On the Parallel Execution Time of Tiled Loops

Karin Högstedt, Member, IEEE, Larry Carter, Fellow, IEEE, and Jeanne Ferrante, Senior Member, IEEE

Abstract—Many computationally-intensive programs, such as those for differential equations, spatial interpolation, and dynamic programming, spend a large portion of their execution time in multiply-nested loops that have a regular stencil of data dependences. Tiling is a well-known compiler optimization that improves performance on such loops, particularly for computers with a multilevel hierarchy of parallelism and memory. Most previous work on tiling is limited in at least one of the following ways: they only handle nested loops of depth two, orthogonal tiling, or rectangular tiles. In our work, we tile loop nests of arbitrary depth using polyhedral tiles. We derive a prediction formula for the execution time of such tiled loops, which can be used by a compiler to automatically determine the tiling parameters that minimize the execution time. We also explain the notion of rise, a measure of the relationship between the shape of the tiles and the shape of the iteration space generated by the loop nest. The rise is a powerful tool in predicting the execution time of a tiled loop. It allows us to reason about how the tiling affects the length of the longest path of dependent tiles, which is a measure of the execution time of a tiling. We use a model of the tiled iteration space that allows us to determine the length of the longest path of dependent tiles using linear programming. Using the rise, we derive a simple formula for the length of the longest path of dependent tiles in rectilinear iteration spaces, a subclass of the convex iteration spaces, and show how to choose the optimal tile shape.

Index Terms—Tiling, blocking, compiler optimization, parallel compilers.

1 INTRODUCTION

Many computationally intensive programs spend a high percentage of their execution time in nested loops that have a regular stencil of data dependences between the iterations (see Figs. 1a and 1b for an example). Obtaining high performance on such loops requires both data locality and efficient use of parallel resources. Tiling [43], [33], [24], [44], [32] is a well-known compiler optimization that is effective at achieving both goals. Conceptually, given a loop nest, tiling operates on the iteration space graph defined by that loop nest. For a loop nest of depth \( K \), the iteration space graph is a polyhedron\(^1\) in \( K \)-dimensional space. The tilings we consider partition this space into a collection of parallelepipeds\(^2\) which are then scheduled for atomic execution (see Fig. 1 for a two-dimensional example).

The execution of a tile is said to be atomic if all input data needed for the execution of any iteration within the tile is available before the execution of the tile begins, and all output data is communicated after the execution completes. See [46] for nonatomic tilings.

We also refer to one of the dimensions in a \( K \)-dimensional iteration space as the vertical dimension. All tiles are assumed to have \( K - 1 \) pairs of vertical sides, thus enabling columns or stacks of tiles to form, which facilitates an efficient assignment of tiles to processors. The tiling parameters that define a tiling are the tile size \( \Pi_{d=1}^K w_d \), where \( w_d \) is the width of a full tile in dimension \( d \), and the tile shape (the set of all \( s_{i}^{e} \), where \( s_{i}^{e} \) is the slope in dimension \( d \) of the nonvertical tile side \( e \)).

There are two well-known benefits of tiling. Tiling can significantly increase the data locality within a code and it also enables more straightforward parallelization. We discuss these advantages, together with the legality requirements of a tiling next. Then, in Section 2, we discuss how the tiling choice affects the execution time of the program. In this section, we also discuss the impacts of tiling at multiple levels of the memory hierarchy and some about block-cyclic distribution of stacks to processors. In Sections 3, 4, 5, and 6, we then introduce some tools used in Section 7 to determine the length of the longest path of dependent tiles. In this section, we also give a formula for predicting the execution time of a tiled iteration space. In Section 8, we then show how to choose the optimal tile shape. Finally, we present related work in Section 9 and conclude in Section 10.

1.1 Locality, Parallelism, and Legality

Tiling enables data locality when the tile size ensures that the necessary data fit in a given level of the memory hierarchy. For instance, consider the loop nest in Fig. 1 and its resulting iteration space graph. As the code is written in Fig. 1a, all the data needed to calculate the \( n \) iterations in the first row of the iteration space graph will be accessed before

\[d \text{ rows, } e \text{ columns}\]

\[s_{i}^{e} \text{ (the set of all } s_{i}^{e} \text{, where } s_{i}^{e} \text{ is the slope in dimension } d \text{ of the nonvertical tile side } e \text{.)}

1. A polyhedron is a solid formed by plane faces.
2. A parallelepiped is a 2\( K \)-faced, \( K \)-dimensional polyhedron, all of whose faces are \( K - 1 \) dimensional parallelepipeds lying in pairs of parallel hyperplanes. A two-dimensional parallelepiped is a parallelogram.

References

- K. Högstedt is with AT&T Labs-Research, 180 Park Ave., Florham Park, NJ 07932. E-mail: karin@research.att.com.
- L. Carter and J. Ferrante are with the Department of Computer Science and Engineering, University of San Diego, 9500 Gilman Ave., Mail Stop 0114, La Jolla, CA 92093-0114. E-mail: {carter, ferrante}@cs.ucsd.edu.
- For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 115930.
an iteration \(2.j\) in the second row will use the result computed in iteration \((1.j)\). If the amount of data needed to calculate these \(M\) iterations is too large to fit in the memory level in question, the data calculated in iteration \((1.j)\) will have been replaced in that memory, and is no longer available when it is needed by iteration \((2.j)\). When \((2.j)\) is executed, the results computed in \((1.j)\) will therefore have to be brought in from slower memory, resulting in a memory miss.

If, on the other hand, we execute the loop as shown in Fig. 1c, i.e., the tiled version with \(n\) by \(m\) tiles, only the data needed to calculate the first \(m\) iterations in the first row of the iteration space graph will be accessed before iteration \((2.j)\) is executed. Therefore, if \(m\) is small enough so that all that data fit in memory, the result from \((1.j)\) will still be available, thus avoiding an access to slower memory. The tile size should not however be chosen too small since overheads that are independent of the tile size, such as loop overhead and communication start-up costs, then will dominate the execution time.

Tiling also enables efficient parallelization since independent tiles can simultaneously be executed on different processors with relatively low communication overhead. Assume, for example, that there are multiple processors available when executing the loop nest from Fig. 1a. One (impractical) approach would be to schedule the iterations to the processors one at a time in an on-demand fashion. Depending on the exact assignment of iterations to processors, communication overhead may be high. If, on the other hand, the loop is tiled as in Fig. 1c, all the iterations within each tile are executed by the same processor. This allows the data transferred from one processor to another to be bundled together. This results either in fewer messages needed to be sent (message-passing machines), or fewer synchronization messages needed (shared memory machines).

Not every tiling is legal, however. Throughout the iteration space, there are dependences between the tiles. These dependences are called *tile dependences* [21]. The tile dependences are imposed by the *iteration dependences*, i.e., the loop-carried data dependences between iteration points [45]. Depending on the tile shape, a chain of dependences between tiles is either cyclic or acyclic. Any tiling that results in a cyclic chain of tile dependences is *illegal* and all other tilings are legal. In [24], it was shown that all tilings where all iteration dependences cross a tile surface in the same direction are guaranteed to be legal. See Fig. 2 for examples of an illegal and a legal tiling. The iteration dependences and the tile dependences are indicated with solid and dashed arrows, respectively. The two iteration dependences that cause the tiling to be illegal by not crossing the tile surface in the same direction are indicated in bold.

This is a sufficient, but not necessary, condition for legality. In Fig. 3a, we give an example of a legal tile shape which does not conform to the legality condition introduced in [24]. The characteristics of this tiling that make it possible to be legal, in spite of the iteration dependences crossing the tile surfaces in different directions, are 1) one of these iteration dependences is longer than the tile width in the horizontal dimension and 2) there is no iteration dependence with a negative component in this dimension. Note that this tiling, or grouping of iterations, corresponds to a “staggered” tiling using a tile shape that does conform to the legality condition (Fig. 3b). It is however, not always possible to find such a tiling. The scheduling of iterations in Fig. 3c, for example, cannot be achieved using a tiling that conforms to the legality condition. Such tilings have been considered by others, particularly in the context of improving processor pipelining in the innermost loops [8]. But like most other previous work, we do not consider tilings that do not conform to the legality condition introduced by [24].
2 Tile Shape versus Execution Time

To calculate the execution time of an iteration space graph, we make the assumption that the execution time of a tile is directly related to the size of the tile, i.e., the number of iterations included in the tile. We work in a continuous, real-number version of the discrete iteration space graph [21]. We call this real space the iteration space. The volume of a tile in the iteration space is approximately proportional to the number of iterations in a tile. The volume of the tile is therefore strongly correlated with the execution time of the tile.

The parallel execution time of a tiled loop nest depends on the dependences between the tiles. If there is a tile dependence between two tiles, those two tiles have to be executed sequentially, independent of the amount of available parallelism. The execution time of the iteration space graph can therefore never be any shorter than the amount of time it takes to execute these two tiles sequentially. In fact, the parallel execution time of a tiled iteration space graph can never be shorter than the amount of time it takes to execute all the tiles on the longest path of dependent tiles within the iteration space graph.

Definition 1. A path of dependent tiles within an iteration space graph is a sequence of tiles, where all but the first tile is dependent on the previous tile. A longest path of dependent tiles is any path of all such paths that takes the longest to execute. We refer to any one of the longest paths of dependent tiles as the longest path of dependent tiles.

The block distribution parallelization strategy allocates one stack of tiles to each processor. It is not hard to show that for block distribution, the time it takes to execute the longest path of dependent tiles is the parallel execution time. When using block-cyclic distribution, however, this is not necessarily the case. When a processor is executing more than one stack, the time at which its second stack can start to execute is determined by the maximum of its first stack’s finishing time, and the finishing time of the tiles upon which its second stack depend. These two cases are illustrated in Fig. 4, where there are six stacks executed by six processors in a total of five phases. The two shaded paths in Fig. 4a and Fig. 4b are the sequences of tiles expected to take the longest to execute: Fig. 4a shows the case in which each processor has some idle time after finishing executing one stack and before it starts with its next. This idle time is due to the fact that the tile dependences between processor $p'_i$’s stack in Phase $i - 1$ and processor $p_i$’s stack in Phase $i$ have not yet been satisfied. Fig. 4b shows the case in which there is no such idle time. In this paper, we will concentrate on the case of block distribution, and refer the reader to [21] for a more detailed discussion on block-cyclic distribution. Using block distribution of stacks to processors, the problem of minimizing the execution time can be translated into the problem of choosing the tiling parameters such that the resulting tile dependences create the shortest possible longest path of dependent tiles. We therefore, explore how the tile shape, given a fixed tile size, affects the number of tiles on the longest path of dependent tiles and, thus, likely the execution time of a tiled iteration space with a couple of examples. The method of prediction used in this discussion is only meant to be a rough estimate, building on the assumption that the execution time is strongly correlated with the number of tiles on the longest path of dependent tiles. We present a more detailed method of prediction which, for example, takes communication into account, in Section 7.

2.1 Single Level Tilings

Consider the tilings in Fig. 5. The iteration dependences are such that all four tilings are legal, and such that the tile dependences are given by the arrows, e.g. $d_1 = (1, 0)$ and $d_2 = (1, 2)$. In addition, the volume of the tiles in each case is the same. In Fig. 5d, however, there are fewer tiles on the longest path of dependent tiles than in any other case. The iteration space in Fig. 5d, therefore, executes in a shorter amount of time than any of the others, under the assumptions previously mentioned. The intuitive reason

![Fig. 4. Block-cyclic distribution of stacks to processors.](image)

![Fig. 5. The same iteration space tiled using an increasing tile slope from left to right.](image)
for this difference is that in Fig. 5a, only one processor can start at time zero, namely the processor executing the first tile in the first column. Due to the tile dependences, the processor executing the next tile, be it the second tile in the first column or the first tile in the second column, has to be idle until the first tile is completed. In Fig. 5d, on the other hand, all processors can start executing simultaneously since the first tile in each stack does not depend on the output of any other tile. The first tile in all columns can thus start at time zero and no processor needs to be idle. In fact, since the longest path of dependent tiles consists of 10 tiles in Fig. 5a, and only six tiles in Fig. 5d, a rough estimate of the parallel execution time would be that the tiling in Fig. 5a should perform about $\frac{10}{2} - 1 = 67\%$ worse than the one in Fig. 5d. And tiling the iteration space using square tiles as in Fig. 5b, results in eight tiles on the longest path of dependent tiles, and will thus perform about $\frac{5}{2} - 1 = 33\%$ slower compared to the tiling in Fig. 5d. Experimental results that confirm these estimates has been presented in [21]. These observations are also consistent with the results in Section 8 stating that when tiling for a single level of the memory hierarchy the optimal tile shape is achieved by using the smallest legal tile shape which results in a vertical longest path of dependent tiles.

### 2.2 Multilevel Tilings

When we tile for multiple levels in the memory hierarchy, however, this result is no longer valid. For example, if we tile for cache and registers, a smaller slope is often more advantageous. This is because the argument used in the example in Fig. 5 makes the assumption that the execution time of a full tile is the same in all four tilings. This is not the case when we tile for multiple levels of the memory hierarchy. By partitioning the tiles into yet smaller tiles, the longest path of dependent (second-level) tiles changes with the shape of the first-level tiles. The execution time of the first-level tiles is hence different in the four cases. Note that the tiles from one level of tiling become the iteration space for the next level. In these terms, the execution time of the iteration space at each level depends on the relationship between the shape of the tile and the shape of the iteration space at that level. See Fig. 6 for an example.

Fig. 6 shows the same four tilings as in Fig. 5 where each tile has been further tiled into smaller tiles. In all four cases, the same tile slope has been used for the second level of tiling. This is also the most beneficial tile slope, assuming the dependence vectors $\hat{d}_1 = (1, 0)$ and $\hat{d}_2 = (1, 2)$. Recall that when tiling for a single level of the memory hierarchy Fig. 5d was the best tiling of the four and Fig. 5a was the worst. When we tile for two levels, however, the shape of the first-level tiles is not the most favorable for tiling a second time. Of the four cases, the tiling in Fig. 6a yields the fewest number of tiles on the longest path of dependent, second-level tiles. Which of the four cases yields the best performance depends, however, on the number of tiles on the longest path of dependent tiles at both levels. Consider the tiling in Fig. 6a. Within each first-level tile (the iteration space of the second level), the longest path of dependent, second-level tiles consists of three full tiles and two half tiles. Assuming at least 4-way parallelism at the second level of tiling, the expected parallel execution time of a first level tile is therefore $3 + 2 \times 0.5 = 4$ times the execution time of a second-level tile. The number of first-level tiles on the longest path of dependent tiles in the original iteration space in Fig. 6a is 10. The total expected parallel execution time of the iteration space is therefore $10 \times 4 = 40$ times the execution time of a second-level tile. Similar calculations in the other three cases yield execution times of $8 \times 5.5 = 44$, $6.5 \times 6.6 = 43$, and $6 \times 7 = 42$ times the execution time of a second-level tile in Figs. 6b, 6c, and 6d, respectively. Again, comparing the performance of Fig. 6a and Fig. 6b to Fig. 6d, we see that the expected performance of Fig. 6a is $1 - \frac{44}{40} = 5\%$ better than that of Fig. 6d and the performance of Fig. 6b is $\frac{44}{42} - 1 = 5\%$ worse than the one of Fig. 6d.

The examples in Figs. 5 and 6, illustrate quite well the difficulties and importance of choosing the tile shape wisely. It is obvious that the tile shape affects the length of the longest path of dependent tiles, but we have yet to determine how. We will now introduce some tools in Section 3, 4, 5, and 6 that will help us in Section 7, where we calculate the length of the longest path of dependent tiles as a function of the tile shape.

### 3 Number of Tiles on the Longest Path of Dependent Tiles

To simplify the processes of determining the number of tiles on the longest path of dependent tiles, we want our model of the iteration space to have the following properties:

4. We assume the execution time of a tile to be roughly proportional to the tile size.
5. Please see the last paragraph of Section 6 for a discussion on the tiling of the partial tiles.
6. We always tile the iteration spaces into $P$ columns if there are $P$ processors available.
Property 1. The $L_1$-norm distance $\|x - y\|_1$ between two vertices of the iteration space polyhedron (i.e., extreme points of the polyhedron) should be equal to the sum of the volume of the tiles on the longest path of dependent tiles between those points. Due to partial tiles, this distance is not always an integer.

Property 2. The volume of each stack (column) of tiles and, thus, the tiled iteration space, should be the same as in the original iteration space.

The first property permits us to determine the number of tiles on the longest path of dependent tiles in a tiled iteration space using linear programming (See [23]). This is a peculiar property since we want a linear distance to be equal to a volume. This can be achieved if we ensure that the extreme points selected by the linear programming algorithm always occur at the midpoint of the top and bottom surface of a single unit cube tile. For example, the $L_1$-norm distance between a unit cube's two extreme points $(0.5, 0.5, 0)$ and $(0.5, 0.5, 1)$ is then equal to $(0.5 - 0.5) + (0.5 - 0.5) + (1 - 0) = 1$. This choice gives the desired property not only for single tiles, but also for paths of dependent tiles.

If the second property does not hold, then if the difference in volume is large, it would be likely that the difference between the execution time determined using the model and the actual execution time also is large.

The models used in previous research do not have these properties. The volume of the iteration space using the model in [22] exactly corresponds to the volume of the original iteration space, but the model does not fulfill the first requirement since the facets bounding the iteration space can intersect at any point in a stack. Using the model in [18] might result in an arbitrarily large difference in the volume of a stack compared to the original iteration space.

We introduce a new model that fulfills the above two properties. The iteration space needs to be transformed so that the linear programming constraints can be chosen such that Property 1 is satisfied. The necessary transformation of the iteration space is illustrated in Fig. 7.

Fig. 7a shows a sample iteration space using the model of partial tiles presented in [22]. The partial tiles are shaded. Fig. 7b shows how the partial tiles are transformed to our model. In our model the bottom and top of a stack is always horizontal. The height of a stack is determined by making a horizontal cut so that the volume cut off is equal to the volume added by the cut, i.e. in the blow-up in Fig. 7b, the area of the dark shading is equal to the area of the light shading. Note that the total area shaded in Fig. 7a is equal to the total area shaded in Fig. 7b, i.e., this transformation preserves the volume of the stacks and, thus, satisfies Property 2. However, also note that after this transformation the slopes of the different facets and even the number of facets might change. In Fig. 7c, we show the transformed iteration space. In Fig. 7d, we show how the constraints of the linear programming problem are determined. For all dimensions the slope between two stacks is equal to the slope of the line between the midpoints of the two stacks in that dimension. Although, we use the model in Fig. 7d in this paper, we might however still draw the iteration space as in Fig. 7a for simplicity.

A comparison to other models is given in Fig. 8. In Fig. 8a, the volume of each stack is chosen such that the volume of the light shading is equal to the volume of the dark shading. In Fig. 8b, the volume of partial tiles is chosen as $p_K \prod_{i=1}^{j+K} f_{i0}$ where $f_{i0}$ is the length of a full tile in dimension $d$ and $p_K$ is the length of the partial tile in the vertical dimension. Reference [21] proves an upper and lower bound on the difference in the number of tiles on the longest path of dependent tiles between our model and the model used in [22].

4 Communication

Apart from the number of tiles on the longest path of dependent tiles, the execution time also depends on the inter-processor communication time. Let a tile surface be the data that needs to be communicated between one tile and a neighbor.8 Communicating a tile surface involves transferring the data in the tile surface from the sending processor to the receiving. On a shared-memory machine it also involves locking and unlocking a mutex.9 The process of communicating a tile surface consists of three stages. 1) The sender does some amount of work to "start" the commu-

---

8. Note that there is a tile surface also between two neighboring tiles executed on the same processor. The communication time of this tile surface is zero since the data is already available at the receiving (i.e., sending) processor.

9. A mutex is a mechanism used to provide synchronization barriers in parallel codes. The mutex is locked before entering a critical section requiring atomic execution, and unlocked after exiting the section.
communication. Depending on the architecture, the amount of work can vary drastically. It can include anything from unlocking a mutex to actually moving the data. 2) The message is “in transition” before it reaches the receiver. This stage consists of the time that both the receiver and the sender can spend performing other tasks. 3) The receiver does some amount of work to get access to the message. This can include the locking of a mutex or unpacking data. Remember that depending on the architecture, any one of these three parts may not exist, i.e., take zero amount of time.

In our communication model, we divide communication into two parts; the part of the communication that can be overlapped by computation by both processors, and the part that cannot. The sum of the amount of time from part 1 and 3 is what we call the nonoverlappable communication time; at least one of the processors can do no other work during this time. This nonoverlappable communication time is considered to be part of the execution time of the tile that was executed on the sender. It is accounted for in the parameter $O$ in (5) presented in Section 7. The error incurred by charging the sender time that actually was spent by the receiver, is negligible since all processors (except the last and the first one) will be both sending and receiving. See Fig. 9 for an example. Notice that even though the first/last processor actually does not receive/send data, assuming they do only affects the longest path of dependent tiles (marked by the long arrow) by one send and one receive (circled).

Let $E_i$ be the execution time of a full tile. We define a parameter $c_d$ such that $c_dE_i$ is equal to the overlappable communication time for data communicated in dimension $d$ (see Fig. 9). So, $c_d$ is equal to zero when the architecture does not allow any overlap of communication by computation. This does not mean that the communication cost is zero, merely that the processor is busy communicating and the communication cost is therefore included in the execution time of the tile. For example, $c_d=0.25$ if the architecture allows for 0.25$E_i$ seconds of the communication of one full tile surface to be overlapped by computation. In this case, immediately after a tile is completed, the execution of the next tile in the same stack may begin, but the dependent tile in the adjacent stack has to wait for 0.25$E_i$ seconds. The reason the value of $c$ varies with the dimension is that the amount of data needed to be communicated varies from dimension to dimension. In particular, since we assume all tiles in each stack to be executed by the same processor, no communication is needed in the vertical $K$th dimension, and we have $c_K = 0$.

10. The overlappable communication corresponds to the latency, and the nonoverlappable to the overhead in the LogP model [16].
6 Rectilinear Iteration Spaces

As was shown in [23], we can formulate the problem of finding the end points of the longest path of dependent tiles in a general convex iteration space as a linear programming problem (restated in Section 7). We can therefore, find the length of the longest path of dependent tiles in polynomial time for general convex iteration space shapes. However, this only gives us a numeric value and not a simple symbolic formula. A formula would allow us to determine the tile shape that optimizes the length of the longest path of dependent tiles directly. It can also be used to predict the length of the longest path of dependent tiles of a given tiling. To achieve the goal of deriving a formula, however, we must further restrict the kind of iteration spaces we consider since finding a symbolic solution for a general convex iteration space corresponds to finding a symbolic solution to a general linear programming problem [23].

Rectilinear iteration spaces is a subclass of convex iteration spaces. For rectilinear iteration spaces, we can use the duality property of a linear programming formulation to derive a symbolic formula for the length of the longest path of dependent tiles (Theorem 2). The key property that makes it easier to analyze rectilinear iteration spaces is the fact that the slopes of two adjacent facets differ only in one dimension. All two-dimensional, convex iteration spaces are therefore rectilinear. A rectilinear iteration space is defined as follows:

**Definition 3.** Let $T(x_1, \ldots, x_{K-1})$ and $B(x_1, \ldots, x_{K-1})$ be the functions describing the top and bottom of the iteration space respectively. A convex iteration space is rectilinear if

\[
\forall i: 1 \leq i \leq K - 1 \left( \frac{\partial B(x_1, \ldots, x_{K-1})}{\partial x_i} = g_i(x_i) \right),
\]

and

\[
\forall i: 1 \leq i \leq K - 1 \left( \frac{\partial T(x_1, \ldots, x_{K-1})}{\partial x_i} = h_i(x_i) \right),
\]

where $g_i(x_i)$ and $h_i(x_i)$ are arbitrary functions of $x_i$.

In other words, a $K$-dimensional, convex iteration space is rectilinear if the following condition holds:

The intersection of any two facets defining the boundaries of the iteration space is either parallel or perpendicular to all coordinate axes.

The iteration spaces generated by matrix multiply [5], various dynamic programming applications [15], sweep 3D [37], 2D SOR [40], 2D seismic migration [1], and all two-dimensional iteration spaces, including iteration spaces of triangular shapes, are examples of rectilinear iteration spaces.

Rectilinear iteration spaces are more general in shape than the iteration space shapes most commonly addressed in the tiling literature. This is an advantage when tiling for multiple levels of the memory hierarchy (see Section 2.2). When tiling for multiple levels the partial tiles, which become the iteration space to be tiled at the next level, might be quite complicated in shape. They are, however, guaranteed to be rectilinear. The technique proposed in this paper can therefore, still be used to tile the partial tiles, whereas other techniques will not be able to since the partial tiles are too complicated in shape.

Handling partial tiles well can be important when the partial tiles form a significant portion of the iteration space. This can happen when the sizes of two levels of the computer’s memory hierarchy aren’t vastly different, e.g., when tiling out-of-core computations into “main memory tiles,” or tiling main memory into “L3 cache tiles.”

7 Length of the Longest Path of Dependent Tiles

We are now ready to define the length of the longest path of dependent tiles in an iteration space. Consider all tiles in a tiled iteration space after the transformations to ensure Property 1. In this iteration, space there are at most two partial tiles in each stack: the bottom and top tile (see Fig. 7d). Furthermore, all tiles (including the partial tiles) have horizontal bottom and top surfaces. Before we continue, we need to introduce some notation to state Assumption 1. Let $S$ be a plane that cuts the iteration space perpendicular to the vertical dimension, and let $C(S)$ be the set of tiles cut by $S$. Then, $P(C(S))$ is any path of dependent tiles that only consists of tiles in $C(S)$, and that starts in a tile that has no predecessors in $C(S)$ and ends in a tile that has no successors in $C(S)$.

**Assumption 1.** In all tiled iteration spaces, there exists a path $P(C(S))$ s. t. all tiles on $P(C(S))$ are full.

Informally, this assumption ensures that each stack in the iteration space contains “sufficiently” many full tiles. This technical assumption allows us to use linear programming to determine the length of the longest path of dependent tiles. It does limit the applicability of our results, but it is our belief that our results are valid also in many cases not covered by Assumption 1. We also assume without loss of generality that full tiles are unit cubes.

12. By the convexity of the iteration space, they must be monotone.
Now, consider the center point at both the top and bottom surfaces of all tiles in the tiled iteration space. Let \(p\) be the center point of the bottom surface of a tile and let \(q\) be the center point of the top surface of a tile such that \(d \geq 1\). The \(L_1\)-norm distance between \(p\) and \(q\) is equal to the sum of the volume of the tiles on the longest path of dependent tiles between \(p\) and \(q\) (Property 1).

Due to Assumption 1 and the fact that all full tiles are unit cubes, there are \(q - p\) tiles and \(q - p\) surfaces to be communicated along the path between \(p\) and \(q\). This observation leads to the following definition of the length of the longest path of dependent tiles between \(p\) and \(q\).

**Definition 4.** Let \(\bar{p} = (p_1, \ldots, p_K)\) be the center point of the bottom surface of a tile and let \(\bar{q} = (q_1, \ldots, q_K)\) be the center point of the top surface of a tile such that \(\bar{p}\) precedes \(\bar{q}\). Let \(v\) be the longest path of dependent tiles between \(\bar{p}\) and \(\bar{q}\) (see Definition 1). We define \(L(v) = L(\bar{p}, \bar{q})\), the length of the longest path of dependent tiles between \(\bar{p}\) and \(\bar{q}\), as follows:

\[
L(v) = L(\bar{p}, \bar{q}) = \sum_{d=1}^{K} (q_d - p_d)(1 + c_d).
\]

Intuitively, this definition says that the length of the longest path of dependent tiles between two tiles is the number of tiles that lie on the path plus the communication between the processors executing these tiles. Note that using this definition, if \(L\) is the length of the longest path of dependent tiles, \(L_E\) yields the execution time of the iteration space. This is because the number of tiles on the longest path of dependent tiles \(\sum_{d=1}^{K} (q_d - p_d)\) times \(E_t\) is the execution time of all the tiles on the path. And \(\sum_{d=1}^{K} (q_d - p_d)c_d\) times \(E_t\) is by definition the sum, over all dimensions, of the number of surfaces to be communicated between the processors \(\sum_{d=1}^{K} (q_d - p_d)\) times the amount of overlappable communication time for each such surface \(c_d\).

We also define the initial and final synchronization points as the end points of the longest path of dependent tiles in an iteration space:

**Definition 5.** Let \(\bar{p}\) be the center point of the bottom surface of a tile and let \(\bar{q}\) be the center point of the top surface of a tile such that \(\bar{p}\) precedes \(\bar{q}\).\(\bar{z}^i\) and \(\bar{z}^f\) are the initial and final synchronization points of the iteration space.

In [23], it was shown that the location of the synchronization points in a general convex iteration space bounded by \(M_t\) planar constraints can be found by the linear programming problem expressed in Table 1, where \(M_b, M_t, M_l,\) and \(M_r\) denote the number of facets defining the bottom, top, left and right side of the iteration space, respectively, and \(M_t = M_b + M_t + M_l + M_r\). Notice that, for the term \(x_{K+d} - x_d\) to denote the distance between the top and bottom of two stacks, the tiles must have first been transformed into unit cubes (Property 1). This is, however, only a conceptual transformation and not actually performed on the iteration space. Here, and in the rest of this...
paper, we use \( \sum_{d=1}^{K-1} (-r_d) x_d + x_K = n^i \) to denote the equation describing facet \( i \), since the rise \( r_d \) is equal to the iteration space slope in dimension \( d \) of the \( i \)th facet when the tiles are unit cubes. The objective function ensures that the solution is indeed the coordinates for the synchronization points, as defined in Definition 5.

It was also shown in [23] that the problem of determining the execution time of a general planar convex iteration space is as hard as general linear programming. For a general convex iteration space, it is, therefore, not likely that we can derive a simple formula for the execution time. A formula would, however, be useful both for predicting the execution time given a tiling and for finding the tile shape that minimizes the execution time. In the next section, we determine a formula for the length of the longest path of that minimizes the execution time. In the next section, we show how to find the synchronization points, as discussed in Section 6. We also present a prediction formula for the execution time of rectilinear iteration spaces.

### 7.1 Rectilinear Iteration Spaces

In this section, we show how to find the synchronization points to a rectilinear iteration space without solving a linear programming problem. This result allows us to derive a formula for the execution time of the iteration space. It also gives us an understanding as to how the longest path of dependent tiles changes with the shapes of the iteration space and the tiles. This is discussed in Section 8, where we also determine the tile shape that minimizes the length of this path.

Apart from the synchronization points, there are two other points in an iteration space that are of importance: the bottom and top corners of the iteration space. These are the coordinate points at which in all dimensions \( d \) the rise of the bordering facets is less than \(-1+c_d\) on one side and greater or equal to \(-1+c_d\) on the other side (see Fig. 11a). We define these points, \( \tilde{q}^b \) and \( \tilde{q}^t \), as follows:

**Definition 6.** For a rectilinear iteration space, the bottom corner, \( \tilde{q}^b \), is the coordinate point such that \( \forall d \) 1) there exists no facet of the bottom surface of the iteration space at a smaller coordinate than \( q^b_d \) that has a rise \( r_d \geq -1+c_d \) and 2) there exists no facet of the bottom surface of the iteration space at a larger coordinate than \( q^b_d \) that has a rise \( r_d < -(1+c_d) \). Similarly, the top corner, \( \tilde{q}^t \), is the coordinate point such that \( \forall d \) 1) there exists no facet of the top surface of the iteration space at a smaller coordinate than \( q^t_d \) that has a rise \( r_d < -(1+c_d) \) and 2) there exists no facet of the top surface of the iteration space at a larger coordinate than \( q^t_d \) that has a rise \( r_d \geq -1+c_d \).

The reason for why the bottom and top corners are so important is that the location of the synchronization points in each dimension changes if the bottom corner, precedes the top corner, in that dimension. In Theorem 1, we show that in all dimensions \( d \) where the tile slope is such that \( q^t_d \leq q^t_d \), we have \( z^t_d = q^t_d \) and \( z^b_d = q^b_d \), and in all dimensions \( d \) such that \( q^t_d > q^t_d \), \( z^t_d \) and \( z^b_d \) are the end points of the longest vertical path in that dimension (see Figs. 11b and 11c). To distinguish between these two kinds of dimensions, we introduce the following terminology.

**Definition 7.** \( \forall d : 1 \leq d \leq K-1 \), if \( q^t_d \leq q^t_d \), we call \( d \) a forward dimension. \( \forall d : 1 \leq d \leq K-1 \), if \( q^t_d > q^t_d \), we call \( d \) a backward dimension. We call the set of all forward dimensions \( F \) and the set of all backward dimensions \( B \).

We prove Theorem 1 by using the duality property of linear programming on the linear programming formulation from Section 7. As always, finding the solution to the dual problem is tricky since we have to determine the values of the dual variables. For rectilinear iteration spaces, the two solution points of the primal problem are surrounded by \( 2^{K-1} \) facets, which limits the problem to finding the dual variables corresponding to those constraints. (The dual variables corresponding to the other constraints are all set to zero.) It turns out that \( K \) of the \( 2^{K-1} \) variables should be picked to be a telescoping series that sums to 1. To determine which those \( K \) variables are, we need to order the constraints to which they correspond. The variables are ordered so that the constraint corresponding to the \( i \)th variable only differs in rise in dimension \( d \) compared to the constraint corresponding to the \((i-1)\)th variable, where \( d \) is the dimension in which the measure is the \( i \)th largest for all dimensions. This will define a unique ordering of \( K \) variables (constraints). The rest of the \( 2^{K-1} \) variables of the dual problem are set to zero. Once we have determined this solution to the dual problem, we prove that the objective function of the dual problem at that point is equal to the objective function of the primal problem at the synchronization points.

**Theorem 1.** Let \( \tilde{i} \) and \( \tilde{j} \) be any two coordinate points on the top and bottom surface of a tiled iteration space, respectively, such that \( \forall d : 1 \leq d \leq K-1 \) (\( i_d = j_d \)). Let \( \tilde{a} \) and \( \tilde{b} \) be the two points on the top and bottom of the same iteration space respectively, such that \( \alpha_K - b_K = \max_{i,j} [\tilde{i}K - jK] \).
The location of the two synchronization points are then
given by
\[ \forall d \in F(z_d^l = q_d^l, z_d^f = q_d^f) \]
\[ \forall d \in B(z_d^l = b_d = z_d^f = a_d). \]

**Proof of Theorem 1.** Due to space limitations, we refer the
reader to [21].

From Theorem 1 and Definition 6, we see that the length
of the longest path of dependent tiles only depends on the
slope of four of the facets on the top and bottom surfaces,
indeed of the dimensionality of the iteration space.
All other facets can change slope without affecting the
execution time, as long as the iteration space remains
convex and rectilinear. We call these four facets the defining
facets of an iteration space. Before we define these facets we
need to introduce some terminology. For simplicity, we do
not introduce the notation necessary to cover iteration
space facets each. For a given dimension \( d \), there are only two
slopes used in those facets. Using this fact, we define the
defining facets as follows (see Fig. 12).

**Definition 8.** Consider all the \( 2^{K-1} \) facets intersecting in the
initial synchronization point, \( z^f \). For all dimensions \( d \), let \( s_d^{low} \)
and \( s_d^{high} \) be the two slopes in dimension \( d \) used in those facets
s. t. \( s_d^{low} \leq s_d^{high} \).

The first defining facet, Facet 1, is the facet with equation
\[ - \sum_{d=1}^{K-1} s_d^{low} x_d + x_K = n^1. \]
\[ \forall d : 1 \leq d \leq K - 1, \text{ the slope in dimension } d \text{ of Facet 1, } r_d^{\phi}, \text{ is thus equal to } s_d^{low}. \]

The second defining facet, Facet \( K \), is the facet with equation
\[ - \sum_{d=1}^{K-1} s_d^{high} x_d + x_K = n_K. \]
\[ \forall d : 1 \leq d \leq K - 1, \text{ the slope in dimension } d \text{ of Facet } K, r_d^{K}, \text{ is thus equal to } s_d^{high}. \]

**Definition 9.** Consider all the \( 2^{K-1} \) facets intersecting in the
final synchronization point, \( z^f \). For all dimensions \( d \), let \( s_d^{low} \)
and \( s_d^{high} \) be the two slopes in dimension \( d \) used in those facets s. t.
\( s_d^{low} \leq s_d^{high} \).

The third defining facet, Facet \( M_b + 1 \), is the facet with equation
\[ \sum_{d=1}^{K-1} s_d^{high} x_d - x_K = n^{M_b+1}. \]
\[ \forall d : 1 \leq d \leq K - 1, \text{ the slope in dimension } d \text{ of Facet } M_b + 1, r_d^{M_b+1}, \text{ is thus equal to } s_d^{high}. \]

The fourth defining facet, Facet \( M_b + K \), is the facet with equation
\[ \sum_{d=1}^{K-1} s_d^{low} x_d - x_K = n^{M_b+K}. \]
\[ \forall d : 1 \leq d \leq K - 1, \text{ the slope in dimension } d \text{ of Facet } M_b + K, r_d^{M_b+K}, \text{ is thus equal to } s_d^{low}. \]

Knowing that there are only \( 2(K-1) \) different slopes in the
\( 2^{K-1} \) facets intersecting at each of the synchronization
points helps us solve the system of linear equations
from Section 7. We present the resulting coordinates in
Observation 1.

**Observation 1.** In the last dimension, \( K \), \( z^f_K \), and \( z^r_K \) are
given by
\[ z^f_K = \sum_{d=1}^{K-1} r_d^{\phi} x_d + n^1 \]
\[ z^r_K = \sum_{d=1}^{K-1} r_d^{K} x_d + n^{M_b+1} \]

For all dimensions \( d : z^f_d \neq x^\text{min}_d \) and \( z^r_d \neq x^\text{max}_d \), \( z_d \) is given by
\[ z^f_d = \begin{cases} 
\frac{q^d+1-n^1}{r_d^{\phi}-r_d^{f}} (\forall d : (r_d^{\phi} 
eq r_d^{f})) \\
\frac{q^f}{z_d^{f}} (\forall d : (r_d^{f} = r_d^{f})) 
\end{cases} \]

For all dimensions \( d : z^f_d \neq x^\text{min}_d \) and \( z^f_d \neq x^\text{max}_d \), \( z_d \) is given by
\[ z^f_d = \begin{cases} 
\frac{n^1 - \hat{z}_d}{r_d^{\phi} - r_d^{f}} (\forall d : (r_d^{\phi} = r_d^{f})) \\
\frac{n^{M_b+1} - \hat{z}_d}{r_d^{K} - r_d^{f}} (\forall d : (r_d^{K} = r_d^{K})) 
\end{cases} \]

where \( \forall i : 1 \leq i \leq K - 1 \) the facets are ordered such that only
the rise in dimension \( d \) changes between facet \( d \) and facet
\( d + 1 \).

Using these coordinates, we can now calculate the
length of the longest path of dependent tiles in the
iteration space as defined in Definition 4. A compiler can
use this formula to predict the optimal tile shape for an
iteration space. In Theorem 2, the length of the longest
path of dependent tiles is given as a function of the
parameter \( T \). As illustrated in Fig. 13, \( T \) is the number of
(possibly empty) tiles between Facet 1 and Facet \( M_b + 1 \).

16. We know that such an ordering exists since the iteration space is
rectilinear.
17. We know by Theorem 1 that \( r_d^{\phi} \), where \( r_d^{\phi} = r_d^{f} \) and \( r_d^{M_b+1} = r_d^{M_b+K} \).
at coordinate \( z^i \). The quantity \( T + \sum_{d \in F} r_d^{M+1} (z_d^f - z_d) \) therefore, represents the vertical distance (in units of tiles) between the top surface at coordinate \( z^f \) and the bottom surface at \( z^i \). In each dimension \( d \in F \), the quantity \( \sum_{d \in F} (1 + c_d)(z_d^f - z_d^i) \) is the number of tiles (stacks) plus communication cost between the processors along the horizontal path in that dimension \( d \) from \( z^i \) to \( z^f \). Table 2 shows the length of the longest path of dependent tiles for some common iteration space shapes. In a 2-dimensional parallelepiped with vertical sides, (Case a in Table 2), the longest path of dependent tiles goes straight up each stack if the first dimension is a backward dimension (see Theorem 1). The length of the longest path of dependent tiles is thus \( T \). If the first dimension is a forward dimension the longest path of dependent tiles goes horizontally through all \( P \) stacks as well as up to the top. Along the horizontal component of the path, \((P_1 - 1)\) tiles are executed and \((P_1 - 1)\) surfaces communicated. Along the vertical component, we execute the number of tiles between \( z_k^i \) and \( z_k^f \), i.e., \( T + (P_1 - 1)\) \( n^{M+1} \). Due to the fact that only the defining facets affect the length of the longest path of dependent tiles, there is no difference in the formula for the length between Case b and c. The only K-dimensional parallelepiped that is rectilinear, is the parallelepiped skewed in at most one dimension other than the vertical. In this table, we call this dimension \( j \). Case e shows the formula for when \( j \in F \) and \( j \in B \). \( P_d \) is the number of processors along dimension \( d \) in the processor grid, i.e., the total number of processors is equal to \( \prod_{d=1}^{K} P_d \). \( T \) is equal to \( \sum_{d=1}^{K} (r_d^{M+1} - r_d) z_d^i + n^{M+1} - n^i \), and is explained in Fig. 13.

**Theorem 2.** The length of the longest path of dependent tiles in a rectilinear iteration space, \( L(z^i, z^f) \), is given by the following expression:

\[
L(z^i, z^f) = \sum_{d \in F} (1 + c_d + r_d^{M+1}) (z_d^f - z_d^i) + T,
\]

where

\[
T = \sum_{d=1}^{K-1} (r_d^{M+1} - r_d) z_d^i + n^{M+1} - n^i
\]

and \( \forall d : 1 \leq d \leq K - 1 \), \( z_d^i \) and \( z_d^f \) are given by Observation 1.

**Proof of Theorem 2.** Due to space limitations, we refer the reader to [21]. \( \square \)

Using Theorem 2 as an expression for the length of the longest path of dependent tiles, we will now model the execution time of a rectilinear iteration space. As discussed in Section 2.2, tiling can be applied to multiple levels of the memory hierarchy. Numbering the levels of memory from 0 to \( z \) where 0 is the fastest memory level, we write the formula for the execution time of the iteration space at level \( k : 0 < k \leq z \), as follows:

\[
E^k_i = L^k E^k_i \quad (1)
\]

\[
E^0_i = E^0_i + O^k \quad (2)
\]

\[
E^0_i = L^k E^0_i \quad (3)
\]

where \( E^k_i \) is the execution time of complete iteration space at level \( k \), \( L^k \) the length of the longest path of dependent tiles at level \( k \), \( E^k_i \) the execution time of a full tile at level \( k \),

<table>
<thead>
<tr>
<th>dim</th>
<th>I.S. Shape</th>
<th>( F ) or ( B )</th>
<th>( L(z^i, z^f) ) from Theorem 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>a: Parallelogram w/ vertical sides</td>
<td>( B )</td>
<td>( T )</td>
</tr>
<tr>
<td></td>
<td>( F )</td>
<td>( 1 + c_1 + r_1^{M+1} (P_1 - 1) + T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B )</td>
<td>( T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( F )</td>
<td>( 1 + c_1 + r_1^{M+1} (z_1^f - z_1^i) + T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B )</td>
<td>( T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( F )</td>
<td>( 1 + c_1 + r_1^{M+1} (z_1^f - z_1^i) + T )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b: Parallelogram</td>
<td>( B ), ( B )</td>
<td>( T )</td>
</tr>
<tr>
<td></td>
<td>( F ), ( F )</td>
<td>( 1 + c_1 + r_1^{M+1} (P_1 - 1) + T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( F ), ( F )</td>
<td>( \sum_{d=1}^{2} (1 + c_d + r_d^{M+1} (P_d - 1) + T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( j \in B )</td>
<td>( \sum_{d \in F} (1 + c_d + r_d^{M+1} (P_d - 1) + T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( j \in F )</td>
<td>( \sum_{d \in F, d \neq j} (1 + c_d + r_d^{M+1} (P_d - 1) + T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( j \in F )</td>
<td>( \sum_{d \in F, d \neq j} (1 + c_d + r_d^{M+1} (P_d - 1) + T )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( j \in F )</td>
<td>( \sum_{d \in F, d \neq j} (1 + c_d + r_d^{M+1} (P_d - 1) + T )</td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 2**

The Length of the Longest Path of Dependent Tiles for Some Common Iteration Space Shapes
The overheads encountered at level $k$, $L^k$ the number of iterations in lowest level tile, and $E_{it}$ the execution time of one iteration. At the base case of the recursion, $k=0$, the iterations in each tile are assumed to be executed sequentially. If they were not, the tile would have been considered further tiled, and that level would therefore not have been the lowest level of tiling. Since the iterations are executed sequentially there is no need for communication between processors. The length of the longest path of dependent "tiles" (=iterations) on the lowest level, i.e., $L^0$, is therefore simply the number of iterations in the smallest tile, i.e., the volume of the tile. For a single level of tiling, (1), (2), and (3) can be rewritten as

$$E_i = LE_i$$

$$E_{it} = V E_{it} + O$$

where $E_i = E_i^0$ is the execution time of complete iteration space, $L = L^1$ the length of the longest path of dependent tiles, $E_i = E_i^1$ the execution time of a full tile, $V = L^0$ the volume of a tile, $O = O^1$ the overheads, and $E_{it}$ the execution time of one iteration. $O$ includes overheads such as synchronization and nonoverlappable communication overheads (see Section 4), and both $E_{it}$ and $O$ can be determined by benchmarks.

**8 TILE SHAPE SELECTION**

In this section, we discuss the implications Theorem 1 has on the preferred choice of the tile shape. We first discuss how the longest path of dependent tiles changes with the tile shape and then determine the tile shape that minimizes the length of this path. In past practice, tiles were chosen either to be rectangular, or of the same shape in the case of parallelepiped iteration spaces, as the iteration space. Our results do not necessarily support either of these choices.

The height of the tallest stack in an iteration space is a lower bound on $E(\bar{z}, \bar{f})$, since all vertical paths are paths of dependent tiles. We therefore know that for any two points $\bar{e}$ and $\bar{f}$ that lie on the bottom and top of a single stack, $E(\bar{e}, \bar{f})$, can never be less than $f_{K} - e_{K}$. Due to the rectilinear property of the iteration space, we know that $E(\bar{z}, \bar{f})$ decreases as the number of dimensions $d$ for which $z_d^j = f_d^j$ increases.

The tile shape should, therefore, be chosen such that the coordinates for the initial and final synchronization points are equal in as many dimensions as possible. If that is not possible, then they should lie as close as possible. Increasing the tile slope in a given dimension decreases the distance between the synchronization points in that dimension. At the point when $q_d^f$ first becomes greater than $q_d^t$, there is no longer a benefit in increasing the slope further. The longest path of dependent tiles is at that point already vertical and can not become shorter. This is illustrated in Fig. 14, where we can see how the location of both the bottom and top corner, as well as the two synchronization points changes with the tile slope. The tile slope increases from Fig. 14a to Fig. 14d, but in all four cases the tile size is the same. The shaded tiles constitute the longest path of dependent tiles. Increasing the tile slope pushes the bottom and top corners of the iteration space to the right and left, respectively. Also, the synchronization points move but only to the point where $z_d^j = f_d^j$. At that point, they specify the coordinate in dimension $d$ for the tallest stack within the iteration space.

Since the execution time of a stack is a lower bound on the execution time of the iteration space, we would like for as many dimensions as possible to achieve this lower bound, i.e., be backward dimensions.

We state a theorem, which says that when tiling at a single level in the memory hierarchy, the tile slope should be chosen as large as possible without violating the legality condition, but that such a large tile slope is not necessary in all cases. We prove this theorem by minimizing the formula for $L(\bar{z}, \bar{f})$ given by Theorem 2.

**Theorem 3.** Let $\forall d : 1 \leq d \leq K - 1 \, t_d$ be the largest legal tile slope in dimension $d$. Let $q_d^b$ and $q_d^t$ be the location of the bottom and top corners in dimension $d$ when the iteration space is tiled with $t_d$ in that dimension. Let $t_d$ be the smallest tile slope in dimension $d$ for which $d \in B$, i.e., $q_d^b \geq q_d^t$. When tiling a rectilinear iteration space and at a single level of the memory hierarchy, the tile slope in dimension $d$, $t_d$, should then be chosen as follows:

$$t_d = t_d \, (\text{if } q_d^b < q_d^t)$$

$$t'_d \leq t_d \leq t_d \, (\text{if } q_d^b \geq q_d^t)$$

**Proof of Theorem 3.** We omit the proof due to space limitations.

The fact that the optimal tile slope when $q_d^b \geq q_d^t$ is not necessarily equal to the largest legal tile slope $t_d$, is quite interesting. Although the length of the longest path of dependent tiles is the same, there might be advantages, e.g., simpler code generation and when tiling for multiple levels of the memory hierarchy, in choosing the tile slope equal to some value other than $t_d$ in the range $[t'_d \ldots t_d]$. 

---

Fig. 14. The same projection through the same iteration space tiled with four different tile slopes.
### 9 RELATED WORK

As previously mentioned, traditional tiling [43], [33], [24], [44], [32] partitions the iteration space of a nested loop with a regular stencil of dependencies into regular size and shape pieces. Tiling has been used to achieve both locality and parallelism, and its beneficial effects are well established [14], [40], [41], [10], [32], [12], [13]. In [40], [14], [3], tiling is used to improve locality, and the effect on the amount of available parallelism is ignored. The research that seeks to improve locality only considers rectangular tiles, or tiles of the same slope as the iteration space [3]. In that work, only tile size selection is therefore of concern [14], [34], [3].

In [41], tiling parameters are chosen to improve parallelism. Unimodular transformations [4] (which include loop skewing) are used to increase the applicability of tiling. The iteration space is skewed by a (integer) factor so that the use of rectangular tiles become legal. This is similar to always choosing a tile shape that includes the extreme vectors in the skewed dimension. In [25], [26], [11], [39], [42], [9], both locality and parallelism are used as optimization goals. References [25], [26], [11] first optimize for locality and, then, use the degrees of freedom left over to improve the available parallelism. References [39], [42] attack the problem in the opposite order, optimizing first for parallelism and then for locality. These approaches are likely to suffer from the disadvantage of other code optimization problems, that a two-step optimization approach is inferior to making a coordinated choice using a unified cost metric [28].

Reference [9] optimizes for both locality and parallelism simultaneously. It uses the notion of balance defined in [8]. It incorporates the effects in latency hiding and cache misses in the computation of loop balance, and are able to optimize for both locality and ILP using one transformation. Our work applies to all levels of the memory hierarchy and is not limited to ILP.

In [7], the authors determines the optimal tile shape by minimizing the amount of data communicated. Their approach determines the tile shape given a set of dependences and does not take the shape of the iteration space into account. Reference [17] uses linear programming to determine the execution time of a convex iteration space under the assumptions that the iteration space is large enough, compared to the size of a tile, to be able to ignore the effects of partial tiles.

The work in [46], [38], [6] also tiles for multiple levels. Reference [6] presents a parameterized code generator that produces architecture-specific code for matrix multiply. These routines are blocked for both cache and register and achieve close to peak performance on several architectures.

There are two bodies of work that take a slightly different approach to tiling. In [47], the reuse of a program is used to minimize the number of dimensions in which the iteration space is tiles. This has the advantage that the loop control overhead is minimized. On the other hand, the shape of the iteration space is ignored which we show in this paper is suboptimal (Theorem 3), especially when tiling for multiple levels of the memory hierarchy. In [27], data-shackling is introduced as an alternative to tiling. Instead of reasoning about the program’s traversal of the iteration space, it takes a more data-centric view and reasons about how the program traverses the data. Apart from giving good performance due to more exact information, it has the additional advantage that it can be applied to imperfectly nested loops with out first transforming them. Another paper that also handles imperfectly nested loops is [35] where they propose new program transformations to enable tiling for nontrivial imperfectly nested loops.

In our work, we only handle perfectly nested loops and, thus, assume that the above mentioned transformations already have taken place. Our previous work [22], introduces the notion of rise and recognizes its importance in tile shape selection for minimizing the idle time of parallel execution, and [23] directly extends [22] from two-dimensional iteration spaces to K-dimensional iteration spaces of a much more general shape. Tiling work that limit itself to parallelepiped-shaped iteration spaces have the disadvantage that they cannot completely automate the tiling of partial tiles when tiling for multiple levels of the memory hierarchy. And work where the shape of the iteration space is ignored altogether might find a suboptimal tiling by not realizing the implications of Theorem 3. This might result in tiles for which it is more complicated to generate code, as well as suboptimal tile shapes when tiling is applied in the context of multiple levels of parallelism [29], [28], such as in hierarchical tiling [12], [13].

Both [22] and [18] considered parallelepiped-shaped tiles for two-dimensional iteration spaces and determined simple formulae for idle time and finishing time. The difference between the two papers is that the latter used a slightly different model of partial tiles, resulting in simpler formulas and proofs. See Section 3 for a detailed comparison of these models.

Like most tiling work, we only consider atomic execution of tiles. In atomic execution of tiles, a tile is not allowed to start execution until all its input data are available. In [46], this assumption is relaxed. Because of this, they can overlap a larger fraction of the communication with computation than would otherwise be possible for given a tile shape and size. In many cases, it is, however, possible to achieve the same overlap with an atomic tile of a different shape.

The goal of our work is to develop analysis techniques that can be used by a compiler that automatically chooses a tiling for a sequential program that achieves both good locality and parallelism. A complete discussion of the automatic code generation of a tiled loop is outside the scope of this paper, but has been studied in [2], [19], [31], [36]. Reference [30] presents a compiler that automatically parallelizes a single program across multiple processors. It does not depend on explicit parallel language features or on user directives. It relies on the program dependence graph [20] to identify regions of the program that can be executed in parallel.

### 10 CONCLUSIONS

We have developed a performance model that allow us to predict the execution time for a tiled perfectly nested loop of an arbitrary depth. To model the performance, we use the notion of rise. The rise of a tiled iteration space is a measure of the relationship between the shape of the tiles and the
shape of the iteration space. It is this relationship that affects the execution time, and not the tile shape per se. The rise determines how soon all the processors can reach full parallel execution rate. For example, for certain rises, all processors can start execution immediately and will never need to wait for communication from another processor. For other rises, the processors will have to wait a long time before they can reach that state of "ideal" parallel execution.

We use the notion of rise to determine the length of the longest path of dependent tiles in an iteration space. The longest path of dependent tiles is the path of dependent tiles through an iteration space that takes the longest to execute, either because of the large number of tiles on the path or because of the large communication cost along the path. The length of the longest path of dependent tiles is hence strongly correlated with the execution time of the iteration space.

It was shown in [23] that determining the length of the longest path of dependent tiles in a general k-dimensional convex iteration space is as hard as general linear programming. We therefore define a subclass of convex iteration spaces, for which we can derive a formula for the length of the longest path of dependent tiles. We call this subclass rectilinear iteration spaces. The class of rectilinear iteration spaces includes all two dimensional iteration spaces. It also includes applications such as matrix multiply [5], various dynamic programming applications [15], sweep 3D [37], 2D SOR [40], and 2D seismic migration [1].

For the rectilinear iteration spaces, we then derive a formula for the length of the longest path of dependent tiles as a function of the rise. We incorporate this formula into our performance model and present a formula for predicting the execution time for tiling both at a single level and at multiple levels of the memory hierarchy. We also show how to choose the optimal tile shape.

An area of future work is to further develop the ideas we have on how to transform nonrectilinear iteration spaces into rectilinear without altering execution time and optimal tiling parameters. This would allow us to use our prediction formula also for nonrectilinear iteration spaces. To validate the accuracy of our predictions we will also examine a wide range of both applications and architectural platforms.

ACKNOWLEDGMENTS

This work was done while K. Högstedt was at the University of California, San Diego. This work was partly supported by US National Science Foundation grants CCR-9504150 and CCR-9808946.

REFERENCES


For more information on this or any computing topic, please visit our Digital Library at http://computer.org/publications/dlib.