Partitioning Strategies for Concurrent Programming

Henry Hoffmann, Anant Agarwal, and Srini Devadas
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
{hank,srini,agarwal}@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 decomposition 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, temporal data partitioning, spatial instruction partitioning, and temporal instruction partitioning, and cataloges the benefits and drawbacks of each. In addition, the practical use of these strategies is demonstrated through a case study in which they are applied to implement several different parallelizations of a multicore H.264 encoder for HD video. This case study illustrates both the application of the patterns and their effects on the performance of the encoder.

1 Introduction

Design patterns for parallel computing help to add structure and discipline to the process of concurrent software development [7, 8, 9, 5]. 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. Such applications 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 [11, 4] 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.

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 partitionings 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. [6]). 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.

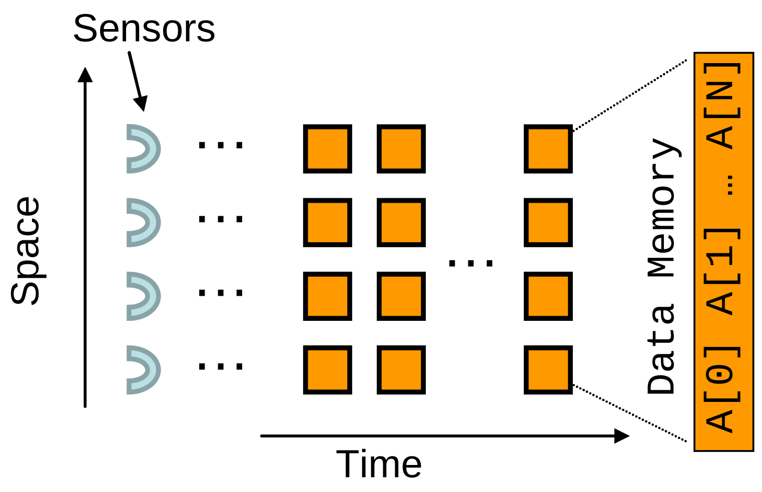


Figure 1. Spatiotemporal indexing of data in the security camera example. Spatially distributed sensors produce a sequence of pixels over time. The pixel's location in the frame is the spatial index while the frame number in the sequence in the temporal index.

The camera executes 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.

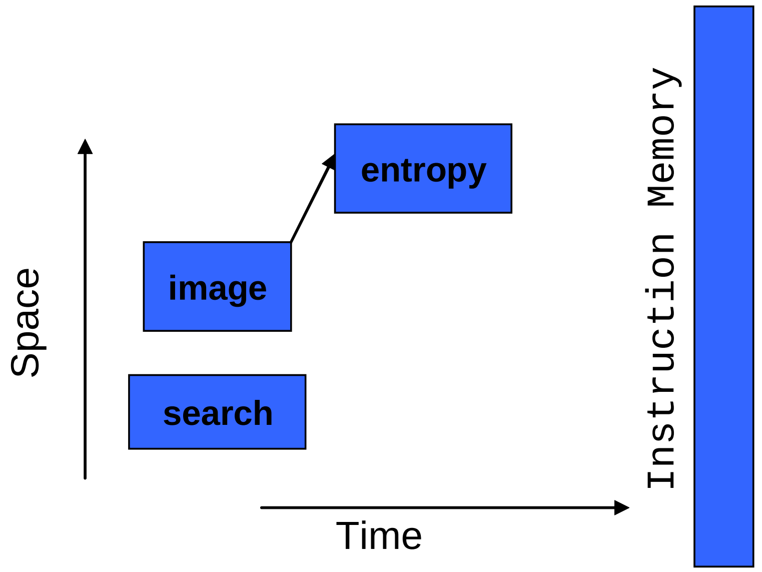


Figure 2. Spatiotemporal indexing of instructions in the security camera. Each frame produced by the camera is searched and encoded. The encoding process is further broken down into image processing and entropy encoding functions. The topological sort of the dependence graph provides a temporal index,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

Programs operate on temporal sequences of spatially distinct data. If each input in the sequence is numbered, then the sequence number serves as a temporal index into the program's data. Each input may consist of multiple distinct items, which are stored in separate variables or array indices. These storage locations serve as spatial indices into the program's data.

The security camera illustrates one example of data indexing as shown in Figure 1. The sequence of frames represent temporal inputs while the location of pixels in the frame represent a spatial index. Another example is a search engine, which receives a sequence of queries consisting of multiple search terms. In this case, the sequence of queries represents a temporal index while the distinct terms in the query represent the spatial index.

A program operates on a data input by executing sequences of distinct instructions which are bundled into functions. The functions, which are stored in distinct memory locations, represent a spatial index into instruction execution. The dependence between functions can be represented in a dependence graph. Performing a topological sort on the dependence graph produces a partial ordering which can be used as a temporal index into a program's instructions.

The security camera illustrates one example of instruction indexing as shown in Figure 2. The three functions, `search`, `image`, and `entropy` represent spatial instruction indices of the camera application. The ordering of function execution dictates that the `image` function must execute before the `entropy` function, while there are no ordering constraints on `search`. This partial order represents the temporal instruction indices of the camera example.

To summarize, partitioning strategies are distinguished by the decomposition of data or instructions and whether that decomposition is performed in time or space. For data, the temporal dimension is defined by the sequence of inputs, while the spatial dimension is defined by the distinct components of a single input. For instructions, the temporal dimension is defined by the sequence of functions used to process an input, while the spatial dimension is defined by the functions themselves.

2.3 Preparing to partition a program

The following procedure is used to prepare a program 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 our 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 of the strategy, or scenarios where it is most useful is discussed.

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 `image`, `encode`, and `search` 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 [10].

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:

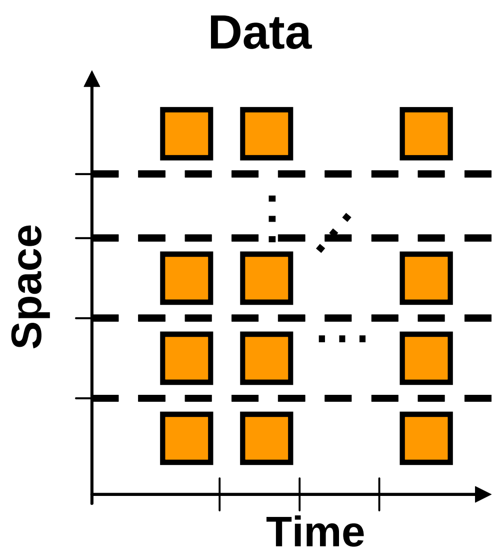
- A single processor will not meet the application's latency demands.
- The spatial data dimension is large and has few dependences.
- The application performs similar amounts of work on each of the spatial indices, which makes it easy to load balance.

3.2 Temporal data partitioning

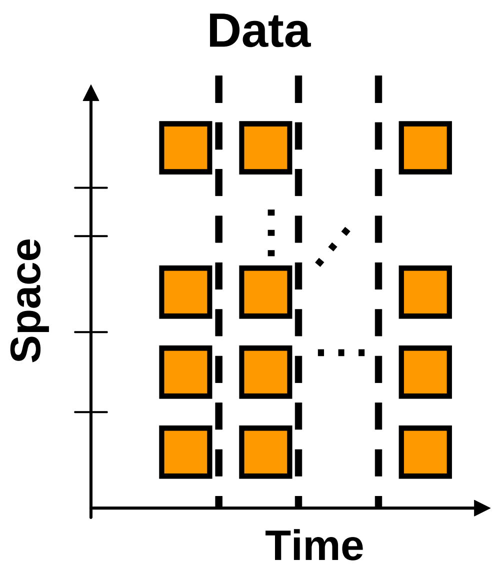
Description. Using the temporal data partitioning (TDP) strategy, data are divided among processes according to temporal index. Following this pattern, each process performs computation on all spatial indices associated with its assigned temporal index as illustrated in Figure 3(b). In a typical TDP 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.

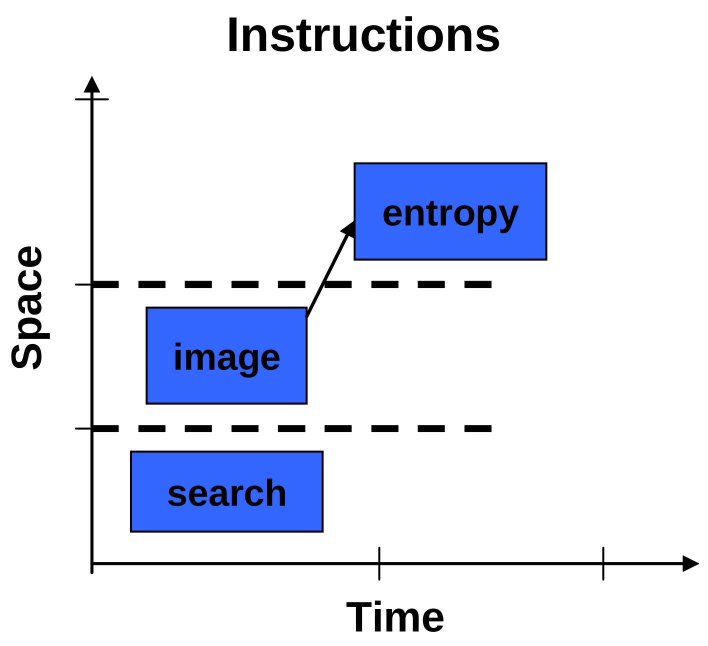
Other common examples. Packet processing is commonly parallelized using the TDP pattern. This application processes a sequence of packets from a network, possibly searching them for viruses. As packets arrive they are assigned to a process which performs the required computation on that packet.



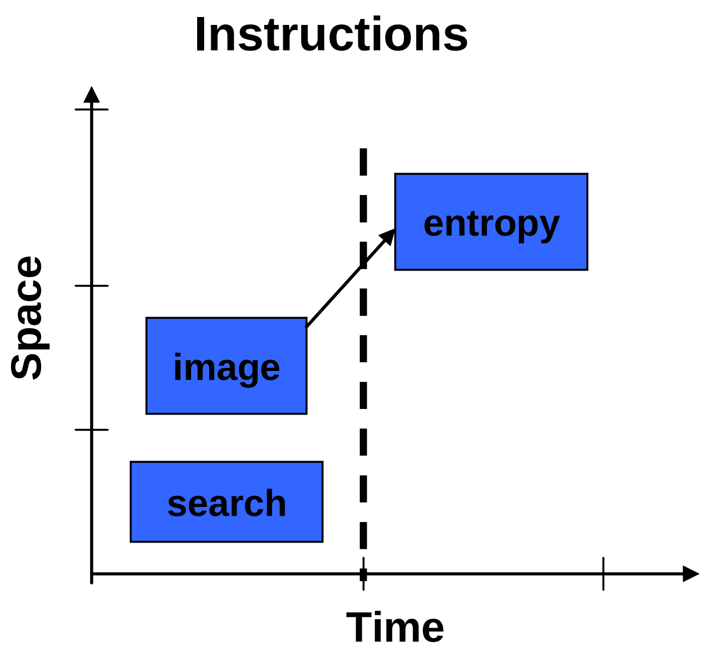
(a) Spatial Data Partitioning



(b) Temporal Data Partitioning



(c) Spatial Instruction Partitioning



(d) Temporal Instruction Partitioning

Figure 3. A Taxonomy of parallelization strategies for embedded real-time systems.

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 use the master to manage the load of the individual workers or a work-queue pattern can allow the individual processes to load-balance 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.

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 [3] 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 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 amortize the cost of the required communication between stages.
- It is useful to split a large and complicated algorithm into several smaller modules.

3.5 Combining 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.

The security camera example demonstrates the benefits of combining multiple strategies. As illustrated above it is possible to use TIP partitioning to split the application into two processes. One process is responsible for the `image` function while another is responsible for the `entropy` function. (For the moment, ignore the `search` function). This TIP partitioning is useful because the `image` and `entropy` functions have very different dependences and splitting them creates two simpler modules, each of which is easier to understand and parallelize. In addition, this TIP partitioning improves throughput, but it suffers from a load-imbalance, because the `image` function requires much more processing power than the `entropy` function. In addition, this partitioning has not improved the camera's latnecy.

To address the load-imblance and latency issues, one may apply a further partitioning strategy. For example, applying the SDP strategy to the process executing `image` will split that computationally intensive function into smaller parts. This additional partitioning helps to improve load-balance and application latency.

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

4 Case study

4.1 H.264 overview

4.2 H.264 with TDP

4.3 H.264 with SDP

4.4 H.264 with TIP

4.5 H.264 with TIP and SDP

4.6 H.264 with TDP, TIP, and SDP

4.7 Summary of case study

5 Related Work

6 Conclusion

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] 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.

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

[5] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[6] 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.

[7] 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.

[8] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[9] S. Siu, M. D. Simone, D. Goswami, and A. Singh. Design patterns for parallel programming, 1996.

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

[11] 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.