Compiler/Run-Time Framework for Dynamic Data-Flow Parallelization of Tiled Programs

OSU Technical Report Number: OSU-CISRC-7/14-TR14
Revised: September 2, 2014

Martin Kong[1], Antoniu Pop[3], R. Govindarajan[4],
Louis-Noël Pouchet[2], Albert Cohen[5], P. Sadayappan[1]
[1]The Ohio State University, [2]University of California-Los Angeles
[1]{kongm,saday}@cse.ohio-state.edu, [2]pouchet@cs.ucla.edu,
[3]antoniu.pop@manchester.ac.uk, [4]govind@serc.iisc.in,
[5]albert.cohen@inria.fr

Task-parallel languages are increasingly popular. Many of them provide expressive mechanisms for inter-task synchronization. For example, OpenMP 4.0 will integrate data-driven execution semantics derived from the StarSs research language. Compared to the more restrictive data-parallel and fork-join concurrency models, the advanced features being introduced into task-parallel models in turn enable improved scalability through load balancing, memory latency hiding, mitigation of the pressure on memory bandwidth, and as a side effect, reduced power consumption.

In this paper, we develop a systematic approach to compile loop nests into concurrent, dynamically constructed graphs of dependent tasks. We propose a simple and effective heuristic that selects the most profitable parallelization idiom for every dependence type and communication pattern. This heuristic enables the extraction of inter-band parallelism (cross barrier parallelism) in a number of numerical computations that range from linear algebra to structured grids and image processing. The proposed static analysis and code generation alleviates the burden of a full-blown dependence resolver to track the readiness of tasks at run time. We evaluate our approach and algorithms in the PPCG compiler, targeting OpenStream, a representative data-flow task-parallel language with explicit inter-task dependences and a lightweight runtime. Experimental results demonstrate the effectiveness of the approach.

1. Introduction and Motivation

Loop tiling and thread-level parallelization are two critical optimizations to exploit multi-processor architectures with deep memory hierarchies [1]. When exposing coarse grain parallelism, the programmer and/or parallelization tool may rely on data parallelism and barrier synchronization, such as the `for` worksharing directive of OpenMP, but this strategy has two significant drawbacks:

- Barriers involve a global consensus, a costly operation on non-uniform memory architectures; it is more expensive than the resolution of point-to-point dependences unless the task dependence graph has a high degree (e.g., high fan-in reductions) [6].
- To implement wavefront parallelism in polyhedral frameworks one generally resorts to a form of loop skewing, exposing data parallelism in a wavefront-based parallel schedule

An alternative is to rely on more general, task-parallel patterns. This requires additional effort, both offline and at runtime. Dedicated control flow must spawn coarse-grain tasks, that must in turn be coordinated using the target language's constructs for the enforcement of point-to-point dependences between them. A runtime execution environment resolves these dependences and schedules the ready tasks to worker threads [20, 7, 6, 23].

In particular, runtimes which follow the data-flow model of execution and point-to-point synchronization do not involve any of the drawbacks of barrier-based parallelization patterns: tasks can execute as soon as the data becomes available (i.e., when dependences are satisfied) and lightweight scheduling heuristics exist to improve the locality of this data in higher levels of the memory hierarchy; no global consensus is required and relaxed memory consistency can be leveraged to avoid spurious communications; loop skewing is not always required, and wavefronts can be built dynamically without the need of an outer serial loop.

Loop transformations for the automatic extraction of data parallelism have flourished. Unfortunately, the landscape is much less explored in the area of task parallelism extraction, and in particular the mapping of tiled iteration domains to dependent tasks. This paper makes three key contributions:

Algorithmic We design a task parallelization scheme following a simple but effective heuristic to select the most profitable synchronization idiom to use. This scheme exposes concurrency and favors temporal reuse across distinct loop nests (a.k.a. dynamic fusion), and further partitions the iteration domain according to the input/output signatures of dependences. Thanks to this compile-time classification, much of the runtime effort to identify dependent tasks is eliminated, allowing for a very lightweight and scalable task-parallel runtime.

Compiler construction We implement the above algorithm in a state-of-the-art framework for affine scheduling and polyhedral code generation, targeting the OpenStream research language [23]. Unlike the majority of the task-parallel languages, OpenStream captures point-to-point dependences between tasks explicitly, reducing the work delegated to the runtime by making it independent of the number of waiting tasks.

Experimental We demonstrate strong performance benefits of task-level automatic parallelization over state of the art data-parallelizing compilers. These benefits derive from the elimination of synchronization barriers and from a better exploitation of temporal locality across tiles. We further characterize these benefits within and across tiled loop nests.

We illustrate these concepts on a motivating example in Sec. 2 and introduce background material in Sec. 3. We present our technique in detail in Sec. 4, and evaluate the combined compiler and runtime task-parallelization to demonstrate the performance benefits over a data-parallel execution in Sec. 5. Related work is discussed in Sec. 6.

2. Motivating Example

We use the Blur-Roberts kernel performing edge detection for noisy images in Fig. 1 as illustrating example, with N=4000 and using double precision floating point arithmetic. Fig. 2 shows the performance obtained when using Intel ICC 2013 as the compiler, using flags `-O3 -parallel -xhost`; PLuTo variants compiled with -O3 -openmp -xhost; task variant corresponds to *task-opt* (See Sec. 5.2). The original code peaks at 2.7 GFLOPS/sec on an AMD Opteron 6274 and 3.6 GFLOPS/sec on an Intel i7-2600, although by using half the number of cores. (See Tab.5.2 in Sec. 5 for complete description of machines used). This program is memory bound but contains significant data reuse potential, so tiling is useful to improve performance. We leverage the power of the PLuTo compiler [5] to tile simultaneously for coarse grained parallelism and locality, considering the two fusion heuristics "minfuse" (cut non-strongly connected components at each level) and "smartfuse" (fuse only loops of identical depth). For this example, the result of the third fusion heuristic of pluto, "maxfuse" (fuse as much as possible) gives an identical output code than the "smartfuse" heuristic.

```
for (i = 1; i < N - 1; i++)
  for (j = 1; j < N - 1; j++) {
S1: B[i][j] = (A[i][j] + A[i][j-1] + A[i][1+j] +
      A[1+i][j] + A[i-1][j] + A[i-1][j-1] +
      A[i-1][j+1] + A[i+1][j-1] + A[i+1][j+1])/8;
}

for (i = 1; i < N-2; i++)
  for (j = 2; j < N-1; j++) {
S2: A[i][j] = abs(B[i][j]-B[i+1][j-1]) +
      abs(B[i+1][j] - B[i][j-1]);
}
```

Figure 1: Blur-Roberts kernel

Proc cores	ref ICC	pluto min fuse	pluto smart fuse	our work
opt-1	1.25	0.4	0.7	0.9
opt-8	1.25	2.7	3.9	4.7
opt-16	1.25	2.0	0.7	6.8
i7-1	3.4	2.6	2.3	2.8
i7-2	4.2	3.6	4.0	5.4
i7-4	4.1	3.5	4.3	10.1

Figure 2: Blur-Roberts kernel performance in GFLOPS/sec for AMD Opteron 6274 and Intel i7-2600, on 1, half and all cores.

The ”minfuse” heuristic distributes all loops that do not live in the same strongly connected component (SCC). Thus, it tiles and parallelizes each loop nest independently, requiring 2 barriers as shown in Figure 3.

```
parfor (ti=0; ti < ubti; ti++)
  for (tj=0; tj < ubtj; tj++)
    { /* tile body of S1*/ }
/* barrier */
parfor (ti=0; ti < ubti; ti++)
  for (tj=0; tj <= ubtj; tj++)
    { /* tile body of S2*/ }
/* barrier */
```

Figure 3: Blur-Roberts Kernel produced with PLuTo minfuse

```
for (w = 0; w < wmax; w++) {
  parfor (ti=f1(w); ti < f2(w); ti++) {
    { /* tile body */ }
  /* barrier */
}
```

Figure 4: Blur-Roberts Kernel produced with PLuTo smartfuse

”Smartfuse” performs a complex sequence of transformations such as multidimensional retiming and peeling to enable the fusion of program statements under a common loop nest, and skewing is used for the wavefronting transformation to expose coarse-grain parallelism. The output exhibits an outermost sequential loop and a second outermost parallel loop followed by its implicit barrier (see sketch in Fig. 4). The complete code of Blur-Roberts tiled with PLuTo smartfuse can be found in Sec. A.

As we observed in Fig. 2, the performance does not increase linearly with the number of processors, instead it either reaches a plateau with the Intel i7 or drastically drops as in the Opteron's case.

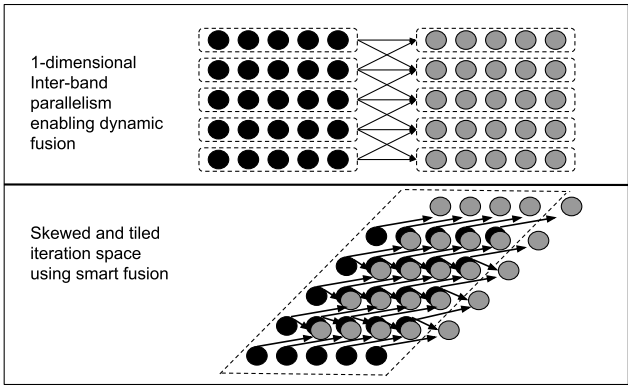

Figure 5: Illustration of benefit of task data-flow over static affine scheduling

Figure 5 illustrates the nature of the data dependences in the Blur-Roberts code and the parallelization options for static affine scheduling, as represented by state-of-the-art polyhedral compilers such as PLuTo, versus dynamic task data-flow. The top half of the figure shows the iteration spaces for an unfused form of the code, with the left square representing the first loop nest and the right square the second loop nest, along with 2D tiling of each loop nest, working on blocks of rows of the matrices. With PLuTo minfuse, a barrier is used between executions of tiles of the first and second loops. With smartfuse, the two loop nests are fused, but skewing is employed to expose coarse-grain parallelism using the parfor/barrier idiom supported in PluTo. After fusion, only wavefront parallelism is feasible with 2D tiling, with barriers between diagonal wavefronts in the tiled iteration space. Thus, there is a trade-off: with minfuse, the tiled execution of each loop nest is load-balanced, but inter-loop data reuse is not immediately feasible; with smartfuse, inter-loop data reuse is improved, but may lead to load imbalance due to the exploitation of wavefront parallelism. We remark that other tiling techniques such as split-tiling [13] or diamond-tiling [2] can enable better load-balance than wavefront-based rectangular tiling, it however still demands explicit barriers in the code between tile "phases". In this work, we focus exclusively on rectangular affine tiling.

With a task data-flow model, it is possible to get the best of both worlds: increased degree of task-level parallelism as well as inter-loop data reuse. This occurs by creating 1D parallel tasks for each loop, and point-to-point task level synchronizations enable ready tasks from the second loop nest to be executed as soon as their input tasks (the ones corresponding to the same block row and the ones on either side) have completed. Thus, a "dynamic fusion" between the loop nests is achieved automatically, without the problem of load imbalance from the wavefront parallelism with a static affine tile schedule for the fused loop nests.

This problem has been recognized in the linear algebra community and specialized solutions have been designed [6]. We propose a solution for affine programs by leveraging properties of the schedule of tiles as computed by polyhedral compilers, and utilize it to determine at compile-time the inter-band (among disjoint loop nests) and intra-band (within a single loop nest) dependences. Our approach produces a fine interleaving of instances of the first and second band (outer iterations of the tiled loop nests). Unlike classical approaches in automatic parallelization, these dependences will then be instantiated at run-time, and fed to a dynamic task scheduler. The three last columns in Table 2 show the performance obtained in GFLOPS/sec when using two of PLuTo's heuristics (See Fig. 3 and Fig. 4) and comparing these to our generated task-parallel code. As one can see, by using point-to-point synchronization and statically mapping tile-level dependences it is possible to greatly improve over state-of-the-art vendor and research compilers: on Intel's i7 we achieve a 2.5$\times$ latency improvement w.r.t. ICC and PLuTo's best; whereas on AMD's Opteron we obtain over 5$\times$ relative to the baseline and 1.5$\times$ over PLuTo.

Our technique can be summarized as follows. We first compute tile-level constructs which are the input

to an algorithm that selects stream idioms to be used for each tile dependence or to a partition routine which splits the loop nests into classes that share identical input/output dependence patterns. This algorithm chooses when to extract parallelism across disjoint loops, while the partition routine allows to create a dynamic wavefront of tiles. Then a (static) task-graph is constructed to prune redundant dependences and to decorate it with dependence information. Finally, code is generated for the OpenStream run-time.

3. Background

The principal motivation for research in data-flow parallelism comes from the inability of the Von Neumann architecture to exploit large amounts of parallelism, and to do so efficiently in terms of hardware complexity and power consumption. In data-flow languages and architectures, the execution is explicitly driven by data dependences rather than control flow [15]. Data-flow languages offer functional and parallel composition preserving (functional) determinism. So-called *threaded* or *task-parallel* data-flow models operate on atomic sequences of instructions, or tasks, whereas early data-flow architectures leveraged data-driven execution at the granularity of a single instruction.

3.1. OpenStream

We selected the task-parallel data-flow language OpenStream [23], a representative of the family of low-level (C language) task-parallel programming models with point-to-point inter-task dependences [20, 7, 8]. OpenStream stands out for its ability to materialize inter-task dependences explicitly using streams, whereas the majority of the task-parallel languages rely on some form of implicit dependence representation (e.g., induced from memory regions in StarSs or OpenMP 4 [21]). The choice of an explicit dependence representation reduces the overhead of dynamically resolving dependences. For a detailed presentation, one may refer to Pop et al. [23], and to the formal model underlying the operational semantics of OpenStream [22].[1]

Like the majority of task-parallel languages, an OpenStream program is a dynamically built task graph. Unlike the majority of streaming languages, OpenStream task graphs do need to be regular or static, but they may arbitrarily and composed of a dynamically evolving set of tasks communicating through data-flow streams. The latter are strongly typed, first class values: they can be freely combined with recursive computations and stored in dynamic data structures. Programming abstractions and patterns allow to construct complex, fully dynamic, possibly nested task graphs with unbounded fan-in and fan-out communications. OpenStream also provides syntactic support for common operations such as broadcasts.

Programmer annotations are used to specify regions, within the control flow of a sequential program, that may be spawned as concurrent coroutines and delivered to a runtime execution environment. These control flow regions inherit the OpenMP task syntax. Without `input`/`output` stream annotations, OpenStream tasks have the same semantics as OpenMP tasks. For example, Figure 6 shows the OpenStream version of the first loop nest in the Ring-Roberts edge detection kernel. Each instance of the outer loop outputs a token to stream s. The second task, following the first loop nest, waits for $N-2$ tokens on stream s, and then proceeds. This pattern implements data-flow concurrency in OpenStream and avoids barrier synchronization.

While OpenStream is very expressive and can be used to express non-deterministic computations, the language comes with specific conditions under which the functional determinism of Kahn networks [16] is guaranteed by construction. These conditions enforce a precise interleaving of data in streams derived from the control flow of the program responsible for spawning tasks dynamically, hereafter called the *control program*. In the following, we will assume the control program is sequential, which is a sufficient condition to enforce determinism.

OpenStream Task Idioms In this work we leverage multiple programming patterns present in modern task-parallel languages.

[1] The source code repository and web site for OpenStream can be found at `http://www.openstream.info`.

```
int s __attribute__((stream));
int w, x, data[N-2];
for (i = 1; i < N - 1; i++)
#pragma omp task output (s << w)
{
  for (j = 1; j < N - 1; j++)
S1: B[i][j] = (A[i][j] + A[i][j-1] + A[i][1+j] +
      A[1+i][j] + A[i-1][j] + A[i-1][j-1] +
      A[i-1][j+1] + A[i+1][j-1] + A[i+1][j+1])/8;
}

#pragma omp task input (s >> data[N-2])
{/* ... */}
```

Figure 6: OpenStream example

1. Input and output clauses: they extend the OpenMP task syntax and allow to explicit the point-to-point synchronization between tasks via streams. By default, all streams are operated through a *window* of stream elements accessible to a given task. The horizon of the window is the number of stream elements accessible through it. The burst size of a window corresponds to the number of elements read/written from/to input/output streams at a given task activation. The default burst size is 1. The input clause can be further specialized into the two following clauses.

2. Peek operation: similar to the input clause, but it does not advance the stream's window, i.e., the window's burst size is zero. It enables the reuse of stream elements across multiple task activations, which is particularly useful when implementing broadcast operations.

3. Tick operation: a collection of peek operations may be followed by a tick operation, to advance the window to new stream elements.

4. Soft-barrier, point-to-point barrier or data-flow barrier: it is an (empty) program statement which waits for a (fixed or parametric) number of elements from one or more streams. Once all its (input) dependences are satisfied, an element is written to each of its output streams. Then, the tasks waiting for these such elements may read them with peek operations.

3.2. The Polyhedral Model

The polyhedral framework is a flexible and expressive representation for imperfectly nested loops with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoP) [10, 11], roughly defined as a set of consecutive statements such that all loop bounds and conditional expressions are affine functions of the enclosing loop iterators and variables that are constant during the SCoP execution (but whose values are unknown at compile-time). Numerous scientific computations exhibit those properties; they are found frequently in image processing filters (such as medical imaging algorithms), dense linear algebra operations and stencil computations on regular grids.

Unlike the abstract syntax trees used as internal representation in traditional compilers, polyhedral compiler frameworks internally represent imperfectly nested loops and their data dependence information as a collection of parametric polyhedra. Programs in the polyhedral model are represented using four mathematical structures: each statement has an *iteration domain*, each memory reference is described by an affine access function, data dependences are represented using *dependence relations/polyhedra* and finally the program transformation to be applied is represented using a *scheduling function* or a *schedule map*. Since the operations and analyses performed in this work heavily rely on the maps and sets representation of the Integer Set Library (ISL) [27] we briefly describe and give examples of these structures using the set and map notations. In addition, we also recapture two concepts that will be used in Section 4, *Bands of Tiles* and the *Polyhedral Reduced Dependence Graph*.

Iteration Domains They represent the set of dynamic (run-time) executions of a syntactic statement, typically enclosed in a loop nest. Each program statement is associated with an iteration domain defined as a set of integer tuples, one for each dynamic execution of the statement. These tuples capture the value of the surrounding loop iterators when the associated statement instance is executed at run-time. For the two statements `S1` and `S2` in Ring-Roberts we have:

$$\{S1[i,j] \in \mathbb{Z}^2 : 1 \leq i,j \leq N-1\}$$
$$\{S2[i,j] \in \mathbb{Z}^2 : 1 \leq i \leq N-2 \wedge 2 \leq j \leq N-1\}$$

Access Relations For a given memory reference (e.g., `B[i][j]`) access relations map statement instances with the set of memory locations of the array (e.g., `B`) which are accessed during the statement execution. Here a few examples from the Ring-Roberts kernel:

$$\{S1[i,j] \rightarrow B[i,j] : 1 \leq i,j \leq N-1\}$$
$$\{S1[i,j] \rightarrow A[i-1,j-1] : 1 \leq i,j \leq N-1\}$$
$$\{S2[i,j] \rightarrow B[i+1,j-1] : 1 \leq i \leq N-2 \wedge 2 \leq j \leq N-1\}$$

Dependence Relations Data dependences in ISL are represented as relations in between two iteration domains, possibly of the same statement for self dependences. The relation indicates the source and target of the dependence. Continuing with our example, two of the four dependences between $S1$ and $S2$ on array B are:

$$\{S1[i1,j1] \rightarrow S2[i2,j2] : 1 \leq i1,j1 \leq N-1 \wedge 1 \leq i2 \leq N-2 \wedge$$
$$2 \leq j2 \leq N-1 \wedge i1 = i2 \wedge j1 = j2\}$$
$$\{S1[i1,j1] \rightarrow S2[i2+1,j2-1] : 1 \leq i1,j1 \leq N-1 \wedge$$
$$1 \leq i2 \leq N-2 \wedge 2 \leq j2 \leq N-1 \wedge i1 = i2+1 \wedge j1 = j2-1\}$$

Schedules Reordering transformations are represented as relations from iteration domains to the logical time space. The relation assigns to each point in an iteration domain a (multi)-dimensional timestamp, which is later used to generate code. In the final code after transformation, each point in the iteration domains will be executed, according to the lexicographic ordering on the timestamps provided by the schedule function [4]. For instance, the following relation permutes dimensions i and j of the first loop nest in the output code of our running example:

$$\{S1[i,j] \rightarrow [j,i] : 1 \leq i,j \leq N-1\}$$

Bands of Tiles When the computed schedule represents a tiling transformation, a band is a consecutive set of non-constant dimensions (e.g., loop levels) in time space with the property that these dimensions are permutable. A band of tiles is a band that only considers some number of consecutive tile dimensions.

Intuitively, one can think of these dimensions as the ones that will become the tile loops after code generation. For instance, in Figure 3, only the tile loops are depicted. These correspond to the dimensions of the bands of tiles and would be generated by a schedule of the form:

$$\{S1[i,j] \rightarrow [0,ti,tj,i2,j2] : 1 \leq i,j \leq N-1 \ldots\}$$
$$\{S2[i,j] \rightarrow [1,ti,tj,i2,j2] : 1 \leq i,j \leq N-1 \ldots\}$$

The leading constant value of 0 and 1 in the range of the relations for statements $S1$ and $S2$, respectively, indicate that it is a scalar dimension (e.g., not a loop) and that the generated code will consist of 2 bands of tiles, composed in both cases by the trailing tile loop dimensions ti and tj.

Polyhedral Reduced Dependence Graph (PRDG) It is a multigraph where each node represents a program statement [9]. Edges in between nodes capture different dependences. Nodes are labeled with iteration do-

mains and edges are labeled with dependence relations. In our particular case, it will be required to construct a Tile-PRDG from the statement-PRDG and the computed tile schedule, as shown below.

Fusion Heuristics: min-fuse and smart-fuse The loop structure produced by a transformation can be viewed as a partition of the program under the criterion of appearing fused in one or more outer loops. *smart-fuse* is the default PluTo heuristic for tiling where statements that do not share any data reuse are placed on different partitions. *min-fuse* attempts to maximize the number of partitions by setting statements into different classes, unless they are required to appear fused under some loop level (or same strongly connected component) [5].

Macro-statements After applying a tiling algorithm such as PLuTo to an input program, statements are naturally fused under a sequence of loops. All statements that live under a common set of loops form a *macro statement*. The set of macro statements of a tiled program depends on the tiling heuristic used (e.g., smart-fuse or min-fuse).

4. Extracting Task Parallelism

The high-level flow that we follow to extract tasks at the granularity of a tile is shown in Figure 7. Its input is a sequential code which is first tiled with PLuTo's algorithm. Then the tile-level polyhedral representation (see Sec. 3.2) is extracted from the original statement representation and from the tile schedule computed in the previous stage. Next, the PRDG is obtained from this representation. The following stage partitions the tile domains according to the dependence signatures of all neighboring nodes. This produces a new set of domains for which a new PRDG is computed and input/output dependence patterns are associated to each new node/domain. Finally, code is generated and loops are decorated with OpenStream's task syntax that describes the input and output dependence instances for each task/tile.

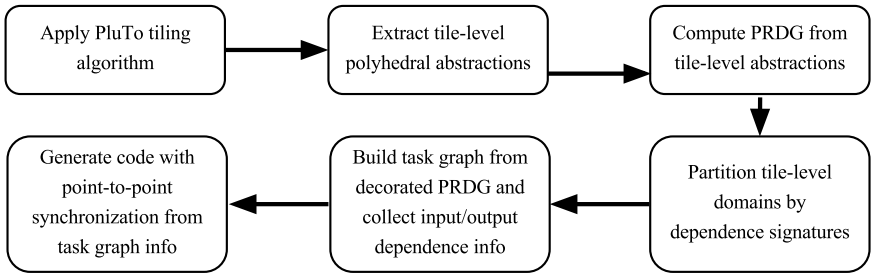

Figure 7: Compiler stages from a sequential input code to a tile level parallel version with point-to-point synchronization

The first stage of our method is to build a polyhedral representation of the *tiled* program, with one (macro-)statement per tile, each having an iteration domain (that is, capturing the set of dynamic executions of each tile). A *Tile-PRDG* is then produced, which captures data dependences between tiles. Next, we introduce an algorithm to select the most profitable task-parallel idiom, in terms of number of required synchronizations (incoming tile dependences). We present a processing step that allows to extract task parallelism from kernels that would otherwise use a (static) wavefront to expose tile-level parallelism, as well as handle cases where two disjoint loop nests have more than one dependence in common. The following step consists of building a static task-graph. Finally, we briefly discuss general details pertaining to polyhedral code generation.

4.1. Tile-Level Processing

The input to this compilation flow is a C program for which a tile schedule can be computed. In this work, we assume the PLuTo algorithm [5] has been used to enable tiling, however we are not limited to this particular tiling algorithm. We call *tile-level* the outermost or outermost and next outermost tile dimensions. Two tile-level abstractions must be constructed from the program statement domains, dependences and schedule: the tile domains and the tile dependences. These are determined by first projecting the intra tile dimensions $k..n$ of the schedule onto the tile dimensions $1..k-1$ in the range of the schedule map M^S of statement S:

$$M^S_{tile} = PROJ_{k,\ldots,n}(range(M^S))$$

This yields a tile map M^S_{tile} for each statement S, where k is the starting dimension to be projected (2 for inter-band parallelism and 3 for intra-band) and n is the number of output dimensions of the map. The modified map is then used to compute the tile domain by simply intersecting the map's domain with the set representing the statement domain. The range of the resulting map is the tile domain.

Tile dependences are constructed in a similar way. Given a dependence map $M^{S\to T}$ describing a dependence from statement S to statement T, and the tile maps M^S_{tile} and M^T_{tile} computed, the tile dependence $M^{S\to T}_{tile}$ is $(M^S_{tile})^{-1} \circ M^{S\to T} \circ M^T_{tile}$, where $\circ$ is the composition of maps. At this stage, we remove existent non-forward dependences, i.e., those which have the exact same source and target tile coordinates (same tile instance). This type of dependence could emerge after the projection step, in which case the dependence was forward in some of the projected out dimensions. From the task-parallel data-flow perspective, these dependendes are part of the task body of each tile instance, and do not represent any real flow of data. We do so by computing identity maps of the tile domain (e.g., from a domain $T\_0[tt,ii]$ we produce the map $T\_0[tt,ii] \to T\_0[tt,ii]$) and subtracting them from the original dependence maps, prior intersection with the tile domain. Non-forward dependences do not arise when considering inter-band dependences (due to the leading scalar dimension in the schedule). However, they do appear in single bands (e.g. seidel kernel), and must therefore be pruned.

In addition, all tile dependences that share the same source and the same target can be further simplified by: (1) combining pairs of basic maps within a single map by using the map coalescing ISL functions (this simplifies the representation); (2) if a new tile dependence is a subset of a previous dependence we do not include it; (3) conversely, if a new tile dependence is a superset of previous tile dependences, we remove the previous ones and add only the new one; (4) if a tile dependence partially intersects with some other tile dependence we take their union.

Finally, each tile dependence represented by a map (in ISL terminology) is massaged by detecting equalities, removing redundancies and making them disjoint (basic maps that constitute a map are not guaranteed to be disjoint unless invoking an explicit ISL function) [27]. An additional pruning stage is later performed during the task graph construction to remove dependence polyhedra which are already captured through transitive dependence relations.

Here we introduce the kernel jacobi-2d (Fig. 8), which we use to explain the partitioning steps. On Fig.9 we show 4 tile dependences computed from 9 statement dependences. These tile dependence will be further processed in order to make them uniform and proceed with the partitioning stage.

4.2. Partitioning

Domain partitioning is required in two scenarios: (1) when extracting inter-band parallelism, two nodes in the PRDG can share more than one normalized dependence (e.g. in blur-Roberts kernel), thereby needing different treatment for each dependence; (2) when the task graph consists of a single band of tiles, i.e. a single node in the PRDG. We note that when partitioning domains for the former case we only target the outermost tile dimension, whereas for the latter we work on the two outermost dimensions if dependences are uniform. Our approach for exposing task-level parallelism from a polyhedral program representation is to implement a partitioning scheme that groups tiles of a loop nest's iteration domain into classes, where each class is associated with a single signature of incoming and outgoing, inter-tile (or intra-tile) dependences.

This single signature in turn enables the generation of OpenStream directives and tasks with well defined input/output clauses.

```
for (t = 0; t < tsteps; t++)
{

 for (i = 1; i <= n-2; i++)
   for (j = 1; j <= n-2; j++)
 S1: B[i][j] = (A[i][j] +  A[i][j-1] + A[i][j+1] + A[i+1][j] + A[i-1][j]) * 0.2;

 for (i = 1; i <= n-2; i++)
  for (j = 1; j <= n-2; j++)
 S2: A[i][j] = B[i][j];
}
```

Figure 8: Jacobi-2d kernel

```
[tsteps, n] -> { T_0[0, 0, i2, i3] -> T_0[0, 0, o2, o3] : i2 >= 0 and n >= 3
and o3 <= 1 + i3 and o2 <= 1 + i2 and 2o2 <= i3 and 16o3 <= 30 + n + 32i2 and
16i3 <= -5 + 2tsteps + n and 16o3 <= -4 + 2tsteps + n and 16i3 <= 29 + n + 32i2
and 16o2 <= -1 + tsteps and 16i2 <= -2 + tsteps and o3 >= i3 and o2 >= i2 and
16o3 <= 12 + n + 16i3 and 32o2 >= -27 - n + 16i3 and 16o3 <= 28 + n + 32o2;
T_0[0, 0, i2, i3] -> T_0[0, 0, o2, o3] : 16o3 <= -4 + 2tsteps + n and 16i2 <=
-2 + tsteps and 16i3 <= 28 + n + 32i2 and 16o3 <= 30 + n + 32i2 and i2 >= 0 and
o3 >= 2o2 and o2 >= i2 and n >= 3 and 32o2 >= -26 - n + 16i3 and o3 >= i3 and
16o3 <= 28 + n + 32o2 and 16i3 <= -6 + 2tsteps + n and o3 <= 1 + i3 and o2 <= 1
+ i2 and 16o2 <= -1 + tsteps and i3 >= 2i2; T_0[0, 0, i2, i3] -> T_0[0, 0, i2,
o3] : 16o3 <= -3 + 2tsteps + n and 16i2  <= -1 + tsteps and o3 >= i3 and 16o3
<= 29 + n + 32i2 and i2 >= 0 and 16i3 <= -4 + 2tsteps + n and  i3 >= 2i2 and n
>= 3 and 16i3 <= 28 + n + 32i2 and o3 <= 1 + i3; T_0[0, 0, i2, i3] -> T_0[0, 0,
o2, o3] : 16o3 <= -3 + 2tsteps + n and 16i2 <= -2 + tsteps and 16i3 <= 29 + n +
32i2 and 16o3 <= 31 + n + 32i2 and i2 >= 0 and 2o2 <= i3 and 16o3 <= 29 + n +
32o2 and n >= 3 and o2 >= i2 and o3 >= i3 and 32o2 >= -27 - n + 16i3 and 16i3
<= -5 +  2tsteps + n and o3 <= 1 + i3 and o2 <= 1 + i2 and 16o2 <= -1 + tsteps
}
```

Figure 9: Jacobi-2d tile dependences

Our partitioning scheme takes as input the bands of permutable dimensions resulting from the PLuTo algorithm. However, any tiling algorithm that outputs a similar structure can also be used instead. We make the two following assumptions:

1. tile dependences of all statements within a permutable band must be subsumed by that of a single (macro-)statement,
2. all statements must live under a common and non-disjoint tile schedule space.

The first assumption is guaranteed by design of the PLuTo algorithm, which effectively groups statements in tiled parts of the schedule where all dependences may be collected as if the band was formed of a single, fused, macro-statement. The second one allows us to propagate the partition generated from a leading statement domain into the follower domains in the band. This is discussed next.

Definition 1 (Leading Domain and Follower Domains) *The iteration domain I^S of a statement S that belongs to a tiled band is called the leading domain if*

(i) the depth of this domain is maximal within the band (at the chosen granularity, i.e., of one dimension for inter-band and two dimensions for wavefront).

(ii) the domain with maximal depth is unique within the band. This implies that all tile dependences within the band are subsumed by the leading domain.

Any domain which is not the leading domain is a follower domain.

When Def. 1 does not hold, we name any statement domain with maximal (tile) depth the leading domain, but consider its tile domain as the union of all tile domains within the band that have maximal depth. In our jacobi-2d example (Fig.8) both statements have the same dimensionality. Therefore, we compute the tile domain of each individual statement, take their union, and name either of the statements as the leading domain. Definition 1 combined with our assumptions on the input schedule guarantees that all dependences are subsumed by the tile domain of the leading domain. Partitioning can safely be applied to the *leading domain*, and then propagated onto the *follower domains*. In the case that the macro-statement (statements sharing a same band of tiles) has no self-dependence, then any statement can be chosen as the leading domain.

Each dependence relation is identified with a unique number $1..n$, for n dependence relations. The dependence signature lists the unique identifiers of dependence relations that are either to or from the domain.

Definition 2 (Dependence Signature) *The dependence signature* SIG^S *of a domain* I^S_{tile} *is composed of two sets: the IN set and the OUT set. For each dependence relation k, k is put in IN (resp. OUT) iff* $\mathcal{D}^{T\to S}$ *(resp.* $\mathcal{D}^{S\to T}$*) has at least one destination (resp. source) in* I^S_{tile}*.*

$$\text{SIG}^S = \{IN^S, OUT^S\}$$
$$IN^S = \{k : Ran(D^{k:T\to S}_{tile}) \cap I^S_{tile} \neq \varnothing\}$$
$$OUT^S = \{k : Dom(D^{k:S\to T}_{tile}) \cap I^S_{tile} \neq \varnothing\}$$

It follows the definition of a disjoint partition of the iteration domain:

Definition 3 (Domain Partition) *The partition* P^S *of a domain* I^S *is defined by the following conditions:*

$$P^S = \{P^S_i\}, \quad I^S = \cup(P^S_i)$$
$$P^S_i \cap P^S_j = \varnothing \quad \wedge \quad \text{SIG}(P^S_i) \neq \text{SIG}(P^S_j), \quad i \neq j$$

where an iteration domain I^S *is divided into* P^S_i *disjoint domains such that they all have different dependence signatures.*

Fig.10 shows all the possible signatures that could be extracted from a 2D tile domain having as input dependences which are parallel to the axes in the transformed space. The actual signatures that arise after partitioning are a function of the tile schedule as well as the parameters. In the figure, shaded circles represent signatures never applicable to jacob-2d's tiled code. Moreover, some partitions might not execute at runtime due to the actual parameter values (e.g., the partition associated to signature 7).

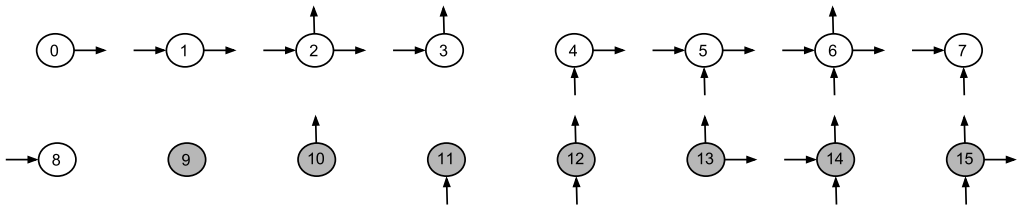

Figure 10: Potential signatures generated by uniform dependences (gray denotes signatures that do not appear in jacobi-2d kernel)

```
[tsteps, n] -> { T_0[0, 0, i2, i3] -> T_0[0, 0, 1 + i2, o3] : o3 >= i3 and
  16i2 <= -17 + tsteps and 16i3 <= 29 + n + 32i2 and 16o3 <= 31 + n + 32i2
  and i2 >= 0 and i3 >= 2 + 2i2 and o3 <= 1 + i3; T_0[0, 0, i2, i3] ->
  T_0[0, 0, o2, 1 + i3] : 16o2 = -1 + tsteps and 16i2 = -1 + tsteps and
  16i3 <= -19 + 2tsteps + n and 8i3 >= -1 + tsteps and tsteps >= 1;
  T_0[0, 0, i2, i3] -> T_0[0, 0, i2, 1 + i3] : 16i3 <= -19 + 2tsteps + n
  and 16i2 <= -2 + tsteps and 16i3 <= 13 + n + 32i2 and n >= 3 and i2 >= 0
  and i3 >= 2i2; T_0[0, 0, i2, 1 + 2i2] -> T_0[0, 0, 1 + i2, 2 + 2i2] :
  i2 >= 0 and 16i2 <= -17 + tsteps and n >= 3 }
```

Figure 11: Jacobi-2d tile dependences prior to transitive reduction pruning

```
[tsteps, n] -> {
* T_0[0, 0, i2, i3] -> T_0[0, 0, 1 + i2, i3]
  T_0[0, 0, i2, i3] -> T_0[0, 0, 1 + i2, 1 + i3]
* T_0[0, 0, i2, i3] -> T_0[0, 0, i2, 1 + i3]
  T_0[0, 0, i2, i3] -> T_0[0, 0, 1 + i2, 1 + i3]
  T_0[0, 0, i2, i3] -> T_0[0, 0, i2, 1 + i3]
  T_0[0, 0, i2, 1+2i2] -> T_0[0, 0, 1 + i2, 2 + 2i2]}
```

Figure 12: Jacobi-2d tile (uniform) dependences. Dependences marked with * are the non-redundant ones and used for partitioning

Making dependences uniform Fig.11 shows the tile dependence map for the jacobi-2d kernel before making all dependences uniform. In order to apply our partitioning algorithm we convert all the basic maps that constitute a tile dependence into their uniform shape. We do so by expanding the output dimensions that produce a range of values from the input dimensions, e.g., the first basic map with constraints $o3 >= i3$ and $o3 <= 1 + i3$ is expanded into 2 basic maps. The result of this expansion is shown in Fig.12. This step allows to further prune dependences by transitive reduction [19] [18], remove new covered dependences, and duplicated dependences. At this point, all output sets are explicit functions of their input sets. This permits to replace the tile dependence constraints by the constraints of the (leader) tile domain. The net effect of this is that dependences become broader, but also enables to remove redundant communication. After this process, only two non-redundant dependences are left, each parallel to one of the outermost axis on the transformed space. The only two required dependences are marked with a *.

Algorithm 1 is used to perform domain partitioning (when required), based on Definition 2 and 3, during the task graph construction and after pruning transitively covered dependences. The domain of a band could be partitioned both according to the incoming dependences as well as the outgoing dependences, e.g. domain S could be partitioned first when processing dependence $R \rightarrow S$ and again partitioned when processing dependence $S \rightarrow T$, and also due to multiple dependences between two nodes (since the PRDG is a multigraph).

Fig. 15 shows the partitions generated by Algorithm 1, the respective dependence signatures, and the distribution of the signatures around the (tile) domain. It may be surprising to see signature 7 in this figure. However, this signature could be triggered depending on the parameter values, especifically for this condition:

```
(2 * ((tsteps - 1) % 16) + ((14 * tsteps + 15 * n + 2) % 16) <= 13 && (tsteps - 1) % 16 <= 6)
```

In the following two sections we will discuss in more detail when partitioning is required.

Partial tiles A band of tiles contains a partial tile when one or more of its (intra-tile) dimensions have a smaller cardinality than the selected tile size. Our algorithm does not distinguish partial tiles from full tiles (tiles which have their cardinality equal to the tile size), since we are only interested in the (outer) tile coordinates, and which are only dependent on the problem size and outer tile coordinates. Furthermore, partial tiles already have a different dependence pattern, and only appear at the end of some domain. As such, they are naturally placed into their own (signature) class. A similar handling applies to all tiles which appear in a domain border, i.e., a first or last tile instance along some dimensions.

ALGORITHM 1: PartitionDomains (G, u, v)

Input: G: Tile-Level PRDG; u: source tile domain; v: target tile domain
Output: Updated G where nodes u and v have been decorated with their partitions

if $parts(u) = \emptyset$ **then** $parts(u) \leftarrow I^u_{tile}$;
if $parts(v) = \emptyset$ **then** $parts(v) \leftarrow I^v_{tile}$;
foreach *tile dependence d from u to v* **do**
  $indom \leftarrow$domain(d);
  $outdom \leftarrow$range(d);
  foreach *p in parts(u)* **do**
    $s \leftarrow p \cap indom$;
    if $s \neq \emptyset$ **then**
      $diff \leftarrow p - s$;
      SIG($diff$) $\leftarrow$SIG(p);
      SIG(s) $\leftarrow$SIG(p) $\cup \{OUT^s = d\}$;
      remove p from parts(u);
      insert s and $diff$ in parts(u);
      $indom \leftarrow indom - p$;
    end
  end
  foreach *p in parts(v)* **do**
    $s \leftarrow p \cap outdom$;
    if $s \neq \emptyset$ **then**
      $diff \leftarrow p - s$;
      SIG($diff$) $\leftarrow$SIG(p);
      SIG(s) $\leftarrow$SIG(p) $\cup \{IN^s = d\}$;
      remove p from parts(v);
      insert s and $diff$ in parts(v);
      $outdom \leftarrow outdom - p$;
    end
  end
end

ALGORITHM 2: ConstructTaskGraph (PRDG)

Input: PRDG: Tile-Level PRDG
Output: T: Task graph with signatures and partitioned domains

foreach *written array A* **do**
  Collect dependences on array A;
  Traverse the PRDG and remove transitively covered dependences on array A;
end
foreach *edge* $u \rightarrow v$ **do**
  if *u or v have more than 1 dependence* **then**
    PartitionDomain(PRDG,u,v);
  end
end
$T \leftarrow \varnothing$;
foreach *edge* $u \rightarrow v \in PRDG$ **do**
  $\pi_u \leftarrow$ Get Partitioned Domains of u;
  $\pi_v \leftarrow$ Get Partitioned Domains of v;
  Add a node to T for each domain in π_u and π_v;
  Add edges for dependences between π_u and π_v to T;
end
return T

4.3. Task Graph Construction

The task graph construction takes as input the decorated PRDG. We contemplate a single-dimensional tile band PRDG in order to limit the number of potential synchronizations, which increases as one considers more tile dimensions by factors that depend on the problem sizes as well as the chosen tile size. This still allows a sufficiently fine interleaving between the tile instances. Thus, this stage is only performed if we have multiple loop nests after tiling. The goal of this step is to traverse the PRDG and consider the minimum number of edges in the task graph so that all dependences are satisfied as well as being able to remove redundant synchronizations that can arise from transitively covered dependences, which often appear in PRDGs composed of several bands. Algorithm 2 shows the pruning steps performed for the dependences (edges of the PRDG) on each array (scalars are 0-dimensional arrays) and the addition of the relevant edges from the PRDG into the task graph. Transitively covered dependences are found by composing the dependence relations in between PRDG nodes. For example, given nodes R, S and T, and dependences $R \rightarrow S$, $S \rightarrow T$ and $R \rightarrow T$, then dependence $R \rightarrow T$ can be removed if it is equal to the composition $R \rightarrow S \circ S \rightarrow T$. These dependences can be of type RAW, WAR and WAW. Then, Algorithm 1 is invoked for nodes u and v of the PRDG when they have more than one dependence relation connecting them, i.e. 2 or more dependences which differ in either source or domain or both (such as in blur-Roberts for instance). Finally, the underlying task graph is constructed from the partitioned domains attached to each node of the PRDG.

4.4. Implementing Inter-Band Parallelism

We now use the 3mm benchmark from Polybench/C to illustrate the various implementations of inter-band paralellism that can be achieved. Our algorithm automatically prioritizes the possible implementations based on data dependence features, as shown later.

```
// Band 1
for (i=0; i<l; i++)
  for (j=0; j<m; j++)
    for (k=0; k<q; k++)
 S1:  A[i][j] += B[i][k] * C[k][j];
// Band 2
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    for (k=0; k<p; k++)
 S2:  D[i][j] += E[i][k] * F[k][j];
// Band3
for (i=0; i<l; i++)
  for (j=0; j<n; j++)
    for (k=0; k<m; k++)
 S3:  G[i][j] += A[i][k] * D[k][j];
```

Figure 13: 3mm kernel (with removed initializations)

In the following the term *band* can be intuitively thought as a single loop nest. We need to decide whether two nodes connected by an edge in the task graph can be effectively parallelized with inter-band (across disjoint loop nests) task parallelism, if a soft barrier should be used to satisfy the dependences of the target node, or if a band should be considered as a single task. *We emphasize that this algorithm is designed for a single level of intra-band parallelism. Thus, it focuses exclusively on 1-dimensional tile-level abstractions.* Consider the dependence between $S1$ and $S3$ on array A, each $S3(i)$ depends on an $S1(i)$. Intuitively one can think of this as a "row-to-row" mapping. Consider now the dependence from $S2$ to $S3$ on array D. Each $S3(i)$ requires all the instances $S2(i, j, k)$, a many-to-1 relation. However, when considering all instances of $S3$ this becomes a many-to-many relation. Thus, depending on the desired granularity, the communication can start pounding the run-time. At the very least each $S3(i)$ will have to wait for the m instances of $S2(i)$, and could become as bad as $(l \times n \times m)$ instances of $S3$ each waiting for $(m \times n \times p)$ instances of $S2$.

This highlights that additional granularity of parallelism is not for free, as synchronization of several orders of magnitude might be required and the additional benefits of parallelism will be mitigated. This granularity criterion also serves a second purpose, which is to exclusively restrict inter-band parallelism to the outermost dimensions of nodes/bands in a dependence relation. Finally, we systematically treat untiled bands (for instance those that result from multi-level reduction statements) as single tasks. The motivation comes from the exponential growth in sychronizations (a factor of tile-size for each dimension) that can arise with such loops.

Algorithm 3 prioritizes the 3 types of communication from less communication (single task), to 1-to-1 mappings, to m-to-n communication patterns that require soft barriers. Along the process nodes are decorated with the type of synchronization and the name of the stream to be used; and new nodes are inserted to handle barriers and tick operations. The latter are OpenStream operations responsible of consuming data tokens which were previously broadcasted. Fig. 14 shows the 1-dimensional tile-PRDG (left) and the result of applying our algorithm to it (right).

Inter-band parallelism is possible and profitable between bands B1 and B3 (akin to a row-to-row mapping). However, the algorithm deems unprofitable the number of synchronizations between bands 2 and 3. Thus the task graph is modified to reflect the insertion of a node BX which collects the output dependences from B2, and outputs a single dependence which is reused via *peek* operations in B3. This is OpenStream's broadcast stream idiom. After B3 is also necessary to insert a new node BY to consume the token that has been used by all instances of B3 (a tick operation). Experimental results show that for this 3mm example, 1D inter-band parallelism (only one barrier synchronization) outperforms by 41% on AMD Opteron a 2D inter-band

ALGORITHM 3: SelectTaskIdiom (G)

Input: G: Task-graph after partitioning
Output: G: Decorated task-graph
for *each edge* $e \in G$ **do**
    $S \leftarrow$ tile-level domain of dependence source ;
    $T \leftarrow$ tile-level domain of dependence target ;
    $m \leftarrow$ dependence map $S \rightarrow T$ of edge e;
    $m \leftarrow$ intersect domain of m with S;
    $m \leftarrow$ intersect range of m with T;
    $m \leftarrow$ decompose m into a union of basic maps in ISL;
    if *S is an untiled band* **then**
        Create a new stream *single*;
        Output a single dependence from S;
        Input dependence of T with peek operation from S;
        Insert a tick operation after band T;
    end
    else
        if *(Def. 4 is true for S and T)* **then**
            Create an array of streams D, one per distinct basic map in m;
            Decorate e with 1-to-1 inter-band parallelism on D;
        end
        else
            Create a new stream *bar* of size 1 for barrier synchronization;
            Insert statement *barout* between nodes S and T;
            Output all dependences of S to *bar*;
            Input dependences of *barout* from S;
            Output a single dependence of *barout* to T;
            Input dependence of T from *barout* with peek operation;
            Insert statement *barin* after node T with tick operation on *bar*;
        end
    end
end

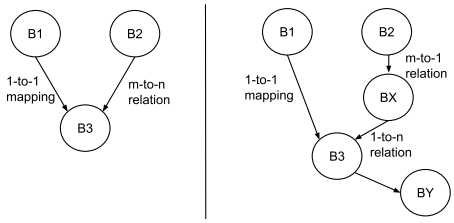

Figure 14: 3mm 1-dimensional tile-PRDG (left) and task graph (right)

parallelism approach (no barriers, three 1-to-1 2D mappings and two many-to-many synchronizations with peek operations). The performance between the two schemes is comparable on Core i7, and also in favor of 1D inter-band parallelism for AMD Phenom.

We now formalize the requirements for inter-band parallelism. Definition 4 establishes that a fine-grain interleaving of loop iterations between two bands can be legally executed. Once an iteration of the source band is executed, the dependent iteration of the target band can initiate execution.

Definition 4 (Inter-band parallelism) *Given two distinct bands A and B. Barrier-less inter-band parallelism is exploitable if:*

1. *exists at least one point in band B that does not depend on all the points of band A*
2. *Neither band A nor band B have dependence cycles*

Def. 4 is a general definition for inter-band parallelism. Condition 1) enforces that some subset of the target band (possibly exhibited after domain partitioning) can initiate early execution, i.e., a barrierless behavior. Condition 2) essentially states when a point-to-point synchronization is possible. Some of the cases that this definition covers are:

- when all points from band A have exactly one outgoing dependence instance, and all points from band B have exactly one incoming dependence instance (e.g. between bands B1 and B3 in Fig. 14);
- when the number of dependence instances outgoing from band A are bounded by some fixed number K_A, and the number of incoming dependence instances to band B are bounded by a fixed number K_B (e.g. in Ring-Roberts kernel, $K_A = K_B = 3$). Note also that not all points in both bands are required to have the same number of incoming or outgoing dependence instances;
- with no particular relation between the cardinalities of band A and B (e.g., $card(A) = card(B)$, $card(A) = 2 \times card(B)$, $card(A) = card(B) + K$, etc).

Definition 5 is used in Algorithm 3 to select when a soft barrier must be inserted in the task-graph. We recall that a soft barrier is a barrier with point-to-point synchronization, that is, it does not affect all bands, but only the one that indeed requires that output data of the source band.

Definition 5 (Soft barrier synchronization) *Given two bands A and B, if no barrier-less inter-band parallelism can be found according to Definition 4 then a soft barrier must be inserted between both bands.*

4.5. Extracting Dynamic Tile-Level Wavefronts

The (static) partitioning scheme presented in Section 4.2 allows to extract dynamic wavefronts of tiles, i.e., all inter-tile instance dependences are made explicit to the run-time, which in turn determines the actual wavefront as tiles go executing and completing. Fig. 15 displays the partition produced for a 2-dimensional (tiled) iteration domain corresponding to the jacobi-2d running example. The arrows depict the direction in

which the data flows, in this case, one token for each tile instance. The x-axis represents space while the y-axis represents time. Partitioning yields 9 (tile) domains, each identified by its signature number. The appearance of a specific 2-dimensional signature depends directly on the domains's shape. For jacobi-2d, the outer space dimension (tile dimension-i) is skewed w.r.t to the time dimension and has as schedule (tt,2*tt + ti), where tt is the time tile coordinate and ti is the space tile coordinate. Fig.10 shows the possible dependence signatures on a given 2D partitioning. The shaded ones are signatures which do not arise in jacobi-2d.

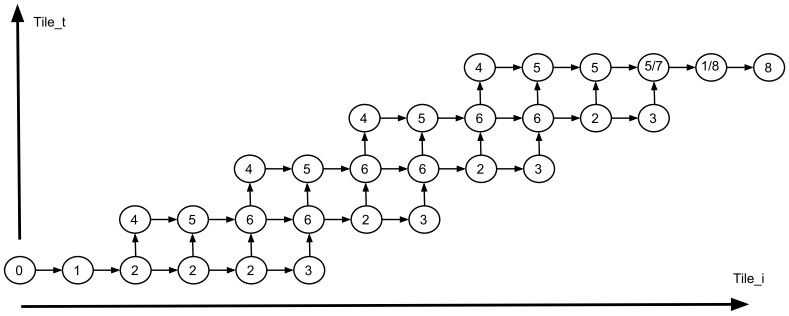

Figure 15: Partition generated for jacobi-2d with tile schedule (tt,2 * tt + ti). Circles represent tile instances and the number within each is the signature id. (see Fig.10)

4.6. Code Generation

During polyhedral code generation, special care must be taken to replicate the structures of partitioned domains as well as pretty-printing OpenStream clauses from the annotated information in the task graph. This includes the generation of input/output clauses from the dependence signature of each node. Streams are allocated and declared before the SCoP. The appropiate scalar values on the tile schedule must be set to represent the correct interleaving of bands.

Separating dimensions Separating dimensions are schedule dimensions which hold scalar values. Their purpose is to keep separated certain domains. In general, polyhedral code generation will attemp to fuse all possible dimensions allowed by the schedule. Thus, in order to keep the generated partitions separated (and their -code- signature unique), we insert an additional scalar dimension at position 1 (offset to 0) for each band parallelized via dynamic wavefront. In standard polyhedral techniques this is not common practice. However, since the dependences of the outer dimensions will be dynamically resolved, inserting an arbitrary and distinct scalar value for each partition remains legal.

Stream declarations The current implementation of OpenStream only supports 1-dimensional arrays of streams. For each inter-band stream we compute the volume of the outermost dimension and use it as the stream array size. For dynamic-wavefront we assume a 2-dimensional linearized array of dimensions $D_0 \times D_1$, where D_0 and D_1 are the volumes obtained by projecting out all dimensions except dimension 0 for the former and dimension 1 for the latter. However, this results in a non-affine parametric product. Thus, the product is computed out of the polyhedral code generation process.

Stream indexation Stream arrays are indexed by the (outer) tile coordinates, which is either the source or target of the tile dependence: all input streams read from their local tile coordinate and write to the tile coordinate determined by the outset of a tile dependence. Here is where the property of having output sets that are explicit functions of the input set in a tile dependence becomes essential. In addition, since the

dynamic wavefront requires a 2-dimensional access to the stream array, which commonly results in a non-affine expression (e.g., $iterator \times parameter$), the linearization process is performed in a pragmatization pass after polyhedral code generation.

Simplification To avoid some corner and degenerate cases such as having only one tile instance on any dimension, we set the parameter context parameters (e.g., problem sizes) as having at least 4 tiles along a dimension. The partition shown in Fig.15 as well as the code in Fig.16 have this simplification.

Dealing with One-Time-Loops (OTLs) OTLs are loops which execute only once. Polyhedral code generators such as CLooG-ISL typically eliminate these to produce more efficient code. Removing a loop implies propagating the iterator's value of the removed OTL into the loops and access functions of array references nested within. This optimization poses a minor problem for the stream mapping process since all tile iterator values are required for proper indexation. OTLs are very common when partitioning, as can be observed from Fig. 15, any tile domain part which lies on a domain border will have at least one loop removed (e.g., part with signature 4 will have dimension $Tile_i$ removed). Unfortunately, removed OTLs are only known after code generation (as soon as a loop dimension is "skipped"), and not before. Thus, we resort to re-generate the explicit representation of the tile iterators on the dimensions used for partitioning by applying the tile schedule (with projected out intra-task dimensions) to the tile domain of the leader domain and tagging this information in the generated code. We note, nonetheless, that other code generators such as non-ISL based CLooG [4], have options to eliminate or to keep OTLs.

Finally, a global *taskwait* is inserted after the SCoP so that the program does not terminate early. As a note, it is particularly useful to use unscaled tile iterators during code generation in order to facilitate the mapping between tile coordinates and stream arrays. The generated code for one of the partitions is shown in Fig. 16. More examples of the generated code can be found in Sec. A.

```
for (int tt = 1; tt < (tsteps - 1) / 16; tt += 1)
 for (int ii = 2 * tt + 2; ii <= 2 * tt + (n - 3) / 16; ii += 1)
  #pragma omp task input(stream_tt[( tt ) * S_1_3 + ii] >> token1_1,\
    stream_ii[( tt ) * S_1_3 + ii] >> token2_1)\
    output(stream_tt[( 1+tt ) * S_1_3 + ii] << token1_2,
    stream_ii[( tt ) * S_1_3 + 1+ii] << token2_2)
  for (int jj = ii; jj <= ii + (n - 3) / 16 + 1; jj += 1)
   for (int t = max(16 * tt, -n + 8 * jj + 2); t <= 16 * tt + 15; t += 1)
    for (int i = max(16 * ii, -n + 16 * jj + 2);
          i <= min(min(16 * ii + 15, 16 * jj + 14), n + 2 * t - 1); i += 1)
     for (int j = max(i + 1, 16 * jj); j <= min(16 * jj + 15, n + i - 2); j += 1) {
      if (n + 2 * t >= i + 2)
       B[-2 * t + i][-i + j] = (0.2 * ((((A[-2 * t + i][-i + j] + A[-2 * t + i][-i + j - 1]) +
        A[-2 * t + i][-i + j + 1]) + A[-2 * t + i + 1][-i + j]) + A[-2 * t + i - 1][-i + j]));
      A[-2 * t + i - 1][-i + j] = B[-2 * t + i - 1][-i + j];
     }
```

Figure 16: Jacobi-2d code snippet for partition with signature 6 (see Fig.15)

4.7. Putting It All Together

Algorithm 4 brings together our approach for extracting both inter-band and intra-band tile-level parallelism, which allows to describe concurrent tasks with explicit, point-to-point dependences. Based on the number of tile bands produced, one of our two main techniques is applied, i.e., inter-band parallelism in the presence of more than one band, and our partitioning scheme which enables a dynamic wavefront of tiles.

4.8. Handling complete applications

The techniques described in this paper can be applied to a variety of real world applications. To do so, the input program needs to be affine or to have affine friendly portions. Also, a number of pre-processing transformations would normally be required to make a program tilable and to further expose parallelism (e.g., array/scalar expansion, renaming and privatization).

Partitioning scalability Our partitioning technique could be easily extended to handle more than the two outermost dimensions. In general, partitioning a domain according to dependence signatures yields $O(2^{2\times nbr.deps})$ parts. This, for a 2D domain represents up to 16 parts per program statement. In practice, a few signatures are impossible to generate for a particular input and tile schedule. However, for higher dimensional domains this number grows fast. A 3D domain could potentially generate up to 64 parts and a 4D domain 256. At this point, the polyhedral code generation could become a bottleneck, especially for complete applications which can have hundreds of statements.

ALGORITHM 4: High-level algorithm for data-flow task-level parallelization

```
Input: Statement level PRDG and tile schedule
Output: Tile level task graph and PRDG with decorated nodes and edges
if (nbr.bands > 1) then
    if (no dependences) then
        Bands fully parallel and independent;
    end
    else
        Tile_PRDG ← build 1-dimensional tile-level PRDG;
        T ← ConstructTaskGraph (Tile_PRDG);
        T ← SelectTaskIdiom (T);
    end
end
else
    if (outer dimension is parallel) then
        Band is fully parallel;
    end
    else
        if (all dependences are uniform) then
            Tile_PRDG ← build 2-dimensional tile-level PRDG;
            T ← ConstructTaskGraph (Tile_PRDG);
            T ← SelectTaskIdiom (T);
        end
    end
    return {Tile_PRDG,T};
end
```

5. Experimental Results

In our experiments we do not perform auto-tuning, and use the same tile size on all tile variants of a given benchmark so that we can have a cleaner comparison in terms of workload. We conducted three sets of experiments to validate our approach. The first one shows the performance achieved by our technique when using all cores of a machine. Our goal here is not to show the best possible performance, but to show that for the same workload we can obtain similar performance to PLuTo's heuristics (both tiled and untiled) for dense linear algebra kernels (which exhibit low imbalance), and strong performance improvements on kernels that are typically parallelized with static wavefronts (e.g., Seidel-like kernels, Jacobi-like kernels and FDTDs) or that exhibit some inherent task parallelism (bicg, mvt). We remark, nonetheless, that auto-tuning would be required to search the tile space for best performance. For dynamic partitioning the search should focus on

tile sizes that produce (roughly) the same number of tile instances on the outer two dimensions. The second set of experiments focuses on showing the scaling and load balancing properties of our technique for a few representative benchmarks. The final set of experiments compares our technique to diamond tiling [2] in terms of performance scalability and code structure.

5.1. Experimental Setup

The machines used for the experiments as well as the benchmarks are described in Tab. 5.2 and Tab. 5.2 We test our approach on a subset of Polybench-3.2/C [24] and focus on dense linear algebra and stencil kernels. The compilers used were GCC 4.8.1, ICC 2013-update5, PoCC-1.3. Our framework was built over PPCG [28]; experiments were performed with two OpenStream versions, the public version [23] and one in development for trace capabilities. The FLOPS for each benchmark are computed statically.

5.2. Experimental Protocol

Figures 17 and 18 report the performance (GFLOPS/sec) on double-precision floating point using all available cores. For each benchmark we report baseline performance on GCC and ICC (-O3 -ffast-math for GCC and -O3 -parallel -xhost for ICC), PLuTo's best tiled performance and PLuTo's best untiled performance, compiled with both ICC and GCC (generated using PoCC's parallel, tile (or not), minfuse/smartfuse, prevector, pragmatizer and vectorizer flags, compiled with -O3 -openmp -xhost), and the performance of two OpenStream variants: one, Task, exclusively compiled with OpenStream's modified GCC (v4.7.0) and the second one, Task opt exporting the task bodies into separate compilation units which are processed by PoCC using pragmatizer, vectorizer and past-hoist-lb flags. Each of the compilation units are then built with Intel's ICC -O3 -xhost for further optimization, and then linked with OpenStream's GCC modified compiler.

	Intel Core i7-2600	AMD Opteron 6274	Intel Xeon E5-2650 v2
Freq	3.40 GHz	2.2 GHz	2.6 GHz
Cores	4	16	8
L1	32 KB	16 KB	32 KB
L2	256 KB	8 x 2 MB	256 KB
L3	8 MB	6 MB	20 MB
RAM	16 GB	32 GB	16 GB

Table 1: Experimental testbed

Benchmarks	Category	Problem Size	Tile Size
2mm,3mm,gemm,syrk	linear algebra	2000^3	32
gemver	linear algebra	8000^2	32
bicg,mvt	linear algebra	8000^2	32
correlation,covariance	data mining	2000^2	32
seidel-1d	stencils	200000^2	1024
blur-Roberts	image processing	4000^2	32
fdtd-2d,jacobi-2d,seidel-2d	stencils	2000^3	16

Table 2: Benchmark description (right)

5.3. Performance Results

We first remark that the achievable performance varies significantly between GCC and ICC, for the same code. This is why we chose to report numbers not only for OpenStream's native compiler (GCC), but also for a hybrid compilation scheme relying on ICC for task bodies and OpenStream's GCC for task creation and

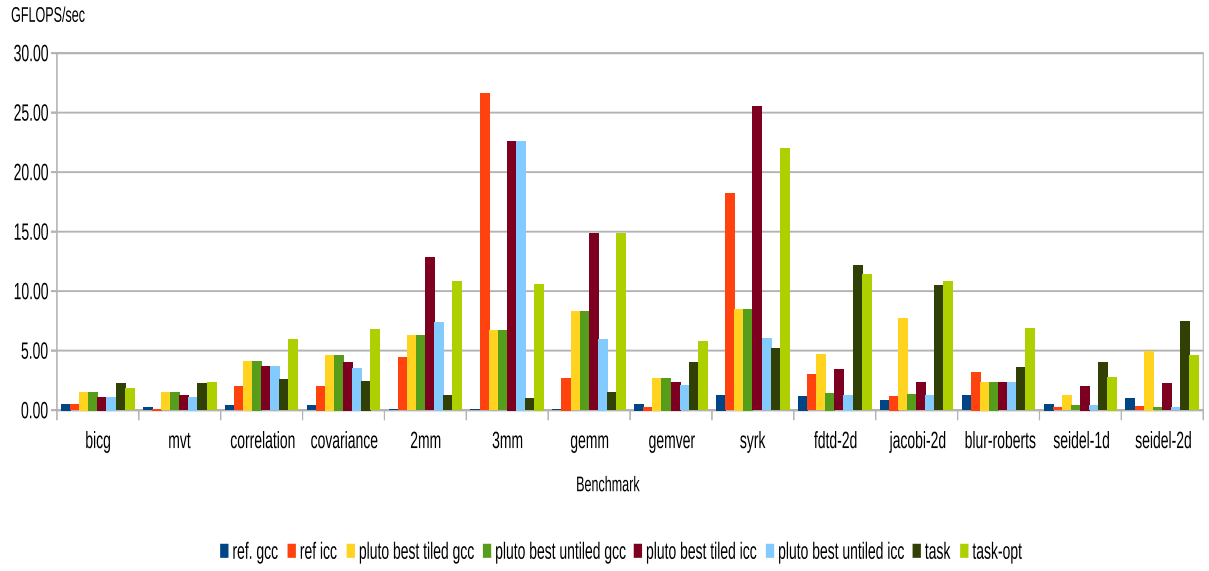

Figure 17: Performance (GFLOPS/sec) on AMD Opteron 6274, double precision, using all 16 cores

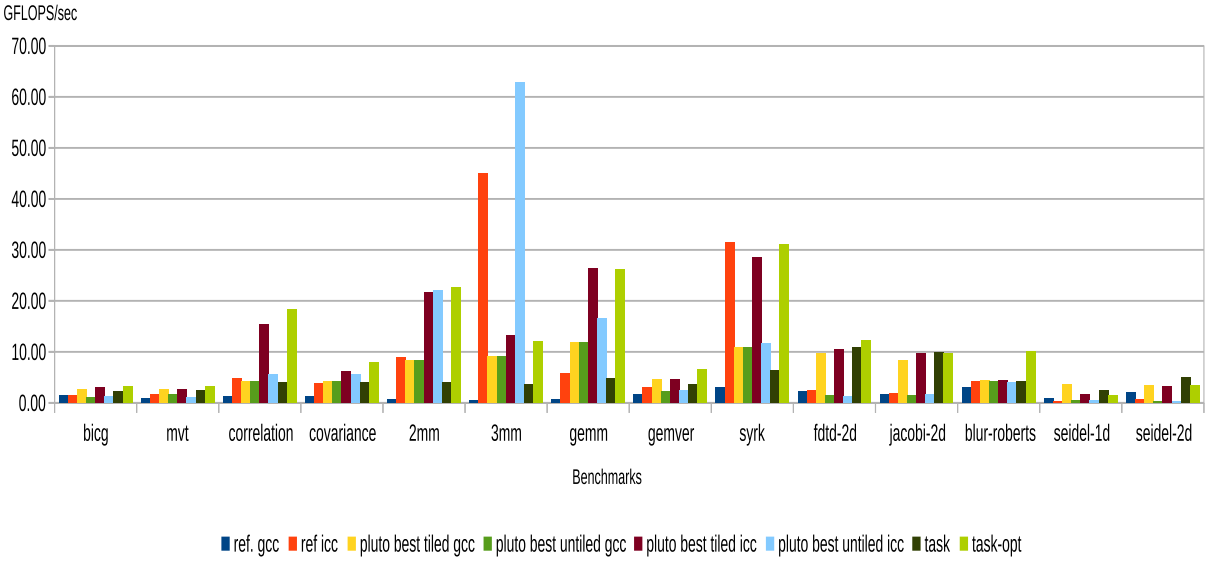

Figure 18: Performance (GFLOPS/sec) on Intel i7-2600, double precision, using all 4 cores

scheduling (*task-opt*). The reason for this is that GCC does not always implement as many optimizations as ICC for these numerical codes, implementing less effective automatic vectorization and less optimizations in the low-level generated code.

Our experiments show that two architectural features have a strong impact in performance: the processor's core count and having a shared L2 cache (e.g., AMD's Opteron). The significance of the core count is evident. More cores imply longer synchronizations in each barrier execution as well as a longer time to reach the

steady state in pipeline/wavefront schedules. The overall impact of having a shared L2 cache is observable by comparing Fig. 17 and Fig. 18, wherein only the well balanced benchmarks (variants of matrix multiply) maintain their relative performance between AMD's Opteron and Intel's i7-2600. In general, smartfuse benefits from shared L2 caches as it is able to exploit more data reuse, but this also hinders vectorization for some benchmarks (e.g., fdtd-2d and jacobi-2d which have tiles with innermost loops that carry data dependences). On the other hand, minfuse exploits less the cache hierarchy, but typically produces more effectively vectorizable loops (due to loop distribution). Finally, task variants implemented with minfuse exhibit better locality due to OpenStream's default task firing policy (when a task completes one of its dependent tasks is activated). Our 14 benchmarks can be classified into the following 4 categories: benchmarks with independent bands (bicg and mvt); benchmarks with fixed but short dependence paths, fewer than 7 bands (2mm, 3mm, gemm, gemver, syrk); benchmarks with fixed but long dependence paths, i.e., 7 or more bands (correlation and covariance); and benchmarks which PLuTo's versions are composed of a single band, resulting from a wavefront transformation with a sequential outer loop and a second outermost parallel loop (blur-Roberts, fdtd-2d, jacobi-2d, seidel-1d and seidel-2d).

Benchmarks with independent bands In this category we have kernels mvt and bicg. Both are 2D kernels that compute a matrix-vector product. PLuTo's heuristics produce the same fusion structure for bicg (4 statements, 4 bands) whereas for mvt smartfuse fuses the two statements under a single band. This is due to matching dimensionalities of statements involved. Both kernels have the particularity that one of the two compute statements references a 2-dimensional array, which incurs on a high-stride w.r.t the innermost loop. Thus, they will both suffer from oversynchronization or from bad locality produced by the high-stride. The profitability of tiling varies between among the architectures and compilers used: bicg's best untiled version improves 50% over the best when compiled with GCC, but only 11% when compiled with ICC in AMD's Opteron; on the contrary, on Intel's i7-2600 performance descreases 60% from the best tiled to the best untiled version with GCC and ICC. Regarding mvt, best untiled improves performance in 9% with GCC, but decreases in 11% with ICC on Opteron, while on Core i7 it decreases performance between 40% and 55%. Fig. 17 and Fig. 18, for these benchmarks show that PLuTo tiled versions improve performance by factors of $2\times$ to $5\times$ w.r.t. ICC. Exploiting minfuse combined with point-to-point synchronizations enables to duplicate the number of independent tasks and to remove the barrier present in PLuTo's minfuse variant. This translates in a $1.05\times$ to $2.2\times$ improvement of our data-flow approach over PLuTo's barrier's based parallelization. Furthermore, due to the simplicity of the kernel, ICC does not show any notable improvement over GCC (single-thread performance). On average, our task variants improve the performance of these kernels by 50% on Opteron and 12% on Core i7.

Benchmarks with short dependence paths Benchmarks 2mm, 3mm, gemm, gemver and syrk fall into this group. The minfuse heuristic essentially places each statement in a separate band. Thus, we have 4, 6, 2, 4, and 2 bands, respectively. For this category of benchmarks we observed a clear performance pattern, untiled versions compiled with GCC generally outperform the tiled variants on both architectures, whereas tiled variants outperform the untiled when compiled with ICC. The exception to this observation is the best 3mm untiled variant (and its reference versions as well). Further inspection revealed that ICC was able to pattern-match matrix-multiplication kernels in the input code and transformed code, replacing all three matrix-multiply in 3mm by MKL calls. Similarly, one of the two matrix-multiply of 2mm got replaced by an MKL call in the reference version. Interestingly, ICC was however unable to recognize and pattern-match the gemm kernel nor its occurrence in 2mm, meaning the 3 kernels kernels for which ICC was able to use MKL instead of the original code were 2mm (in part), 3mm-ref (in full) and 3mm-untiled. Our approach achieves performance comparable to PLuTo for 2mm, 3mm, and gemm, as these are dense linear algebra kernels which do not suffer from an inherent imbalance, even after tiling. We observe in Figures 17 and 18 that composing matrix multiply (kernels 2mm and 3mm) produces a gradual loss in performance on both machines for PLuTo compiled with ICC. On gemver we observe a $1.6\times$ to $2.2\times$ improvement with *task-opt* w.r.t. PLuTo's best across both architectures. Tiled-Smartfuse produces 3 bands while tiled-minfuse produces 4. Inter-band

parallelism effectively removes the second and third barriers (we do not count the barrier following the last band).

Regarding kernel syrk, PLuTo outperforms the task variants by approx. 16% on Opteron and yields almost the same performance on Intel's i7-2600. We note that for syrk, PoCC's vectorizer flag was disabled, since it was sinking a loop that produced a high-stride on the symmetric matrix (enabling this flag produces about 6 GFLOPS/sec, instead of 25 GFLOPS/sec).

A complementary experiment showed that by converting 3mm into three consecutive gemm kernels, performance on Intel's i7-2600 fell to 9.9 GFLOPS. Finally, we note that no significant differences were observed between PLuTo's fusion heuristics.

Benchmarks with long dependence paths Here we have correlation and covariance kernels, both of which exhibit between 7 and 12 barriers in PLuTo tiled versions, but between 4 to 7 in their respective untiled variants (4-7 for correlation and 4-5 for covariance). No drastic performance gap was observed between PLuTo's tiled fusion heuristics which consistently outperform *ref icc*, however, untiled variants exhibited a high variation. Overall, the performance of untiled variants varies between 66% slowdown to 60% improvement w.r.t. their untiled counterpart. PPCG produces a schedule with several untiled 1-dimensional loops, which are parallelized as single tasks. Performance gains for these benchmarks vary from $1.2\times$ to $2.1\times$, relative to PLuTo's. Furthermore, opportunities for inter-band parallelism are very limited. Thus, the main sources of improvement are the better load balance enabled by task parallelism, and the lightweight stream idiom used for barrier synchronization. Task variants improvement over all PLuTo variants varies from $1.2\times$ to $1.5\times$.

Benchmarks with inner barriers in smartfuse (single band) This category involves the two seidel kernels, blur-Roberts, fdtd-2d and jacobi-2d. We note that these benchmarks are difficult to optimize for multiple reasons: i) structured grid computations typically require skewing the iteration space to enable tiling (thus making all dependences forward); ii) coarse grained parallelism is enabled by the wavefront technique, thus making the outermost loop serial and the second outermost loop parallel, an inner barrier executed multiple times; iii) seidel kernels are not fully vectorizable; iv) all of these kernels, except for blur-Roberts, are bandwidth bound. PLuTo applies skewing, tiling and wavefronting. This yields a $4\times$ improvement w.r.t. the best sequential performance. Moreorver, since most of these kernels are memory bound, tiling is beneficial. The only benchmark wherein not tiling is necessary to obtain good performance is Blur-Roberts. Untiled versions consistently outperform, though by a small margin (12%), their tiled counterparts on Opteron, but produce an 8% slowdown on Core i7. Dynamic wavefront yields 8 partitions for both seidels and 9 partitions for jacobi-2d and fdtd-2d. On the other hand. Kernel blur-Roberts is parallelized with the combination of inter-band parallelism and partitioning, which yields 3 partitions for each of its two bands. Performance improvements for these benchmarks vary from a 34% slowdown on seidel-1d to a $2.6\times$ speedup on fdtd-2d. Tab. 5.3 shows the signatures used for the generated partitions before and after code generation.

Kernel	Tile Schedule	Signatures Used	#Tile Partitions	#Code Generated Partitions
fdtd-2d	(tt, tt + ti)	0, 2 - 8	8	9
jacobi-2d	(tt, 2 * tt + ti)	0 - 8	9	12
seidel-2d	(tt, tt + ti)	0, 2 - 8	8	8

Table 3: Partition summary for kernels fdtd-2d, jacobi-2d and seidel-2d. ID signatures follow the numbering of Fig. 10

We will further analyze seidel-2d's load imbalance and scalability properties in the following section, and conclude the experimental section with a case study comparing jacobi-2d's dynamic wavefront with diamond tiling.

5.4. Load Balancing

We now analyze the degree of parallelism and load balance achieved by our method enhanced with further ICC optimization (*task-opt*) and contrast it to PLuTo's heuristics, minfuse and smartfuse, which are parallelized with OpenMP's *for* work-sharing construct. We focus the analysis to the AMD Opteron platform, and select 3mm, seidel-2d, covariance and gemver as case studies. For each benchmark, we decompose its execution time into 3 parts: the sequential time, the parallel balanced time and the parallel unbalanced time. The breakdown of the execution time has been obtained through the following methodology:

- we use a version of OpenStream capable of generating traces. In particular, we consider the time spent in task creation (serial), task initialization, task execution and task seeking states, of which the last three are total time across all cores. The breakdown shown was converted from cycles to seconds considering the number of executing cores and their frequency. We consider the task execution time as parallel balanced, and the initialization and seeking times as unbalanced.
- PLuTo's variants are compiled with ICC v11, using the same flags as in the previous section, with the exception of -openmp which is replaced by -openmp-profile. The output obtained includes sequential time and per core minimum, average and maximum parallel time, both balanced and unbalanced. Here we define $time^{unbalanced}$ as $\max(time^{unbalanced}_{core\_id})$ and $time^{balanced}$ as $\max(time^{parallel}_{core\_id}) - time^{unbalanced}$.

Although we use different compiler versions for this set of experiments than in the previous section, we note that no significant difference was observed in the performance.[2]

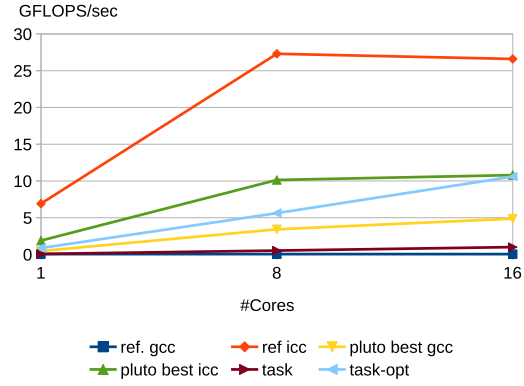

Figure 19: Performance Scalability for kernel 3mm

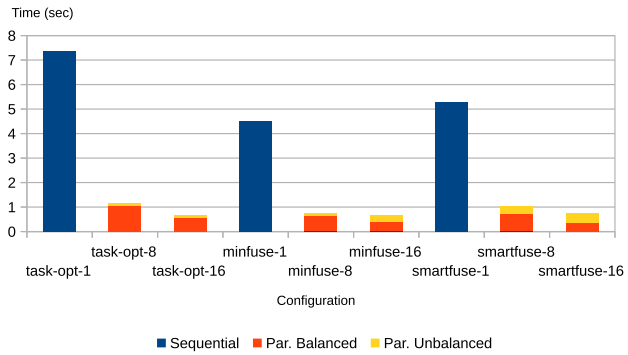

Figure 20: Time breakdown of 3mm kernel

3mm A dense linear algebra kernel such as this is not expected to benefit much from the data-flow parallel model; minfuse and smartfuse produce 6 tiled loop nests, but with different loop structure. In both cases, these are mapped to 6 parallel regions. In contrast, the inherent task parallelism is limited to the initializations of each of the 3 product matrices, followed by the computation of the left and right side matrices required for the final product. Figure 19 shows that *ref icc* and PLuTo's variants saturate at 8 cores. This is due to the 8×2MB distributed structure of Opteron's L2 cache, which forces many threads to access data from a distant core. Variant *task*, being fully compiled with GCC, is compute bound, but after further optimizing the task bodies with ICC (see the *task opt* graph) this is no longer the case, achieving linear scaling with the number of cores. Regarding the impact of barrier removal, we show 3mm's time breakdown in Figure 20. The performance achieved is slightly superior to both of PLuTO's. *task opt* suffers from load imbalance due to initialization

[2]The OpenMP profiling option has disappeared from recent versions of ICC, hence the use of v11 in this specific experiment.

and task creation overhead, as these steps run on a single core. This phenomenon is visible in most of our experiments. The fraction of the unbalanced execution time for OpenStream represents approx. 8% on 8 cores and 17% on 16 cores, whereas for PLuTo it varies between 15% and 50%.

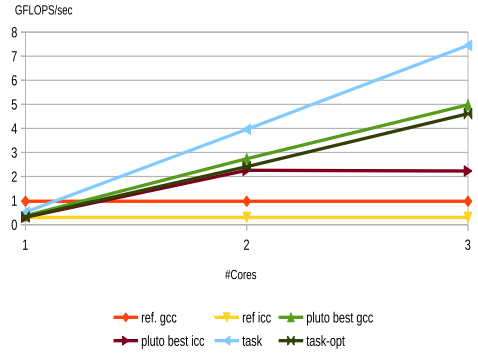

Figure 21: Performance Scalability for kernel seidel-2d

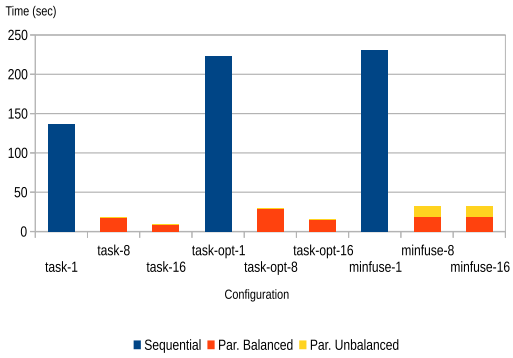

Figure 22: Time breakdown of seidel-2d kernel.

Seidel-2d This kernel becomes highly unbalanced when applying PLuTo's tiling heuristics combined with the (static) wavefront technique. It consists of a single statement and both tiling heuristics produce the same fusion structure. Fig. 21 shows the lack of scalability of this benchmark when compiled with GCC and ICC. PLuTo minfuse, on the other hand, scales well with GCC, but saturates at 8 cores with ICC. Further inspection revealed that task variant incurred in 50% less L1 accesses when scaling from 8 to 16 cores, whereas minfuse-ICC experienced a 15% reduction and minfuse-GCC only a 10%. L1 misses also reduced by 50% for task variant, 23% for minfuse-ICC and minfuse-GCC remained the same. Regarding the L2 cache behavior, task variant showed 25% less accesses than minfuse for ICC and GCC for 8 cores. For 16 cores, the L2 accesses difference between task variant and minfuse variant varied between 4% and 6%. L2 misses also reduced by 54% for task variant from 8 to 16 cores, but only 13% for minfuse-ICC and 30% for minfuse-GCC. When using all 16 cores the 3 variants exhibit the same number of L2 misses. Fig. 22 shows that the unbalanced fraction of time for minfuse-ICC is approx. 50% on 8 and 16 cores. Task variant (compiled only with GCC) spends 5% of its time in an unbalanced state for 8 cores and it increases to 10% on 16, while the balanced time is reduced to half as there are twice more cores for computing. Finally, Fig. 21 shows that speedups range from 1.5× to 3×.

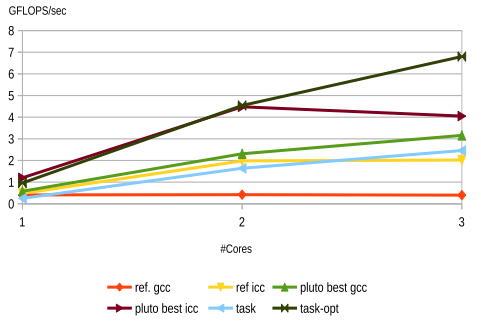

Figure 23: Performance Scalability for covariance kernel

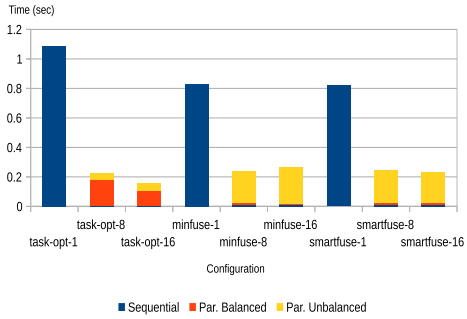

Figure 24: Time breakdown of Covariance kernel

Covariance PLuTo's heuristics and parallelization method limit the potential performance of this kernel. Both smartfuse and minfuse heuristics yield 7 parallel regions. While PLuTo variants saturate at 8 cores (See Figure 23) and reach a maximum of 4× speedup, *task-opt* continues scaling, although at a lower rate, achieving a final speedup of 7× w.r.t. single-threaded execution time. Figure 24 shows that approximately 96% of the total time is spent in unbalanced parallel execution with PLuTo-generated variants, while this is reduced by a factor of 4 with task-parallel execution.

5.5. Dynamic wavefront vs. Diamond tiling

We now compare our dynamic wavefront technique to diamond tiling [2], which aims at allowing concurrent start and improving the load imbalance in stencil computations that are parallelized with static wavefronts. It allows concurrent start along (at least) one of the faces of the iteration domain. It assumes that all tile dependences (in the transformed space) are unit vectors, thus similar to our usage of uniformizing dependences prior to the domain partitioning stage. A sufficient condition for diamond tiling is that the tile schedule must have the same direction as a face of the iteration domain.

We compare the scalability properties as well as the code characteristics of both approaches using the jacobi-2d kernel, a 5-pt stencil which meets diamond tiling's requirements. Diamond tiling variants were generated with PLuTo v.0.10, using flags –partlbtile (1-dimensional load balance tiling) and –parallel for OpenMP pragmatization, considering the default fusion heuristic (smartfuse). Experiments were conducted on the AMD Opteron and Intel Xeon (See Tab.5.2). The diamond tile variants were compiled with GCC 4.8 using flags -O3 -openmp and -ffast-math. We use the same tile sizes as the experiments reported in [2].

On the performance aspect, our dynamic wavefront outperforms diamond tiling on the two machines (Fig. 5.5 and Fig. 5.5). Both techniques scale very well on Intel processors. However, as in the previous experiments, diamond tiling neither scales beyond 8 cores on the Opteron, whereas the dynamic wavefront achieves a speedup of 11× w.r.t to its sequential performance. As AMD's Opteron has an L1 data cache of 16 KB, a tile of 32×32 on double precision and two arrays occupies most of it. Increasing the tile size to 64 does not improve performance for the task variant, but represents a 3 GFLOPS/sec increment for diamong tiling. On Intel's Xeon we see a similar tile size behavior, but without the Opteron's plateauing, and both techniques scale almost linearly. In general, our task variants will benefit more from having tasks with tiles that are slightly smaller than the L1 cache.

Regarding the code structure, diamond tiling's code still suffers from the inherent load imbalance of having a parallel loop nested within a sequential loop. Furthermore, [2] also states that, in practice, partial concurrent execution (only the outermost dimension) yields better performance than exploiting parallelism across all faces of the iteration domain due to code complexity, manifested as code explosion and numerous modulo conditions along the code (about 8000 lines and 400 modulo conditions for full dimensional concurrent start, but about 900 lines and 7 modulo conditions when using the 1-dimensional option). Two more features hinder the performance of this technique: conditions nested within the innermost loops and the tile shape. Both inhibit vectorization in many cases. On the contrary, partitioning for generating dynamic wavefronts produces a more compact code, about 160 lines for the same kernel and the modulo conditions appear mostly on the outermost dimensions.

Finally, we note that diamond tiling is not applicable to kernels such as fdtd-2d or seidel-2d, because of the nature of their dependences on the tiled (transformed) program. On the other hand, our framework seemlessly handles these kernels, as long as the processed tile dependences used for partitioning are uniform.

6. Related Work

A number of runtimes have been proposed for dynamic, task parallelism with inter-task dependences. In particular, the DAGuE runtime [6], the U. of Delaware codelet model [26] and the related SWARM environment [14], and the T* runtime of OpenStream [23] are all "feed-forward" or "argument fetch" data-driven models. In such runtimes, tasks notify their successors upon completion. While this increases the programming or

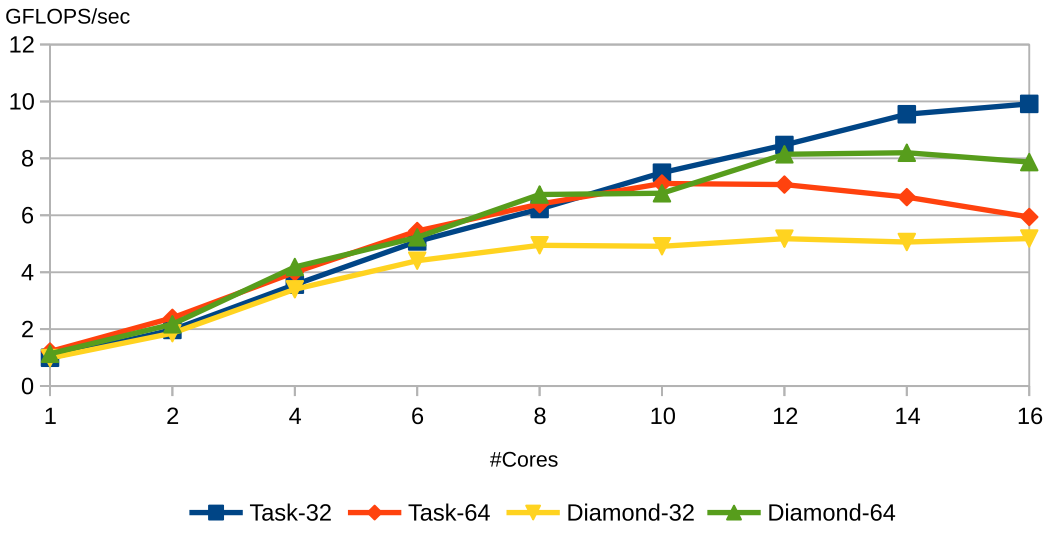

Figure 25: Performance of dynamic wavefront vs diamond tiling on AMD Opteron for tile sizes of 32 and 64

compilation burden—continuations may have to be exposed and passed explicitly, the dependence resolution algorithm can be fully distributed and its complexity is generally independent of the number of blocked tasks [6, 23]. This contrasts with a family of runtime systems where a dynamic dependence resolver identifies the ready tasks, such as the current runtimes supporting the StarSs [20] and CnC [7] languages.

DAGuE is unique in that it uses a symbolic representation of the acyclic dependence graph to avoid building it in extension. This approach is reminiscent of context-based data-flow execution and architectures, where the iteration space structure is directly embedded into the data-driven execution mechanics [29, 17]. While such symbolic approaches are more (local-) memory-efficient than explicit graph representations à la Star-PU, the exact benefits remain largely unknown compared to the fully dynamic dependence graphs of SWARM or T* (OpenStream). Nevertheless, the evaluation is orthogonal to our work which can make use from both approaches.

Moving on to polyhedral compilation, the most closely related technique is the one of Baskaran et al. [3]. Their approach also aims at generating tasks from a polyhedral compiler (PLuTO [5]), with the goal of mitigating load imbalance. The notable differences w.r.t. their work are: we resolve tile dependences entirely statically, whereas their DAGs are generated dynamically; our approach also includes a specific code generation step that takes care of not overloading the runtime; and we perform tile dependence pruning to reduce the number of synchronizations.

Our technique relies heavily on the partitioning of the iteration domain of a loop nest. Griebl et al. proposed the technique known as Index Set Splitting (ISS) [12] and Pugh and Rosser proposed Iteration Set Slicing (also ISS) [25]. Both techniques partition iteration domains based on dependence patterns (e.g., when a dependence changes direction, or based on transitive closure computations). Their technique is applied as a pre-processing step that allows to extract more parallelism, or to apply more aggressive affine transformations. We devised a form of ISS that operates—on tile dependences—after any affine transformation is performed and not before.

Finally, diamond tiling [2] and split tiling [13] enable concurrent tile execution, but use barriers between phases. These approaches do reduce imbalance induced by loop skewing and wavefront parallelization, but do not address the fundamental reason of load imbalance: the presence of barriers that oversynchronize bands of tiles. Our focus is the generation of more flexible inter-task synchronization instead of barriers. While we have evaluated it in the context of wavefront tile schedules, the approach could also be applied to the

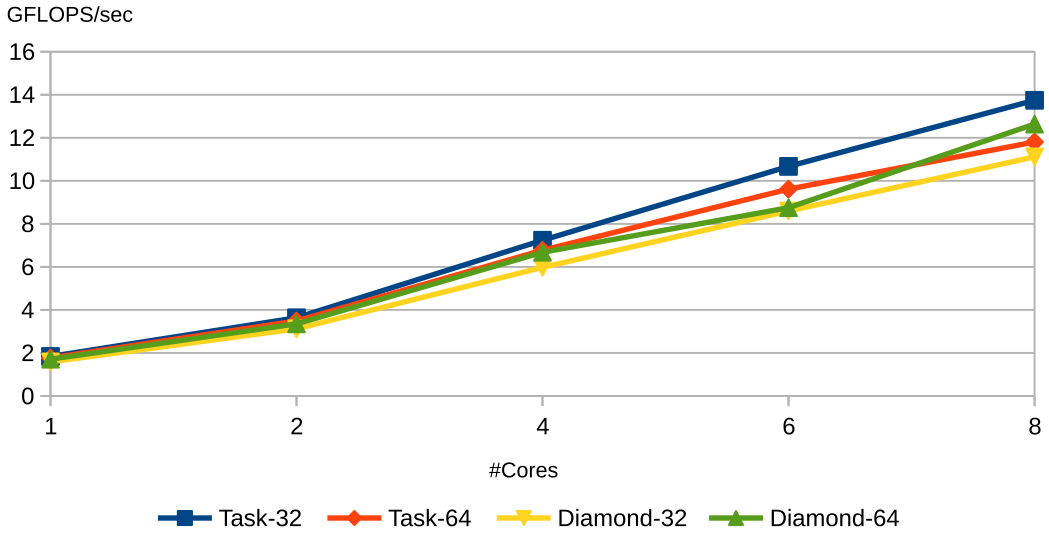

Figure 26: Performance of dynamic wavefront vs diamond tiling on Intel Xeon for tile sizes of 32 and 64

elimination of barriers associated with diamond and split tiling.

7. Conclusions and Future Work

We presented a systematic approach to compile a loop nest into concurrent, dependent tasks. We formulated a partitioning algorithm based upon the tile-to-tile dependences represented as affine polyhedra. This algorithm takes out much of the burden of runtime dependence enforcement, while preserving its load balancing and lightweight synchronization benefits. We implemented the algorithm in the PPCG research compiler, targeting the OpenStream data-flow language. Our results confirm the advantage of task-parallel execution over barrier-based, data-parallel patterns.

These results push for the generalization of the partitioning algorithm to more complex, irregular control flow. Further efforts to reduce code size should also be considered, implementing a hybrid approach where some of the partitioning occurs offline, while less performance-sensitive dependence patterns are deferred to conditional data-flow and implicit dependence resolution at run time.

References

[1] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann San Francisco, 2002.

[2] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 40. IEEE Computer Society Press, 2012.

[3] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. *ACM Sigplan Notices*, 44(4):219–228, 2009.

[4] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *PLDI*, June 2008.

[6] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[7] Z. Budimlič, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent collections. *Sci. Program.*, 18:203217, 2010.

[8] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

[9] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–496, 1997.

[10] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.

[11] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[12] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6), 2000.

[13] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 13–24. ACM, 2013.

[14] E. International. SWARM (SWift Adaptive Runtime Machine). `http://www.etinternational.com/index.php/products/swarmbeta`, 2014.

[15] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, March 2004.

[16] G. Kahn. The semantics of a simple language for parallel programming. In N. Holland, editor, *IFIP'94*, pages 471–475, 1974.

[17] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Trans. on Parallel Distributed Systems*, 17(10):1176–1188, 2006.

[18] S. P. Midkiff and D. A. Padua. Compiler generated synchronization for do loops. In *ICPP*, pages 544–551, 1986.

[19] S. P. Midkiff and D. A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on computers*, 36(12):1485–1495, 1987.

[20] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[21] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *International Journal on High Performance Computing Architecture*, 23(3):284–299, 2009.

[22] A. Pop and A. Cohen. Control-driven data flow. Technical Report RR-8015, INRIA, July 2012.

[23] A. Pop and A. Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9, 2013.

[24] L.-N. Pouchet. Polybench: The polyhedral benchmark suite, 2012.

[25] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In *Proceedings of the 11th international conference on Supercomputing*, pages 221–228. ACM, 1997.

[26] J. Suettlerlein, S. Zuckerman, and G. R. Gao. An implementation of the codelet model. In *Euro-Par*, pages 633–644, 2013.

[27] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010.

[28] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[29] I. Watson and J. R. Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, 1982.

A. Appendix

A.1. Blur-Roberts PLuTo tile with smartfuse

Fig.27 shows the code generated by the PoCC compiler using PLuTo's tiling with the smartfuse heuristic. Here we observe that in fusion is enabled by statement retiming, thus producing a parallel loop nested within the outer sequential loop.

A.2. Gemm kernel inter-band code

Fig.28 shows kernel gemm tiled with PPCG's implementation of the PLuTo tiling algorithm using the minfuse heuristic. Parallelization is performed with our inter-band idiom (cross barrier), which enables point-to-point synchronization between disjoint loops.

A.3. Dependence processing

Fig. 29 shows the evolution of the statement dependences to tile-level dependences, but which are still not in their uniform standard representation and which are (not) obviously parallel to the axes after the change of basis.

A.4. Jacobi-2d-imper dynamic wavefront code

Fig. 30-33 show the generated code for kernel jacobi-2d-imper. This code was generated with a tile size of 16 and has 9 signatures. However, due to the disjointness of some domain parts, 12 code partitions are generated in practice.

```
if ((_PB_N >= 3)) {
 for (c1 = 0; c1 <= floord((_PB_N + -2), 8); c1++) {
  if ((_PB_N >= 4)) {
   lb1 = max(0, ceild((((16 * c1) + (-1 * _PB_N)) + 2), 16));
   ub1 = min(floord((_PB_N + -2), 16), c1);
   #pragma omp parallel for private(c4, c3) firstprivate(lb1, ub1)
   for (c2 = lb1; c2 <= ub1; c2++) {
    if ((c1 == c2)) {
     #pragma ivdep
     #pragma vector always
     #pragma simd
     for (c4 = max(1, (16 * c1)); c4 <= min((_PB_N + -2), ((16 * c1) + 15)); c4++) {
      B[1][c4]=(A[1][c4]+A[1][c4-1]+A[1][1+c4]+A[1+1][c4]+A[1 -1][c4]+A[1 -1][c4-1]+
       A[1 -1][c4+1]+A[1 +1][c4-1]+A[1 +1][c4+1])/8.0;
     }
    }
   for (c3 = max(2, ((16 * c1) + (-16 * c2)));
        c3 <= min((_PB_N + -2), (((16 * c1) + (-16 * c2)) + 15)); c3++) {
    if ((c2 == 0)) {
     #pragma ivdep
     #pragma vector always
     #pragma simd
     for (c4 = 1; c4 <= 2; c4++) {
       B[c3][c4]=(A[c3][c4]+A[c3][c4-1]+A[c3][1+c4]+A[1+c3][c4]+A[c3-1][c4]+
        A[c3-1][c4-1]+A[c3-1][c4+1]+A[c3+1][c4-1]+A[c3+1][c4+1])/8.0;
      }
     }
    for (c4 = max(3, (16 * c2)); c4 <= min((_PB_N + -2), ((16 * c2) + 15)); c4++) {
     B[c3][c4]=(A[c3][c4]+A[c3][c4-1]+A[c3][1+c4]+A[1+c3][c4]+A[c3-1][c4]+
      A[c3-1][c4-1]+A[c3-1][c4+1]+A[c3+1][c4-1]+A[c3+1][c4+1])/8.0;
     A[(c3 + -1)][(c4 + -1)]=fabs(B[(c3 + -1)][(c4 + -1)]-B[(c3 + -1)+1][(c4 + -1)-1])+
      fabs(B[(c3 + -1)+1][(c4 + -1)]-B[(c3 + -1)][(c4 + -1)-1]);
     }
         if ((c2 >= ceild((_PB_N + -16), 16))) {
      A[(c3 + -1)][(_PB_N + -2)]=fabs(B[(c3 + -1)][(_PB_N + -2)]-B[(c3 + -1)+1][(_PB_N + -2)-1])+
       fabs(B[(c3 + -1)+1][(_PB_N + -2)]-B[(c3 + -1)][(_PB_N + -2)-1]);
     }
    }
   }
  }

  if (((_PB_N == 3) && (c1 == 0))) {
   B[1][1]=(A[1][1]+A[1][1 -1]+A[1][1+1]+A[1+1][1]+A[1 -1][1]+A[1 -1][1 -1]+
    A[1 -1][1 +1]+A[1 +1][1 -1]+A[1 +1][1 +1])/8.0;
  }
  if ((c1 >= ceild((_PB_N + -1), 16))) {
   if ((((_PB_N + 15) % 16) == 0)) {
    #pragma ivdep
    #pragma vector always
    #pragma simd
    for (c3 = max(2, (((16 * c1) + (-1 * _PB_N)) + 1));
         c3 <= (((16 * c1) + (-1 * _PB_N)) + 16); c3++) {
     A[(c3 + -1)][(_PB_N + -2)]=fabs(B[(c3 + -1)][(_PB_N + -2)]-B[(c3 + -1)+1][(_PB_N + -2)-1])+
      fabs(B[(c3 + -1)+1][(_PB_N + -2)]-B[(c3 + -1)][(_PB_N + -2)-1]);
    }
   }
  }
 }
}
```

Figure 27: Blur-Roberts kernel tiled with PLuTo smartfuse

```
#define min(x,y)    ((x) < (y) ? (x) : (y))
#define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
#define W 1
long T_0_T_1_0_SIZE = (floor((15 + ni)/16));
int T_0_T_1_0[T_0_T_1_0_SIZE] __attribute__((stream));
int rwin0[W];
int wwin0[W];
{

  for (int c1 = 0; c1 <= floord(ni - 1, 16); c1 += 1)
#pragma omp task  output(T_0_T_1_0[c1] << wwin0[W])
    for (int c2 = 0; c2 <= floord(nj - 1, 16); c2 += 1)
      for (int c3 = 16 * c1; c3 <= min(ni - 1, 16 * c1 + 15); c3 += 1)
        for (int c4 = 16 * c2; c4 <= min(nj - 1, 16 * c2 + 15); c4 += 1)
          C[c3][c4] *= beta;



  for (int c1 = 0; c1 <= floord(ni - 1, 16); c1 += 1)
#pragma omp task  input(T_0_T_1_0[c1] >> rwin0[W])
    for (int c2 = 0; c2 <= floord(nj - 1, 16); c2 += 1)
      for (int c3 = 0; c3 <= floord(nk - 1, 16); c3 += 1)
        for (int c4 = 16 * c1; c4 <= min(ni - 1, 16 * c1 + 15); c4 += 1)
          for (int c5 = 16 * c2; c5 <= min(nj - 1, 16 * c2 + 15); c5 += 1)
            for (int c6 = 16 * c3; c6 <= min(nk - 1, 16 * c3 + 15); c6 += 1)
              C[c4][c5] += ((alpha * A[c4][c6]) * B[c6][c5]);


}
#pragma omp taskwait
```

Figure 28: Code generated for gemm kernel with inter-band parallelism

```
0) [tsteps, n] -> { S_1[t, i, j] -> S_0[1 + t, i, 1 + j] : t >= 0 and t <= -2 +
        tsteps and i >= 1 and i <= -2 + n and j <= 999 and j >= 1 and j <= -3 + n;
        S_1[t, i, j] -> S_0[1 + t, 1 + i, j] : t >= 0 and t <= -2 + tsteps and i >= 1
        and i <= -3 + n and j <= 999 and j >= 1 and j <= -2 + n; S_1[t, i, j] -> S_0[1
        + t, i, j] : (t >= 0 and t <= -2 + tsteps and i >= 1 and i <= -2 + n and j <=
        999 and j >= 1 and j <= -2 + n) or (t >= 0 and t <= -2 + tsteps and i >= 1 and
        i <= -2 + n and j <= 999 and j >= 1 and j <= -2 + n); S_1[t, i, j] -> S_0[1 +
        t, i, -1 + j] : t >= 0 and t <= -2 + tsteps and i >= 1 and i <= -2 + n and j <=
        999 and j >= 2 and j <= -2 + n; S_1[t, i, j] -> S_0[1 + t, -1 + i, j] : t >= 0
        and t <= -2 + tsteps and i >= 2 and i <= -2 + n and j <= 999 and j >= 1 and j
        <= -2 + n; S_0[t, i, j] -> S_0[1 + t, i, j] : t >= 0 and t <= -2 + tsteps and i
        >= 1 and i <= -2 + n and j <= 999 and j >= 1 and j <= -2 + n; S_0[t, i, j] ->
        S_1[t, ii, jj] : t >= 0 and t <= -1 + tsteps and i >= 1 and i <= -2 + n and j
        >= 1 and j <= -2 + n and ii >= 1 and ii <= -2 + n and jj >= -1 + i + j - ii and
        jj <= 1 - i + j + ii and jj <= 999 and jj >= 1 and jj >= -1 - i + j + ii and jj
        <= 1 + i + j - ii and jj <= -2 + n; S_0[t, i, j] -> S_1[t, i, j] : t >= 0 and t
        <= -1 + tsteps and i >= 1 and i <= -2 + n and j <= 999 and j >= 1 and j <= -2 +
        n; S_1[t, i, j] -> S_1[1 + t, i, j] : t >= 0 and t <= -2 + tsteps and i >= 1
        and i <= -2 + n and j <= 999 and j >= 1 and j <= -2 + n }
1) [tsteps, n] -> { T_0[0, 0, i2, i3] -> T_0[0, 0, o2, o3] : i2 >= 0 and n >= 3
        and o3 <= 1 + i3 and o2 <= 1 + i2 and 2o2 <= i3 and 16o3 <= 30 + n + 32i2 and
        16i3 <= -5 + 2tsteps + n and 16o3 <= -4 + 2tsteps + n and 16i3 <= 29 + n + 32i2
        and 16o2 <= -1 + tsteps and 16i2 <= -2 + tsteps and o3 >= i3 and o2 >= i2 and
        16o3 <= 12 + n + 16i3 and 32o2 >= -27 - n + 16i3 and 16o3 <= 28 + n + 32o2 }
2) [tsteps, n] -> { T_0[0, 0, i2, i3] -> T_0[0, 0, o2, o3] : 16o3 <= -4 + 2tsteps
        + n and 16i2 <= -2 + tsteps and 16i3 <= 28 + n + 32i2 and 16o3 <= 30 + n + 32i2
        and i2 >= 0 and o3 >= 2o2 and o2 >= i2 and n >= 3 and 32o2 >= -26 - n + 16i3
        and o3 >= i3 and 16o3 <= 28 + n + 32o2 and 16i3 <= -6 + 2tsteps + n and o3 <= 1
        + i3 and o2 <= 1 + i2 and 16o2 <= -1 + tsteps and i3 >= 2i2 }
3) [tsteps, n] -> { T_0[0, 0, i2, i3] -> T_0[0, 0, i2, o3] : 16o3 <= -3 + 2tsteps
        + n and 16i2 <= -1 + tsteps and o3 >= i3 and 16o3 <= 29 + n + 32i2 and i2 >= 0
        and 16i3 <= -4 + 2tsteps + n and i3 >= 2i2 and n >= 3 and 16i3 <= 28 + n + 32i2
        and o3 <= 1 + i3 }
4) [tsteps, n] -> { T_0[0, 0, i2, i3] -> T_0[0, 0, o2, o3] : 16o3 <= -3 + 2tsteps
        + n and 16i2 <= -2 + tsteps and 16i3 <= 29 + n + 32i2 and 16o3 <= 31 + n + 32i2
        and i2 >= 0 and 2o2 <= i3 and 16o3 <= 29 + n + 32o2 and n >= 3 and o2 >= i2 and
        o3 >= i3 and 32o2 >= -27 - n + 16i3 and 16i3 <= -5 + 2tsteps + n and o3 <= 1 +
        i3 and o2 <= 1 + i2 and 16o2 <= -1 + tsteps }
[tsteps, n] -> { T_0[0, 0, i2, i3] -> T_0[0, 0, 1 + i2, o3] : o3 >= i3 and 16i2
    <= -17 + tsteps and 16i3 <= 29 + n + 32i2 and 16o3 <= 31 + n + 32i2 and i2 >= 0
    and i3 >= 2 + 2i2 and o3 <= 1 + i3; T_0[0, 0, i2, i3] -> T_0[0, 0, o2, 1 + i3]
    : 16o2 = -1 + tsteps and 16i2 = -1 + tsteps and 16i3 <= -19 + 2tsteps + n and
    8i3 >= -1 + tsteps and tsteps >= 1; T_0[0, 0, i2, i3] -> T_0[0, 0, i2, 1 + i3]
    : 16i3 <= -19 + 2tsteps + n and 16i2 <= -2 + tsteps and 16i3 <= 13 + n + 32i2
    and n >= 3 and i2 >= 0 and i3 >= 2i2; T_0[0, 0, i2, 1 + 2i2] -> T_0[0, 0, 1 +
    i2, 2 + 2i2] : i2 >= 0 and 16i2 <= -17 + tsteps and n >= 3 }
```

Figure 29: Jacobi-2d-imper dependence evolution from statement dependences to (disjoint) tile dependences prior to transitive reduction pruning

```
/* Band x dimension volumes. */
long S_1_0 = 1;
long S_1_1 = 1;
long S_1_2 = (floor((15 + tsteps)/16));
long S_1_3 = (floor((13 + 2*tsteps + n)/16));
long S_0_0 = 1;
long S_0_1 = 1;
long S_0_2 = (floor((15 + tsteps)/16));
long S_0_3 = (floor((12 + 2*tsteps + n)/16));

/* Band volumes. The variables declared here are
 used to declare stream windows and stream arrays */
long T_0_vol = S_1_2 * S_1_3;
long T_0_T_0_1_SIZE = S_1_2 * S_1_3;
int T_0_T_0_1[T_0_T_0_1_SIZE] __attribute__((stream));
int token1_1; int token1_2; int token1_3;
long T_0_T_0_2_SIZE = S_1_2 * S_1_3;
int T_0_T_0_2[T_0_T_0_2_SIZE] __attribute__((stream));
int token2_1; int token2_2; int token2_3;
{
  #pragma omp task  output(T_0_T_0_2[( 0 ) * S_1_3 + 1+0] << token2_2)
  for (int c4 = 0; c4 <= (n - 3) / 16 + 1; c4 += 1)
    for (int c5 = 0; c5 <= min(7, 8 * c4 + 6); c5 += 1)
      for (int c6 = max(2 * c5 + 1, -n + 16 * c4 + 2); c6 <= min(15, 16 * c4 + 14); c6 += 1)
        for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(16 * c4 + 15, n + c6 - 2); c7 += 1) {
          B[-2 * c5 + c6][-c6 + c7] =
            (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] + A[-2 * c5 + c6][-c6 + c7 - 1]) +
            A[-2 * c5 + c6][-c6 + c7 + 1]) + A[-2 * c5 + c6 + 1][-c6 + c7]) +
            A[-2 * c5 + c6 - 1][-c6 + c7]));
          if (c6 >= 2 * c5 + 2)
            A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
        }

  if (2 * ((tsteps - 1) % 16) + ((14 * tsteps + 15 * n + 2) % 16) <= 13 && (tsteps - 1) % 16 <= 6) {
  #pragma omp task \
    input(\
      T_0_T_0_1[( (tsteps-1)/16 ) * S_1_3 + (2*tsteps+n-3)/16] >> token1_1,\
      T_0_T_0_2[( (tsteps-1)/16 ) * S_1_3 + (2*tsteps+n-3)/16] >> token2_1)
    for (int c4 = (2 * tsteps + n - 3) / 16; c4 <= (tsteps + n - 3) / 8; c4 += 1)
      for (int c5 = max(-n + 8 * c4 + 2, -((tsteps - 1) % 16) + tsteps - 1);
          c5 < tsteps; c5 += 1)
        for (int c6 = max(((14 * tsteps + 15 * n + 2) % 16) +
                2 * tsteps + n - 18, -n + 16 * c4 + 2);
            c6 <= min(n + 2 * c5 - 1, 16 * c4 + 14); c6 += 1)
          for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(16 * c4 + 15, n + c6 - 2); c7 += 1) {
            if (n + 2 * c5 >= c6 + 2)
              B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
                A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
                A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
            A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
          }

  } else
  #pragma omp task\
  input(T_0_T_0_2[( (tsteps-1)/16 ) * S_1_3 + (2*tsteps+n-3)/16] >> token2_1)
  for (int c4 = (2 * tsteps + n - 3) / 16; c4 <= (tsteps + n - 3) / 8; c4 += 1)
    for (int c5 = max(-n + 8 * c4 + 2, -n + n / 2 + 8 * ((2 * tsteps + n - 3) / 16) + 1);
        c5 < tsteps; c5 += 1)
      for (int c6 = max(-n + 16 * c4 + 2, -((2 * tsteps + n - 3) % 16) + 2 * tsteps + n - 3);
          c6 <= min(n + 2 * c5 - 1, 16 * c4 + 14); c6 += 1)
        for (int c7 = max(16 * c4, c6 + 1); c7 <= min(n + c6 - 2, 16 * c4 + 15); c7 += 1) {
          if (n + 2 * c5 >= c6 + 2)
            B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
              A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
              A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
          A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
          }
```

Figure 30: Code generated for jacobi-2d-imper kernel with intra-band (dynamic wavefront) parallelism - Part I

```
for (int c2 = 1; c2 < (tsteps - 1) / 16; c2 += 1)
  for (int c3 = 2 * c2 + 2; c3 <= 2 * c2 + (n - 3) / 16; c3 += 1)
  #pragma omp task\
    input(T_0_T_0_1[( c2 ) * S_1_3 + c3] >> token1_1,\
          T_0_T_0_2[( c2 ) * S_1_3 + c3] >> token2_1)\
    output(T_0_T_0_1[( 1+c2 ) * S_1_3 + c3] << token1_2,\
          T_0_T_0_2[( c2 ) * S_1_3 + 1+c3] << token2_2)
    for (int c4 = c3; c4 <= c3 + (n - 3) / 16 + 1; c4 += 1)
      for (int c5 = max(16 * c2, -n + 8 * c4 + 2); c5 <= 16 * c2 + 15; c5 += 1)
        for (int c6 = max(16 * c3, -n + 16 * c4 + 2);
              c6 <= min(min(16 * c3 + 15, 16 * c4 + 14), n + 2 * c5 - 1); c6 += 1)
          for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(16 * c4 + 15, n + c6 - 2); c7 += 1) {
            if (n + 2 * c5 >= c6 + 2)
              B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
                A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
                A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
            A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
          }



for (int c2 = 1; c2 <= (tsteps - 1) / 16; c2 += 1) {
  if (tsteps >= 16 * c2 + 17) {
    #pragma omp task\
      input(T_0_T_0_1[( c2 ) * S_1_3 + 2*c2+1] >> token1_1,\
            T_0_T_0_2[( c2 ) * S_1_3 + 2*c2+1] >> token2_1)\
      output(T_0_T_0_2[( c2 ) * S_1_3 + 1+2*c2+1] << token2_2)
    for (int c4 = 2 * c2 + 1; c4 <= 2 * c2 + (n - 3) / 16 + 2; c4 += 1)
      for (int c5 = 16 * c2; c5 <= min(8 * c4 + 6, 16 * c2 + 15); c5 += 1)
        for (int c6 = max(max(32 * c2 + 16, -n + 16 * c4 + 2), 2 * c5 + 1);
             c6 <= min(32 * c2 + 31, 16 * c4 + 14); c6 += 1)
          for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(n + c6 - 2, 16 * c4 + 15); c7 += 1) {
            B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
              A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
              A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
            if (c6 >= 2 * c5 + 2)
              A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
          }
  } else
    for (int c3 = 2 * c2 + 1;
         c3 <= min((2 * tsteps + n - 3) / 16 - 1, 2 * c2 + (n - 3) / 16); c3 += 1)
    #pragma omp task\
      input(T_0_T_0_1[( (tsteps-1)/16 ) * S_1_3 + c3] >> token1_1,\
            T_0_T_0_2[( (tsteps-1)/16 ) * S_1_3 + c3] >> token2_1)\
      output(T_0_T_0_2[( (tsteps-1)/16 ) * S_1_3 + 1+c3] << token2_2)
      for (int c4 = c3; c4 <= c3 + (n - 3) / 16 + 1; c4 += 1)
        for (int c5 = max(-n + 8 * c4 + 2, 16 * c2);
             c5 <= min(8 * c4 + 6, tsteps - 1); c5 += 1)
          for (int c6 = max(max(2 * c5 + 1, 16 * c3), -n + 16 * c4 + 2);
               c6 <= min(min(n + 2 * c5 - 1, 16 * c3 + 15), 16 * c4 + 14); c6 += 1)
            for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(n + c6 - 2, 16 * c4 + 15); c7 += 1) {
              if (n + 2 * c5 >= c6 + 2)
                B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
                  A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
                  A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
              if (c6 >= 2 * c5 + 2)
                A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
            }
}
```

Figure 31: Code generated for jacobi-2d-imper kernel with intra-band (dynamic wavefront) parallelism - Part II

```
  for (int c2 = 1; c2 <= (tsteps - 1) / 16; c2 += 1)
  #pragma omp task\
    input(T_0_T_0_1[( c2 ) * S_1_3 + 2*c2] >> token1_1)\
    output(T_0_T_0_2[( c2 ) * S_1_3 + 1+2*c2] << token2_2)
    for (int c4 = 2 * c2; c4 <= 2 * c2 + (n - 3) / 16 + 1; c4 += 1)
      for (int c5 = 16 * c2; c5 <= min(min(tsteps - 1, 16 * c2 + 7), 8 * c4 + 6); c5 += 1)
        for (int c6 = max(2 * c5 + 1, -n + 16 * c4 + 2);
             c6 <= min(16 * c4 + 14, 32 * c2 + 15); c6 += 1)
          for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(16 * c4 + 15, n + c6 - 2); c7 += 1) {
            B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
              A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
              A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
            if (c6 >= 2 * c5 + 2)
              A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
          }

  for (int c2 = 0; c2 < (tsteps - 1) / 16; c2 += 1) {
    if (c2 == 0)
      for (int c3 = 2; c3 <= (n - 3) / 16; c3 += 1)
      #pragma omp task\
        input(T_0_T_0_2[( 0 ) * S_1_3 + c3] >> token2_1)\
        output(T_0_T_0_1[( 1+0 ) * S_1_3 + c3] << token1_2,\
               T_0_T_0_2[( 0 ) * S_1_3 + 1+c3] << token2_2)
        for (int c4 = c3; c4 <= c3 + (n - 3) / 16 + 1; c4 += 1)
          for (int c5 = max(-n + 8 * c4 + 2, 0); c5 <= 15; c5 += 1)
            for (int c6 = max(-n + 16 * c4 + 2, 16 * c3);
                 c6 <= min(min(16 * c3 + 15, 16 * c4 + 14), n + 2 * c5 - 1); c6 += 1)
              for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(16 * c4 + 15, n + c6 - 2); c7 += 1) {
                if (n + 2 * c5 >= c6 + 2)
                  B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
                    A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
                    A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
                A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
              }

  #pragma omp task\
    input(T_0_T_0_2[( c2 ) * S_1_3 + 2*c2+(n-3)/16+1] >> token2_1)\
    output(T_0_T_0_1[( 1+c2 ) * S_1_3 + 2*c2+(n-3)/16+1] << token1_2,\
           T_0_T_0_2[( c2 ) * S_1_3 + 1+2*c2+(n-3)/16+1] << token2_2)
    for (int c4 = 2 * c2 + (n - 3) / 16 + 1; c4 <= 2 * c2 + 2 * ((n - 3) / 16) + 2; c4 += 1)
      for (int c5 = max(-n + 16 * c2 + n / 2 + 8 * ((n - 3) / 16) + 9, -n + 8 * c4 + 2);
           c5 <= 16 * c2 + 15; c5 += 1)
        for (int c6 = max(-n + 16 * c4 + 2, -((n - 3) % 16) + n + 32 * c2 + 13);
             c6 <= min(min(n + 2 * c5 - 1, 16 * c4 + 14), -((n - 3) % 16) + n + 32 * c2 + 28);
             c6 += 1)
          for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(16 * c4 + 15, n + c6 - 2); c7 += 1) {
            if (n + 2 * c5 >= c6 + 2)
              B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
                A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
                A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
            A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
          }
  }
```

Figure 32: Code generated for jacobi-2d-imper kernel with intra-band (dynamic wavefront) parallelism - Part III

```
  for (int c2 = 0; c2 < (tsteps - 1) / 16; c2 += 1)
  #pragma omp task\
    input(T_0_T_0_2[( c2 ) * S_1_3 + 2*c2+(n-3)/16+2] >> token2_1)\
    output(T_0_T_0_1[( 1+c2 ) * S_1_3 + 2*c2+(n-3)/16+2] << token1_2)
    for (int c4 = 2 * c2 + (n - 3) / 16 + 2; c4 <= 2 * c2 + (n - 3) / 8 + 2; c4 += 1)
      for (int c5 = max(-n + 16 * c2 + n / 2 + 8 * ((n - 3) / 16) + 17, -n + 8 * c4 + 2);
           c5 <= 16 * c2 + 15; c5 += 1)
        for (int c6 = max(-n + 16 * c4 + 2, -((n - 3) % 16) + n + 32 * c2 + 29);
             c6 <= min(n + 2 * c5 - 1, 16 * c4 + 14); c6 += 1)
          for (int c7 = max(16 * c4, c6 + 1); c7 <= min(n + c6 - 2, 16 * c4 + 15); c7 += 1) {
            if (n + 2 * c5 >= c6 + 2)
              B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
                A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
                A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
            A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
          }

#pragma omp task\
  input(T_0_T_0_2[( 0 ) * S_1_3 + 1] >> token2_1)\
  output(T_0_T_0_2[( 0 ) * S_1_3 + 1+1] << token2_2)
  for (int c4 = 1; c4 <= (n - 3) / 16 + 2; c4 += 1)
    for (int c5 = 0; c5 <= min(15, 8 * c4 + 6); c5 += 1)
      for (int c6 = max(max(2 * c5 + 1, -n + 16 * c4 + 2), 16);
           c6 <= min(31, 16 * c4 + 14); c6 += 1)
        for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(n + c6 - 2, 16 * c4 + 15); c7 += 1) {
          B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
            A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
            A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
          if (c6 >= 2 * c5 + 2)
            A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
        }

  if (2 * ((tsteps - 9) % 16) + ((n - 3) % 16) >= 14 && (tsteps - 9) % 16 <= 7)

#pragma omp task\
  input(T_0_T_0_2[( (tsteps-1)/16 ) * S_1_3 + 2*((tsteps-1)/16)+(n-3)/16+1] >> token2_1)\
  output(T_0_T_0_2[( (tsteps-1)/16 ) * S_1_3 + 1+2*((tsteps-1)/16)+(n-3)/16+1] << token2_2)
    for (int c4 = (2 * tsteps + n - 3) / 16 - 1;
         c4 < 2 * tsteps - 2 * (15 * tsteps / 16) + 2 * ((n - 3) / 16) + 1; c4 += 1)
      for (int c5 = max(-n + 8 * c4 + 2,
           8 * tsteps + 7 * n + n / 2 - 8 * ((14 * tsteps + 15 * n + 2) / 16) - 15);
           c5 < tsteps; c5 += 1)
        for (int c6 = max(((14 * tsteps + 15 * n + 2) % 16) + 2 * tsteps + n - 34,
             -n + 16 * c4 + 2);
             c6 <= min(min(n + 2 * c5 - 1, 16 * c4 + 14),
               ((14 * tsteps + 15 * n + 2) % 16) + 2 * tsteps + n - 19); c6 += 1)
          for (int c7 = max(c6 + 1, 16 * c4); c7 <= min(n + c6 - 2, 16 * c4 + 15); c7 += 1) {
            if (n + 2 * c5 >= c6 + 2)
              B[-2 * c5 + c6][-c6 + c7] = (0.2 * ((((A[-2 * c5 + c6][-c6 + c7] +
                A[-2 * c5 + c6][-c6 + c7 - 1]) + A[-2 * c5 + c6][-c6 + c7 + 1]) +
                A[-2 * c5 + c6 + 1][-c6 + c7]) + A[-2 * c5 + c6 - 1][-c6 + c7]));
            A[-2 * c5 + c6 - 1][-c6 + c7] = B[-2 * c5 + c6 - 1][-c6 + c7];
          }
}
#pragma omp taskwait
```

Figure 33: Code generated for jacobi-2d-imper kernel with intra-band (dynamic wavefront) parallelism - Part IV