves their execution, and are therefore race condition free.

Since the Sketch solver will unroll loops into pfor instruction sequences before checking satisfiability, we can expand on our understanding of a pfor instruction sequence to validate our race detection assertions are effective, that they will all be satisfiable if and only if the sequence in which they were injected is good. First, we really only need to consider instructions the read or write global variables, and we must be aware of which are reads and writes. Second, since the assertions are not actually executed, we can consider them as part of an atomic tandem with the instruction they are checking. Therefore, exchanging instructions will also move the assertions the instruction triggered.

We can now prove by cases of exchanging operations which access the same location from different iterations that our assertions guarantee correct sequences.

Case 1 exchanging reads. If both exchanged instructions are reads, then program output will only change if the location is also being written by at least one iteration. If this were true, either at least one of the read’s assertions would fail after seeing another threads ownership or the write assertions would fail upon seeing an exclusively read or shared location. If the location is only being read, there will be no output change and the assertions will pass.

Case 2 exchanging writes. If both exchanged instructions are writes, then the second write will fail because the first iteration will own the location.

Case 3 exchanging a read and a write. Either the write assertions will see the location is exclusively read, shared, or owned and fail, or the read assertions will see that the location is owned and fail.

Since operations to different locations cannot directly cause race conditions, we do not need to consider them in our cases. If such an exchange did cause a race condition, the offending operations would have to be applied to the same location, which means they would be included in our cases and the sequence would be incorrect, which means the pre-exchange sequence was incorrect as well.
4.3 Implementation

4.3.1 Basic State

The race detection is done by an AST visitor pattern that tracks certain global state variables while passing over the code. Upon reaching a parallel for loop, a global Boolean is set so that visits that occur within the loop body can easily check whether they are in a parallel region. The Boolean is cleared when the pfor visit is finished. While within a parallel region, it is important to differentiate between reads and writes. Therefore, a Boolean tracks if the visitor is on the left hand side of an assignment within a parallel loop. This Boolean is turned off, for example during visits to array ranges, if an object is on the left hand side but not actually being written. Each parallel loop maintains its own symbol table that registers variable declarations found in the loop. This is used to differentiate between local variables, which do not need race detection analysis, and global variables. Additionally, the table registers the ghost variables used in our analysis to ensure that they are declared once and the analysis code is valid. The visitor also tracks a list of declaration and initialization statements for the analysis variables. These statements are created whenever a global variable is read or written within a pfor statement, and they are inserted into the code prior to the pfor so that they are valid and the sequential analysis does not consider them to be overwritten with each iteration.

4.3.2 Reaction to key statements

For Loop

When visiting a for loop, an error is thrown if it is a nested pfor, i.e. the loop is parallel and the global state indicates a pfor visit is active. If the loop is a non-nested parallel loop, we set the global state to indicate that we’re visiting a pfor loop and initialize the other global state variables. After visiting the body of the loop, the ghost variable initializations, if any, are inserted before the loop and a new loop, including assertions, is returned.
visitStatementFor (loop) {
    if ( NestedPFor() ) throw error
    if ( loop.isParallel() ){
        visitingPFor = true
        InitializeLoopState()
        loop.body = loop.body.accept(this)
        AddGhostInitializers()
        ClearPForState()
        visitingPFor = false
        return loop
    }
}

Variable Declaration

Visiting a variable declaration statement causes the variable to be registered in the parallel for symbol table.

Assignment

When visiting an assignment, the global state is changed to reflect visiting the left side, checking for writes. We then visit the left hand side, rollback state change, and visit the right hand side. Assertions are added before the assignment by the visits to each side.

visitStatementAssignment (assn) {
    if ( visitingPFor ){
        visitingPAssign = true
        assn.leftSide = assn.leftSide.accept(this)
        visitingPAssign = false
        assn.rightSide = assn.rightSide.accept(this)
        return decl
    }
}
Variable Expression

If the variable is on the left hand side of an assignment, perform the race detection write check, otherwise perform the race detection read check.

```java
visitStmtVariableExpression(var) {
    if (visitingPFor) {
        if (visitingPAssign) {
            variableWriteCheck(var)
        } else {
            variableReadCheck(var)
        }
    } else {
        return var
    }
}
```

Array Range Expression

If the array location is being assigned to, then we turn off the assignment Boolean while visiting the range expression to check for references to global variables in the index. We then perform the race detection write check. Otherwise, we perform the race detection read check.

```java
visitArrayRangeExpression(arrRange) {
    if (visitingPAsn) {
        visitingPAsn = false;
        arrRange.accept(this)
        visitingPAsn = true;
        arrayWriteCheck(arrRange);
    } else if (visitingPFor) {
        arrayReadCheck(arrRange);
    }
    return arrRange
}
```
Access Checks

These are the checking functions called from variable and array range expressions. If the location is not local these function handle the assertion insertion. First the ghost variables are initialized if the variable is previously unused. The functions then generate the assertions for validating legal state transitions and the code for updating the ghost variables.

```c
1  readCheck ( var ) {
2       if (!PForSymbolTable . has ( var )) {
3           CheckGhostVariableInitialized ( var )
4           AssertUnwrittenOrOwned ( var )
5           UpdateGhostRead ( var )
6       }
7  }
8
9  writeCheck ( var ) {
10     if (!PForSymbolTable . has ( var )) {
11        CheckGhostVariableInitialized ( var )
12        AssertUnwrittenOrOwned ( var )
13        AssertNotShared ( var )
14        SetOwned ( var )
15     }
16  }
```
Function Call

This function is used to inline called functions. First, a copy of the local pfor symbol table is made. Then the parameters in the function call are checked against the parameters in the function definition to determine which are going to be passed by reference. If any of the variables passed by reference are local, then the name from the function definition is registered in the local symbol table as well. The function call is replaced with the code returned from visiting the functions body with the return statement omitted. The reference variables that were inlined, as well as all references to their associated ghost variables, are then renamed to match the variable name used in the function call. The symbol table is then restored to prevent renaming related corruption.

```c
visitFunctionCall(func) {
    if (visitingPFor) {
        BackupSymbolTable();
        MapVariablesToReferenceParams();
        Inline = (func.getBody()).accept(this);
        RenameReferences(Inline);
        RestorSymbolTable();
    }
    return Inline;
}
```
Chapter 5

Parallel For Loop Evaluation

Parallel for loop was evaluated using canonical test cases and small scale benchmarks that are not inherently parallel.

5.1 Canonical Tests

5.1.1 Passes

These tests represent common, legal use cases and are all permitted by the race detection.

Scalar Access

This test demonstrates straightforward, legal scalar accesses.
```c
int Scalar () {
    int i = 0;
    int a = 0;

    // Read only global scalar
    // Read and write local scalar
    pfor (i = 0; i < 8; i++) {
        int j = i;
        int b = a + 1;
        j = 0;
    }
    return 0;
}
```

Figure 5-1: Permissible scalar access.

**Global Scalar Write**

This test shows that a single thread can write to a global scalar, since no other threads have read from it.

```c
int GlobalScalarWrite () {
    int i = 0;
    int a = 0;
    pfor (i = 0; i < 8; i++) {
        int j = i;
        j = 0;

        // A single thread reads and writes globally
        if (i == 7) {
            a = 5;
            j = a;
        }
    }
    return a;
}
```

Figure 5-2: Permissible global scalar write.
Local Array

This test shows that arrays are allocated locally to each iteration and therefore cannot create race conditions.

```c
int LocalArray () {
    int i = 0;
    // Read and write local array
    for (i = 0; i < 8; i++) {
        int[8] c;
        c[0] = 1;
        c[1] = c[0];
        c[i] = c[i % 4];
    }
    return 0;
}
```

Figure 5-3: Local arrays are race free.
Global Array Indirection

This is a substantially more complex example. An indirection array is used to hold indexes into an output array. First the index array is populated with a permutation of the valid index values. Afterwards, each iteration in a pfor is assigned to use a different contiguous chunk of the index array to determine where in the output array to write its iteration id. This test passes if the indirection array contains a true permutation of the output indexes or if any duplicated indexes are assigned to the same thread.

```c
int GlobalArrayIndirection() {
    int [64] output;
    int [64] indirect;
    for (int i=0; i<8; i++) {
        for (int j=0; j<8; j++) {
            indirect[i + j * 8] = (8 * i + j);
        }
    }
    pfor (int j=0; j<8; j++) {
        for (int i=0; i<8; i++) {
            output[indirect[j * 8 + i]] = j;
        }
    }
    return 0;
}
```

Figure 5-4: Indirection array that legally permutes the accesses.
Function Call

This test will inline the function code and include it in the analysis. The test passes because each thread only reads from globalInd and c.

```c
int globalInd = 0;
int arrayAccess () {
    return globalInd;
}

harness int FunctionCall () {
    int i=0;
    int j=0;
    int [8] a;
    int [8] b;
    int [8] c;
    pfor (i=0; i<8; i++) {
        // global index w/o dependency
        a [i] = i;
        // indirect index w/o dependency
        b [a [i]] = i;  // b[i]=i;
        a [i] = c [arrayAccess ()];
        b [i] = c [arrayAccess ()+i];
    }
    return 0;
}
```

Figure 5-5: Function call legally accessing global scalar.
5.1.2 Failures

This section presents test cases that contain race conditions and cause assertion failures in Sketch.

Loop Carried Dependency

This test shows a parallel for loop in which the value of an array element depends on the value set in an adjacent slot by a different iteration. This is known as a loop carried dependency and is a common obstacle to the use of parallel loops. If the dependence cannot be broken, then the loop cannot be executed in parallel.

```c
harness int loopCarriedDependency (){  
    int i=0;  
    int j=0;  
    int [8] a;  
    int [9] aa;  
    for (i=0; i<8; i++)  
        aa[i]=i;  
        a[i] = aa[i+1] + i;  
    }  
    return 0;  
}
```

Figure 5-6: Loop carried dependency.
Global Races

These tests show two obvious data races created by having all iterations accessing the same global variable.

```c
int globalRaceScalar(){
    int i=0;
    int a=0;
    for (a=0; a<8; a++){
        int j = i;
        int b = a+1;
        j+=b;
        i=j;  // race writing to i
    }
    return i;
}

int globalRaceArray(){
    int i=0;
    int[8] a;
    for (i=0; i<8; i++){
        a[0] = i;  // race at a[0]
    }
    return 0;
}
```

Figure 5-7: Obvious race conditions.
**Invalid Indirection**

This test demonstrates an invalid array indirection. The indexes contained in the first array lead only to the first and second slot of the second array.

```c
harness int invalidIndirection() {
    int i = 0;
    int[8] a;
    int[8] aa;
    for (i = 0; i < 8; i++) {
        aa[i] = (i % 2);  // indirect index w/ dependency
        a[aa[i]] = i;
    }
    return 0;
}
```

Figure 5-8: Indirections that collide are detected.
Race In Function

This test checks that race conditions can be detected in functions called within parallel loops. In this test, the indirect access pattern will result in independent array accesses in serial execution, but the function updates a global scalar in a non-thread-safe manner.

```c
int globalIndex = -1;
int arrayAccess (){
    globalIndex += 1;
    return globalIndex;
}

harness int outOfScopeRace (){
    int i=0;
    int [8] a;
    int [8] b;
    int [8] c;
    pfor (i=0; i<8; i++){
        a[i]=i;
        b[i] = a[arrayAccess ()];
    }
    return 0;
}
```

Figure 5-9: Race detected in function call.

5.2 Small Scale Benchmarks

The performance of the parallel for loop was examined in part using two small scale benchmarks, a reduce iterator that applies a function to every element of a list and an image processing program. These benchmarks were chosen to explore different methods of parallelizing code and give an understanding of the overhead created by parallelization. All tests were performed on the authors desktop computer, which was running Ubuntu 12.04 on an Intel i7-970 hexacore processor.
5.2.1 Reduce

The reduce benchmark is based on an iterator over a linear array. The iterator is supplied with a function that takes an input and the previous output as arguments. The benchmark generates an array of a specified length containing random integers and applies six different functions to the array.

Four different implementations were tested. First, there was a straightforward serial implementation. This was converted to a fine grain parallel implementation by changing the serial for loop to a pfor. Two coarse grain implementations were also created. One splits the input array into one continuous segment for each pfor iteration, and the other assigns one reduce function to each iteration of a pfor.

In 5.2.1, we can see how the run times for each implementation scale with respect to the size of the input array. The fine grain parallelism is strictly slower than the serial implementation. This result was not surprising since an unstructured parallel iteration over a one dimensional array should yield poor locality. The coarse grain implementations perform better, but they require a large input array before they can overcome the treading overhead. 5.2.1 shows the speedup that is achieved by the various implementations when the input array has $10^7$ elements. The coarse grain implementations scale well with the first few cores, but the speedup saturates before all cores are used. A substantial speedup is still provided, though.
5.2.2 Image Processing

The other small scale benchmark used was an image processing program that applies a filter to an input bmp in order to detect edges. The program takes a bmp image as input and creates a three-dimensional array that stores the RGB values of each pixel. A simple iterator of two nested for loops is used to iterate over the rows and columns of the picture to apply the filter. Since the two dimensional array should
yield reasonable locality, and Sketch is meant to produce clean code, this benchmark was parallelized by simply changing the inner and outer loops of the iterator from "for" to "pfor."

5.2.2 shows the speedup that was achieved by each implementation when they were run on the same image, an 800px x 611px image of the MIT’s Great Dome. Both implementations provided a substantial speedup and saturated similarly to the reduce benchmark. The fact that inner and outer loop parallelization yield similar results is a positive sign for the ease of use, but also likely indicates that improved performance should be possibly since the methods use inverted memory access patterns that should yield different results.

![Image processing speedup scaling.](image-url)
Chapter 6

Lulesh Case Study

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a benchmark published by the Lawrence Livermore National Laboratory as a high performance computing challenge problem. It is a simplified hydrodynamics simulator that solves a "simplified Sedov blast problem." [5] The program represents a mass of material as a mesh of volumetric elements in order to be representative of the operation of more complex general purpose simulators. [5] Lulesh was transliterated from C++ to sketch with the aim of remaining faithful to the intention of Lulesh. Although, using sketch, it is possible to replace repetitive C++ with generators that make the code appear cleaner, the Sketch Lulesh implementation does not add substantial abstractions or optimizations to the reference Lulesh code provided at [6].

6.1 Serial Performance

6.1.1 Initial results

Initially, the serial Sketch version of Lulesh was implemented using method signatures that matched the C++ reference version. This meant passing ref parameters and using non-generator function calls. This required substantial array copying that proved to yield poor performance, especially since some kernels require large arrays and are called many times. 6.1.1 shows that this version performed poorly on a small 125
element, 5 elements per side, mesh and scaled to even worse performance. However, there was substantial room for improvement remaining before adding parallelism.

Figure 6-1: Performance of serial C++ and default Sketch.

6.1.2 Serial Optimizations

This section discusses the two major performance optimizations that were employed to make Sketch competitive with the C++ reference.

Generators

From the nature of the data structures used in Lulesh and the semantics of sketch, it was not surprising that regular functions provided slow performance. The most obvious optimization was to take advantage of Sketch’s "generator" functionality that inlines the code thereby sidestepping the array copying. 6.1.2 Shows that generators offered a substantial speedup from the naive implementation but fell roughly 20% short of the reference.
Loop Memory Optimization

Analysis of the generator code revealed inefficiencies in the memory management. Sketch employs explicit memory management. The use of generators meant that memory management calls were being inlined inside of loops, which is an obvious inefficiency. To make the memory management more efficient, a visitor pattern was designed that would relocate variable declarations, and therefore their allocations, without affecting correctness. The sketch implementation that employs this strategy is referred to as pre-allocation, and the relaxed memory model can be referred to as declarative consistency. This model works because Sketch guarantees that there are no references made to a variable before its declaration. This means that no errors will arise from moving the declaration earlier in the program, and the output will not change as long as the declaration is replaced with an assignment to the initial value. This does require though, that all values associated with the declaration, e.g. array dimensions, are identical in the original and final declaration locations, otherwise consistency would be violated for those secondary variables. A simplified visitor implements this optimization for constant length arrays, although synthesis may be leveraged to generalize this approach. The visitor pattern simply keeps track of the beginning of the highest level loop it is currently inside of, it safely moves array declaration in front of the highest loop, and it replaces the declaration with an assignment to the initial value. 6.1.2 shows that a serial Sketch implementation employing pre-allocation will perform nearly identically to the reference C++ implementation.
6.2 Parallel Implementations

Lulesh was used to provide the largest test for the shared memory parallelism. It is larger and more realistic than the small scale benchmarks, and it includes a reference version implemented in openmp [6]. The pfor based implementation was created by directly transliterating the changes between the serial and parallel references. This parallelized all large for loops while only needing to add dependency resolution code in a few places.

As was the case with the serial implementation, the initial performance suggested there was room to optimize. Pre-allocation was applied to the pfor implementation, but this yielded virtually no speedup. This can be explained by considering that most loops had been converted to parallel loops, which requires providing each iteration with a working set of local variables that cannot be moved outside of the loop. This lead to a new optimization, hand modifying the generated code to stack allocate iteration-local variables. This change yielded substantial speedup, but, as shown in figures 6.2.1, 6.2.1, and 6.2.1, the openmp reference remained the fastest.

6.2.1 Results
Figure 6-3: Performance of parallel implementations on a $5^3$ element mesh.

Figure 6-4: Performance of parallel implementations on a $15^3$ element mesh.
Figure 6-5: Performance of parallel implementations on a $25^3$ element mesh.
6.2.1 and 6.2.1 show the speedup that was achieved by moving from the fastest serial implementation to the fastest parallel implementation. These values seem consistent with the speedups that were witnessed in the small benchmarks, so although it may have been possible to further optimize the pfor memory management in ways beyond what was included in the reference, some other aspect of the generated code seems to be limiting the scalability.

Figure 6-6: A comparison of the fastest serial and parallel implementations.

<table>
<thead>
<tr>
<th>Elements</th>
<th>$5^3$</th>
<th>$15^3$</th>
<th>$25^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.684404</td>
<td>2.14352</td>
<td>2.173001</td>
</tr>
</tbody>
</table>

Table 6.1: Speedup achieved with parallel Sketch
Chapter 7

Conclusion

7.1 Conclusion

In this thesis, I briefly surveyed parallel programming problems and HPC solutions in order to motivate mapping the fork-join shared memory parallelism to a Sketch virtual multiprocessor. I then discussed how the serial constraint checker used in Sketch can be leveraged to ensure correct, deterministic operation on all inputs. I then detailed the tests and benchmarks that were used to evaluate the performance of the extended programming model. Overall, the new model demonstrated an ability to speed up Sketch implementations without adding the complexity of reasoning about thread safety.

7.2 Future Work

In addition to further exploration of the parallel loop performance, I have identified a few areas for potential future work.

7.2.1 Syntax Improvements

Parallel for loops could retain their beneficial properties while becoming even easier to use with a few simple syntactic improvements.
Read-only input copies

In order to resolve dependencies, it is often necessary to instantiate a copy of an array so that one copy can remain read-only within a parallel region. This requires explicit management by the programmer and can introduce errors while making the code more complex. Sketch could allow for the programmer to specify, as arguments to the loop, arrays that they would like to have copies of. This would be implemented such that iterations may not overlap writes but can see the writes to locations they own while seeing only pre-loop values in other locations.

Barrier

A barrier primitive is a fairly standard primitive that is easy to reason about. It can be replaced in a program by using two separate loops and checked for correctness in the same way.

Semaphore

Partially to complement the barrier, a limited, protected semaphore primitive could enhance expressiveness without introducing undesirable complexity. Each loop could have a designated semaphore that can be safely set and read by any number of iterations. This would allow for the early termination of loops by surrounding segments with conditional checks to the semaphore and/or barriers.

7.2.2 Per Iteration hole resolution

Another area for future work that will aid code simplicity but goes beyond being a simple syntax change would be for holes, unknown integers filled in by Sketch, to be synthesized on a per thread basis in parallel regions. For example, it may be necessary to change the bounds on a for loop to avoid a buffer overrun, but it is not true that all iterations need the same buffer. This would eliminate complex setup code when holes are required in parallel regions.
7.2.3 Locality and Thread Pooling

The initial implementation of pfor described in this thesis relies on openmp to parallelize the code, but a more specialized threadpooling could be developed and bring better results by better integrating with sketch. For example, holes could be used to determine characteristics of the memory accesses in a parallel region. This information could then determine how best to parallelize the region. As we saw from the Reduce benchmark, some of the one dimensional pfor array traversals in the parallel Lulesh implementations may actually perform faster when executed serially.

7.2.4 Asynchronous Join

The current distributed shared memory implementation blocks when joining threads, which imposes a bulk-synchronous style to the program. If these joins could be performed asynchronously then the program model would become substantially more flexible. The joins would still serve as synchronization points by guaranteeing that updates made before the joins are visible to operations after the join while only blocking as little as is necessary to ensure no updates are lost.
Appendix A

Sample of Generated Analysis Code

```c
harness int test6(){

    int[64] output;
    int[64] indirect;

    for(int i=0; i<8; i++){
        for(int j=0; j<8; j++){
            indirect[i + j*8] = (8*i + j);
            // indirect[i+j*8] = 0;
        }
    }

    pfor(int j=0; j<8; j++){
        for(int i=0; i<8; i++){
            output[indirect[j*8 + i]] = j;
        }
    }

    return 0;
}
```
void test6 (ref int _out_a) /* largePass.sk:92 */
{
{
{
    _out_a = 0;
    int[64] output_b;
    int[64] indirect_c;
    int i_d;
    for(i_d = 0; i_d < 8; i_d = i_d + 1) /* Canonical */
    {
        int j_10;
        for(j_10 = 0; j_10 < 8; j_10 = j_10 + 1) /* Canonical */
        {
            int _pac_s18;
            _pac_s18 = i_d + (j_10 * 8);
            assert ((_pac_s18 >= 0) && (_pac_s18 < 64)); //
            largePass.sk:99: Either null pointer or Array
            out of bounds
            indirect_c[_pac_s18] = (8 * i_d) + j_10;
        }
    }
    int j_11;
    int64 j_11_ghost_write;
    j_11_ghost_write = 0 - 1;
    int64 j_11_ghost_read;
    j_11_ghost_read = 0 - 1;
    int64[64] indirect_c_ghost_write;
}
for(int i = 0; i < 64; i = i + 1)/*Canonical*/
    indirect_c_ghost_write[i] = 0 - 1;
int64[64] indirect_c_ghost_read;
for(int i = 0; i < 64; i = i + 1)/*Canonical*/
    indirect_c_ghost_read[i] = 0 - 1;
int64[64] output_b_ghost_write;
for(int i = 0; i < 64; i = i + 1)/*Canonical*/
    output_b_ghost_write[i] = 0 - 1;
int64[64] output_b_ghost_read;
for(int i = 0; i < 64; i = i + 1)/*Canonical*/
    output_b_ghost_read[i] = 0 - 1;
for(j_11 = 0; j_11 < 8; j_11 = j_11 + 1)/*Canonical*/
{
    int i_14;
    for(i_14 = 0; i_14 < 8; i_14 = i_14 + 1)/*Canonical*/
    {
        int _pac_s20;
        assert (((j_11_ghost_write == j_11) || (j_11_ghost_write == (0 - 1))) ||
                  //Race condition
                  // at largePass.sk:107: scalar j_11 was not
                  //unwritten or exclusively written
                  j_11_ghost_read = ((j_11_ghost_read != j_11) &&
                                      (j_11_ghost_read != (0 - 1)) ? (8 - 1) + 1 : j_11)
                        );
        _pac_s20 = (j_11 * 8) + i_14;
        assert ((_pac_s20 >= 0) && (_pac_s20 < 64)); //
                    // largePass.sk:107: Array out of bounds
        int _pac_s19;
        assert (((indirect_c_ghost_write[_pac_s20]) == j_11
                   ) || ((indirect_c_ghost_write[_pac_s20]) == (0 -

59
1)); //Race condition at largePass.sk:107: array was not
unwritten or exclusively written

indirect_c_ghost_read[_pac_s20] = (((
indirect_c_ghost_read[_pac_s20] != j_11) && ((
indirect_c_ghost_read[_pac_s20] != (0 - 1))
? (8 - 1) + 1 : j_11);

_pac_s19 = indirect_c[_pac_s20];
assert ((_pac_s19 >= 0) && (_pac_s19 < 64)); //
largePass.sk:107: Either null pointer or Array
out of bounds

assert (((output_b_ghost_write[_pac_s19]) == j_11)
|| ((output_b_ghost_write[_pac_s19]) == (0 - 1)))
); //Race condition at largePass.sk:107: array
was not unwritten or exclusively written

assert (((output_b_ghost_read[_pac_s19]) == j_11)
|| ((output_b_ghost_read[_pac_s19]) == (0 - 1))))
; //Race condition at largePass.sk:107: array
was being shared by multiple iterations

output_b_ghost_write[_pac_s19] = j_11;
assert ((j_11_ghost_write == j_11) || (j_11_ghost_write == (0 - 1)));
//Race condition
at largePass.sk:107: scalar j_11 was not
unwritten or exclusively written
j_11_ghost_read = ((j_11_ghost_read != j_11) && (j_11_ghost_read != (0 - 1))
? (8 - 1) + 1 : j_11);

output_b[_pac_s19] = j_11;
}
}
_out_a = 0;
return;
}
}
}
References


