Memory access classification for vertical task parallelism

Jens Gustedt, Maxime Mogé

Memory access classification for vertical task parallelism

Jens Gustedt, Maxime Mogé

RESEARCH REPORT
N° 9182
June 2018
Project-Team Camus

ISSN 0249-6399 ISRN INRIA/RR--9182--FR+ENG

Memory access classification for vertical task parallelism

Jens Gustedt, Maxime Mogé

Project-Team Camus

Research Report n° 9182 — June 2018 — 20 pages

Abstract: We present a paradigm and implementation of a parallel control flow model for algorithmic patterns of two nested loops; an outer *iteration* loop and an inner *data traversal* loop. It is centered around memory access patterns. Other than dataflow programming it emphasizes on upholding the sequential modification order of each data object. As a consequence the visible side effects on any object can be guaranteed to be identical to a sequential execution. Thus the set of optimizations that are performed are compatible with C's abstract state machine and compilers could perform them, in principle, automatically and unobserved. We present two separate implementations of this model. The first in `C++` uses overloading of the `operator[]` to instrument the memory accesses. The second in `Modular C` uses annotations and code transformations for the two nested loops. Thereby the code inside the loops may stay as close as possible to the original code such that optimization of that code is not impacted unnecessarily. These implementations show promising results for appropriate benchmarks from `polybench` and `rodinia`.

Key-words: automatic parallelization, iterative algorithms, parallel control flow, memory access classification

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Classification d'accès mémoire pour la parallélisation verticale en tâches

Résumé : Nous présentons un paradigme et une implémentation d'un modèle de flot de contrôle parallèle pour des motifs algorithmiques de deux nids de boucles imbriqués; une boucle externe d'itération et une boucle interne de parcours de données. Ce paradigme est basé sur les motifs d'accès mémoire. Contrairement à la programmation par flot de données, il assure l'ordre séquentiel des modifications de chaque objet. Par conséquent, les effets de bords visibles sur tous les objets sont garantis identiques à une exécution séquentielle. Ainsi l'ensemble des optimisations effectuées sont compatibles avec la machine abstraite d'état de C, et les compilateurs peuvent les effectuer, en principe, de façon automatique et non-observée. Nous présentons deux implémentations distinctes de ce modèle. La premiere, effectuée en `C++`, utilise la surcharge de l'opérateur `[]` pour instrumenter les accès mémoire. La seconde, effectuée en *Modular C*, utilise des annotations et transformations de code pour les deux boucles imbriquées. Ainsi le code à l'intérieur des boucles peut rester aussi proche que possible de l'original pour ne pas impacter inutilement son optimisation. Ces implémentations montrent des résultats prometteurs pour des codes de référence tirés de `polybench` et `rodinia`.

Mots-clés : parallélisation automatique, algorithmes itératifs, flot de contrôle parallèle, classification d'accès mémoire

1 Introduction and overview

Race conditions are one of the principle challenges of automatic parallelization, and over last decades a lot of effort has been put into understanding and mastering them, in particular into their detection in faulty executions, see `e.g` [2, 11, 15]. Another direction of research has been to attempt to provide a race free task schedule once that a dependency graph is given [13]. But generally, the need for a programmer to manually specify a dependency graph, even in the form a data flow graph is a burden that defeats automatic parallelization.

In our opinion, to be closer to current programming practice, detection of dependencies and thus possible parallelism should be based on the features that are present in every day's imperative programming languages. In these, dependency between program statements are usually not explicit but they are forced indirectly via data accesses when one statement that reads a data has to be scheduled after another statement that wrote it.

Therefore parallelization of loops with irregular access patterns requires a runtime data dependency analysis that is followed by a tool that ensures that dependencies are enforced. There are currently mainly two types of tools: speculative and scheduling based.

Speculative techniques such as LRPD [5] identify the dependencies during the execution of the loop. The loop is run in parallel as a `doall` and followed by a test if the computations are correct. If not, a rollback to the previous correct state is performed and a sequential execution of the loop is carried out. For certain types of data accesses, namely affine or nearly affine functions over loop counts, polyhedral models can be used to provide race-free static task parallelism [3], and these approaches have also recently be used successfully to build runtime systems that are able to rollback execution if access violations occured [10]. For programs with low effective parallelism, this can lead to a significant overhead due to rollback followed by sequential execution when many conflicts occur between threads.

Scheduling based methods preprocess the loop to compute a dependency graph at runtime and use this information to do a parallel scheduling. The Inspector-Executor model [12, 14] consists of three phases: inspection, scheduling and execution. First the program is instrumented to explicitely compute a dependency graph. Then a parallel scheduling of the iterations is derived, and then the iterations are run in parallel in wavefronts, using synchronization between consecutive wavefronts. Huang et al. [1] propose a scheduling based method that does not explicitely compute the dependency graph, but instead overlaps the inspection and scheduling. A dedicated scheduling thread dynamically ensures that there is no conflict between threads, and allows cross invocation parallelism. However this method still needs an inspector that is run at the beginning of each iteration. The limitations of these methods are inherent to the Inspector-Executor model: it needs one inspector per iteration. In the case of a cyclic dependency between data and computation of shared addresses the inspector is basically the whole loop body.

In contrast to that, we will not restrict ourselves to a specific fine grained access projection model, but instead focus on certain type of applications, namely those that repeat the same data access pattern (coined data traversal) over a set of iterations, Section 2. This is a pattern that is found widely in the field, *e.g* many iterative algorithms that traverse matrices or geometric objects fall into that category. Using the assumption of a constant data access pattern, we detect dependencies and derive an implicit scheduling at runtime using Ordered Read-Write Locks (ORWL) [4] during some initial iterations. After this instrumentation phase, we do not need to check any condition or recompute dependencies, thus eliminating the drawbacks of the Inspector-Executor model.

As a basis for our argumentation, here, we will use C's model of *side effects* on data [9]. C has an *abstract state machine* that can be used to describe the effects of any valid program and compilers are allowed to perform all optimizations that respect that machine model. Our present work emphasizes on the fact that our parallelization fully respects the computation in that machine model. A fine grained (theoretical) model is presented that guarantees that all computational results are *exactly* as they would have been produced by the originating sequential program, Section 3. Based on [4] we are able to prove that our parallel execution model is correct, fair and deadlock free.

Being much too fine-grained for practical utilization, we have to coarsen our model by grouping programming steps into "meta-steps" and by classifying objects into "meta-objects", Section 4. We are able to prove that by doing so the good properties of our model are maintained, and present two different strategies that can be used for a memory access classification.

Our approach has been implemented with two different approaches, Section 5. The first uses `C++` and its ability to overload the `operator[]`. By that we are able to dynamically instrument the access pattern of complex code without having to rely on the programmer. A second implementation in *Modular C* [7] provides #pragma annotations for a code transformation that shows to be more efficient, but that needs manual annotations of the data accesses. Both implementations are tested with a set of benchmarks from the polybench and the rodinia collections, Section 6, and show very satisfying speedups.

2 Iterations over data transversals

In this work we restrict ourselves to a specific framework, namely programs that are dominated by two levels of loops, see Fig. 1.

1. An outer loop that we call *iteration*, *e.g*, an iteration over a simulated time or an iterative approximation of some numerical quantity.
2. One or several inner loops that we call *data traversal tasks* or just *tasks*, *e.g* loops that visit all elements of a matrix or that iterate over a geometric domain such as the facets of a polyhedral object description.

Our main assumption for data accesses is the following:

The data access pattern does not depend on the iteration.

Fig. 1. Loop exchange for vertical task parallelism

```
for (i in iteration) {
    // task T^0
    for (p in first data traversal) {
        // do steps of first task
        T^0(i, p);
    }


    // task T^1
    for (p in second data traversal) {
        // do steps of second task
        T^1(i, p);
    }

}
```

⟹

```
do parallel {
  for (p in first data traversal) parallel {
    for (i in iteration) sequential {
      /* insert some miraculous data
         coherence enforcement */
      T^0(i,p);
    }
  }
  for (p in second data traversal) parallel {
    for (i in iteration) sequential {
      /* insert some miraculous
         data coherence enforcement */
      T^1(i,p);
    }
  }
}
```

That is, we assume that each outer iteration visits the same data in the same order as the previous ones, *e.g* a traversal of a matrix or of geometry elements would be done in the same order in each iteration. That does not mean that a task can't use the iteration variable for its computation, only that its data access should not depend on it.

The inner part of a data traversal task t is called a `step`, denoted by $T^t(i, p)$, characterized by the triple (i, t, p), where i is the actual iteration, t is the task and p is the actual position in the traversal. That is, we assume that the program identifies code segments that perform these steps, and that the information about the current iteration, task and position in the traversal is available.

Steps are *sequenced*, that is there is a linear order in which the steps are executed by the program. This order corresponds to the lexicographic ordering of the indices (i, t, p).

Listing 1 illustrates what we have in mind here. It shows an idealized code excerpt of rodinia's `hotspot3D` benchmark, where we use a split directive (pragma) and a special task :: for construct to identify a data traversal task. A step is then the entire code inside the {}. It has access to the iteration variable i and the traversal position z.

Another feature of that implementation is also shown, namely data accesses are instrumented explicitly by means of directives that proclaim that the step may modify (lvalue) or just read (rvalue) a specific data.

2.1 Execution model: loop exchange and parallelism

In general, we suppose that a sequential program P has the following form:

```
for (all iterations i)
  for (all tasks t)
     for (all positions p in T^t data traversal) T^t(i, p);
```

Our goal is to improve the performance of P by parallelizing it. The main idea is to automatically perform a loop exchangetransformation as exemplified in Fig. 1. After the exchange the outer loops are in fact over the positions p of the

Listing 1. annotation of a data traversal loop of rodinia's `hotspot3D` benchmark

```
for (size_t i = 0; i < numiter; ++i) {
# pragma CMOD insert mctask split
  task:for(size_t z = 0; z < nz; z++) {
    size_t z0 = ... ; size_t z1 = ... ;
    /* insert some miraculous data coherence enforcement */
#   pragma CMOD insert mctask lvalue = tOut[z]
#   pragma CMOD insert mctask rvalue = tIn[z], tIn[z0], tIn[z1]
    for(size_t y = 0; y < ny; y++) {
      size_t y0 = ... ; size_t y1 = ... ;
      for(size_t x = 0; x < nx; x++) {
        size_t x0 = ... ; size_t x1 = ... ;
        /* do the computation */
        tOut[z][y][x] = tIn[z][y][x]*cc + tIn[z][y0][x]*cn + ... ;
      }
    }
  }
  // same data traversal, but inverting roles of tOut and tIn
  ...
}
```

data traversals and (at least conceptually) an independent thread is launched for each such position. Inside such a thread, the iterations are performed over the individual steps that are all associated to the same position p.

As remarked in the comments, data races could occur when two inner steps would access the same data concurrently. Therefore we will have to develop a mechanism that helps us to detect when one step has to wait for another one because of such a concurrent access.

As already implicit in the `hotspot3D` example, our notion of dependency of steps is not an abstract model of data or control flow, but directly deduced from access to objects of the program. It directly follows C's notion of *side effects* [9], that is, of modifications effected to data. Step S_1 *directly depends* on step S_0 if

- S_1 is sequenced after S_0 and
- S_1 reads or writes a data object o that S_0 has written.

By that, direct dependency between two steps can never be induced by only read accesses to a common data.

Step S_n (indirectly) *depends* on step S_0 if there is a direct dependency chain between them, that is, if there is a, possibly empty, sequence $S_{n-1}, \ldots, S_1$ such that S_j directly depends on S_{j-1} for all $0 < j \leq n$.

Dependence in that sense establishes a *partial ordering* $\wp$ the *dependency relation* on the steps that are executed by a program. Two steps S_0 and S_1 are *independent* if neither of them depends on the other and if so, they can be executed concurrently without introducing a *race condition*. Observe that two independent steps may be from the same or different tasks, iterations and traversal positions.

The notions of dependence and independence easily extend to sets of steps $\mathcal{S}_0$ and $\mathcal{S}_1$. In fact, $\mathcal{S}_0$ depends on $\mathcal{S}_1$ if there is a $S_0 \in \mathcal{S}_0$ that depends on $S_1 \in \mathcal{S}_1$; $\mathcal{S}_0$ and $\mathcal{S}_1$ are independent if all pairs $S_0 \in \mathcal{S}_0$ and $S_1 \in \mathcal{S}_1$ are independent. Observe that here the dependency relation between sets of steps may have cycles if we are not careful enough in choosing such sets.

2.2 Parallelizations

Fig. 2 shows an example of the initial section of a direct dependency graph and different scheduling strategies that have been applied. In all four examples, lines corresponds to concurrent executions of steps at a given time. Columns visualize all steps that deal with the same position in their respective data traversal. For the sake of the example, we make the simplified assumption that all steps have the same execution time.

Sequential execution. A sequential scheduling of the graph is shown in Fig. 2(a). Here, all steps are strictly executed in iteration and traversal order. *E.g* first we schedule all steps $T^0(0,.)$ of task T^0. Among these we respect the traversal order, that is we execute first $T^0(0,0)$, then $T^0(0,1)$, $T^0(0,2)$ and $T^0(0,3)$. The rectangle in the figure shows the pattern of the steady state, that is the pattern of the graph that is repeated over and over again until the final iteration.

Horizontal parallel execution is similar to what OpenMp would do with a for loop that is prefixed with a `parallel for` directive. But without keeping track of dependencies, OpenMp would allow all steps from the same task to run in parallel, and thus lead to incorrect results.

In fact, introducing horizontal parallelism has several problems that are not easily solved with static scheduling

- how about horizontal data dependencies?
- how about cache accesses?
- how about execution time variations?

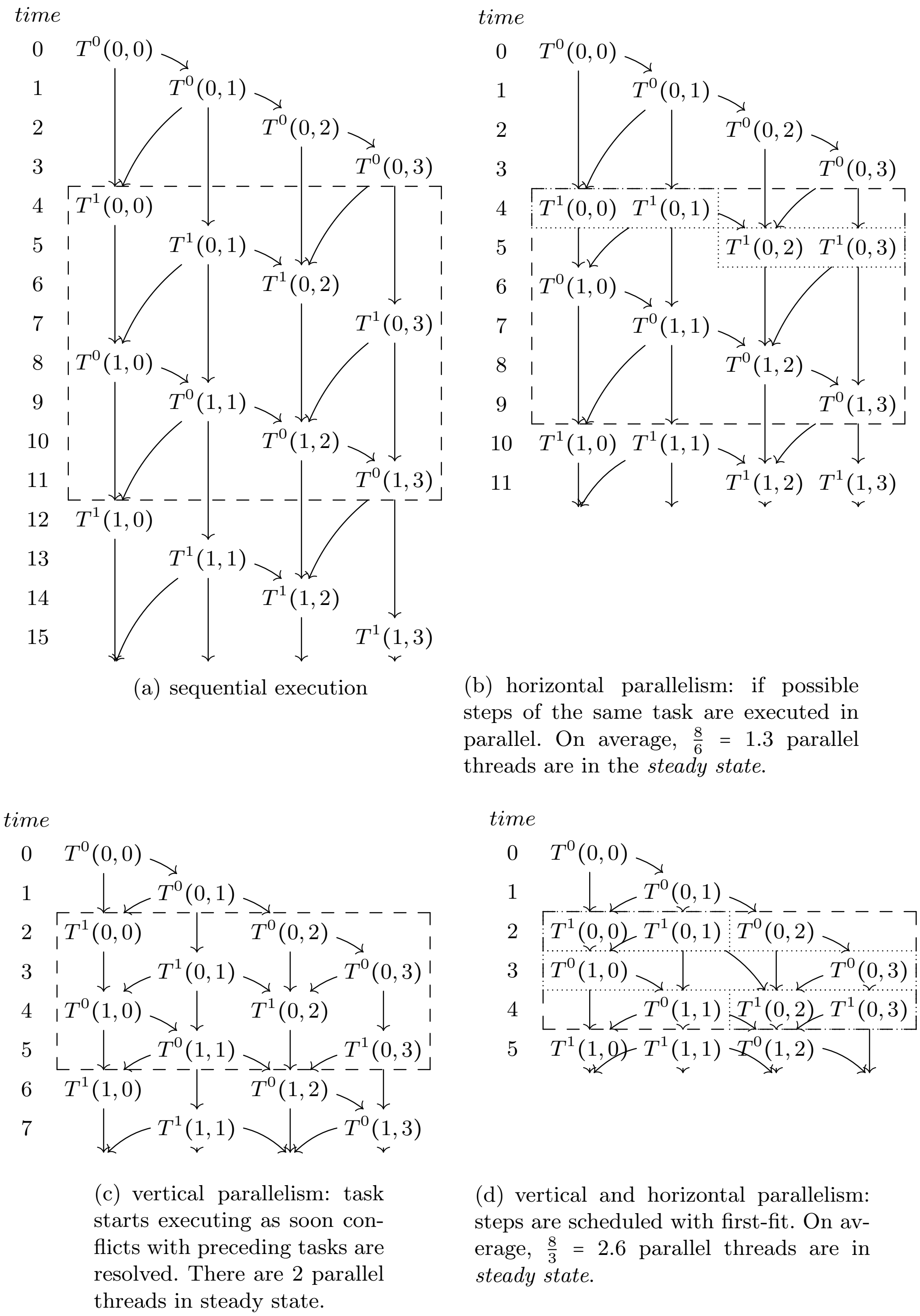
Fig. 2(b) shows that our example only allows for a limited horizontal parallelization that respects dependencies: the small rectangles inside the steady state visualize the steps from the same task that can be parallelized. We see that we can place our 8 steps in 6 time units, so on average we have $\frac{8}{6} = 1.3$ parallel threads in the *steady state.*

Vertical parallel execution is illustrated in Fig. 2(c). Here, we start the steps as early as possible under the constraint to still execute the steps of the same task and iteration in order. In our example, we see that at time 2, step $T^1(0,0)$ already starts in parallel to step $T^0(0,2)$. Generally, for all $i > 0$, the four types of sets of steps

$$\begin{gathered}\left\{T^0(i,0), T^1(i-1,2)\right\} \ \left\{T^0(i,1), T^1(i-1,3)\right\} \\ \left\{T^0(i,2), T^1(i,0)\right\} \ \left\{T^0(i,3), T^1(i,1)\right\}\end{gathered}$$

are always scheduled in parallel. Thus there are 2 parallel threads running at all times in the *steady state.*

Vertical and horizontal parallel execution as a combination of both parallel modes, shows the best available parallelism, see Fig. 2(d). Here it not only allows to run steps of the same iteration in parallel (*e.g* $T^1(0,0)$ and $T^1(0,1)$) but also steps of the same task but from different iterations, *e.g* $T^0(0,3)$ and $T^0(1,0)$. Or



(a) sequential execution

(b) horizontal parallelism: if possible steps of the same task are executed in parallel. On average, $\frac{8}{6} = 1.3$ parallel threads are in the *steady state*.

(c) vertical parallelism: task starts executing as soon conflicts with preceding tasks are resolved. There are 2 parallel threads in steady state.

(d) vertical and horizontal parallelism: steps are scheduled with first-fit. On average, $\frac{8}{3} = 2.6$ parallel threads are in *steady state*.

Fig. 2. Schedulings of the direct dependency graphs of steps. Arrows represent data dependencies, numbers on the left represent execution times of the steps in their line. Rectangles illustrate patterns of the steady state.

stated differently, a new iteration of a task may already start before the previous has completely finished. On average we have $\frac{8}{3} = 2.6$ parallel threads in *steady state*.

3 A fine grained execution model

We aim to develop an execution model that is compatible with C's abstract machine in a way that the parallelizations that we propose do not change the observable state of a program.

Observable changes in the states of that abstract machine occur through so-called *side effects*. By neglecting IO and similar external events for the moment, the interesting side effects in our context are modifications of the data objects that a program manipulates. Such manipulations occur through operations (such as assignment $=$ or increment $++$) or C library functions (such as memcpy).

For each object o in an execution of a program the C standard requires that accesses to o are properly *sequenced*, that is, that it can be deduced from the program structure if

- a write operation A on o provides the value for a read operation B of o;
- two write operation A and B on o are not separated by another write operation.

Read-only operations in turn are not necessarily expected to be properly sequenced. *E.g* the two evaluations of i in a function call $f(i,i)$ may be executed in any order.

For our context here, we can assume that an access A to object o is characterized by a triple (i,t,p) for step $T^t(i,p)$, and, if it is a write access, by the new value that is stored in o at the end of step $T^t(i,p)$. In particular, we don't distinguish if a step $T^t(i,p)$ does several accesses to the same object o and whether read and write access occur inside the same step.

The *modification order* $M(o) = M_0, M_1, \ldots$ is the precise list of changes that are applied to an object o. The *access order* $L(o)$ is $M(o)$ together with the read accesses to o, that is where we insert the read accesses in sequence order between write accesses:

$$L(o) = W_0, R_0^0, R_0^1, \ldots, W_1, R_0^0, R_0^1, \ldots$$

Since we are assuming that memory accesses do not depend on the iteration, $L(o)$ is of the form

$$L(o) = \hat{L}(o,0), \hat{L}(o,1), \ldots$$

where $\hat{L}(o,x)$ is a predetermined list $(x,t_0,p_0),(x,t_1,p_1),\ldots$ where we textually substitute x by i for each iteration i.

By changing the perspective from data objects to steps, for each step $T^t(i,p)$ we can now identify the objects $o = o_0, o_1, \ldots$ that it accesses as well as the list position $\ell(i,t,p,o)$ in $\hat{L}(o,i)$ where this access occurs.

3.1 Fifo execution

The lists that we defined in the previous section can now be used to establish a parallel execution model for the program steps, that takes care of the coherence requirements of Fig. 1.

For each object o we establish a FIFO data structure F(o) that follows $L(o)$ and for which we define the following operations:

- init(F(o)) initializes the FIFO position to the start of $L(o)$.
- release(F(o), i, t, p) announces that iteration i of task t at position p has ended.
- acquire(F(o), i, t, p) blocks step $T^t(i,p)$
  - if $T^t(i,p)$ is a write access until all previous accesses to position (i,t,p) in $L(o)$ have been released;
  - if $T^t(i,p)$ is a read-only access until the last write access previous to position (i,t,p) in $L(o)$ has been released.

Observe that the conditions for acquire ensure that each write access can be performed exclusively, and that all read-only accesses that follow the same write access can be honored concurrently.

Now we can augment each step of the program such that it is protected from race conditions by acquiring all FIFOs for all accessed objects and by releasing them thereafter. A *protected step* `Prot`$\langle\!\langle T^t \rangle\!\rangle(i,p)$ derived from $T^t(i,p)$ is:

```
Prot⟪$T^t$⟫(i,p) {
  for (all objects o accessed by $T^t$(i, p))
     acquire(F(o), i, p);
  $T^t$(i,p);
  for (all objects o accessed by $T^t$(i, p))
     release(F(o), i, p);
}
```

Clearly, substituting `Prot`$\langle\!\langle T^t \rangle\!\rangle$ for T^t for all t in the whole program does not change its visible side effects: all acquire operation would only be issued after all previous positions in $L(o)$ are released, and all operations T^t themselves would still appear in sequence order.

We can now proceed to our final goal, the loop exchange. A *step iteration* `StepIt`$\langle\!\langle T^t \rangle\!\rangle$ performs the iteration loop

```
StepIt⟪$T^t$⟫(p) {
  for (i in iteration) Prot⟪$T^t$⟫(i,p);
}
```

Now our parallelized program can be formulated as

```
for (all tasks t) do parallel {
  for (all positions p in $T^t$ data traversal) do parallel
    StepIt⟪$T^t$⟫(p);
}
```

Theorem 1. *The parallelized program is well defined and produces the same side effects as the original program.*

A proof of this theorem follows from the fact that the dependency relation $\wp$ is acyclic and from the theory developed in [4] for Ordered Read-Write Locks (ORWL). This formalizes a model for autonomous execution of tasks or processes that manage their shared resources by means of cyclic FIFOs that are attached to data resources. The ORWL model here can guarantee correctness, equity and liveness of execution. It has a reference implementation that has first been described in [8], and that has proven to be very efficient for coarse grained parallelism.

Expected efficiency: Generally for the framework that we described so far, we are unfortunately not in a coarse grained setting. We have made no restriction about the code that is found inside a step, and for many sensible programs that we want to cover a step could just be several instructions. In particular, we have to resolve the following problems:

- Data traversals can have thousands, millions or billions of steps. Thus our parallel program would have to run as many threads or processes, exceeding the possible number of threads for current architectures.
- A protected step $\texttt{Prot}\langle\!\langle T^t \rangle\!\rangle$ introduces runtime overhead for each memory access, namely calls to acquire and release the corresponding FIFO. Therefore we'd have to expect the cost of a step at least to triple by that procedure.
- Additionally, we also would have to maintain a FIFO for each data object. So the memory footprint of the application would at least double.
- There is an important overhead to initiate FIFOs for all objects.

4 Coarsening

To be useful as such, our step-model of is too fine grained. We have to coarsen it with respect to two aspects. First, we have to ensure that the number of steps (and thus the number of threads) does not exceed the usual limits of existing platforms. Second, we have to avoid that each data object gives rise to its own FIFO, but must regroup objects into classes such that the three types of overhead that we describe above are limited.

4.1 Groups of steps

To reduce the number of steps, we transform the original program by grouping steps:

```
for (i in iteration)
  for (t in tasks)
    for (group g of positions in $T^t$ traversal)
      for (p in g) $T^t$(iteration, p);
```

The easiest strategy to form such groups of positions is to divide the positions $p_0, p_1, \ldots, p_r$ in the traversal into a fixed a amount m of intervals of successive positions.

$$g_k = (p_{a_k}, \ldots, p_{b_k}), \text{with } a_k = \frac{kr}{m},\ b_k = \frac{(k+1)r}{m}.$$

This allows to rewrite a group g of steps into a "meta-step" $\texttt{MetaStep}_m\langle\!\langle T^t \rangle\!\rangle$:

```
MetaStep_m ⟪T^t⟫(i, g) {
    for (p in g) T^t(i, p);
}
```

And thus we can reformulate the program as

```
for (i in iteration)
  for (t in tasks)
    for (g of meta-positions in MetaStep_m ⟪T^t⟫ traversal)
      MetaStep_m ⟪T^t⟫(i, g);
```

As a consequence this rewrite of the program fits exactly the same model as the original program; it consists of an iteration loop that is nested with data traversal loops such that the data access pattern does not change with the iterations. In particular we have

Lemma 1. *The induced dependency relation $\wp_m$ between meta-steps is acyclic.*

With that grouping strategy the total amount of threads can be limited as a function of m, the number of groups that we fix per task.

4.2 Classes of objects

To bundle the management of FIFOs for several data objects at a time we partition the set of data elements D into m' *meta-objects* $C = \{c_k \subset D\}$.

$$D = \dot{\bigcup_{k<m'}} c_k$$

Such a classification of objects into meta-objects gives rise to an induced dependency relation $\wp_m|_C$ between meta-steps in a obvious way. We have

Lemma 2. *$\wp_m|_C$ is an partial order extension of $\wp_m$.*

and in particular

Corollary 1. *$\wp_m|_C$ is an acyclic, irreflexive and transitive relation.*

It is also easy to see that the access list $L(c)$ of a meta-object $c \in C$ is the lexicographic merge of all lists $L(o)$, with $o \in c$. That is, we consider that an meta-object is accessed if any of its members is accessed. We then attribute one FIFO F(c) to each meta-object $c \in C$ and define the protected meta-step analogously as before.

```
MetaProt⟪ MetaStep⟪T^t⟫⟫(i,g) {
  for (c ∈ C touched by MetaStep⟪T^t⟫(i,g)) acquire(F(c), i, g);
  MetaStepMetaStep⟪T^t⟫(i,p);
  for (c ∈ C touched by MetaStep⟪T^t⟫(i,g)) release(F(c), i, g);
}
```

Clearly, using the resulting scheduling is more restrictive, but, because of Corollary 1 its validity is not impacted.

Theorem 2. *The parallelized program that is coarsened with respect to a step grouping of m and an object classifier C is well defined and produces the same side effects as the original program.*

To assemble the meta-objects we investigated two strategies. Objects such as vectors or matrices for which sub-objects occupy contiguous memory can be classified in a similar way as we grouped steps by using access intervals. If this is not the case or if the access pattern is too irregular, we have to keep track of individual accesses.

Classify groups of contiguous memory accesses in ranges. Suppose the main data access is through a *vector* V[j] for $j = 0, ..., N - 1$ and that the access to that vector is regular such that the progression of the data traversal corresponds to a linear access to the vector.

To classify the access to V we can then fix a chunk size M and define the meta-objects as

$$\begin{aligned} \mathrm{contig}(\mathrm{V[k]}) &= \lfloor k/M \rfloor \\ \mathrm{Contig(j)} &= \left\{ \mathrm{V[k]} \mid \mathrm{contig}(\mathrm{V[k]}) = j \right\} \end{aligned}$$

The advantage of Contig(.) is that the membership for any data object to its class is easily computed. Therefore the overhead that is needed to establish the FIFOs for the meta-steps is relatively small.

Classify random memory access by ownership. Vector classification does not work if data access is through lists, e.g if we do a data traversal over a complicated data structure or if our access pattern follows some random order. In these cases for each object o we can designate the *owner* (owning step) and use that owner to define a partition of the set of objects:

$$\begin{aligned} \mathrm{owner(o)} &= \text{first mega-step } g \text{ that writes to } o \text{ in the sequence ordering} \\ \mathrm{Owner(g)} &= \left\{ o \mid o \text{ object, } \mathrm{owner(o)} = g \right\} \end{aligned}$$

We may prepare the computation of Owner(g) by first running several sequential iterations:

- In a first *counting* phase we determine the number of objects that will be accessed by all mega-steps and dimensions a hash table with the appropriate size.
- A second *owning* phase notes the first write access for all objects in the hash table.
- A third *attribution* phase determines the owner of each object. Now the data objects are partitionned by their access in the meta-step. We have at most as many meta-objects as there are meta-steps.

After this initial setup of the meta-objects, the remaining iterations can be done in parallel.

Compared to Contig, Owner classification has a startup that is is more expensive. On the other hand, no assumption is made about access order or about a specific organization of the data.

5 Implementations

We have implemented the Owner classifier with two different approaches. The first uses `C++` and overloading of the `operator[]` and the second uses *Modular C* [7] and #pragma directives for a source-to-source transformation.

Both build on the `EiLck` library for the FIFO functionality. `EiLck` is a standalone library written in *Modular C* for Linux. It implements cyclic access FIFOs on the basis of Linux' `futex` system call that efficiently combines atomic access with a scheduling queue in the system kernel, [6]. All scheduling of tasks is done implicitly via these FIFOs; `EiLck` executes the application without dedicated management threads. `EiLck` is compatible with different thread libraries (C11, C++11, POSIX, OpenMp) and is interfaced to C and `C++`. It efficiently avoids overhead for busy waiting if a FIFO access is congested and provides a minimal run time overhead in case it is not.

5.1 C++: Operator overloading and explicit thread creation

Besides the use of `Eilck`, our `C++` implementation is based on standard tools, mainly `operator[]` and the explicit creation of threads for the execution of the tasks. `operator[]` is used to instrument every single memory access to a set of vectors that the programmer designates as being shared between steps. In addition, the programmer also identifies the iteration loop and registers the data traversal tasks as functions that are executed in separate C++11 threads.

Then, during execution, data traversals are grouped and protected automatically. For the grouping of the steps into meta-steps, each task counts its number of calls to `operator[]` and uses this count as an indication of the traversal position. For the classification of the memory accesses we use a dynamic `std::map` to register the owners and then classify each access according to that owner. A set of FIFOs is constructed from this, and used in subsequent iterations.

This approach has the advantage that we don't even have to identify steps. Any code that supplies sufficient amount of work can automatically be split by this `operator[]` approach into steps and meta-steps, respectively. The programmer only needs to designate the iteration loop, a set of tasks (data traversal) and the data vectors they share. The modifications needed for this parallelization are simple, as shown in Listing 2 for the `hotspot3D` benchmark. The approach has several disadvantages, though:

- All meta-steps of a task are executed in the same thread. Parallelism is only achieved between meta-steps that are part of different tasks.
- Control operations by `operator[]` interleave the computation. This breaks local optimization opportunities, *e.g* vectorization.
- The `operator[]` needs a thread local variable for the position in the traversal.Access to such variables is generally slow.

Listing 2. Parallelization of rodinia's `hotspot3D` benchmark using operator overloading - the code was simplified for clarity (e.g. omission of template parameters).

```
void computeTask(VectorWrapper tOut_w, VectorWrapper tIn_w, ...) {  // encapsulate computation in a separate function
  for(size_t z = 0; z < nz; z++) {
    size_t z0 = ... ; size_t z1 = ... ;
    for(size_t y = 0; y < ny; y++) {
      size_t y0 = ... ; size_t y1 = ... ;
      for(size_t x = 0; x < nx; x++) {
        size_t x0 = ... ; size_t x1 = ... ;
        tOut_w[z][y][x] = tIn_w[z][y][x]*cc + tIn_w[z][y0][x]*cn + ... ;
  } } }
}
VectorWrapper tOut_w(tOut), tIn_w(tIn);  // overload operator[] using a vector wrapper
IterationLoopIdentifier loopId = ...;    // identify an iteration loop
for (size_t i = 0; i < numiter; ++i) {
  beginLoopIteration(loopId);
  threadPool->submit(new TraversalTask(computeTask,tOut_w,tIn_w));  // start task 1
  threadPool->submit(new TraversalTask(computeTask,tIn_w,tOut_w));  // start task 2
  endLoopIteration(loopId);
}
```

Listing 3. The complete paralellization of rodinia's `hotspot3D` benchmark. alternate and duplicate unroll the iteration loop four times such that four tasks can be generated.

```
#pragma CMOD insert mctask groups = 8
#pragma CMOD amend mctask iterate  = numiter
#pragma CMOD amend mctask duplicate = 2        // unroll for more parallelism
#pragma CMOD amend mctask alternate tOut tIn  // unroll to alternate buffers
# pragma CMOD insert mctask split
  task:for(size_t z = 0; z < nz; z++) {
    size_t z0 = ... ; size_t z1 = ... ;
    /* insert some miraculous data coherence enforcement */
#   pragma CMOD insert mctask lvalue = tOut[z]
#   pragma CMOD insert mctask rvalue = tIn[z], tIn[z0], tIn[z1]
    for(size_t y = 0; y < ny; y++) {
      size_t y0 = ... ; size_t y1 = ... ;
      for(size_t x = 0; x < nx; x++) {
        size_t x0 = ... ; size_t x1 = ... ;
        /* do the computation */
        tOut[z][y][x] = tIn[z][y][x]*cc + tIn[z][y0][x]*cn + ... ;
      }
    }
  }
#pragma CMOD done
#pragma CMOD done
#pragma CMOD done
```

5.2 Modular C: pragmas and source-to-source transformation

Listing 3 shows a complete parallelization of the `hotspot3D` benchmark using our *Modular C* implementation. Here we use two forms of *Modular C* directives. The first, insert,

```
#pragma CMOD insert mctask keyword [[ = ] arguments ]
    ... prefixed statement or block ...
```

is for a construct that precedes some statement or block and potentially changes its interpretation. The second, amend,

```
#pragma CMOD amend mctask keyword [[ = ] arguments ]
    ... code snippet to be rewritten ...
#pragma CMOD done
```

can be used to rewrite an entire code snippet as necessary. Here, both directives refer to an external script, mctask, that implements our parallelization.
The main features shown in that example are the following

- An iterate directive replaces the outer iteration loop.
- A split directive prefixes the data traversal.
- A task::for (or task::while) loop specifies the data traversal.
- Data access directives lvalue and rvalue indicate all memory accesses that could be subject to race conditions.

The iterate directive uses amend to transform the code of the iteration. It creates two modified copies of the code, one for an initial instrumentation phase (similar to the C++ version) and one for the steady state iterations. For the first phase, lvalue and rvalue are replaced by calls to appropriate access classification functions, for the second they are removed such that the steady-state iterations can use the original code directly.

The split directive together with the task::for ensures that the loop is interpreted as a single task. It groups the steps into meta-steps and ensures that the protection against race conditions is inserted between them. But in contrast to a steps directive, see below, it still only launches one thread per task, and we only will see vertical parallelism if we only use split.

The transformed parallel program can be quite effective because the inner data traversal is exactly as programmed originally. In particular, the program maintains the same optimization opportunities, *e.g* for vectorization. Also, compared to our C++ version, there is no need for thread local variables, the source-to-source transformation has enough possibilities to provide the necessary information . On the other hand, this implementation here requires that the structure of the program makes the iteration and data traversal loops apparent. *E.g* hiding the data traversal inside a function would not be possible.

Listing 3 also features two other directives that are be quite useful. With the innermost amend, the alternate directive implements an automatic duplication that alternates the roles of tOut and tIn. The next level of amend with a duplicate directive duplicates its inner part, again, such that at the end we have four copies of the inner part, giving rise to four different tasks.

In total we have $4 \times 8 = 32$ meta-steps. Here, because of the split directive, we have one thread for each task, so 4 tasks in total that are parallelized vertically. To gain also horizontal parallelism we can replace

```
#pragma CMOD insert mctask split
  task::for(size_t z = 0; z < nz; z++) {
     ...
  }
```

by

```
#pragma CMOD insert mctask steps = nz
  {
     size_t z = task::step();
     ...
  }
```

The steps directive creates a separate thread for each meta-step, so here this results in 32 threads in total.

6 Benchmarks

We tested our implementations with those benchmarks from the polybench and the rodinia series that fulfill our requirements, and inside a real life application, the SOFA framework.

The test platform is a Linux machine with an Intel Xeon E5-2650 v3 processor, 2 cpu sockets, 10 real cores (and 20 hyperthreaded cores) each. Although we did not expect much parallelism beyond a speedup factor of 2 or 3, we chose such a platform with relatively many cores to avoid an artificial bound on the parallelism. For the `C++` implementation, we adapted the code using `C++` vectors.

Fig. 3 shows the speedups that we obtained with these relatively simple tests, once we have reached the steady state. We compare the original sequential implementation `seq` to our implementations using *Modular C* with split and steps directives, and `C++`. For some tests we unroll the iteration loop (`2x` or `4x` factor) to increase the number of tasks. With *Modular C*, we see that all but one of the benchmarks achieve worthy speedup factors that range from 1.2 or 1.3 (`srad2D`), over 1.5 (`fdtd`), 1.7 (`adi` and `heat3d`), 2.2 (`hotspot3D`), to up to 4 for `seidel_2d`. Indeed, the latter parallelizes perfectly with the number of duplicates of the data traversal that we produce.

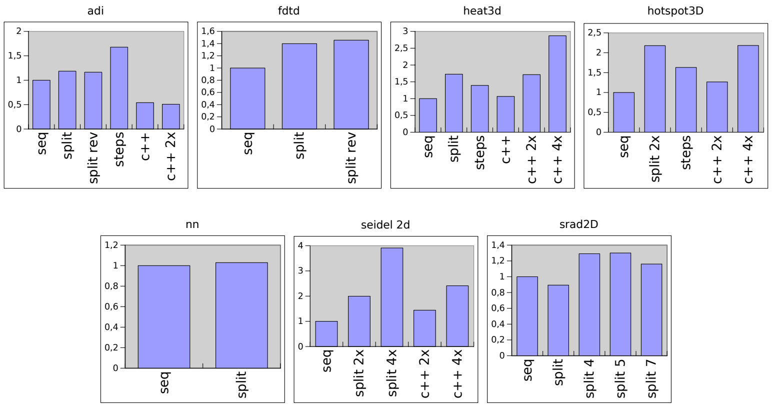


Fig. 3. Speedups of steady state iterations achieved for benchmarks from polybench and rodinia with the Modular C and the C++ implementations.

Most of these parallelizations are straight forward code annotations as we have seen them for the `hotspot3D` benchmark above. There is no clear winner between the split and steps directives; adding more threads is not always an advantage. The parallelization of `srad2D` is a little bit more involved than the others, because the original sequential code concentrates most of the computation in one of the traversals. So here a simple split even slows the execution down. But when we divide that traversal loop into several such that we obtain 4, 5 or 7 tasks in total, we observe a speedup of about 1.3.

Only the *nearest neighbor* (`nn`) benchmark from the rodinia suite detaches from this. It is completely dominated by the IO for reading the input data. The computation itself is negligible, so we can't gain much by parallelization. Nevertheless, we also don't lose anything, which shows the validity of our approach, where we are able to integrate the IO seamlessly into a data flow parallelism.

With the `C++` version, most of the tested benchmarks achieve worthy speedup, too, although lower than with *Modular C*. The overhead induced by the overloading and the inhibited compiler optimizations can seriously impact the performances (e.g. only 1.1 speedup for `heat3d` and even a slowdown for `adi`). However, a simple unrolling of the iteration allows us to create more tasks and gives a much better speedup (1.7 for `heat3d` with duplicated tasks, and up to 2.9 with 4x unrolling).

The cost of the startup steps is highly dependent on the data objects considered and the application. With default settings, the *Modular C* version uses fine grained data objects (cache lines) with a cost ranging from 2 to 120 sequential iterations. This granularity can be manually adapted to reduce this cost. The `C++` version, uses coarser data objects (rows of 2D matrices) with a cost of 3 sequential iterations for `seidel2D`, 12 for `heat3d`, 16 for `hotspot3D`.

We also tested our `C++` implementation on a large scale application, the SOFA framework, with interesting performance: we were able to maintain the runtime overhead induced by our parallelization mechanism low enough to get a significant speedup on a specific test simulation.

7 Conclusion and outlook

We have shown that a semi-automatic parallelization tool for a special class of iterative algorithm can be implemented. Provided that the data access pattern does not change in the iteration loop, execution is guaranteed to follow the sequential modification order for each data. Thereby we can *e.g* guarantee that numerical iterations lead to *exactly* the same results as sequential execution and that all proofs for convergence translate directly to the parallel version. We showed that our implementations leads to satisfying speedups for a variety of benchmarks.

For the moment our implementations both need intervention from the programmer. Both need an identification of the iterations and data traversals. Also, they are limited in their ability to instrument the memory access. The `C++` version has a certain runtime overhead for the overloaded `operator[]`. The *Modular C* version needs manual annotation of the accesses and has a high startup cost because of its use of fine grained data objects. We think that all these shortcomings can be overcome by implementing the memory access instrumentation directly in the compiler.

The need of identification could be circumvented by using a framework such as Apollo [10] that allows to speculate on a particular pattern, and then to rollback if the expected pattern was not respected.

References

[1] August, D.I., Huang, J., Beard, S.R., Johnson, N.P., Jablin, T.B.: Automatically exploiting cross-invocation parallelism using runtime information. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 1–11. CGO '13, IEEE Computer Society, Washington, DC, USA (2013). https://doi.org/10.1109/CGO.2013.6495001

[2] Banerjee, U., Bliss, B., Ma, Z., Petersen, P.: A theory of data race detection. In: PADTAD '06. pp. 69–78. ACM, New York, NY, USA (2006). https://doi.org/10.1145/1147403.1147416

[3] Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT '04. pp. 7–16. IEEE Computer Society, Washington, DC, USA (2004). https://doi.org/10.1109/PACT.2004.11

[4] Clauss, P.N., Gustedt, J.: Iterative Computations with Ordered Read-Write Locks. Journal of Parallel and Distributed Computing **70**(5), 496–504 (2010). https://doi.org/10.1016/j.jpdc.2009.09.002

[5] Dang, F., Yu, H., Rauchwerger, L.: The r-lrpd test: speculative parallelization of partially parallel loops. In: Proceedings 16th International Parallel and Distributed Processing Symposium. pp. 10 pp– (April 2002). https://doi.org/10.1109/IPDPS.2002.1015493

[6] Franke, H., Russell, R., Kirkwood, M.: Fuss, futexes and furwocks: Fast userlevel locking in Linux. In: Ottawa Linux Symposium (2002)

[7] Gustedt, J.: Modular C. Research Report RR-8751, INRIA (Jun 2015)

[8] Gustedt, J., Jeanvoine, E.: Relaxed Synchronization with Ordered Read-Write Locks. LNCS, vol. 7155, pp. 387–397. Springer, Bordeaux, France (May 2012)

[9] JTC1/SC22/WG14 (ed.): Programming languages - C. No. ISO/IEC 9899, ISO, cor. 1:2012 edn. (2011)

[10] Martinez Caamaño, J.M., et al.: Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. Concurrency and Computation: Practice and Experience **29** (2017)

[11] Matsubara, M., et al.: Model checking with program slicing based on variable dependence graph. In: First International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2012). p. 88 (2012)

[12] Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Transactions on Computers **40**(5), 603–612 (May 1991). https://doi.org/10.1109/12.88484

[13] Sánchez, C., et al.: On efficient distributed deadlock avoidance for real-time and embedded systems. In: IPDPS 2006. vol. 2006 (2006). https://doi.org/10.1109/IPDPS.2006.1639370

[14] Venkat, A., Mohammadi, M.S., Park, J., Rong, H., Barik, R., Strout, M.M., Hall, M.: Automating wavefront parallelization for sparse matrix computations. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 41:1–41:12. SC '16, IEEE Press, Piscataway, NJ, USA (2016)

[15] Wang, G., Matsubara, M.: Data race detection based on dependence analysis. In: Software Symposium 2017 (SS 2017). Software Engineers Association (SEA) (2017)

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399