Parallel Design Patterns

Michael Herwig

German Research School for Simulation Sciences GmbH
Laboratory for Parallel Programming
michael.herwig@rwth-aachen.de

Supervisor: Zia Ul Huda (z.ul-huda@grs-sim.de)

Abstract: While multicore architectures get more distributed in modern computers, taking advantage of them gets more and more important. This paper introduces design patterns integrating structured concurrency into modern software. Therefore environment and suitable patterns get categorized. Further several patterns get presented with focus on their underlying domain. Following these patterns get compared against each other and in combination for different cases of application.

1 Introduction

In recent years multicore architectures took over the cpu market. Nearly every modern cpu consists out of multiple *units of execution*. This is a consequence of the heat problem from increasing frequency[4, p. 23].

To take advantage of this new environment programs need to be overworked and code design changes. This opens a wide variety of new design patterns challenging the balance between flexibility, efficiency and simplicity.

- **flexibility**

  System environments change in time. A flexible solution is portable and reusable. This requires a high modularity in code design. This allows advancing with upcoming trends and testing multiple environments with less effort.

- **efficency**

  While the count of *units of execution* on modern CPUs increases it is unforeseeable to consider a limitation. Despite todays CPUs normally exists of four to sixteen *units of execution* it is presumable that this number increases expeditious in the next time. A good design pattern covers this progression and scales with the underlying hardware otherwise it cannot be denoted as efficient since it does not exploit all resources.

- **simplicity**

  Often code optimization and parallelization results in more complex relationships and code design. Nevertheless it needs to be simple enough to be maintained and debugged.

Unfortunately these three mutual exclude each other. Flexibility comes with the price of efficiency and vise versa. The key for a good modern parallel design is to find the best suitable balance for the environment. The following sections will address some techniques to overcome this challenge.

2 Characteristics

Finding a suitable design pattern is a difficult task and more an iterative process then a clear decision. Every approach got its own strength and weaknesses and choosing one results always in trade offs. There is no holy grail of patterns that fits best and often it is not even possible to estimate the real effort. Design decisions get evaluated after implementation and may be altered. This makes comparing different patterns an elaborate task.

This chapter gives a brief explanation and overview of some characteristics used to compare and categorize the design patterns presented later. The key is to distinguish between hardware and software dependencies. Some patterns may fit better on special memory layouts while others requires some kind of data sharing. Additionally patterns run faster with different implementations on varying platforms. This opens a zoo of possible comparisons and categorizations. Since this paper gives a shorthand and more abstract overview everything cannot be covered. Therefore we will split all patterns into two subcategories and treat them as far as possible separately.

2.1 Algorithm Structure Design Patterns

This category includes all patterns that takes a more administrative and abstract part in the implementation and are mostly platform independent. Their purpose is to organize concurrency independent from the underlying realization. They are distinguished into classes by the way they organize the commanding part of concurrency found in earlier development stages. This results in three different classes that are shortly explained in the following itemization.

- **organized by task**

  The domain is dominated by different concurrent tasks. This does not mean data dependencies are excluded furthermore to overlook as a result of their magnitude. Tasks are braked down in a collection of independent subtasks and executed in parallel.

- **organized by data decomposition**

  The same task is repeated plenty of times on different sets of data that may be subsets of huge data blocks. Patterns in this class parallelize execution by organizing the availability of data to different units of execution performing the same task.

- **organized by flow of data**

  Data is commonly shared among tasks and different tasks depend on their ordering of execution. Tasks get scheduled statically or dynamically to different units of execution, based on their internal dependencies and workload.

With this classification it is auxiliary to create a decision tree for choosing the right pattern like in figure 1. It is uncommon to find one best suitable pattern but it gives a great overview and introductory overview of possibilities. In practice multiple patterns from different spaces will be considered and may be compound to more complex and nested solutions.

While algorithm structure patterns aims to encapsulate organization from software structure it is worth ignoring specific platform features and implementation characteristics. Flexibility always comes with the costs of efficiency and vise versa. Choosing the right algorithm means choosing an appropriate balance between level

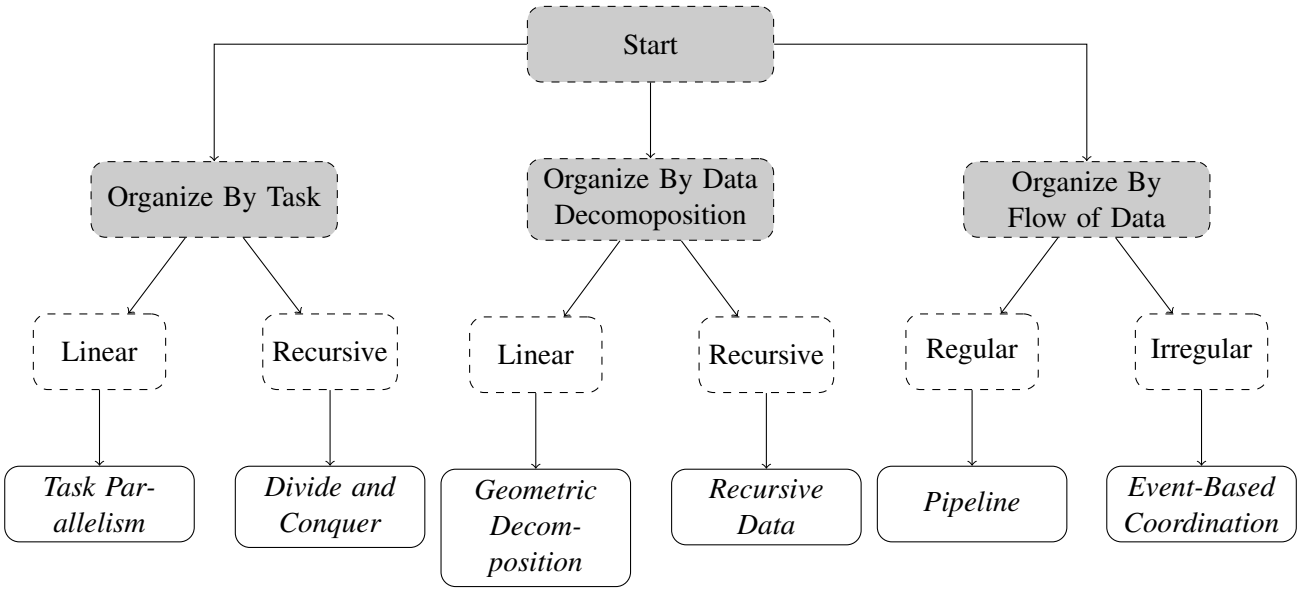


Figure 1: decision tree for algorithm structure design patterns, taken from [1, figure 4.2]

of abstraction and exploiting the underlying architecture. This leads to a mutual dependence on code structure and implementation details.

2.2 Supporting Structure Design Patterns

Patterns of this category helps parallelization by structuring and encapsulating source code to be compatible to the chosen algorithm patterns without leaking implementation detail.

Similar to algorithm structure design patterns the supporting structure design patterns are partitioned into two separate classes.

- **program structures**

  This classifies patterns to organize code structure and abstracts their execution to embed them into *algorithm structure design patterns*.

- **data structures**

  Managing shared data requires fine granulated and suitable data structures to maintain data dependencies and consistency. A pattern in this partition addresses the problem of sharing data structures.

In theory this partitioning of algorithm and supporting design patterns is a clean solution for seperating organization from implementation but in practice they get nest together. Some patterns from these spaces work better together then other while other patterns are better supported in different environments. Even various programming language can have different support and performance for suitable patterns.

3 Divide-And-Conquer Pattern

The divide and conquer pattern is a *algorithm structure design pattern* and is classified as *organized by task*. Therefore it splits problems into smaller sub problems that can be solved in parallel.

3.1 Amplification

Problems can often be divided in several sub problems that can be solved independently. Their solutions are afterwards merged to a solution for the primal problem. Thereby the primal problem depends on the solutions of its subtasks and this limits the number of concurrent tasks as shown in following figure.

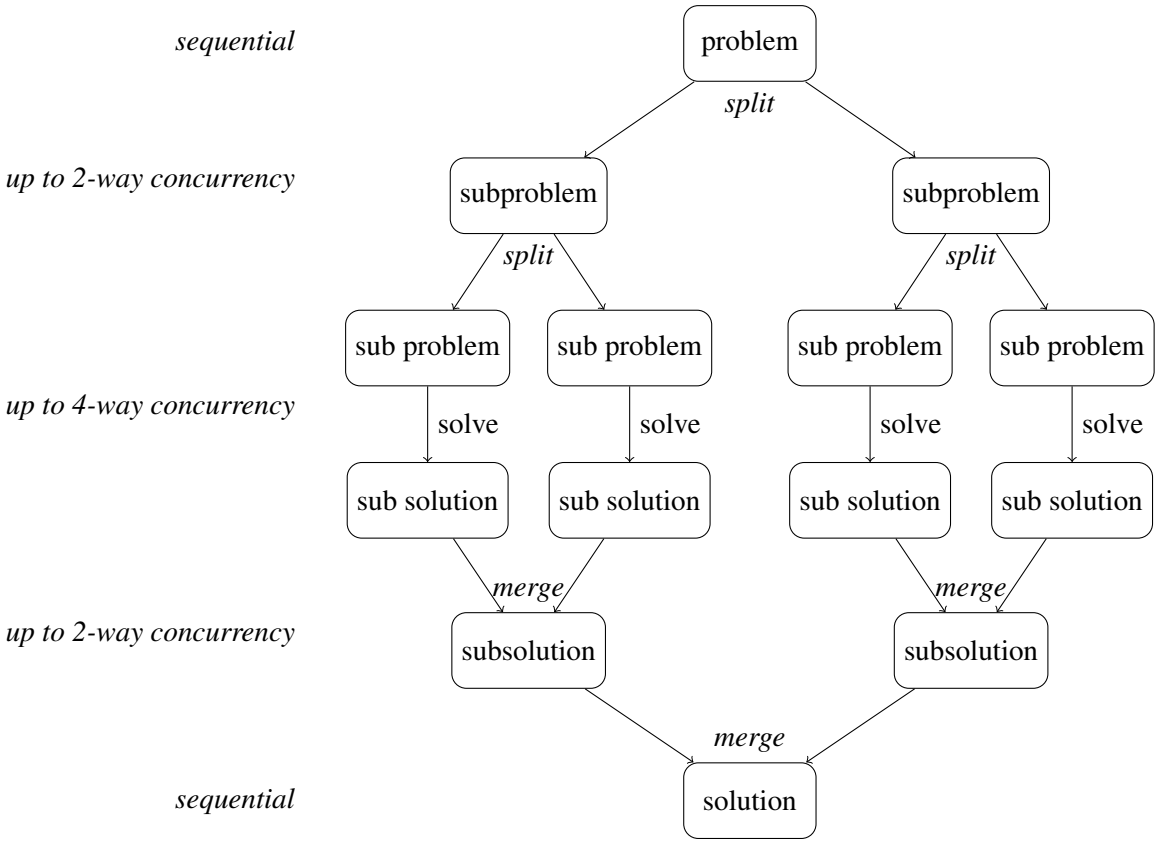


Figure 2: exemplary divide and conquer workflow[1, figure 4.5]

While the depth of divided problems determine the number of concurrent tasks it is important to find a suitable depth. Choosing an oversized depth results in undersized tasks and an overhead of merge and split operations, especially if sub problems are generated dynamically. Therefore usually some kind of limitation or threshold is implemented to conclude whether a split operation is worth it.

Equally important is the mapping from emerging tasks to *processing elements*. A simple approach would be to create a *unit of execution* for every available *processing element* and then mapping every deep most subtask to a busy *unit of execution* until everything is solved. Nevertheless this can cause context switches. If a higher order task executes on a different *processing element* then its subtask at least the data the subtask depends on has to be transmitted, because usually the data is already available for the parent task. To overcome this problem sets of high order tasks and their subtasks are mapped to *units of execution*. Tasks out of this set are then preferred for the respective *unit of execution*. This guarantees the availability of data and tasks are only moved to other *processing element*if they are busy.

3.2 Fork-Join Pattern

One related pattern to the *divide and conquer pattern* is the fork-join pattern. Due to its strong relationship this section gives a brief overview of this pattern and its impact.

The fork-join pattern is a *supporting structure design pattern*. It organizes code by grouping them into three stages shown in figure 3. The fork stages dynamically generate concurrent tasks. The execution stages recursively results in a new fork or process a data subset. The join stage indicates the termination of the subroutine. Unlike the *divide and conquer pattern* this pattern addresses implementation and fork and join has no performance leakage. The merge and split operation can therefore be considered as separate tasks. The *divide and conquer pattern* is usually implemented via the fork-join pattern and mapping rules from tasks to *processing elements* applies from the divide and conquer pattern.

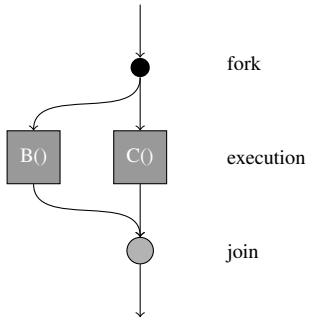


Figure 3: sequential grouping of the fork-join pattern[2, figure 8.1]

4 Map Pattern

The map pattern is a *algorithm structure design pattern* designed to work efficiently with sets of elemental functions, further called instructions, that operates on distinct data hence it is classified to be *organized by data*.

4.1 Amplification

The map pattern itself does not specify the way instructions are executed. Therefore they can run in parallel or in sequence as shown in figure 4.

Instructions are not permitted to communicate with each other or share data that is not read only. In return data can be vectorized which enables *memory alignment*, smaller cache lines, faster caching[4, section 1.3.1] and radical reduces the occurrence of *false sharing*. Additionally this simplifies the pattern and makes it enormously flexible.

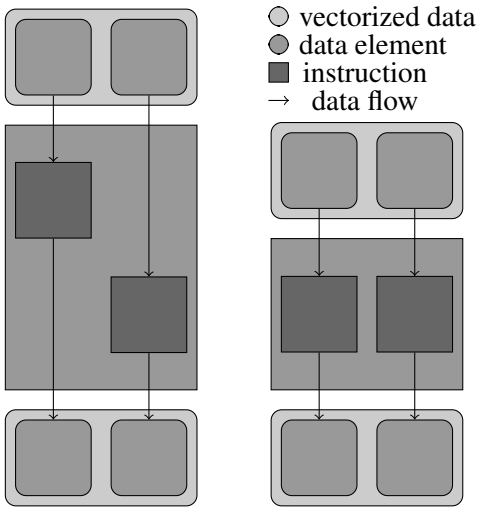


Figure 4: Serial and parallel execution of map pattern. taken from[2, fig. 4.1]

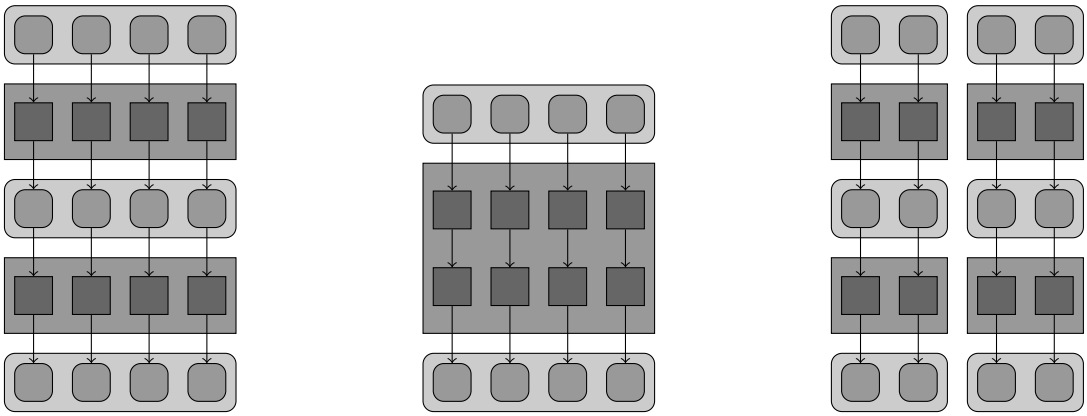
Various combinations with other patterns exists to manage instruction execution and data flow. To support full nesting of patterns without dropping the performance boost from vectorization it is very important to guarantee a serial control flow and data access[2, p. 122]. Otherwise, depending on underlying patterns, execution will raise in *race conditions*, *cache misses* and other unpredictable issues.

While it is permitted to access shared data that is not packed into the vectorized data for read only purpose it is not unusual to store indices to oblique refer to data with random access[2, p.124]. The pattern itself does not forbid varying instructions and choosing instructions may rely on data read at execution.

Notice that this will give some flexibility in execution but at the price of performance leakage to instruction and data *cache misses*. In consequence it is more common to keep the instruction set small and reference data not indirect. In fact sometimes it is more performant to store redundant data that does not alter repetitive into the vectorized data set to improve performance.

4.2 Sequence of Maps

Sometimes multiple instructions have to be executed in sequence because every next instruction attaches to the processed data from the previous instruction. If you optimize every instruction step of your algorithm with the map pattern this leads to a sequence of maps. While every map does not know each other you get synchronizations between every instruction step. Otherwise the next map could not be sure about the availability of its data. This result in less cpu workload because some instructions could already be executed while waiting for the most slow execution of the previous instruction step, without the need of its processing data. To overcome this problem you need to either reduce synchronization points or encapsulate data blocks that need to be synchronized.



(a) *no optimization* **(b)** *code fusion* **(c)** *cache fusion*

Figure 5: comparison of optimization models for sequences of maps. Legend applies from figure 4[2, figure 4.2 & 4.3].

code fusion

Instruction chains get recomposed calling each other in sequence with the processed data. This does not mean that the instruction is executed in the same *unit of execution* but it is common to do so because of the availability of data what disables latency and spares bandwidth. Nevertheless fusing to much instructions results in larger code size and may be to large to hold in cache what in turn slows down computation[4, p.81].

cache fusion

It is not always possible to fuse code or following instruction steps depends on multiple preprocessed data blocks. In this case it is not possible to fuse code, but to encapsulate data blocks that need to be synchronized for the following instruction. This does not overcome the problem entirely but reduces the count of instructions that waits for completion of other tasks.

4.3 Organize Data Flow

In the previous section data flow was straight forward. This is not always the case. Some instructions may produce multiple data sets that can further be processed in parallel or their data flow gets invalidated and discarded. Therefore the data structure needs to be reorganized.

pack

Some data flows do not need to proceed any more and get discarded. This would result to gaps in the vectorized data set. The pack pattern moves further used data blocks to align them in memory.

unpack

The unpack pattern is the converse part of the pack pattern. It moves data blocks to fill in gaps into the vectorized data. These gaps can then be used to bind data from external sources or unrelated tasks for example.

5 Pipeline Pattern

The *map pattern* assumes that all data is available and the task can run in parallel on different sets of data because they not influence each other. Unfortunately this does not fit every domain. Data may get streamed or needs to be processed from pervious stages and it is not worth waiting for it. The *map pattern* tries to overcome this problem with *cache fusion* but i.e. online algorithms requires to change computational behavior from previous processed data by definition. With the *map pattern* this would lead to serial execution. The pipeline pattern solves this problem by encapsulating parallel stages in execution and organizing a regular flow of data.

First tasks get divided into a sequence of stages, depending on their influence to concurrent tasks of the same type[2, p. 254]. A suitable segmentation is itemized below.

- **parallel stage** is a subtask that can run in parallel without the requirement to synchronize with other instances caused by shared data.
- **serial out of order stage** is a subtask that influences the execution of parallel instances without requirements to ordering.
- **serial in order stage** is like the serial out of order stage but with ordering constraints.

Dividing a task into these stages leads to an execution schema like in figure 6. If the first serial stage finished the state is updated and another instance can start working on it although the primary stage is not yet finished but in parallel stage. This leads to a linear data flow from serial stages through every concurrent execution of the task. The stages itself are not permitted to be serial. Further themselves can be implemented by other design patterns to run in parallel.

While it is possible to keep synchronization points as small as possible with such segmentation it also improves maintainability by separating the task into independent chunks. Nevertheless this pattern gives a great flexibility at the price of ignoring hardware environment and increasing complexity and it is worth considering dataflow optimization and error handling[1, p.107-109] due to data locality and *livelocks*.

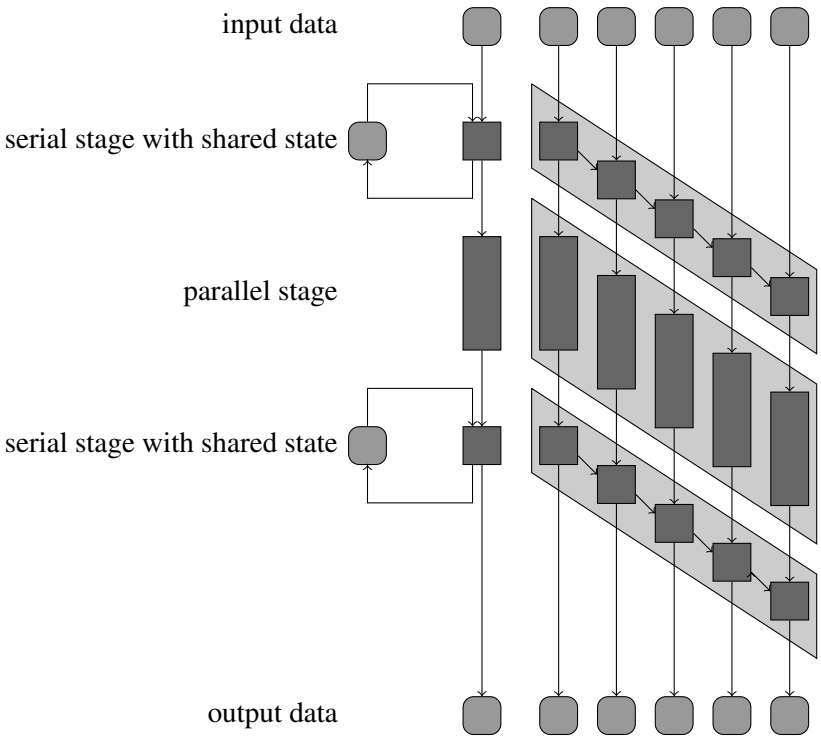


Figure 6: Illustration of a pipeline with two serial stages, taken from [2, figure 9.1 & 9.2]. On the left hand side is the pipeline in a single execution environment showing the data flow dependencies. On the right hand side the concurrent execution model of the task with five different data items.

6 Event-Based Coordination Pattern

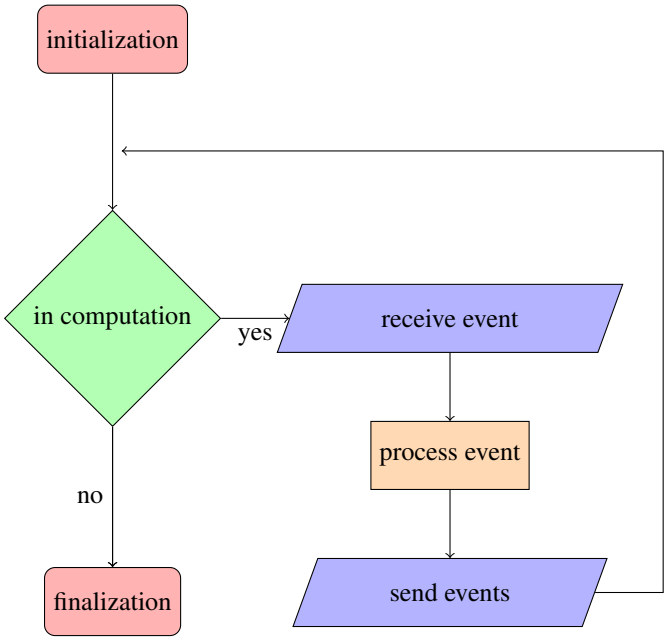


Figure 7: flow chart of task execution in the event base coordination pattern

Sometimes it is not possible to design program execution without irregular data flow. Hence it is a *algorithm structure design pattern* classified as *organized by flow of data*. Subtasks may invoke higher order tasks, depending on their data. This creates circular constraints that cannot be solved by static ordered execution schedules. In fact this would lead to a *deadlock* whenever a subtask invokes a higher order task whose execution again results in execution of the subtask. The event based coordination addresses this problem and introduces a dynamic scheduling mechanism.

In the event based coordination pattern tasks do not get invoked directly but through events. An event is a pattern specific structure that holds two tasks. The creator and the receiver what keeps the data flow transparent and defines ordering

constraints. Additionally events mostly hold some environment specific data to customize execution. Calling a subtask then gets overloaded to creating an event that abstracts the call. Created events gets passed to an interface, that manages passing them to the receiving task, if the current task can not proceed any further. This leads to a uniform structure of execution for every task as shown in figure 7. While tasks execute concurrently events get send and received concurrent. This requires a safe data structure to manage the events asynchronously like a shared queue, see section 7 for more information about concurrent data structures.

event ordering

The strait of this pattern is the coordination of the events. While constraints are stored into the event structure it is sometimes not possible to choose the right ordering and wrong ordering sometimes results in wrong outcome. This happens if some different tasks execute a common subtask that implies changes to data both tasks rely on. If both subtasks get executed before returning to one of the parent tasks, this parent task may accidentally assumes a wrong state. This means that both events were ordered before the parent task. The corresponding event is then called as out-of-order event.

Whether out-of-order is a problem for the execution model depends on the environment. Nevertheless if out of order execution is problematic the chosen ordering strategy apportions in 2 categories optimistic and pessimistic.

Optimistic ordering is an approach that orders everything normally until a critical out-of-order execution happens. At this point the corresponding changes of the out-of-order event execution get led back. This rollback implies that every event execution generated by the out-of-order event and so forth need to be led back. Nevertheless this requires a mechanism to rollback the execution of events and this is not always provided especially if execution invokes external routines or is invoked by an external routine.

Pessimistic ordering is used if rollbacks are not provided or not feasible due to their performance leakage. In this case events only get ordered if it is insured to be linear and no out-of-order event can occur. This leads to high latency due to the waiting time to ensure the ordering and requires more communication what again costs throughput.

7 Shared Data Pattern

7.1 Motivation

Most of the presented patterns assume that data is processed independently of other tasks. This makes it easy to replicate or separate these data to not interfere with other tasks working on these like the *map pattern with cache fusion*. The shared data pattern is applicable if data need to be consistent over several tasks. Imagine a queue of working items where every task wants to grab one of it if it is finished. In this case you need to ensure that two tasks do not get the same working item and updating the queue from multiple *units of execution* do not break the internal structure or make it inconsistent.

In general if one data item is processed by more then one concurrent task and one of these task modifies this data in a way that another concurrent computation needs to be updated the shared data pattern is expedient.

7.2 Solutions

While the shared data pattern is more a hyponym for a collection of techniques this section shows some of them. These should not be read as a fixed set of solutions, more like a set of patches that fits best in related variations.

concurrent data structures

Developing concurrent data structures is a huge topic itself. The key is to keep your data structures as simple as possible without unacceptable performance leakage. Simplicity not only keeps your developing time low but also ease debugging and understanding the process. Therefore starting with an abstract interface and designing a simple implementation is a good point for custom domain suited optimizations and can be used later as consistency oracle[1] for testing[3, p.305]. This ensures correctness of more complex concurrency-control protocols implementations.

one at a time execution

This approach uses mutual exclusions to ensure that only one *unit of execution* can execute an operation, out of a set, at a time. These operations normally share a critical resource like a buffer or more complex data structure and can be considered as critical sections. This will always force every execution of these operations to be serial. Typically mutex, semaphores or critical sections, if provided, implement this. One at a time execution provides an easy and feasible solution to implement and test *concurrent data structures* on consistency with serial results. Therefore the abstract data interface is implemented as a wrapper declaring all operations as critical sections and forward them to a serial standard implementation. Take to account that this is intended to be slow and interlaced virtual calls leads to nested critical sections hence probably nested locks that should be avoided for deadlock safety reasons[3, p.49].

read/write-lock

As mentioned in section *motivation* this problem of shared data only occurs if a task needs to write to a shared set of data that leads to computational update for at least one concurrent task. In a domain where several tasks exists that only read from such data and depends on its consistency during their execution it is usually more performant to lock write access. This enables to run these tasks in parallel since they all only read the shared data hence they do not interfere each other and prevents *cache ping pong*[3, p.235-238]. This gets implemented via read/write-locks. Think of them as a special kind of mutex that be can locked for read or write access. While write access it is granted that no one else can obtain access to read or write. Yet read access can be obtained multiple times and stores an internal counter to secure that write access is only granted if no read access is currently omitted. Certainly this can lead to a continuous read and *deadlock* tasks that acquire write access, as a consequence more complex implementations are usually required.

[1] *consistency oracle*: generates results that are assumed to be right and compares them with the results of an implementation that needs to be tested[5]

8 Conclusion

Multiple *algorithm structure design patterns* and *supporting structure design patterns* were introduced to show their application and usage in modern multicore software development. Since this paper just give a brief overview and can not cover everything in detail it focused more on an abstract point of view.

Therefore some basic related patterns were explained and how they are used in combination to ensure *flexibility*, *efficiency* and *simplicity*. This resulted in multiple dependencies between software design and hardware architecture. For that reason performance challenging dependencies were shown and how to consider these in code structure and design evaluation for specific system environments.

Nevertheless parallelization comes with effort in development and sometimes it is not worthwhile. Unfortunately effort and value cannot always be estimated in advance. This leads to an iterative process of parallelization that adjusts software for new challenges and hardware dependencies. More then once this shows the importance of *flexibility* and *simplicity* despite its cost of *efficiency*.

While parallelization increased dramatically in the last years this field becomes more and more important in modern software design. This especially applies to distributed environments, like cloud services and high performance computing cluster. In industry more and more companies challenges via performance in industrial and multimedia applications. Additionally web development grows more and more and programming in parallel results in programming asynchronously. There a plenty more application areas but this reaches to show the importance of multicore environments, parallel patterns and it is worth knowing it.

Glossary

cache miss continues aligned memory gets prebuffered on accessed memory addresses. A cache miss occurs if a requested memory block is not prebuffered. This results in a loading routine. 5, 6

cache ping-pong multiple threads are executing concurrently on different processors and reading the same data [...] if one thread modifies the data, this change then has to propagate to the cache of other cores.[3, p.235]. 10, 12

deadlock each thread is waiting for the other. Neither can proceed because it is waiting for the other to release its mutex.[3, p47]. 8, 10, 12

false sharing small data items get aligned in cache lines and shared to multiple threads. Changing these items results in *cache ping pong*[3, p.237]. 5

livelock similar to *deadlock* in that one thread is waiting for another, which is in turn waiting for the first. The key differnce here is that the wait is not a blocking wait but an active checking loop.[3, p.301]. 7

memory alignment processor architecture always access a chunk of data, usually 64bit. vectorized memory gets aligned to prevent spacing between data with a word size that does not fit the chunk size.. 5

processing element a generic term for hardware element that executes a stream of instructions[1, p. 17]. 4, 5

race condition the outcome depends on the relative ordering of execution of operations on two or more *units of execution*[3, p. 36]. 5

unit of execution a thread is the fundamental unit of execution in modern operating systems and is associated to a process.[1, p.16]. 1, 4, 6, 9, 10, 12

References

[1] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill: *Patterns for Parallel Programming*. Addison-Wesley, Boston, sixth printing, June 2010.

[2] Michael McCool, Arch D. Robison, James Reinders: *Structured Parallel Programming - Patterns for Efficient Computation*. Morgan Kaufmann, Walthamm, digital print 2012.

[3] Anthony Williams: *C++ Concurrency In Action*. Manning, Shelter Island, first printing, 2012.

[4] Georg Hager, Gerhard Wellein: *Introduction to High Performance Computing for Scientists and Engineers* . CRC Press, Boca Raton, first printing, June 2010.

[5] wikipedia: *Oracle (software testing)*. `http://en.wikipedia.org/wiki/Oracle_(software_testing)`