

Partitioning Strategies for Concurrent Programming

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

Abstract

This work presents four partitioning strategies, or design patterns, useful for decomposing a serial application into multiple concurrently executing parts. These partitioning strategies augment the commonly used task and data parallel design 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, this work describes four decomposition strategies: spatial data partitioning (SDP), temporal data partitioning (TDP), spatial instruction partitioning (SIP), and temporal instruction partitioning (TIP), while cataloging the benefits and drawbacks of each. These strategies can be combined to realize the benefits of multiple patterns in the same program. The practical use of the partitioning strategies is demonstrated through a case study which implements several different parallelizations of a multicore H.264 encoder for HD video. This case study illustrates the application of the patterns, their effects on the performance of the encoder, and the combination of multiple strategies in a single program.

1 Introduction

Design patterns for parallel computing help to add structure and discipline to the process of concurrent software development [11, 13, 14, 15, 18, 19]. Two of the most commonly referenced parallel patterns are *task* and *data* parallelism. Using the task parallel pattern, a program is decomposed into concurrent units which execute separate instructions simultaneously. Using the data parallel pattern a program is decomposed into concurrent units which execute the same instructions on distinct data.

This work extends both the task and data parallel patterns by noting that applications execute in time and space. If one assigns spatial and temporal indices to a program's data and instructions, then it is possible to decompose both data and instructions in time and space. Thus, this work recognizes four partitioning strategies for finding 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 several reasons. First, this distinction provides an additional set of options for finding concurrency in an application. Second, it provides greater descriptive power for documenting and characterizing a parallel application. Finally, and 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 presents a case study in which several different strategies are applied to create parallel implementations of an H.264 video encoder [8] on a multicore architecture. The case study

demonstrates how the additional options of temporal and spatial partitioning can aid programming. In addition, the effects of different strategies on the throughput and latency of the encoder are cataloged and the combination of multiple strategies is illustrated.

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 case study illustrating the use of these patterns. Section 5 covers related work and 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: an intelligent security camera

An intelligent security camera processes a sequence of input images, or frames, from a camera (e.g. [12]). 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.

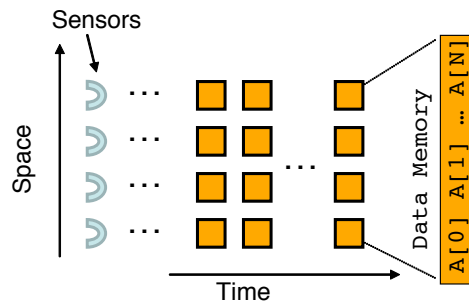


Figure 1. Spatiotemporal indexing of data in the security camera example. Spatially distributed sensors produce a sequence of pixels over time. The pixel's location in the frame is the spatial index while the frame number in the sequence in the temporal index.

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 remove 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`, `image`, and `entropy`. 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 case `image` must be executed before `entropy`, but `search` is entirely independent. The spatiotemporal indexing of the camera's instructions is illustrated in Figure 2.

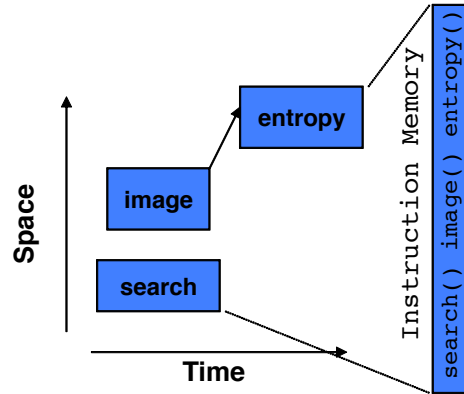


Figure 2. Spatiotemporal indexing of instructions in the security camera. Each frame produced by the camera is searched and encoded. The encoding process is further broken down into image processing and entropy encoding functions. The topological sort of the dependence graph provides a temporal index, while the function name provides a spatial index.

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.

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 design 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 on the program dependence graph 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`, `image`, and `entropy`. These functions define the spatial dimension of the camera's instructions.
4. For a given frame, there is a dependence between the `image` and `entropy` functions while the `search` function is independent. These dependences determine the temporal dimension of the camera's instructions.

Applying this procedure defines the dimensionality of a program's data and instructions. Once this dimensionality is defined, it is possible to explore different spatiotemporal partitioning strategies.

3 A taxonomy of 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. For each of these strategies, this section presents

- A brief description of the strategy.
- An example illustrating how the strategy could apply to the security camera.
- Other common examples of the strategy.
- A description of the effects of a partitioned application relative to the serial one.
- The applicability, or common usage, of the strategy is discussed.

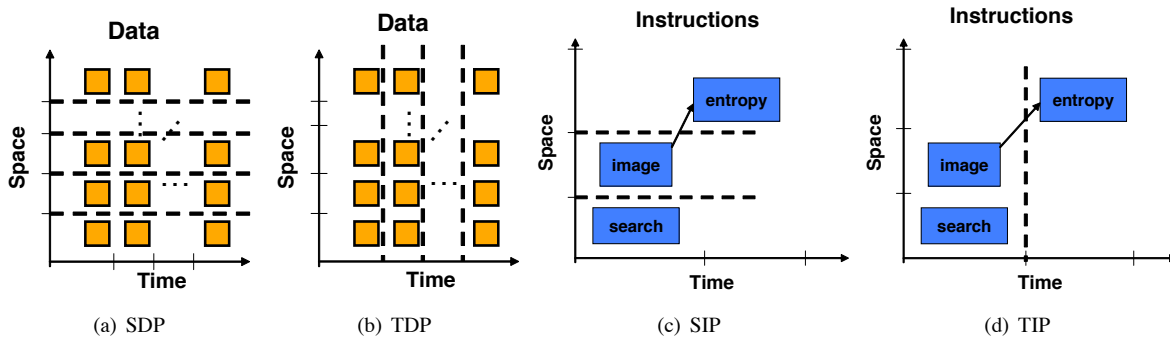


Figure 3. Parallelization strategies.

3.1 Spatial data partitioning

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).

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`, `image`, and `entropy` functions on its assigned pixels. Processes communicate with other processes responsible for neighboring spatial indices.

Other common examples. Jacobi relaxation is often parallelized using the SDP pattern. This application iteratively updates a matrix A . At each iteration the new value of a matrix element is computed as the average of its neighbors. In the SDP implementation of Jacobi relaxation, each process is assigned a region of the matrix and is responsible for computing the updates to that region for each iteration. This pattern is also common in many parallel linear algebra implementations like ScaLAPACK [1] and PLAPCK [21]. Also, the high performance Fortran language is designed to express and manipulate SDP [7].

Effects on application. An application parallelized with the SDP pattern generally has the following effects relative to a serial implementation. The throughput of the application improves. The latency of the parallelized application decreases. The load-balancing of the parallelized application tends to be easy, as the same instructions are often executed in each process. Finally, the communication in the parallel implementation is always application dependent.

Applicability. Use the SDP strategy to parallelize an application when:

- A single processor will not meet the application's latency demands.
- The spatial data dimension is large and has few dependences.
- The application performs similar amounts of work on each of the spatial indices, which makes it easy to load balance.

3.2 Temporal data partitioning

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.

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 `image`, `entropy`, 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 common examples. Packet processing is commonly parallelized using the TDP pattern. 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.

This pattern is sometimes called round-robin parallelism as temporal indices are often distributed among processes in a round-robin fashion. Additionally this pattern is often associated with a master-worker pattern; however, it is distinct as no master process is required to implement the TDP strategy.

Effects on application. An application parallelized with the TDP pattern generally has the following behavior compared to a serial implementation. The throughput of the application increases. The latency of the application remains the same. The load-balancing of the parallelized application tends to be easy. Even when the computation varies tremendously between inputs, it is often easy to load-balance applications written using this strategy by combining it with another pattern. For example, a master-worker pattern can allow the master to manage the load of the individual workers or a work-queue pattern can allow the individual processes to load-balance themselves through the work queue. Finally, the communication required in the parallel implementation is always application dependent.

Applicability. Use the TDP strategy to parallelize an application when:

- A single processor cannot meet the application's throughput requirement, but can meet the application's latency requirement.
- The temporal data dimension is large and has few dependences.
- The application performs widely different computation on inputs and may benefit from load-balancing patterns that synergize well with TDP.

3.3 Spatial instruction partitioning

Description. Using the spatial instruction partitioning (SIP) strategy, instructions are divided among 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).

Example in the security camera. To implement SIP in the security camera, the `image` and `entropy` 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.

Other common examples. 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. Additionally, this is the type of parallelism exploited by multiple-instruction single-data (MISD) architectures [5].

Effects on application. An application parallelized with the SIP pattern generally has the following behavior relative to a serial implementation. The throughput of the application increases. The latency of the application decreases. The load-balancing of the parallel application tends to be difficult as it is rarely the case that each function has the same compute requirements. The communication required in the parallel implementation is always application dependent, but generally low as functions that process the same input simultaneously typically require little communication. Finally, note that the SIP pattern breaks a large computation up into several smaller, independent functional modules. This can be useful for dividing application development among multiple engineers or simply for aiding modular design.

Applicability. Use the SIP strategy to parallelize an application when:

- A single processor cannot meet the application’s latency requirements.
- An application computes several different functions on the same input.
- It is useful to split a large and complicated algorithm into several smaller modules.

3.4 Temporal instruction partitioning

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.

Example in the security camera. To implement TIP in the security camera, the `image` function is assigned to one process while the `entropy` 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 execute the `image` function for frame N while a second process executes the `entropy` function for frame $N - 1$.

Other common examples. This pattern is commonly used and sometimes called task parallelism, functional parallelism, or pipeline parallelism. 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 many streaming languages like StreamIt [6] and Brook [2].

Effects on application. An application parallelized with the TIP pattern has the following characteristics compared to a serial implementation. The throughput of the application increases. The latency of the application remains the same, and sometimes can get worse due to the added overhead of communication. The load-balancing of the parallel application is generally hard as the individual functions rarely have the same computational needs. TIP applications always require communication, although the volume of communication is application independent. Finally, note that the SIP pattern breaks a large computation up into several smaller, independent functional modules. This can be useful for dividing application development among multiple engineers, or simply to create a modular application.

Applicability. Use the TIP strategy to parallelize an application when:

- A single processor cannot meet the application’s throughput requirement, but it can meet the application’s latency requirements.
- An application consists of sequence of stages where each function is a consumer of data from the previous stage and a producer of data for the subsequent stage.
- The amount of work in each stage is enough to allow overlap with the communication required between stages.
- It is useful to split a large and complicated algorithm into several smaller modules.

3.5 Combining multiple strategies in a single program

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.

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 `image` and `entropy` 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 these simple modules is easier to understand and parallelize. 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 in the camera, see [12].)

The case study presented in Section 4 includes other examples using multiple strategies to partition an application.

4 Case study

This section presents a case study using the partitioning strategies presented in Section 3 to create multicore implementations of an H.264 encoder for HD video. This study shows how the use of different strategies can affect both the development and performance of the parallelized application.

To begin, an overview of H.264 encoding covers the aspects of the application relevant to the partitioning. Next the hardware and software used in the study is briefly described. Then the employment of various partitioning strategies is described and the performance of each is presented.

First, the TDP strategy is used and is found to increase throughput, but to be detrimental to latency. Next SDP partitioning is used and demonstrates both throughput and latency improvements but requires some application specific sacrifice. TIP partitioning helps to simplify the design by decomposing a complicated set of dependences into simpler modules, but this simplification comes with a performance loss. Next a combination of TIP and SDP is found to provide a compromise between latency, throughput, and application specific concerns. Finally, the combination of TDP, TIP, and SDP is found to provide the maximum flexibility and allow the developer to tune latency and throughput requirements as desired.

4.1 H.264 overview

This section presents a brief overview of the aspects of H.264 encoding relevant to understanding the parallelization of the application. For a complete description of H.264 see [8, 16, 22].

H.264 is a standard for video compression. In fact, the H.264 standard could be used to implement the video compression discussed in the security camera example. The video consists of a stream of frames. In H.264 each frame consists of one or more *slices*. A slice consists of *macroblocks* which are 16×16 regions of pixels. These concepts are illustrated in Figure 4.

An H.264 encoder compresses video by finding temporal and spatial redundancy in a video stream. Temporal redundancy refers to similarities that persist from frame to frame, while spatial redundancy refers to similarities within

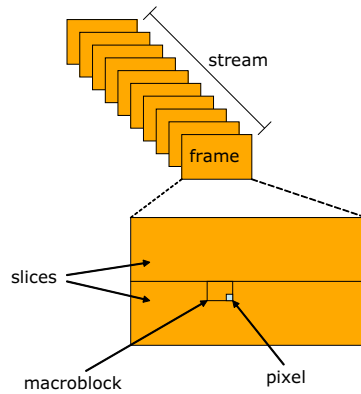


Figure 4. Video data for H.264 encoding.

the same frame. Once this redundancy is found and eliminated, the encoder may remove some additional information which is often unnoticeable to the human eye. (This loss of information is why H.264 is described as a "lossy" encoder.) Finally, the encoder performs entropy encoding on the transformed data for further compression.

Figure 5 shows the block diagram representing H.264 processing. The functions shown in the block diagram are used to process every macroblock in a frame. The analysis and motion estimation block is used to find temporal redundancy by searching for a similar region in a previously encoded frame and determining the best encoding modes for the macroblock. The frequency transform function is used to find temporal redundancy. Quantization removes low-frequency data to which the human eye is less sensitive. The inverse quantization and inverse transform blocks are used to reconstruct the frame as the decoder will see it. The deblocking filter removes artifacts inserted by the frequency transform. Finally, the frame store saves the reconstructed frame to serve as a reference for subsequently encoded frames. These functions represent the image processing portion of the encoding application and are analogous to the `image` function in the security encoder example.

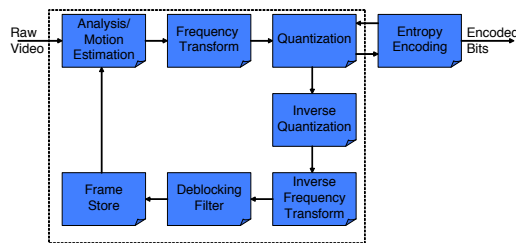


Figure 5. Block diagram illustrating instructions executed by H.264 encoder.

The remaining block in the diagram is labeled CAVLC/CABAC and performs entropy encoding on the transformed data. H.264 supports two separate entropy encoding modes. CAVLC stands for context adaptive variable length coding, while CABAC stands for context adaptive binary arithmetic coding.

During the image processing phase, H.264 observes the following data dependences. Frames are dependent on reference frames (except for specially designated frames which are encoded without a reference and allow the process to begin). Slices are completely independent as defined by the standard, but this independence comes at the cost of reduced ability to find spatial redundancy. Within a slice, there is a dependence between macroblocks. A macroblock depends on four neighboring macroblocks: above and to the left, above, above and to the right, and to the left. This dependence among macroblocks in a slice is illustrated in Figure 6.

During the entropy encoding phase each, H.264 observes the following dependences. The entropy encoding of a frame is dependent on previous frames, but this can be relaxed for a small overhead in the encoded bitstream. The entropy encoding of slices is independent, but this independence also adds extra bits to the encoding. Within a slice, macroblocks must be written to the output in raster order. Thus, the entropy phase as defined here is serial within a slice.

Having described the salient features of H.264, the experimental methodology used to evaluate the partitioning



Figure 6. Macroblock dependence in image processing section of H.264. A macroblock is dependent on neighboring macroblocks above-left, above, above-right, and to the left. All macroblocks labeled “1” must be processed before the macroblock labeled “2.”

strategies is discussed.

4.2 Experimental methodology

This section discusses the software and hardware used in the case study.

The open source x264 software code base is used as a basis for encoder development [23]. x264 is implemented in C with assembly to take advantage of SIMD instructions and hardware specific features. x264 is currently parallelized using the pthreads library¹ 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 this study x264 is configured with a computationally aggressive parameter set to enable H.264 Main profile encoding of 1080p high-definition video³.

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 system.

Each of the partitionings is implemented by modifying the x264 code and running the partitioned version on 2, 4, 8, and 16 cores with the number of processes equal to the number of cores. For each implementation and process count, the throughput (in frames/second) and latency (in milliseconds) are measured. In addition, as discussed above, some features of H.264 allow programmers to relax data dependences at the cost of lost quality and decreased compression. For this reason, the quality of each partitioning is also measured by recording the peak signal to noise ratio (PSNR measured in decibels) and the achieved compression in terms of output bitrate (measured in kilobits/second). For reference Table 1 shows these measurements for a single processor.

	Single Threaded x264
Throughput (frames/s)	1.24
Average frame latency (ms)	642
Image Quality (Average PSNR)	39.654
Encoded Bitrate (Kb/s)	11167.78

Table 1. Performance of single-threaded x264.

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 is optionally made up of slices and these slices are composed of macroblocks. Both the slice and the macroblock represent spatial data indices.
3. The image processing functions (implemented in x264 as `x264_macroblock_analyse` and `x264_macroblock_encode`) and the entropy encoding functions (implemented in x264 as `x264_cavlc_write` and the `x264_cabac` family of functions) represent the spatial instruction indices.

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

²The 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 the study represent original work.

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

- The dependence between the image processing and entropy encoding functions represents the temporal data index.

At this point, x264 is prepared to be partitioned.

4.3 H.264 with TDP

The first strategy studied is temporal data partitioning. As in the security camera, this strategy is implemented by assigning frames, or temporal data indices, to processes. Each process is responsible for executing the image processing and entropy encoding instructions on all pixels, or spatial data indices, of its assigned frames and the processes work on separate frames concurrently. (As noted above, this form of parallelism is supported in the x264 distribution.)

As described above, frames encoded later in the sequence use previously encoded frames for reference and may themselves be used as reference frames for subsequently encoded frames. Thus, a process is both a consumer and producer of reference frames. For the parameters used in this case study, a process may use up to five reference frames. To increase parallelism, processes make rows of reference frame pixels available as they are produced. This scheme comes at a cost of reducing the parallelized encoder’s ability to find temporal redundancy in some video streams.

Load-balancing the parallel encoder using this strategy is fairly easy. Processes are only assigned new frames after completing previously assigned work. This way, a process which is assigned a particularly difficult frame can work on it without becoming a bottleneck. Even if the slow process is producing reference frame data, the fact that the encoder is limited to five reference frames minimizes the impact of one slow frame in a system with sixteen processes.

Table 2 shows the performance of the serial implementation and the TDP encoder for 2, 4, 8, and 16 processes. As expected, the TDP strategy helps to improve throughput, which steadily increases with increasing numbers of processes. However, the latency of encoding an individual frame actually gets worse. There is a constant amount of overhead to pay for the communication and synchronization associated with communicating the reference frames. The encoder quality remains the same for a very modest increase in output bitrate.

Processes	Serial x264	TDP x264			
	1	2	4	8	16
Throughput (frames/s)	1.24	2.06	4.07	7.93	14.03
Average frame latency (ms)	642	910	899	893	916
Image Quality (Average PSNR)	39.654	39.654	39.654	39.654	39.654
Encoded Bitrate (Kb/s)	11167.78	11167.74	11175.38	11179.15	11210.06

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

The TDP strategy is excellent for improving throughput, and easy to load-balance; however, it actually increases the latency of encoding a single frame.

4.4 H.264 with SDP

In an attempt to decrease the latency of the encoder, spatial data partitioning is employed. Following this strategy, a frame is divided into multiple slices and each slice is assigned to a separate process. Each process executes the image processing and entropy encoding functions on its assigned slice, and all processes wait until the current frame is encoded before moving on to a new frame. Thus, processes encode separate portions of the same frame concurrently. (As mentioned above, previous distributions of x264 implemented this strategy; for the case study SDP is re-implemented in the current x264 code base.)

As described above, slices provide independence in both the image processing and entropy encoding functions. This independence is especially important during entropy encoding which requires all macroblocks to be processed in order. However, the independence comes at the cost of reduced ability to find temporal redundancy and a decrease in the compression of the entropy encoding process.

Although load-balancing SDP programs is generally easy, it is difficult in the case of the H.264 encoder. The time taken to decode a macroblock is data dependent making some macroblocks easier to encode than others. Generally,

regions of the frame with high motion take longer to encoder than low-motion regions. Thus some slices may take longer than others and become a bottleneck.

Table 3 shows the performance of the serial implementation and the SDP encoder for 2, 4, 8, and 16 processes. As expected, both the latency and throughput of the SDP parallelized encoder improve compared to the serial version. Unlike the TDP encoder, however, the performance of the SDP encoder appears to be reaching an asymptote as 16 processes provide little added benefit over 8 processes. This knee in the performance curve is due to the load-imbalance described above. Additionally, the quality of the encoder decreases while the encoded bit rate increases as more processes are used. This is because each additional process requires splitting the frame into additional slices and each slice reduces quality and adds overhead to the output bitstream.

Processes	Serial x264		SDP x264			
	1	2	4	8	16	
Throughput (frames/s)	1.24	2.5	4.31	6.77	7.04	
Average frame latency (ms)	642	347	179	96	89	
Image Quality (Average PSNR)	39.654	39.443	39.439	39.423	39.407	
Encoded Bitrate (Kb/s)	11167.78	11690.43	11746.12	11852.75	12407.42	

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

Part of the difficulty implementing the SDP encoder comes from the dependence requiring each macroblock within a slice to be written in raster order. Although the entropy encoding is a small fraction of the total compute time, this concern dominates the design of the SDP encoder.

4.5 H.264 with TIP

In order to separate the image processing and entropy encoding into two separate and individually simpler modules, temporal instruction partitioning is applied. Following this strategy, the image processing and entropy encoding functions are each assigned to separate processes. The image processing thread works on frame N while the entropy encoding thread works on frame $N - 1$. (As noted above, this and subsequent partitionings are original work.)

The image process finds and removes redundancy from the video stream. Having removed the redundancy, the remaining data is forwarded to the entropy encoding thread for additional compression. The entropy encoding process sends data on the output bitrate back to the image thread so that it can adjust its own computation to meet a target bitrate.

Load-balancing the TIP encoder is extremely difficult. The image processing functions are much more computationally intensive than those performing entropy encoding. In addition, a large amount of communication is required to forward the result of image processing to the entropy encoder.

Table 4 shows the performance of the serial implementation and the TIP encoder for 2 processes. As only two temporal instruction indices were identified for the H.264 encoder, that serves as a limit for the number of processes in a TIP implementation. While the throughput shows a very modest improvement, the latency of the TIP encoder is considerably worse than the serial case. This performance loss is due to the added communication of the parallelization. This strategy results in a small gain in image quality, although this is too small to be significant and likely just a lucky consequence of the test input. This slight increase in image quality also comes with a slight increase in the size of the encoded video.

Processes	Serial x264	TIP x264
	1	2
Throughput (frames/s)	1.24	1.35
Average frame latency (ms)	642	1435
Image Quality (Average PSNR)	39.654	39.659
Encoded Bitrate (Kb/s)	11167.78	11288.25

Table 4. Performance of parallel x264 decomposed with the TIP strategy.

The TIP strategy has several drawbacks when applied to x264: it is difficult to load-balance, it increased the

latency, and it decreased the quality. However, it does allow the instructions to be split into two modules. Now, instead of having to handle the dependences of image processing and entropy encoding simultaneously, they can be addressed separately.

4.6 H.264 with TIP and SDP

In order to take advantage of the separation of the image processing and entropy encoding functions provided by the TIP implementation, SDP parallelism is applied to the image process. However, in this case, the frame is not split into slices, but rather rows of macroblocks. Because image processing and entropy encoding are now independent, the SDP partitioning of the image process only has to respect the data dependence of Figure 6. Thus, process P can work on macroblock row i while $P + 1$ can work on $i + 1$ as long as the processes synchronize so that $P + 1$ does not read data from row i until it has been produced. Following this scheme, multiple processes work simultaneously on the image processing for frame N while a single process performs the entropy encoding for frame $N - 1$.

As described above, each image process must send its results to the entropy process. In addition to this communication, processes responsible for image processing must communicate and synchronize to observe the dependence illustrated in Figure 6.

Load-balancing in this implementation is somewhat difficult. Even when the image processing is parallelized amongst as many as 15 processes, the entropy process still requires less communication. In addition, as described above for the SDP partitioning, some macroblocks require more image processing work than others, so even balancing the image processes is difficult.

Table 5 shows the performance of the serial implementation and the parallel encoder implemented with a combination of TIP and SDP. Results are shown for 2, 4, 8, 12, and 16 processes with one process always allocated to entropy encoding and the others allocated for image processing. Use of the SDP strategy in combination with TIP increases throughput and decreases latency using up to twelve processes. After that point, increasing the number of processes reduces performance due to the overhead of communication and load imbalance. However, the use of the TIP strategy to separate the image processing and entropy encoding allows the SDP pattern to be applied without resorting to the use of slices. This is reflected in the results as, unlike in the purely SDP pattern, the quality of the encoded video does not change as more processes are added.

Processes	Serial x264	TIP+SDP x264				
	1	2	4	8	12	16
Throughput (frames/s)	1.24	1.35	3.30	5.94	7.06	5.21
Average frame latency (ms)	642	1435	553	284	229	330
Image Quality (Average PSNR)	39.654	39.659	39.659	39.659	39.659	39.659
Encoded Bitrate (Kb/s)	11167.78	11288.25	11288.25	11288.2	11288.25	11288.25

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

The hybrid approach using TIP and SDP represents a compromise between throughput, latency, and image quality. It does not achieve the high throughput of the pure TDP strategy, nor does it achieve the low latency of the pure SDP strategy. At the same time, the hybrid approach achieves improved throughput and latency without sacrificing the quality of the encoded image.

4.7 H.264 with TDP, TIP, and SDP

The final partitioning in the case study combines TDP, TIP, and SDP in a effort to produce a single code base that can be tuned to meet differing throughput and latency needs. In this implementation, the TDP strategy is applied to allow multiple frames to be encoded in parallel. Each frame is encoded by one or more processes. If one process is assigned to a frame, this implementation acts like the pure TDP approach. If two processes are assigned to a frame, each frame is parallelized using the TIP strategy. If three or more processes are assigned to a frame, each frame is parallelized using the combination of SDP and TIP discussed above. By applying these three strategies in one

application, a variable number of frames can be encoded in parallel using a variable number of processes to encode each frame.

The communication in this implementation is the union of the required communication of the composite strategies. Processes assigned a frame must communicate reference frame data. Within a frame, the processes assigned image functions communicate results to the process assigned entropy encoding and the image processes coordinate among themselves.

The load-balancing of this implementation depends on the number of frames encoded in parallel and the number of processes assigned to each frame. If only one process encodes a single frame, the load balance is good as in the pure TDP implementation. If only one frame is encoded at a time, the load balance is that of the hybrid TIP and SDP solution.

Table 6 shows the performance of the final parallelization in the case study. In this table all results are presented using sixteen processes, but the number of processes assigned to a single frame is varied. On the left side of the table, the performance is the same as the sixteen process implementation of the TIP+SDP strategy. On the right side of the table, performance is the same as the sixteen process implementation of the pure TDP strategy. The data in the middle of the table demonstrates that this approach allows a tradeoff between latency and throughput.

	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)	1.24	5.21	7.74	11.02	12.99	14.03
Average frame latency (ms)	642	330	443	652	1151	916
Image Quality (Average PSNR)	39.654	39.659	39.652	39.652	39.646	39.654
Encoded Bitrate (Kb/s)	11167.78	11288.25	11169.48	11205.90	11224.71	11210.06

Table 6. Performance of parallel x264 decomposed with a combination of the TDP, TIP, and SDP strategies.

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.

4.8 Summary of case study

This case study demonstrates the application of three of the four partitioning strategies to develop an H.264 encoder for HD video and it illustrates how the encoder’s throughput and latency are affected by the choice of partitioning strategy. In addition, the study demonstrates how different strategies influence load balance and the ability to deal with data dependences in the parallelized application. The case study shows that the pure TDP implementation achieves the best throughput, while the pure SDP implementation achieves the best latency although with a loss of quality. The TIP implementation demonstrates that some parallelizations may not help performance but can allow complicated programs to be split into simpler modules. A hybrid approach combining TIP and SDP represents a compromise between throughput, latency, and quality. Figure 7 compares the throughput and latency of these four implementations. Finally, a combination of TDP, TIP, and SDP allows maximum flexibility in a single code base.

5 Related Work

This section discusses related work in both parallel design patterns and parallel software implementations of H.264 encoding.

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 [11, 13, 14, 15, 18, 19]. 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

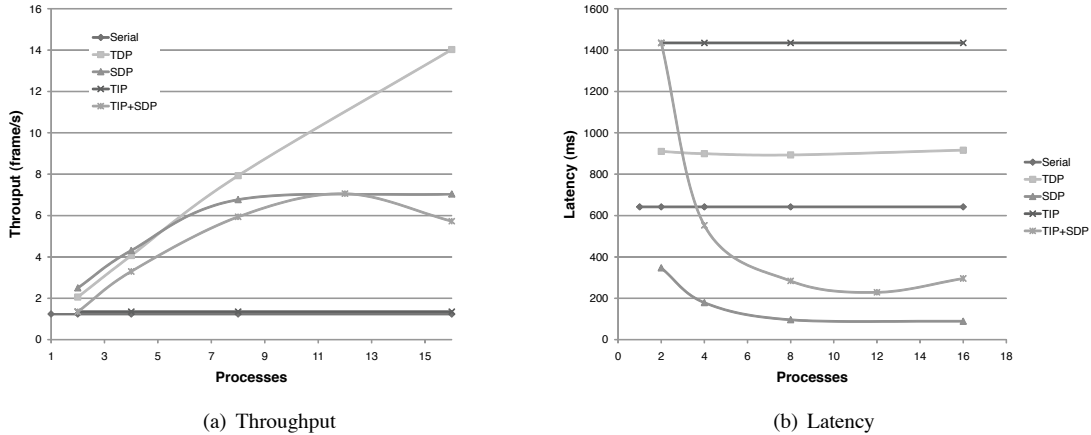


Figure 7. Performance of various parallelization strategies applied to H.264 encoder.

for specific applications. Parallel pattern languages [14, 15] guide users through the application of both high and low level patterns.

Two of the most commonly invoked high-level patterns are task and data parallelism. These terms are so common that they are often used by developers who are not explicitly thinking in terms of design patterns or pattern languages. The work presented in this paper extends the task and data parallel patterns by recognizing that the tasks and data in a program have spatial and temporal components. Furthermore, partitioning in space can have a different effect on program behavior than partitioning in time. These differences are particularly relevant to applications which have both a latency and throughput requirement, such as the large class of applications that interact with the outside world.

In addition to the performance differences associated with spatial and temporal partitionings, noting the difference is useful for its descriptive power. Describing an application as exploiting spatial data parallelism provides more information than describing the same application as simply data parallel. For example, while both the TDP and SDP implementations of the video encoder are data parallel, they have fundamentally different structures, and different performance. Providing additional information enhances understanding and collaboration.

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 [12]. 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.

Rodriquez 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 [17]. Several authors describe slice-based SDP partitioning strategies [3, 9]. Jung et al. adaptively choose the slice sizes in order to address the load-balancing issue discussed above [10]. Sun et al. describe a pure SDP implementation that does not rely on slicing, but still respects the data dependences described above [20].

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 motion estimation and analysis block of Figure 5 into one process and place all other functions 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 motion estimation and analysis process.

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). A case study has demonstrated how these strategies

can be applied to implement a multicore HD encoder. In addition this case study illustrates how multiple partitioning strategies can be combined to yield the benefits of each.

Parallel software development is currently experiencing a surge in interest due to the rise of multicore architectures. The PARSEC benchmark suite is designed to emphasize the workloads that will stress emerging multicore processes [4]. In fact, x264 is included as a PARSEC benchmark. Future case studies could explore the application of these strategies to other parsec benchmarks, perhaps finding other useful combinations of strategies, or even demonstrating that certain combinations occur frequently enough to be labeled design patterns themselves.

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 (CDROM)*, page 5, Washington, DC, USA, 1996. IEEE Computer Society.
- [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] 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*, pages 63–, April 2004.
- [4] B. Christian, K. Sanjeev, S. J. Pal, and L. Kai. The parsec benchmark suite: Characterization and architectural implications. In *PACT'08*, 2008.
- [5] M. J. Flynn. Very high-speed computing systems. pages 519–527, 2000.
- [6] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, 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*, San Jose, CA USA, Oct 2002.
- [7] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.
- [8] ITU-T. H.264: Advanced video coding for generic audiovisual services.
- [9] T. Jacobs, V. Chouliaras, and D. Mulvaney. Thread-parallel mpeg-4 and h.264 coders for system-on-chip multi-processor architectures. *Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. International Conference on*, pages 91–92, Jan. 2006.
- [10] 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. Edited by Rahardja, Susanto; Kim, JongWon; Luo, Jiebo. Proceedings of the SPIE, Volume 6777, pp. 67770J (2007).*, volume 6777 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, Sept. 2007.
- [11] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
- [12] 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*, pages 29–32. IEEE, 2006.
- [13] 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.
- [14] T. Mattson. Our pattern language (opl). http://parlab.eecs.berkeley.edu/wiki/_media/patterns/opl_pattern_language-feb-13.pdf.
- [15] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.
- [16] I. E. G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. John Wiley and Sons, 2003.
- [17] A. Rodriguez, A. Gonzalez, and M. P. Malumbres. Hierarchical parallelization of an h.264/avc video encoder. In *PAR-ELEC '06: Proceedings of the international symposium on Parallel Computing in Electrical Engineering*, pages 363–368, Washington, DC, USA, 2006. IEEE Computer Society.
- [18] S. Siu, M. D. Simone, D. Goswami, and A. Singh. Design patterns for parallel programming, 1996.
- [19] M. Snir. Resources on parallel patterns.
- [20] 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*, pages 577–585, Berlin, Heidelberg, 2007. Springer-Verlag.
- [21] R. van de Geijn. *Using PLAPACK – Parallel Linear Algebra Package*. MIT Press, Cambridge, MA, 1997.
- [22] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.
- [23] x264. <http://www.videolan.org/x264.html>.