PARTITIONING STRATEGIES: SPATIOTEMPORAL PATTERNS OF PROGRAM DECOMPOSITION

Henry Hoffmann, Anant Agarwal, and Srinivas Devadas
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
email: {hank,agarwal,devadas}@csail.mit.edu

ABSTRACT
We describe four partitioning strategies, or patterns, used to decompose a serial application into multiple concurrently executing parts. These partitioning strategies augment the commonly used task and data parallel patterns by recognizing that applications are spatiotemporal in nature. Therefore, data and instruction decomposition are further distinguished by whether the partitioning is done in the spatial or in temporal dimension. Thus, we arrive at four decomposition strategies: spatial data partitioning (SDP), temporal data partitioning (TDP), spatial instruction partitioning (SIP), and temporal instruction partitioning (TIP), and catalog the benefits and drawbacks of each. In addition, the practical use of this work is demonstrated by applying these strategies, and combinations thereof, to implement several different parallelizations of a multicore H.264 encoder for high-definition video.

KEY WORDS
Design patterns, parallel software engineering, H.264 encoding

1 Introduction

Parallel programming is often viewed as an ad hoc and artistic practice best left to experts. Because of this belief, a great deal of research has been done to transform the development of parallel applications into a rigorous engineering discipline. An important component of this work is the recognition and documentation of patterns, or commonly occurring problems and their solutions, in the field of parallel software development [14, 15, 17, 18, 20, 21]. The documentation of these design patterns for parallel programming creates a shared repository and language for discussing parallel software development.

Perhaps the most common dilemma faced by parallel programmers is that of finding the initial parallelism in an application. Two common patterns have arisen that describe solutions to this problem: *Task* and *Data* Parallelism [14] (also called TaskDecomposition and DataDecomposition [18]). These two patterns capture the most basic ways that parallel programming experts think about their programs. Task parallelism refers to viewing a problem as a collection of separate activities, or tasks, which can be executed concurrently. Data parallelism refers to viewing a problem as the manipulation of a data structure with independent substructures, which can be operated on concurrently.

Capturing the way expert parallel programmers think about parallelism helps create a structured and disciplined way for novice parallel programmers to approach these problems, but it can also lead to ambiguity. Such ambiguity arises when the use of two different patterns leads to equivalent parallel programs. As an example, consider the parallelization of matrix multiplication as described in [18]. A task parallel matrix multiplication is presented in which each task consists of the production of a single element of the result. A data parallel matrix multiplication is also presented where elements of the result matrix are distributed among independent units of execution. Depending on the assignment of tasks to execution units, both the task and data parallel implementations of matrix multiplication might be the same. The fact that using different patterns can result in the same parallel implementation means that using these patterns provides limited insight into the effects on the parallel program.

This work presents four spatiotemporal partitioning strategies designed to complement the existing and well-established task and data parallel patterns. Where existing patterns are designed to capture the way the programmer thinks about a parallel program, these spatiotemporal strategies are designed to capture the structural effects on the parallelized application.

The distinction between spatiotemporal patterns is based on determining an index set which describes how an application executes in time and space. Spatial and temporal indices are assigned to a program's data and instructions and different strategies are based on the division of these indices amongst units of execution. Thus, this work recognizes four partitioning strategies for exploiting concurrency in an application: spatial data partitioning (SDP), temporal data partitioning (TDP), spatial instruction partitioning (SIP), and temporal instruction partitioning (TIP).

Recognizing patterns that distinguish between temporal and spatial partitioning is important for the following reasons:

- This distinction provides an additional set of options for finding concurrency in an application.
- Recognizing the difference between spatial and tem-

poral partitionings provides additional descriptive power for documenting and characterizing a parallel application.

- Perhaps most significantly, temporal and spatial partitioning affect the performance of an application in separate ways.

To understand the effect on performance, consider an application that continuously interacts with the outside world by processing a sequence of inputs and producing a sequence of outputs. Examples include desktop applications that interact with a human, embedded applications that interact with sensors, and system software that provides quality of service guarantees to other applications. These interactive programs typically have both *throughput* and *latency* requirements. The throughput requirement specifies the rate at which inputs are processed while the latency requirement specifies the speed with which an individual input must be processed. While both spatial and temporal partitioning patterns improve throughput, only spatial partitionings can improve latency.

To illustrate the use of these patterns, this paper discusses the results of applying several different strategies to parallelize an H.264 video encoder for high-definition video [11] on a multicore architecture.

The rest of this paper is organized as follows. Section 2 defines terminology and presents an example application that is used to illustrate concepts. Section 3 presents the four partitioning strategies describing both spatial and temporal decomposition of data and instructions. Section 4 presents the results of applying these patterns, and combinations of the patterns, to an H.264 encoder. Section 5 covers related work and describes how the partitioning strategies in this paper relate to those in [18] and [14]. Section 6 concludes the paper.

2 Basics and Terminology

This section presents the context and terminology used to describe the partitioning strategies listed in Section 3. It begins by introducing an example application: an intelligent security camera. The security camera example is used to illustrate many of the concepts in the remainder of the paper. Next, the terminology used to describe spatial and temporal indexing of a program is presented. Finally, the section discusses a simple procedure used to prepare an application for decomposition using the spatiotemporal design patterns described in Section 3.

2.1 Example Application: Security Camera

An intelligent security camera processes a sequence of input images, or frames, from a camera (e.g., [16]). The camera compresses the frames for efficient storage and searches the frames to detect objects of interest. The compressed video is stored to disk while a human is alerted to the presence of any objects of interest. Both the data (frames) and the instructions (compression and search) of the camera have spatial and temporal dimensions.

The primary data object manipulated by the camera is the frame. The frame consists of pixels where each pixel is generated by a spatially distinct sensor. The position of a pixel in a frame represents a spatial index into the camera's data. Frames are produced at regular time intervals and processed in sequence. Each frame is assigned a unique identifier corresponding to the order in which it was produced. The sequence of frames represents a temporal index into the camera's data. The spatiotemporal dimensions of the security camera are illustrated in Figure 1.

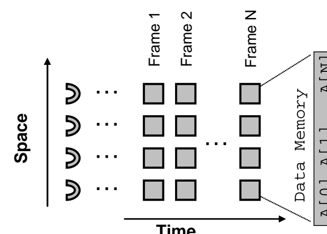


Figure 1. Spatiotemporal indexing of data in the security camera example. Spatially distributed sensors produce a sequence of pixels over time. The pixel's location in the frame is the spatial index while the frame number in the sequence is the temporal index.

The computation in the camera application consists of two primary functions: searching the frame for objects of interest and compressing the frames for storage. The compression operation is, in turn, made up of two distinct functions. First a series of image processing operations are executed to find and eliminate redundancy in the image stream, and, once the redundancy is eliminated, the remaining data is entropy encoded. Thus, there are three high-level functions which constitute the instructions of the camera: `search`, `eliminate`, and `encode`. As these functions occupy distinct regions of memory, their names can serve as spatial indices. To determine the temporal indices of these functions, note that the functions must be executed in a particular order to preserve correctness. The order of function execution represents an index into the temporal dimension of the camera's instructions. In this example `eliminate` must be executed before `encode`, but `search` is entirely independent. The spatiotemporal indexing of the camera's instructions is illustrated in Figure 2.

Note that the camera is typical of many interactive applications in that it has both latency and throughput requirements. The camera's throughput must keep up with the rate at which the sensor can produce frames. In addition, the camera must report objects of interest to the user with a low latency so that timely action can be taken.

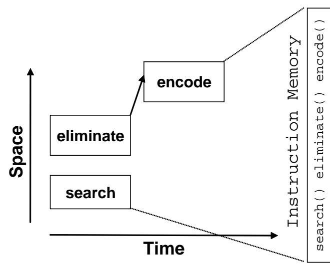


Figure 2. Spatiotemporal indexing of instructions in the security camera. Each frame produced by the camera is searched and encoded. The encoding process is further broken down into the eliminate redundancy and entropy encoding steps. The function name provides a spatial index. To define the temporal index, topological sort is performed on the dependence graph. The order of execution determined by the sort provides a temporal instruction index.

2.2 Terminology

In this paper, the term *program* or *application* is used to refer to the problem to be decomposed into concurrent parts. A *process* is the basic unit of program execution, and a parallel program is one that has multiple processes actively performing computation at one time. A parallel program is created by *partitioning* or *decomposing* a program into multiple processes. A partitioning strategy represents a common pattern for performing this decomposition.

Programs operate by executing *instructions* to manipulate *data*. Both the data and instructions of a program have spatial and temporal components of execution. In order to partition a program, one must first define the spatial and temporal dimensions of the program's execution.

2.3 Preparing to Partition a Program

The following procedure determines the dimensionality of a program's instructions and data to prepare it for partitioning:

1. Determine what constitutes a single input to define the temporal dimension of the program's data. For some programs an input might be a single reading from a sensor. In other cases an input might be a file, data from a keyboard or a value internally generated by the program.

2. Determine the distinct components of an input to define the spatial dimension of the program's data.

3. Determine the distinct functions required to process an input to define the spatial dimension of the program's instructions.

4. Determine the partial ordering of functions using topological sort to define the temporal dimension of the program's instructions.

To illustrate the process, it is applied to the security camera example:

1. A single frame is an input, so the sequence of frames represents the temporal dimension of the camera data.

2. A frame is composed of individual pixels arranged in a two-dimensional array. The coordinates of pixels in the array represent the spatial dimension of the camera data.

3. The three major functions in the camera are: `search`, `eliminate`, and `encode`. These functions define the spatial dimension of the camera's instructions.

4. For a given frame, there is a dependence between the `eliminate` and `encode` functions while the `search` function is independent. These dependences determine the temporal dimension of the camera's instructions.

3 Spatiotemporal Partitioning Strategies

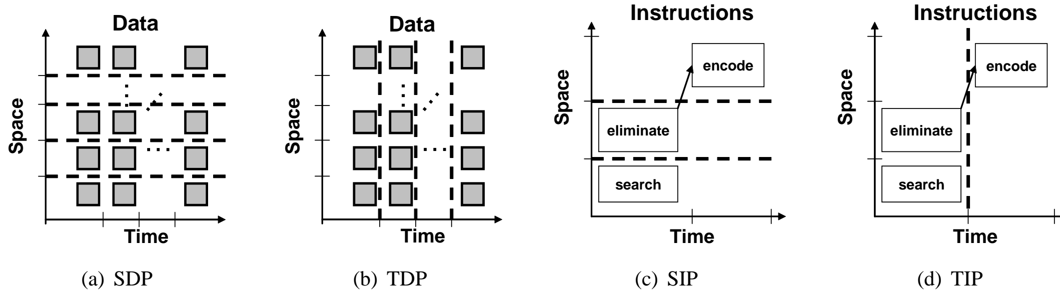
The procedure described in Section 2.3 defines the spatial and temporal dimensionality of a program's instructions and data. Given this definition it is possible to apply one of the following four partitioning strategies: spatial data partitioning, temporal data partitioning, spatial instruction partitioning, and temporal instruction partitioning. This section presents the following for each of the four strategies:

- A brief description of the strategy.
- The applicability of the strategy.
- The forces, or trade offs, that influence the application of the strategy.
- An example showing how to apply the strategy to the security camera.
- Other example uses of the strategy.
- Related strategy patterns from Our Pattern Language (OPL) [14] and Patterns of Parallel Programming (PPP) [18].

3.1 Spatial Data Partitioning (SDP)

Description. Using the spatial data partitioning (SDP) strategy, data is divided among processes according to spatial index. Following this pattern, processes perform computation on spatially distinct data with the same temporal index. Typically, each process will perform all instructions on its assigned data. Additional instructions are usually added to SDP programs to enable communication and synchronization. This partitioning strategy is illustrated in Figure 3(a).

Applicability. SDP generally increases throughput and decreases latency, so it is useful when a single processor cannot meet either of these requirements for a given



(a) SDP (b) TDP (c) SIP (d) TIP

Figure 3. Spatiotemporal partitioning strategies.

application. The performance of an SDP application will be best if the spatial data dimension is large and has few dependences.

Forces. The driving force behind the use of the SDP strategy is the improvement of both throughput and latency for the partitioned application. However, when applying this strategy one must consider the opposing forces of communication and load-balancing.

Communication can oppose the performance gains of the SDP strategy if the time spent transferring data is large enough to negate the performance benefit of parallelization. Unfortunately, communication in an SDP implementation is always application specific and, thus, there are no general guidelines for analyzing communication using this strategy. However, if a particular application is found to require a performance-limiting amount of communication, spatial instruction partitioning may be considered, as SIP can also improve throughput and latency.

Load-balancing is another force limiting the potential performance gains of SDP. Since each process is responsible for executing all instructions on its assigned data the load-balance of an SDP application is determined by the relationship between the instructions and data. If the same instructions are executed regardless of the value of data, then an SDP implementation will be easy to load-balance. If the instructions executed are conditional on the value of data then an SDP implementation may be very difficult to load-balance as some data may cause one process to do more work than others. This difficulty can offset some of the performance gains of an SDP implementation. Implementation strategy patterns like the task-queue and the master-worker pattern can help address load-imbalance [14].

Example in security camera. To implement the SDP strategy in the security camera example, separate processes work simultaneously on pixels from the same frame. Each process is responsible for executing the `search`, `eliminate`, and `encode` functions on its assigned pixels. Processes communicate with other processes responsible for neighboring spatial indices.

Other examples of SDP. This pattern is common in many parallel linear algebra implementations like ScaLAPACK [1] and PLAPACK [24]. The High Performance Fortran language is designed to express and manipulate SDP at a high-level [8]. Single-instruction, multiple-data (SIMD) processor architectures are designed to exploit SDP [5].

Related Strategy Patterns. DataDecomposition from PPP and Data Parallelism from OPL are both used to take a data-centric approach to partitioning an application into concurrent units. In order to perform this decomposition an index set is defined over the program's data. The spatial data partitioning strategy is a sub-strategy of DataDecomposition and Data Parallelism in which the index set is restricted to spatial indices.

The TaskDecomposition (PPP) and Task Parallelism (OPL) patterns both take a task-centric approach to application decomposition. These patterns allow individual tasks to be composed of the same set of instructions. The case where each task is the same function applied to different spatial indices is equivalent to the SDP strategy.

3.2 Temporal Data Partitioning (TDP)

Description. Using the temporal data partitioning (TDP) strategy, data are divided among processes according to temporal index. Following this pattern, each process performs computation on all spatial indices associated with its assigned temporal index as illustrated in Figure 3(b). In a typical TDP implementation each process executes all instructions on the data from its assigned temporal index. Often, communication and synchronization instructions need to be added to allow processes to handle temporal data dependences.

Applicability. TDP can increase throughput but will not decrease latency. Thus, this strategy is useful when a single processor can meet the application's latency requirement, but not the throughput requirement. The performance of a TDP application will be best when the temporal dimension is large and has few dependences.

Forces. The driving force behind the use of the TDP pattern is the improvement in throughput. Opposing this force is the possible overhead of communication and load-balancing.

Communication overhead can counter the throughput gains of applying the TDP strategy. In addition, it is pos-

sible that the added communication can increase the application's latency compared to a single processor implementation. Communication in this pattern occurs when data produced at one temporal index is needed to process data at another temporal index. If the amount of communication required is found to limit performance, any of the other three spatiotemporal partitioning strategies can be tried as they all improve throughput.

Load-balancing in a TDP application tends to be easy. If the same work is done for each temporal data index then achieving good load-balance is trivial. If the computational load varies per temporal data index it can be more difficult to achieve an efficient load-balance; however, the TDP strategy naturally lends itself to implementation through patterns such as the work-queue and master-worker that can be used to address a load-imbalance [14, 18].

Example in security camera. To implement TDP in the security camera example, each frame is assigned to a separate process and multiple frames are encoded simultaneously. A process is responsible for executing the `eliminate`, `encode`, and `search` functions on its assigned frame. Processes receive data from processes working on earlier temporal indices and send data to processes working on later temporal indices.

Other examples of TDP. Packet processing applications like SNORT are commonly parallelized using the TDP pattern [10]. This application processes a sequence of packets from a network, possibly searching them for viruses. As packets arrive they are assigned to a process which performs the required computation on that packet. The back-end of the StreamIt compiler for MIT's Raw processor generates code parallelized according to the TDP strategy [6].

Related Strategy Patterns. The DataDecomposition (PPP) and Data Parallelism (OPL) patterns are both used to take a data-centric approach to partitioning an application into concurrent units. In order to perform this decomposition an index set is defined over the program's data. The temporal data partitioning strategy is a sub-strategy of DataDecomposition and Data Parallelism in which the index set is restricted to temporal indices.

Using the Pipeline strategy pattern from OPL, an application is partitioned into stages and data flows from one stage to the next. If each stage consists of the same instructions processing data from different temporal indices then this instance of the Pipeline strategy is equivalent to TDP.

The TaskDecomposition (PPP) and Task Parallelism (OPL) patterns both take a task-centric approach to partitioning an application. These patterns do not necessarily require that the tasks represent distinct functions or instructions. The specific case where each task represents the same set of functions applied to distinct temporal data indices is equivalent to TDP.

3.3 Spatial Instruction Partitioning (SIP)

Description. Using the spatial instruction partitioning (SIP) strategy, instructions are divided amongst processes according to spatial index. Following this pattern, each process performs a distinct computation using the same data. Often no communication is needed between processes. This strategy is illustrated in Figure 3(c).

Applicability. The SIP strategy can increase application throughput and decrease latency, so it is useful when a single processor cannot meet the application's throughput or latency requirements. In addition, the SIP strategy divides an application into several separate functional modules. Splitting a large, complicated application into multiple simpler pieces aids modular design and allows multiple engineers to work on an application independently. Furthermore, splitting the instructions of the application may simplify data dependences as the single set of dependences for the entire application is divided into several smaller sets of dependences for each module.

Forces. The performance gain and increase in modularity offered by the SIP strategy is opposed by the forces of communication overhead and load-balance.

Communication in SIP implementations is generally rare, given the definition of the temporal and spatial dimensions of instructions. Communication in SIP applications typically involves making the same data available to all processes. In some instances processes might need to communicate while executing their assigned instructions. In both cases it is important to ensure that the cost of communication does not negate any performance gains from applying SIP. If communication is found to limit performance, then spatial data partitioning should be investigated as it can also increase throughput while decreasing latency.

Load-balance is often an issue in SIP applications. Instructions are assigned to processes because they represent logically coherent units. Unfortunately it is rare that these units represent the same amount of computation and, thus, it is often the case that one process is assigned more work than another. The work-queue and master-worker patterns can be used to implement the SIP strategy and help address load-balance, but their applicability can be limited by the (generally low) amount of parallelism that exists in the spatial instruction dimension.

Example in the security camera. To implement SIP in the security camera, the `eliminate` and `encode` functions are coalesced into one `compress` function. This function is assigned to one process while the `search` function is assigned to a separate process. These two processes work on the same input frame at the same time. In this example, the two processes need not communicate. However, one can envision a camera that alters the quality of the compressed video based on the presence of an object of interest. In this scenario, the `search` function sends messages to the `compress` function to indicate when the quality should change.

Other examples of SIP. This pattern is sometimes

used in image processing applications when two filters are applied to the same input image to extract different sets of features. In such an application each of the two filters represents a separate function and therefore a separate spatial instruction index. This application can be parallelized according the SIP strategy by assigning each filter to a separate process. Compilers that exploit instruction level parallelism often make use of this pattern as Butts and Sohi found that $35\,\%$ of dynamically created values are used multiple times [3]. Some speculative execution systems, use SIP partitioning to execute multiple paths through a series of conditional instructions [23]. Additionally, this is strategy is the basis of multiple-instruction single-data (MISD) architectures [5].

Related Strategy Patterns. The TaskDecomposition (PPP) and Task Parallelism (OPL) patterns both take a task-centric view of an application. The SIP strategy corresponds to instances of these patterns where the tasks represent distinct functions and all tasks can execute concurrently while processing data from the same temporal index.

3.4 Temporal Instruction Partitioning (TIP)

Description. Using the temporal instruction partitioning (TIP) strategy, instructions are divided among processes according to temporal index as illustrated in Figure 3(d). In a TIP application each process executes a distinct function and data flows from one process to another as defined by the dependence graph. This flow of data means that TIP applications always require communication instructions so that the output of one process can be used as input to another process. To achieve performance, this pattern relies on a long sequence of input data and each process executes a function on data that is associated with a different input or temporal data index.

Applicability. The TIP strategy can increase throughput but does not improve latency. Therefore, this strategy is most useful when a single processor cannot meet the application's throughput requirement, but can meet the latency requirement. Like the SIP strategy, TIP can also be useful for splitting a large application into smaller functional modules, which can simplify application development and separate a complicated set of data dependences into smaller units that are each individually easier to implement.

Forces. Countering the driving forces of throughput increase and increased modularity are the opposing forces of increased communication and inefficient load-balance.

The TIP strategy is guaranteed to require communication because the temporal instruction indices are defined according to dependences. Therefore, every process in a TIP application can be expected to communicate with at least one other process. In addition to (possibly) limiting the throughput gain, this communication will also increase latency unless it is completely overlapped with computation. If the communication in a TIP application proves too deleterious to performance, any of the other three strategies can be explored as they all increase throughput. The SIP strategy is a useful alternative for modular development.

Load-balancing in TIP applications is often difficult. As with SIP applications, it is rare that the functions represented by different temporal instruction indices require the same amount of computation. Thus, some processes may be assigned more work than others. Again, both the work-queue and master-worker patterns may be used to implement TIP and address the load-balancing issue [14, 18]; however, in practice these patterns are sometimes conceptually difficult to reconcile with the TIP strategy.

Example in the security camera. To implement TIP in the security camera, the `eliminate` function is assigned to one process while the `encode` function is assigned to another. The `search` function can be assigned to a third process or it can be coalesced with one of the other functions to help with load balancing. In this implementation, one process executes the `eliminate` function for frame N while a second process executes the `encode` function for frame $N-1$.

Other examples of TIP. This pattern is often used to implement digital signal processing applications as they are easily expressed as a chain of dependent functions. In addition, this pattern forms the basis of streaming languages like StreamIt [7] and Brook [2]. The Pipeline construct in Intel's Threading Building Blocks also exploits the TIP strategy [9].

Related Parallel Patterns. The TaskDecomposition (PPP) and Task Parallelism (OPL) patterns both take a task-centric view of an application. The TIP strategy corresponds to instances of these patterns where the tasks are distinct functions connected through a chain of dependences. Similarly, the TIP pattern is equivalent to the Pipeline strategy pattern (OPL) if all stages of the pipeline represent distinct functions which process data from different temporal indices simultaneously.

3.5 Combining Strategies

It can often be helpful to combine partitioning strategies within an application to provide the benefit of multiple strategies, or to provide an increased degree of parallelism. The application of multiple strategies is viewed as a sequence of choices. The first choice creates multiple processes and these processes can then be further partitioned as if they were serial applications themselves.

Combining multiple choices in sequence allows the development of arbitrarily complex parallel programs and the security camera example demonstrates the benefits of combining multiple strategies. As illustrated above it is possible to use SIP partitioning to split the camera application into two processes. One process is responsible for the `search` function while another is responsible for the `eliminate` and `encode` functions. This SIP partitioning is useful because the searching computation and the compression computation have very different dependences and splitting them creates two simpler modules. Each of these simple modules is easier to understand and paral-

lelize. While this partitioning improves throughput and latency, it suffers from a load-imbalance, because the searching and compressing functions do not have the same computational demand.

To address the load-imbalance and latency issues, one may apply a further partitioning strategy. For example, applying the SDP strategy to the process executing the compression will split that computationally intensive function into smaller parts. This additional partitioning helps to improve load-balance and application latency. (For a more detailed discussion of using SIP and SDP together in the camera, see [16].)

The next section includes other examples using multiple strategies to partition an application.

4 Results

This section presents the results of applying various partitioning strategies to an H.264 encoder for high-definition video. These results demonstrate how the choice of strategy can affect both the latency and the throughput of the encoder.

This study uses the x264 implementation of the H.264 standard [25]. x264 is implemented in C with assembly to take advantage of SIMD instructions and hardware specific features. x264 is currently parallelized using the pthreads library[1] to implement the TDP strategy, while earlier versions of the encoder implemented a limited form of the SDP strategy (which is no longer supported)[2]. x264 is highly configurable via the command line. For these results, x264 is configured with a computationally aggressive parameter set to enable H.264 Main profile encoding of 1080p high-definition video[3]. All partitionings of the x264 code base are tested on an Intel x86 system. This system consists of four 2.4 GHz quad-core x86 processors communicating through shared memory. For this study, the hardware platform is treated as a sixteen-core multicore.

To prepare x264 for partitioning the procedure described in Section 2.3 is applied:

1. An input is a frame and the sequence of frames represents the temporal data index.
2. Each frame can be divided into one or more *slices*. A slice consists of *macroblocks* which are 16×16 regions of pixels. Slices are defined by H.264 to be independent. Both the slice and the macroblock represent spatial data indices.
3. The instructions that eliminate redundancy (implemented in x264 in the functions `x264_macroblock_analyse` and `x264_macroblock_encode`) and the entropy encoding instructions (implemented in x264 as the x264_cavlc_write and the x264_cabac family of functions) represent the spatial instruction indices.
4. The encoding functions are dependent on the completion of the elimination function, so the elimination function must be executed first. This ordering represents the temporal data index.

At this point, various partitioning strategies are employed using the indices described above. For each parallelization the latency and throughput of the encoder is measured, and both the actual values and the speedups over the serial throughput and latency are reported. Additionally, some features of H.264 allow programmers to relax data dependencies at the cost of lost quality and decreased compression, so image quality (in decibels of peak signal to noise ratio [PSNR]) and achieved compression (measured in kilobits/s) are also recorded. All values are compared to those of a serial implementation of the encoder.

First, the TDP strategy is used to assign frames to processes and each process executes all instructions on its assigned frame. Processes need data from previously compressed frames and this requires communication. As shown in Table 1 this strategy increases throughput, but the added communication is detrimental to latency.

To reduce latency each frame is divided into slices (or spatial indices); using the SDP strategy slices are assigned to processes and all processes execute the elimination and encoding functions on their assigned slice. As shown in Table 2, the SDP strategy increases throughput and reduces latency, but does so at the expense of image quality and compression. One would like to find a parallelization that does not rely on slices, but that is difficult because the elimination and entropy encoding functions have different data dependences and handling both in the same process is extremely complicated. Slices make this easy by eliminating dependences.

An alternative to using slices is to separate the instructions that eliminate redundancy from those that encode the resulting data. To accomplish this the TIP strategy is employed, and all elimination functions are assigned to one process while all encoding functions are assigned to another. The elimination process works on frame N while the encoding process concurrently works on frame $N-1$. Clearly the processes must communicate to produce correct results. As shown in Table 3 the throughput of the encoder decreases, while the latency becomes worse due to the added communication. In addition, the load-balancing of this parallelization is extremely poor as there is much more work in the elimination functions than in the encoding functions. However, the use of TIP has accomplished the goal of separating a complicated application into two separate modules and has done so while maintaining image

[1] Although the case study uses the pthreads library, the term process will continue to be used to describe units of program execution.

[2] The TDP implementation of the case study uses x264 as is. The SDP implementation recreated the earlier version of x264 in the current code base. The other three partitionings in presented here represent original work.

[3] Specifically, x264 is invoked with the following command line parameters: `-qp 25 --partitions all --bframes 3 --ref 5 --direct auto --no-b-adapt --frames 120 --weightb --bime --mixed-refs --no-fast-pskip --me umh --subme 5 --scenecut -1`. The input video is a 1080p sequence of a tractor plowing a field.

	Serial x264	TDP x264				
Processes	1	2	4	8	12	16
Throughput (frames/s)	1.62	2.75	4.06	7.96	11.64	14.66
Throughput Speedup	1	1.62	2.39	4.68	6.85	8.62
Average frame latency (ms)	580	719	962	968	977	1004
Latency Speedup	1	0.81	0.60	0.60	0.59	0.58
Image Quality (Average PSNR)	39.371	39.372	39.373	39.372	39.374	39.372
Encoded Bitrate (Kb/s)	10647.57	10651.38	10656.11	10657.83	10670.81	10694.58

Table 1. Performance of parallel x264 decomposed with the TDP strategy.

	Serial x264	SDP x264				
Processes	1	2	4	8	12	16
Throughput (frames/s)	1.70	3.08	5.71	10.14	12.55	11.36
Throughput Speedup	1	1.81	3.36	5.96	7.38	6.68
Average frame latency (ms)	580	313	163	90	67	82
Latency Speedup	1	1.85	3.56	6.44	8.66	7.07
Image Quality (Average PSNR)	39.371	39.179	39.169	39.151	39.127	39.112
Encoded Bitrate (Kb/s)	10647.57	11078.18	11175.96	11367.62	11574.76	11787.89

Table 2. Performance of parallel x264 decomposed with the SDP strategy.

quality and compression.

Having used TIP to separate the application into elimination and encoding modules, SDP is applied to the process responsible for the elimination functions to help load balancing and improve performance. Now that elimination and encoding are separate, one need not resort to using slices for independence. In this implementation, rows of macroblocks are assigned to processes to parallelize the elimination of redundancy. Thus, the TIP strategy is used to split the elimination and encoding functions, then the SDP strategy is used to further parallelize the elimination step. Table 4 shows that this parallelization represents a compromise, improving both latency and throughput, but not achieving the best result for either metric. In addition, this performance is achieved without a significant loss of quality or compression.

Finally, TDP, TIP, and SDP can be combined to create a single code-base that can be tuned to meet differing performance and quality tradeoffs. In this approach, the TDP strategy is used to allow the encoding of multiple frames in parallel. Each frame is encoded by one or more processes. If one process is assigned to a frame, this approach functions as the pure TIP approach. If two processes are assigned to a frame, they are parallelized using the TIP strategy. If three or more processes are assigned to a frame the combination of TIP and SDP is used. Table 5 shows the performance of this approach. In this table all results are presented using sixteen processes, but the number of processes assigned to a single frame is varied. Combining the TDP, TIP, and SDP strategies creates a code base that is flexible and can be adapted to meet the needs of a particular user.

The results presented in this section demonstrate the benefits of these strategies. Different strategies have different effects on throughput and latency and understanding these effects a priori can save valuable engineering time. For example, an engineer knows that her video encoder will be used as part of a video conferencing system with strict latency requirements. In this case, it makes sense to use the SDP strategy, because it achieves the best latency. Knowing the effects ahead of time saves the effort of trying multiple approaches until the desired result is found.

5 Related Work

5.1 Related Patterns

The use of design patterns for parallel software development can add rigor and discipline to what is otherwise an ad hoc and artistic process. A number of parallel design patterns have been identified [17, 18, 20, 15, 21, 14]. These patterns range from very high-level descriptions, such as the commonly used task and data parallel patterns which may be appropriate for any application, to low-level patterns, such as a parallel hash table, which may be considered only for specific applications. Parallel pattern languages [18, 14] guide users through the application of both high- and low-level patterns.

Throughout Section 3 the four partitioning strategies described in this paper are compared to patterns from Our Pattern Language (OPL) [14] and Patterns of Parallel Programming (PPP) [18]. It is clear from this comparison that there is not a one-to-one mapping of these existing patterns onto the strategies presented here. For example, using the TaskDecomposition pattern of PPP can result in a program that is partitioned according to any one of the four strategies presented here. This property creates some ambiguity when reasoning about the effects that a decomposition will have on a parallelized program.

	Serial x264	TIP x264
Processes	1	2
Throughput (frames/s)	1.70	1.54
Throughput Speedup	1	0.90
Average frame latency (ms)	580	1435
Latency Speedup	1	0.86
Image Quality (Average PSNR)	39.371	39.387
Encoded Bitrate (Kb/s)	10647.57	10913.49

Table 3. Performance of parallel x264 decomposed with the TIP strategy.

	Serial x264	TIP+SDP x264				
Processes	1	2	4	8	12	16
Throughput (frames/s)	1.70	1.54	4.11	8.1	10.25	8.41
Throughput Speedup	1	0.91	2.42	4.76	6.03	4.95
Average frame latency (ms)	580	672	264	145	118	140
Latency Speedup	1	0.86	2.20	4.00	4.92	4.14
Image Quality (Average PSNR)	39.371	39.387	39.387	39.387	39.387	39.387
Encoded Bitrate (Kb/s)	10694.58	10913.49	10913.49	10913.49	10913.49	10913.49

Table 4. Performance of parallel x264 decomposed with a combination of the TIP and SDP strategies.

This ambiguity arises from the fact that the strategy patterns of OPL and PPP are designed to capture the way programmers think about the program while the partitioning strategies presented in this document are designed to capture the effects parallelism has on the application. These two approaches are complementary. The patterns of PPP and OPL give software engineers a language for describing common approaches to parallelism while the partitioning strategies presented here provide a framework for about the performance (in terms of throughput and latency) application. Understanding both allows a more complete description of a parallel system, better communication about the design of the system, and clearer reasoning about how high-level choices affect the performance of the parallelized application.

5.2 Related Video Encoders

Liu et al. explore a parallel implementation of a security camera which combines multiple partitioning strategies in the same application [16]. These authors first use the SIP strategy to split the search and compression functions of the camera into separate processes, each of which is further partitioned using the SDP strategy. Rodriguez et al. describe a TDP partitioning of an H.264 encoder as well as an implementation that uses both TDP and SDP through the employment of slices [19]. Several authors describe slice-based SDP partitioning strategies [12, 4]. Jung et al. adaptively choose the slice sizes in order to address the load-balancing issue discussed above [13]. Sun et al. describe a pure SDP implementation that does not rely on slicing, but still respects the data dependences described above [22]. Park et al. apply TIP partitioning to create their parallel H.264 encoder, but rather than splitting the image and entropy functions, they split the elimination of redundancy into two functions: the first consists of finding temporal redundancy and the second consists of all other elimination functions. The first function is assigned to one process and all other redundancy elimination and encoding functions are placed in a second process. This implementation suffers from the same load imbalance described for the TIP implementation above. To help address this issue, SDP partitioning is applied to the process responsible for finding redundancy.

We note that these alternative parallelizations of video encoders are generally presented without a discussion of the possibility of other parallel implementations or a discussion of the tradeoffs inherent to the chosen parallelization. Incorporating the partitioning strategies shown here into the presentation of novel parallelizations would provide the foundation for exploring these issues and quickly putting new work into context.

6 Conclusion

This paper has presented an extension to the commonly used task and data parallel design patterns. This extension is based on the spatiotemporal nature of program execution and the different effects of parallelizing in space and time. The four strategies discussed are spatial data partitioning (SDP), temporal data partitioning (TDP), spatial instruction partitioning (SIP), and temporal instruction partitioning (TIP). Results implementing a substantial application demonstrate how these strategies can be applied to affect both latency and throughput. In addition these results illustrate how multiple partitioning strategies can be combined to yield the benefits of each.

	Serial x264	TDP+TIP+SDP x264				
Parallel frames	1	1	2	4	8	16
Processes per frame	1	16	8	4	2	1
Throughput (frames/s)	1.70	8.41	12.11	10.67	7.73	14.66
Throughput Speedup	1	4.95	7.12	6.28	4.55	8.62
Average frame latency (ms)	580	140	188	391	1032	1004
Latency Speedup	1	4.14	3.09	1.48	0.56	0.58
Image Quality (Average PSNR)	39.371	39.387	39.378	39.382	39.377	39.372
Encoded Bitrate (Kb/s)	10647.57	10913.49	10847.80	10634.21	10294.74	10694.58

Table 5. Performance of parallel x264 decomposed with a combination of the TDP, TIP, and SDP strategies.

References

[1] L. S. Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, 1996.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004.

[3] J. A. Butts and G. S. Sohi. Characterizing and predicting value degree of use. In *MICRO*, pages 15–26, 2002.

[4] Y.-K. Chen, X. Tian, S. Ge, and M. Girkar. Towards efficient multi-level threading of H.264 encoder on Intel hyper-threading architectures. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004.

[5] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12), Dec. 1966.

[6] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[7] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffmann, D. Z. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[8] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225.

[9] Intel Threading Building Blocks 2.1 for Open Source. http://threadingbuildingblocks.org/.

[10] Supra-linear packet processing performance with Intel multi-core processors. Technical report, Intel, 2006.

[11] ITU-T. H.264: Advanced video coding for generic audiovisual services.

[12] T. Jacobs, V. Chouliaras, and D. Mulvaney. Thread-parallel MPEG-4 and H.264 coders for system-on-chip multi-processor architectures. *International Conference on Consumer Electronics*, Jan. 2006.

[13] B. Jung, H. Lee, K.-H. Won, and B. Jeon. Adaptive slice-level parallelism for real-time H.264/AVC encoder with fast inter mode selection. In *Multimedia Systems and Applications X.*, 2007.

[14] K. Keutzer and T. Mattson. Our Pattern Language (OPL): A design pattern language for engineering (parallel) software. In *ParaPLoP: Workshop on Parallel Programming Patterns*, 2009.

[15] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. 1996.

[16] L.-K. Liu, S. Kesavarapu, J. Connell, A. Jagmohan, L. hoon Leem, B. Paulovicks, V. Sheinin, L. Tang, and H. Yeo. Video analysis and compression on the STI Cell Broadband Engine processor. In *ICME*.

[17] B. L. Massingill, T. G. Mattson, and B. A. Sanders. A pattern language for parallel application programs (research note). In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 678–681, London, UK, 2000. Springer-Verlag.

[18] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. 2004.

[19] A. Rodriguez, A. Gonzalez, and M. P. Malumbres. Hierarchical parallelization of an H.264/AVC video encoder. In *PARELEC '06: Proceedings of the international symposium on Parallel Computing in Electrical Engineering*, 2006.

[20] S. Siu, M. D. Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*.

[21] M. Snir. Parallel programming patterns. http://www.cs.uiuc.edu/homes/snir/PPP/.

[22] S. Sun, D. Wang, and S. Chen. A highly efficient parallel algorithm for H.264 encoder based on macro-block region partition. In *HPCC '07: Proceedings of the 3rd international conference on High Performance Computing and Communications*, 2007.

[23] K. Theobald, G. R. Gao, and L. J. Hendren. Speculative execution and branch prediction on parallel machines. In *In Conference Proceedings, 1993 International Conference on Supercomputing*, 1993.

[24] R. van de Geijn. *Using PLAPACK – Parallel Linear Algebra Package*. MIT Press, Cambridge, MA, 1997.

[25] x264. http://www.videolan.org/x264.html.