Verifying Parallel Loops with Separation Logic*

Stefan Blom Saeed Darabi Marieke Huisman
University of Twente
Enschede, The Netherlands
s.c.c.blom,s.darabi,m.huisman@utwente.nl

This paper proposes a technique to specify and verify whether a loop can be parallelised. Our approach can be used as an additional step in a parallelising compiler to verify user annotations about loop dependences. Essentially, our technique requires each loop iteration to be specified with the locations it will read and write. From the loop iteration specifications, the loop (in)dependences can be derived. Moreover, the loop iteration specifications also reveal where synchronisation is needed in the parallelised program. The loop iteration specifications can be verified using permission-based separation logic.

1 Introduction

Parallelising compilers can detect loops that can be executed in parallel. However, this detection is not perfect. Therefore developers can typically also use a pragma to declare that a loop is parallel. Any loop annotated with such a pragma will be assumed to be parallel by the compiler.

This paper addresses the problem of how to verify that loops that are declared parallel by a developer can indeed safely be parallelised. The solution is to add specifications to the program that when verified guarantee that the program can be parallelised without changing its meaning. Our specifications stem from permission-based separation logic [4, 5], an extension of Hoare logic. This has the advantage that we can easily combine the specifications related to parallelisation with functional correctness properties.

We illustrate our approach on the PENCIL programming language [1]. This is a high-level programming language to simplify using many-core processors, such as GPUs, to accelerate computations. It is currently under development as a part of the CARP project[1]. However, our approach also applies to other languages that use the concept of parallel loops, such as OpenMP [6]. In order to simplify the presentation in this paper, we limit ourselves to single loops. At the end of this paper, we will briefly discuss how to extend our approach to nested loops.

Below, we first present some background information, and then we introduce the specification language for parallel loops. Next, we sketch how we can implement automated verification of the specifications. Finally, we conclude with future work.

2 Background

Parallel Hardware. Modern hardware offers many different ways of parallelising code. Most main processors nowadays are multi-core. Additionally, they often have a set of vector instructions that can operate on small vectors instead of just a single value at once. Moreover, graphics processing units (GPUs) nowadays also can be used for general-purpose programming. Writing and tuning software for such accelerated hardware can be a very time-consuming task.

*This work is supported by the EU FP7 STREP project CARP (project nr. 287767).

[1] See http://www.carpproject.eu/.

Alastair F. Donaldson, Vasco Vasconcelos (Eds.): Proceedings
of the 7th Workshop on Programming Language Approaches
to Concurrency and Communication-cEntric Software (PLACES 2014)
EPTCS 155, 2014, pp. 47–53, doi:10.4204/EPTCS.155.7

The PENCIL Language. The PENCIL programming language is developed as a part of the CARP project. It is designed to be a high-level programming language for accelerator programming, providing support for efficient compilation. Its core is a subset of sequential C, imposing strong limitations on pointer-arithmetic. In addition to traditional C, it allows loops to be specified with two pragmas: *independent* and *ivdep*, indicating that a loop can be parallelised, because it is independent, or only contains forward dependences, respectively.

Loop Dependences. Several kinds of loop dependences can be identified. There exists a *loop-carried dependence* from statement S_{src} to statement S_{sink} in the body of a loop if there exist two iterations i and j of that loop, such that:

- Iteration i is before iteration j, *i.e.*, $i < j$.
- Statements S_{src} on iteration i and S_{sink} on iteration j access the same memory location.
- At least one of these accesses is a write.

When S_{src} syntactically appears before S_{sink} (or if they are the same) there is a *forward loop-carried dependence*, otherwise there is a *backward loop-carried dependence*. The distance between two dependent iterations i and j is defined as the *distance of dependence*.

On the right, we show examples of first a forward and then a backward loop carried dependence. In both cases there is a dependence between S_1 and S_2. In the first loop, the read in S_2 reads the value written in S_1 in the previous iteration of the loop. In the second loop, the read in S_2 must be done before the value is overwritten in S_1 during the next iteration.

```
for(int i=1;i<=N;i++){
  S1: a[i] = c[i] + 1;
  S2: c[i] = a[i-1] + 2;
}

for(int i=0;i<N;i++){
  S1: a[i] = c[i] + 1;
  S2: c[i] = a[i+1] + 2;
}
```

The distinction between forward and backward dependences is important. Independent parallel execution of a loop with dependences is always unsafe, because it may change the result. However, a loop with forward dependences can be parallelised by inserting an appropriate synchronisation in the code, while loops with backward dependences cannot be parallelised.

Separation Logic. Our approach to reason about loop (in)dependences uses permission-based separation logic to specify which variables are read and written by a loop iteration. Separation logic [9] was originally developed as an extension of Hoare logic to reason about pointer programs, as it allows to reason explicitly about the heap. This makes it also suited to reason modularly about concurrent programs [8]: two threads that operate on disjoint parts of the heap do not interfere, and thus can be verified in isolation. The basis of our work is a separation logic for C [10], but extended with permissions [5], to denote either the right to read from or to write to a location. The set of permissions that a thread holds are often known as its *resources*. We write access permissions as $\mathbf{perm}(e, \pi)$, where e is an expression denoting a memory location and $\pi \in (0, 1]$ is a fraction, where any value permits reading and 1 provides write permission. The logic prevents the sum of permissions for a location over all threads to exceed 1, which prevents data races. In earlier work, we have shown that this logic is suitable to reason about kernel programs [3].

```
for(i=0;i<N;i+=1)
/*@ requires perm(a[i],1) ** perm(c[i],1) ** perm(b[i],1/2);
    ensures  perm(a[i],1) ** perm(c[i],1) ** perm(b[i],1/2); @*/
  { S1: a[i] = b[i] + 1;
    S2: c[i] = a[i] + 2; }
```

Listing 1: Specification of an Independent Loop

3 A Specification Language for Loop Dependence

The classical way to specify the effect of a loop is by means of an invariant that has to hold before and after the execution of each iteration in the loop. Unfortunately, this offers no insight into possible parallel execution of the loop. Instead we will consider every iteration of the loop in isolation. To be able to handle dependences, we specify restrictions on how the execution of the statements for each iteration is scheduled. In particular, each iteration is specified by its own contract, *i.e.*, its *iteration contract*. In the iteration contract, the precondition specifies resources that a particular iteration requires and the postcondition specifies the resources which are released after the execution of the iteration. In other words, we treat each iteration as a specified block [7].

Listing 1 gives an example of an *independent loop*, specified by its iteration contract. The contract requires that at the start of iteration i, permission to write both c[i] and a[i] is available, as well as permission to read b[i]. The contract also ensures that these permissions are returned at the end of iteration i. The iteration contract implicitly requires that the separating conjunction of all iteration preconditions holds before the first iteration of the loop, and that the separating conjunction of all iteration postconditions holds after the last iteration of the loop. In Listing 1, the loop iterates from 0 to $N-1$, so the contract implies that before the loop, permission to write the first N elements of both a and c must be available, as well as permission to read the first N elements of b. The same permissions are ensured to be available after termination of the the loop.

To specify *dependent loops*, in addition we need the ability to specify what happens when the computations have to synchronise due to a dependence. During such a synchronisation, permissions should be transferred from the iteration containing the source of a dependence to the iteration containing the sink of that dependence. To specify a *permission transfer* we introduce the **send** keyword:

```
//@ send φ to L, d;
```

This specifies that the permissions and properties expressed by the separation logic formula ϕ are transferred to the statement labelled L in the iteration $i+d$, where i is the current iteration and d is the distance of dependence.

Below, we will give two examples that illustrate how loops are specified with **send** clauses. The **send** clause alone completely specifies both how permissions are provided and used by the iterations. However, for readability, we also mark the place where the permission are used with a corresponding **receive** statement as a comment. Listing 2 gives a specified program with a forward dependence, similar to our earlier example, while Listing 3 gives an example of a program with a backward dependence.

We discuss the annotations of the first program in some detail. Each iteration i starts with write permission on a[i] and c[i]. The first statement is a write to a[i], which needs write permission. The second statement reads a[i−1], which is not allowed unless read permission is available. For the first iteration, this read permission is available. For all subsequent iterations, permission must be transferred. Hence a **send** annotation is specified after the first assignment that transfers a read permission on a[i] to

```
for(int i=1;i<=N;i++)
/*@ requires i==1 ==> perm(a[i-1],1/2);
    requires perm(c[i],1) ** perm(a[i],1);
    ensures  perm(c[i],1) ** perm(a[i],1/2) ** perm(a[i-1],1/2);
    ensures  i==N ==> perm(a[i],1/2); @*/
{
    S1: a[i] = c[i]*CONST +a[i]*(1-CONST);
    //@ send perm(a[i],1/2) to S2,1;
    // if (i>1) receive perm(a[i-1],1/2);
    S2: c[i] = min(a[i],a[i-1]);
}
```

Listing 2: Specification of a Forward Loop-Carried Dependence

```
for( i=0;i<N;i++)
/*@ requires i==0 ==> perm(a[i],1/2);
    requires perm(c[i],1) ** perm(a[i],1/2) ** perm(a[i+1],1/2);
    ensures  perm(c[i],1) ** perm(a[i],1);
    ensures  i==N-1 ==> perm(a[i+1],1/2); @*/
{
   // if (i>0) receive perm(a[i],1/2);
   S1: a[i] = c[i]*CONST + a[i]*(1-CONST);
   S2: c[i] = min(a[i+1],a[i]);
   //@ send perm(a[i+1],1/2) to S1,1;
}
```

Listing 3: Specification of a Backward Loop-Carried Dependence

the next iteration (and in addition, keeps a read permission itself). The postcondition of the iteration contract reflects this: it ensures that the original permission on c[i] is released, as well as the read permission on a[i], which was not sent, and also the read permission on a[i−1], which was received. Finally, since the last iteration cannot transfer a read permission on a[i], the iteration contract's postcondition also specifies that the last iteration returns this non-transferred read permission on a[i].

The specifications in both listings are valid. Hence every execution order of the loop bodies that respects the order implied by the **send** annotations yields the same result as sequential execution. In the case of the forward dependence example, this can be achieved by adding appropriate synchronisation in the parallelised code. All parallel iterations should synchronise each **send** annotation with the location of the specified label to ensure proper permission transfer. For the backward dependence example, only sequential execution respects the ordering.

4 Verifying Dependence Annotations

To verify an iteration contract, we encode it as a standard method contract that can be verified using the VerCors tool set [2]. Suppose we have a loop specified with an iteration contract as below:

```
S_pre;
for(int i=0;i<N;i++)
/*@ requires pre(i);
    ensures  post(i); @*/
{   S; }
S_post;
```

To prove that this program respects its annotations, the following proof obligations have to be discharged:

- after S_{pre}, the separating conjunction of all of the iteration preconditions holds;
- the loop body S respects the iteration contract; and
- the statement S_{post} can be proven correct, assuming that the separating conjunction of the postconditions holds.

To generate these proof obligations, we encode the original program by generating several annotated procedures by the following steps:

1. We replace every loop in the program with a call to a procedure loop_main, whose arguments are the free variables occurring in the loop. The contract of this procedure requires the separating conjunction of all preconditions and ensures the separating conjunction of all postconditions. After this replacement, we can verify the program with existing tools to discharge the first and the last proof obligations.
2. To discharge the remaining proof obligation, we generate a procedure loop_body, whose arguments are the loop variable *i* plus the same arguments as loop_main. The contract of this procedure is the iteration contract of the loop body, preceded by a requirement that states that the value of the iteration variable is within the bounds of the loop.

The result of this encoding is as follows:

```
void block(){
    S_pre;
    loop_main(N, free(S));
    S_post;
}
```

```
/*@ requires (\forall* int i;0<=i && i<N; pre(i));
    ensures  (\forall* int i;0<=i && i<N; post(i)); @*/
loop_main(int N,free(S)));

/*@ requires (0<=i && i<N) ** pre(i);
    ensures  post(i); @*/
loop_body(int i,int N,free(S))){ S; }
```

Verification of the **send** instruction is done by replacing the **send** annotation with a procedure call send_phi(i); and by inserting a procedure call recv_phi(i); at the location of the label L. The contracts of these methods encode the transfer of the resources specified by $\phi(i)$ from the sending iteration to the receiving iteration, subject to two conditions:

1. Permissions can only be transferred to future iterations ($d > 0$).
2. Transfer only happens if both the sending and the receiving iterations exist.

The existence of iteration i is expressed by the predicate is_iteration(i), whose definition is derived from the loop bounds. For example, the loop **for**(**int** i=0;i<N;i++) gives rise to

```
boolean is_iteration(int i){return 0 <= i && i < N;}
```

Using this notation the generated (abstract) methods and contracts are:

```
/*@ requires is_iteration(i+d) ==> φ(i);
 @*/
void send_phi(int i);

/*@ ensures  is_iteration(i-d) ==> φ(i-d);
@*/
void recv_phi(int i);
```

Note that instead of a constant d, we may use any invertible function $d(i)$.

5 Conclusion and Future Work

This paper sketches how to verify parallel loops, even in the presence of dependences from one loop iteration to the next. The idea is to specify each iteration of a loop with its own iteration contract and to use the **send** annotation to transfer permission between iteration if needed. We conjecture that if verification of a loop is possible without using **send** then it is correct to tag the loop as independent, *i.e.*, an iteration never reads a location that was written by a different iteration. Moreover, if **send** is used with labels occurring after the statement then it is correct to use PENCIL's **ivdep** tag to indicate parallelisability.

The method described is modular in the sense that it allows us to treat any parallel loop as a statement, thus nested loops can be dealt with simply by giving them their own iteration contract. Alternatively one iteration contract can be used for several nested loops.

It is future work to provide a formal proof for our conjecture, as well as to develop fully automated tool support for discharging the proof obligations. We also plan to link our PENCIL specifications with our kernel logic [3] and to define compilation of PENCIL specifications.

Another possible direction for future work is to extend our approach to reason about the correctness of OpenMP [6] pragmas in parallel C programs. From the point of view of verification, many concepts in OpenMP and PENCIL are the same. For example, the **simd** pragma in OpenMP is used in the same way as PENCIL uses **ivdep**. In general, our method can be applied for verification of any high-level parallel programming language which uses compiler directives for parallelisation.

Finally, we will also investigate how the iteration contracts for the verifier and parallelisation pragmas for the compiler can support each other. We believe this support can work in both ways. First of all, the parallelising compiler can use verified annotations to know about dependences without analysing the code itself. In particular, the PENCIL language has a feature, called *function summaries*, that allows the programmer to tell the compiler which memory locations are written and/or read by a function by writing a fake function that assigns to the writable locations and reads from the readable locations. Such summaries are easily extracted from specifications, and thus in this way specifications can help to produce better code. Conversely, if the compiler performs an analysis then it could emit its findings as a specification template for the code, from which a complete specification can be derived.

References

[1] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters & A. F. Donaldson (2013): *PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs*. *CoRR* abs/1302.5586. Available at `http://arxiv.org/abs/1302.5586`.

[2] S. Blom & M. Huisman (2014): *The VerCors Tool for Verification of Concurrent Programs*. In: *FM 2014: Formal Methods, Lecture Notes in Computer Science* 8442, Springer, pp. 127–131, doi:10.1007/978-3-319-06410-9_9.

[3] S. Blom, M. Huisman & M. Mihelčić (2013): *Specification and verification of GPGPU programs*. *Science of Computer Programming*, doi:10.1016/j.scico.2014.03.013.

[4] R. Bornat, C. Calcagno, P.W. O'Hearn & M.J. Parkinson (2005): *Permission accounting in separation logic*. In: *POPL*, pp. 259–270, doi:10.1145/1040305.1040327.

[5] J. Boyland (2003): *Checking Interference with Fractional Permissions*. In: *Static Analysis Symposium, LNCS* 2694, Springer, pp. 55–72, doi:10.1007/3-540-44898-5_4.

[6] L. Dagum & R. Menon (1998): *OpenMP: an industry standard API for shared-memory programming*. *Computational Science & Engineering, IEEE* 5(1), pp. 46–55, doi:10.1109/99.660313.

[7] E.C.R. Hehner (2005): *Specified Blocks*. In: *VSTTE*, pp. 384–391, doi:10.1007/978-3-540-69149-5_41.

[8] P. W. O'Hearn (2007): *Resources, concurrency and local reasoning*. *Theoretical Computer Science* 375(1–3), pp. 271–307, doi:10.1016/j.tcs.2006.12.035.

[9] J.C. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures*. In: *Logic in Computer Science*, IEEE Computer Society, pp. 55–74, doi:10.1109/LICS.2002.1029817.

[10] H. Tuch, G. Klein & M. Norrish (2007): *Types, bytes, and separation logic*. In: *POPL*, pp. 97–108, doi:10.1145/1190216.1190234.