A Comparison of Parallel Design Patterns for Game Development

En Jämförelse av Parallella Designmönster för Spelutveckling

Robin Andblom
Carl Sjöberg

Thesis: Bachelor 180 hp
Main field: Computer Science
Program: Game Development
Date of final seminar: 2018-01-15

Supervisor: Carl Johan Gribe
Examiner: Carl-Magnus Olsson
A Comparison of Parallel Design Patterns for Game Development

Robin Andblom, Carl Sjöberg
Malmö, Sweden

Abstract
As processor performance capabilities can only be increased through the use of a multicore architecture, software needs to be developed to utilize the parallelism offered by the additional cores. Especially game developers need to seize this opportunity to save cycles and decrease the general rendering time. One of the existing advances towards this potential has been the creation of multithreaded game engines that take advantage of the additional processing units. In such engines, different branches of the game loop are parallelized. However, the specifics of the parallel design patterns used are not outlined. Neither are any ideas of how to combine these patterns proposed. These missing factors are addressed in this article, to provide a guideline for when to use which one of two parallel design patterns; fork-join and pipeline parallelism. Through a collection of data and a comparison using the metrics speedup and efficiency, conclusions were derived that shed light on the ways in which a typical part of a game loop most efficiently can be organized for parallel execution through the use of different parallel design patterns. The pipeline and fork-join patterns were applied respectively in a variety of test cases for two branches of a game loop: a BOIDS system and an animation system.

Keywords: Flocking, BOIDS, Skeletal animation, Parallelism, Multithreading, Rendering time, Speedup
1. Introduction

Although processor speed has increased exponentially the last decades, it has in recent times come to a halt. It is no longer possible to increase the processor clock rates because of the limited hardware heat tolerance; a phenomenon referred to as a power wall [1]. This is where multithreading becomes important. To develop processor performance even further, more processing units have been added to the computing component to make it possible for processing to be made in parallel, which is especially important with regards to performance-heavy software like video games, which require a vast number of complex calculations to be completed at a blazing pace [2].

It is still a challenge to implement parallelism efficiently nonetheless, because of multithreading scheduling problems such as overheads, deadlocks and load imbalance [1, 3]. This is especially the case for the CPU, since a bottleneck effect easily arises due to its lesser capability of operating in parallel, compared to the GPU which consists of many thousands of cores. It has been shown that parallelizing games brings about a decrease in rendering time, thus eradicating or, at least, minimizing the bottleneck effect [2]. The question of how to most efficiently utilize these resources remains to be answered though, due to the many differences in system hardware complexity.

Nonetheless, not much has been published when it comes to multithreaded programming techniques for game engine architecture, because of the historical focus on always trying to break the performance limit of single-core CPUs [2]. Video games have been programmed as sequential loops which take user input, performs necessary computations and updates data. As hardware becomes more complex, other ways of structuring game logic has to be developed [4]. Attempts at multithreading games based on specific hardware components have been made, but such implementations lack in hardware adaptability due to their tailor-made nature and can therefore only run on platforms similar to what was originally intended. A more efficient strategy would be to find ways to add parallelism to the entire game loop, to more generally meet the potential of the parallel hardware and thus increase performance. A way of carrying out this proposal would be to run heavy calculations like physics and AI in a branch parallel to the game loop [5]. However, within game logic certain dependencies restrain the possibility of applying this solution. Task dependencies, as they are called, make parallelization impossible and force it to occur only in the chunks of code located between them. For instance, draw calls are dependent upon accessing the same memory location, namely the GPU, and must therefore execute sequentially [4]. Nevertheless, the ways in which parallelization can take shape in between dependencies is a surprisingly untouched subject. On account of
this, our study is an ambition to discover properties of parallel design patterns through the parallelization of two important facets of video game logic: flocking behaviour and skeletal animation.

The ultimate purpose is to find an efficient way to decrease rendering time in games. Since parallelism will be increasingly important in video games, an efficient way to parallelize them must be documented. The focus of parallelism is important, since the performance scalability depends on the critical path (the longest path in the program that must remain sequential). In order to avoid bottlenecks, one should try to shorten the critical path by analyzing which segments that can be parallelized [1].

A comparison between two parallel design patterns is conducted. The patterns in focus are fork-join and pipeline parallelism. In a variety of tests, the speedup, efficiency and latency of the two algorithms are measured respectively and used for a definitive comparison. The process through which all of this is done follows the design research approach, since it’s important to iteratively implement, test and document the different patterns and their level of parallelism.

This study presents an experience report, because it opens up for an unexplored area: parallel design patterns in games. Game programmers need to follow studies on parallelism to improve the capacity of their work and thereby the user experience of their products. The rest of this paper will describe how to analyze parallelism, task dependencies, the parallel design patterns in focus as well as our experiences from implementing the software and conclusions derived from a discussion of the results.

2. Background

Parallelism is not as one might think, something brand new; rather it has existed since the earliest computers all the way back to late 1980s. In more recent times, studies concerning parallelism in relation to games have been conducted to contribute with a range of results.

Sometimes it may be beneficial to parallelize as much of a game loop as possible. This can be done by finding already existing loops using the Synchronous function parallel model. The parallel tasks should be completely independent of each other, to reduce communication as much as possible. Typical game aspects that can be processed in parallel are animation and physics, because these branches often don’t depend on each other’s output. Furthermore, the functionality should be divided into small tasks followed by the creation of a task-dependency graph representing the tasks and how they precede one another. After this, the graph should be supplied to a framework, which will schedule the independent tasks with a more balanced
use of available processors. Without this approach, segments of code would have to be rewritten for each core to be utilized [5].

As technology has increased in complexity, it has been observed that multicore processors have been integrated into more systems. To develop games for multicore systems, a multithreaded game engine is necessary. The many components of today’s game engines provide a good use of parallelism, even though some game engines still rely on single-threaded execution. The structure of parallelism in games greatly depends upon the parallelism used in game engines [4, 2].

Some game engine performance tests have been performed with regard to the amount of processor units. The three games Serious Sam 2, Quake 4, and FarCry are some examples of these performance tests. These three games were executed on both a single-core, Athlon 64 3800+ computer and a dual core, Athlon 64 X2 3800+ machine. The goal was to measure the difference in performance in frames per second (FPS). All three games decreased in performance between 14-18% when running on a dual core processor compared to the single-core processor. The more amount of sequential processing, the fewer processors were required to gain the most of the speedup, which meant that not only more processors but also more efficiently implemented parallelism was necessary to gain a relevant difference in speedup [2].

Game engines for multicore systems has been constructed using task and data parallelism, topped by thread synchronization and load balancing. The experimental results, from testing a game created with the use of such an engine at hand, in different architectural environments, showed that the multithreaded model outperformed the single-threaded one [9].

It has also been shown that parallelism can be used for greater effects through the amount of rigid body physics, as well as smarter AI behaviour in the sense of resembling human behaviour [10].

3. Related work

3.1. Measuring parallelism

Parallelism is when program execution is subdivided between many processor units such that processing can be made simultaneously, thereby shortening the total amount of work. Large problems can often be divided into smaller ones, which then can be solved at the same time [6]. The operating system kernel includes a scheduler that runs threads on physical processors. Parallelism is built upon this scheduler, which uses a thread pool at its base. Parallelism uses the divide-and-conquer paradigm to divide tasks into subtasks and solve them on different processing units when they’re small enough. After each subtask has been concurrently solved they are joined together in
the main thread from which they were spawned (called the master thread). Sometimes it can even be an advantage to do the same computation twice on different units [7].

It is important to be able to measure and analyze the level of parallelism and speedup. Before measuring though, it is important to know what to analyze and why:

- **Latency:** The time it takes to complete a task. The units of time can be anything from nanoseconds to days depending on the system complexity. Since execution time is the center of attention for this study, latency is a necessary measurement.

- **Speedup:** Compares the latency for solving the identical computational problem on one processor (“worker”) versus on P processors. This is essential when comparing parallel design patterns to see where the biggest performance benefit lies.

When measuring the computational speedup from parallelism the variables $T$ and $P$ are used, where $T$ is the time of computations and $P$ is the amount of processors. $T_P$ represents the minimum amount of computations needed to execute a multithreaded program on a system using $P$ processors. $T_P$ is thereby the program latency. $T_P$ is a good generalization of the level of parallelism in a multithreaded program. The level of parallelism is the maximum possible speedup, or in other words the average amount of work along the critical path [1]. To achieve a relevant difference in speedup, as many threads as possible should continuously process tasks. This information gives a good estimation of the number of processors one should devote to a computation [7].

Speedup can be measured in the following way:

$$S_P = \frac{T_1}{T_P}$$

Where $T_1$ is the latency of the program with one worker (processor) and $T_P$ is the latency of $P$ workers.

- **Efficiency, $\varepsilon$:** The speedup divided by the number of workers:

$$\varepsilon = \frac{S_P}{P} = \frac{T_1}{PT_P}$$

Efficiency measures the return of hardware investment, and an ideal efficiency is 1 (or 100%), which is linear speedup [7].
3.2. Task dependency

A **task** is an asynchronous operation in a program. In an ideal setting, all tasks would be independent of one another and execute completely in parallel. This is almost never the case. Behind the scenes, tasks are queued to the thread pool, which has been enhanced with algorithms that adjust to the number of threads in order to maximize throughput. This makes tasks relatively lightweight, and you can create many of them to enable fine-grained parallelism [7, 1].

Most tasks depend on data to be processed by other tasks and hence they have to wait for each other to finish processing. It is therefore crucial to determine the pinpoints of the why and wherefore of such dependencies before attempting to parallelize, to avoid overheads. This is done through constructing a **task-dependency graph**, which is a data structure that represents dependencies between tasks and the order of their execution. To construct a task-dependency graph, one has to do a dependency analysis to map and determine the nature of all dependencies [4].

Ultimately, a game loop is a chain of parallelizable blocks of code; because of task dependencies in the game loop, certain statements need to come in a certain order. Our tests involve one such parallelizable block and how that particular part can be parallelized.

3.3. Analyzing parallelism

Below are some well known methods for analyzing parallelism:

3.3.1. Amdahl’s law

“The effort expended achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude” (Gene Amdahl [Amd67])

According to Amdahl execution time \( T_1 \) can be split into two categories:

- Time spent doing non-parallelizable serial work.
- Time spent doing parallelizable work.

\( W_{ser} \) and \( W_{par} \) represents the serial and parallel execution times. With \( P \) workers the execution times for each of these are:

\[
T_1 = W_{ser} + W_{par}
\]

\[
T_P > W_{ser} + \frac{W_{par}}{P}
\]
$T_P$ assumes no linear speedup, and is only exact if the parallel parts are perfectly parallelizable. Using these relations in the definition of speedup defines Amdahl’s law:

$$S_P \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{P}}$$ (4)

Speedup is limited by the amount of non-parallelizable code, the critical path, the part of a program that must remain sequential. This is the case even if an infinite number of processors are used. If 10% of the application is part of the critical path, then the maximum speedup is only 10, in the same way if 1% of the application cannot be parallelized, the maximum speedup will be 100%. Amdahl’s law interprets programs as fixed and computers as changeable, but as computer technology becomes more and more complex applications provides an exploitation of these features. He states that the level of parallelism is limited by the amount of sequential code, and that the serial workload remains constant. Amdahl also states that bad multithreading is worse than no multithreading at all [8].

3.3.2. Gustafson-Barsis’ Law

“Speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size” (John Gustafson [Gus88])

Gustafson-Barsis’ law states that any problem large enough can be efficiently parallelized. It is closely related to Amdahl’s law in the limitations of how a program can be sped up using parallelism. In difference to Amdahl’s law it describes how the problem size is related to the system capabilities. According to Gustafson it is most realistic to assume that run time is constant unlike problem size, which is constant in the case of Amdahl’s law [1, 8].

Gustafson-Barsis’ parallel measurements are calculated using:

$$S_P = W_{ser} + W_{par} \times P$$ (5)

Where $P$ is the number of processors and $W_{ser}$ and $W_{par}$ represents the serial and parallel execution times [13].

The size of the critical path determines the relevance of each law. If the critical path makes it possible to divide tasks into a larger amount than that of physical cores, Amdahl’s law becomes irrelevant. However, if the critical path amounts to more work than what can be divided to match the amount of cores, Amdahl’s law is relevant. Either way, each law has to be kept in mind when deciding how to parallelize software to know if more cores can be utilized. It all depends on the critical path within the parallelizable...
workloads, which differ throughout the game loop. In our practical work which represents one such workload, the critical path allows for more tasks than cores, and the parallel work is hence distributed in accordance with the amount of cores.

3.4. Fork-join

Fork-join is a parallel design pattern, where the main thread branches off (forks) at designated points in the program. The forked threads need to be independent of each other. After tasks have been parallely solved, they join together in the main thread again.

Fork-join usually forks tasks or lightweight threads in practice, rather than OS “heavyweight” threads. A thread pool is used to execute the tasks, allowing the user to specify potential parallelism. This user specification is then used to map the actual parallel execution on the underlying hardware. The overhead from spawning threads is circumvented by using a thread pool [1]. Forking should go deep enough to permit plenty of parallelism, although, not too deep. If it forks too much other problems are encountered, at the sub-level, such as scheduling overheads [1].

![Figure 1: How tasks fork into many parallel threads](image)

Calculating the execution of a basic fork-join is somewhat simple. The total work $T_1$ is the sum for each work along the multiple forks and joins. The $T_\infty$ is the maximum span of any path. Let tasks $B || C$ denote the fork-join composition of $B$ and $C$. The allover workspan is then:

$$T_1(B || C) = T_1(B) + T_1(C)$$

$$T_\infty(B || C) = \max(T_\infty(B), T_\infty(C))$$

[1]

3.5. Pipeline

A pipeline is basically a linear sequence of stages. It can be seen as a real factory pipeline, where items flow through the pipeline from one end to the other. Pipelines have several stages that transforms the data in flow, where at each stage the data is partitioned into pieces or items [1].

[1]
In a parallel pipeline several items can be processed concurrently at various stages in the pipeline, in difference to a serial one. The input items can be sent into different stages directly, and thereby delivering the output out-of-order. As long as stages don’t have dependencies, the out-of-order output won’t be a problem. When implementing a parallel pipeline, more efficient scalability is gained. The most common implementation of a parallel pipeline is the serial-parallel-serial one, where serial and parallel stages are mixed, such that stages with dependencies are forced to run before any parallel stages.

![Figure 2: A serial compared to a serial-parallel-serial pipeline](image)

If each task takes $T$ units of time, and each parallel task takes $4T$ (4 units of time), then the total work is $6n$ for $n$ input items. This is because each item requires two serial and one parallel task. The span $T_\infty$ is then $n + 5$, because the longest paths through the graph must pass through some combination of $n + 1$ serial tasks before and after passing any parallel one. This means that speedup is limited to $\frac{6n}{n+5}$ [1].
4. Method

**Figure 3: The Design Science Research Method (DSRM) process model applied to our study**

4.1. Research approach

*Leonard Bruce Archer* defined the six main steps of the Design Science Research Model (DSRM), which he summarized as: programming (project objectives), data collection and analysis, synthesis of the analytical results, development (to enhance design), prototyping, and documentation (to communicate the results). He claimed that by using these steps, designers would be able to approach problems more systematically, by looking at problems at a functional level in the form of goals and requirements [12]. Using the Design Research approach, we implemented the two parallel design patterns fork-join and pipeline parallelism with Intel’s library Threading Building Blocks (TBB). The TBB library supports scalable memory allocation, work-stealing, task scheduling, and utilizes full advantage of multicore performance. With TBB, one can work on how to focus parallelism, rather than the base of parallelism, which involves *asymptotic complexity logic*.

Initially, there were no distinctive game algorithms to parallelize. Rather, the software constituted a full game loop with scraps of gaming functionality which together formed an early prototype of a game engine. The idea was to parallelize the entire game loop using two different design patterns, which is an element of the study that has persisted throughout. Although, the choice of what design patterns to use is not a factor that has endured the entire span of creating this article. Instead of pipeline parallelism, the *flow_graph* algorithm was used together with the fork-join algorithm to create two test cases of parallelizing the entire game loop in separate projects. From these projects, data was generated and analyzed. After the results and experiences of the first iteration, a decision was made to approach the project differently, the reasons for which are outlined in section 6.1. The TBB templates used for each design pattern to implement became *parallel_pipeline*, for implementing a parallel pipeline and thereby replacing the flow graph, and *parallel_invoke*...
for implementing fork-join parallelism. These templates were applied to two different game algorithms in separate software such that relevant results could be generated and used in a definitive comparison, followed by a discussion and conclusion.

4.2. Data analysis

The data to be collected was the execution time, speedup and efficiency for the two algorithms respectively. To get a hold of the current amount of passed milliseconds since the initialization of the program, the SDL library was used, such that the execution time, i.e. latency, for the chosen block of code could be measured. Flocking and animation were tested separately and together for each parallel design pattern. The execution time was written to a .csv file during runtime every frame. This data was then used to calculate the speedup and efficiency gained from the different setups, from which a discussion and conclusion could derive. In this last part, the theoretical backbone described under background was used.

4.3. Limitations

One of the limitations with TBB was that there was no way to control all functionality; for instance how the task scheduler operated or how the data structures were used for storing and popping tasks. Neither were there any measures of coping with other factors that could limit the processing, such as RAM and cache-usage, hard drive speed as well as background OS-services. Because the software hadn’t been written from a cache-friendly perspective, the efficiency in utilizing the cache memory was unknown. Another factor which added to this uncertainty was the fact that other processes in the OS competed for that very same space of memory. Nor did we know how the template functions for the corresponding design patterns manage the cache and how that could affect the end results. Besides hardware obscurities there were the ways in which the game algorithms could change the conditions for themselves. This especially referred to the flocking solution where the entities moved freely across cells which gave rise to an unpredictability and uniqueness for each test case.
### 4.4. Table of iterations

<table>
<thead>
<tr>
<th>Iteration</th>
<th>DSRM process stage</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define objectives of a solution</td>
<td>A game engine prototype was developed for parallelization. Different types of parallelism were selected; fork-join and the flow graph.</td>
</tr>
<tr>
<td></td>
<td>Design and development</td>
<td>The game engine prototype was adapted to and integrated with the algorithms for implementing parallelism in separate projects.</td>
</tr>
<tr>
<td></td>
<td>Demonstration</td>
<td>The programs were executed to generate a collection of data consisting of latency values for the different projects.</td>
</tr>
<tr>
<td></td>
<td>Evaluation</td>
<td>See section 6.1.</td>
</tr>
<tr>
<td>2</td>
<td>Define objectives of a solution</td>
<td>Two distinct game algorithms to parallelize came to mind; BOIDS and skeletal animation. The parallel_pipeline algorithm corresponded to the specific design pattern pipeline parallelism and became the replacement for the flow_graph algorithm.</td>
</tr>
<tr>
<td></td>
<td>Design and development</td>
<td>The game algorithms were created and implemented for parallelization, which in turn was made with the two templates representing the different design patterns.</td>
</tr>
<tr>
<td></td>
<td>Demonstration</td>
<td>The general execution time for the varying software was collected during their execution for a series of cases consisting of differences in the amount of threads and game algorithms used.</td>
</tr>
<tr>
<td></td>
<td>Evaluation</td>
<td>See section 6.2</td>
</tr>
</tbody>
</table>

### 5. Implementation

The flocking behaviour and skeletal animation algorithms both use buffers to store data; in the flocking solution, artificially intelligent entities are stored in a container that is iterated upon and in the skeletal animation the buffer consists of bone transform matrices. The point of parallelization is thereby the iterations of the different buffers and their corresponding sets of operations.
Our flocking behaviour can best be described as a grid-based BOID system [11], where the behaviour of the entities are only affected by entities currently within the same cell. Thus, the buffer consists of a collection of cells with respective lists of entities, which is why the behavioral operations for all cells can execute independently. This implies an absence task of dependencies which is the condition for parallelization. The grid amounts to sixteen cells for the sake of balancing unity within the flock - when not using enough cells the potential for parallelization diminishes while using a large amount of cells cuts the entities off from each other frequently which disturbs the formation of the flock. Two hundred entities are spread out randomly across the grid at the initialization of the program. The amount of entities is due to make the algorithm sufficiently demanding of the CPU in terms of performance to generate results measurable in milliseconds.

As far as the skeletal animation is concerned, the bone transforms can be calculated in parallel because of their global nature - they are not hierarchically bound to one another, meaning that if all but one bone were to disappear, the remaining bone would still move about in mid air as if being maneuvered by a ghost. In the case of a hierarchical skeleton, all bones are connected in some way or another to a root node, from which a traversal of the skeleton can initiate. The parent transform is recursively passed down the joint hierarchy and used to calculate the global transforms of the child nodes. However, at the initialization of our program, all of these hierarchically oriented calculations are made for all bones in relation to their respective sets of keyframes for interpolation. The resulting data which holds the truth of the bones’ global condition is stored in a buffer made up of global rather than local bones. All bone transforms add up to an amount of nineteen to resemble the amount of cells, for the sake of combining the buffers in a single parallel pipeline.
The 3D-model for the BOID entity was modeled, rigged and animated using Autodesk Maya. To retrieve the necessary information in code, the model data was exported as an fbx file. Through the use of our programming environment consisting of game logic written in C++ and OpenGL as the graphics API, the data for rigging and animating the BOID mesh could be imported.

The two parallel design patterns were implemented on a system using an Intel Core i7-4710HQ processor with a 2.50 GHz clock rate, 4 physical and 8 logical cores, a 6 MB SmartCache on a x64-based PC with 16 GB RAM installed. Windows 10 was the operating system used.

6. Experiences

6.1. Iteration 1

As mentioned before, the original code base for parallelization was an entire game loop, or an early prototype of a game engine. The tasks for thread distribution would consist of the major branches constituting the loop and their subsequent parts. All task dependencies were determined, which functioned as the starting point for parallelization. While the process of pinpointing the dependencies was easy, implementing the design patterns was not; partly because of not being accustomed to TBB, but also due to the many
ways of solving the problem. Much of the code could be organized through a range of potential choices with respect to the TBB functions. Definitive representations of the design patterns could therefore not be created. If carried through, the conclusions would have been undermined by arbitrary choices of implementing parallelism. But this was not the only reason for deeming the game loop as insufficient, since it failed in representing game logic itself due to neither being a distinct algorithm nor a complete set of functionality for games. It could not in any way resemble the reality of game development.

Nevertheless, two projects differing in the use of parallelism were created and executed such that values of latency could be collected. When comparing the values generated by the parallel execution with the single-core data, no speedup whatsoever had been achieved. In fact, in some instances the parallel software performed even worse in terms of latency than that of the single-core version. In short, the data was inadequate to arrive at any conclusion about parallel design patterns.

Another element which added weight to the need for change was a lacking aspect in the flow_graph template function. The choice of it as the contender against parallel_invoke was based on the assumption that it corresponded to a specific parallel design pattern. However, after delving deeper into the subject of what the flow_graph function actually was, no such correlation could be found. The challenge of finding another function arose, tied in with the issues of using the game loop as the basis for implementing parallelism as well as the inadequate test results.

Rather quickly the parallel pipeline design pattern was discovered and chosen as the replacement for the flow graph algorithm. Since the pipeline pattern demanded a set of items to flow through subsequent stages, it made even more sense to focus the test code on distinctive algorithms instead of an entire loop. Knowledge regarding BOIDS and skeletal animation condensed with an understanding of parallelism made way for the creation of ideas (see section 5) which sufficiently met the criteria established by the failures of the first iteration.
6.2. Iteration 2

Fork-join

![Graph: Fork-join speedup and efficiency](image)

*Figure 5: The speedup peaks when using four threads for all cases except flocking. This is due to the way entities cluster across cells resulting in a load imbalance.*

There were some difficulties in wrapping up the sequential code with the fork-join template. Not because of the template itself, which on the contrary was easy to use, but because of the necessity to split the buffers such that all work could be spread evenly across all threads. This was done through the use of a method which through integer division split buffers into parts which could then be iterated upon in parallel, leaving integer leftovers to the last thread. The better the amount of threads fit for dividing the buffers and leaving small leftovers, the more efficient was the spread of work across threads. The animation skeleton contained nineteen bones whereas the grid consisted of sixteen cells.

An interesting difference between the two pieces of logic when increasing the amount of threads is that the flocking speedup reaches its peak at eight threads where animation steadily decreases in speedup after the use of four threads, which makes sense according to *Gustafson-Barsis Law*, since parallelization is only possible to the degree of the amount of cores available - in this case four. This means that the speedup when using eight threads is in fact not because of parallelization. Rather, it is a consequence of how the entities flock and transition into other cells, meaning that the workloads differ as the lists of entities within each cell is in a constant flux. When splitting these cells into fewer chunks due to a lesser amount of threads, the risk for a load imbalance increases. What this tells us is that a larger amount of tasks reduces the risk for an uneven workload in the case of a grid based flocking system - not a larger amount of threads, although that seems to be the case due to the fact that the software generates the same amount of tasks as threads. This effect is counteracted by the load imbalance which
occurs when the buffer leftover for the last thread increases in size due to the increasing amount of tasks, hence also threads.

What’s interesting to note is the amount of overhead generated when increasing the amount of threads - the efficiency rapidly decreases when comparing two to four threads despite the increase in speedup.

**Pipeline**

![Pipeline Graph](image)

*Figure 6: Unlike fork-join, there are no exceptions as to what amount of threads generate the largest speedup, which is even the case when the buffers for both algorithms share a common pipeline.*

Experiences regarding the fork-join design pattern apply to some extent to this design pattern as well. The decrease in efficiency is not quite as severe in this case but it is nonetheless a fact and the general tendency for all setups to peak in speedup when using four threads is also the case here.

Another, although vague, similarity is the speedup tendency regarding the flocking algorithm. As can be seen in the graph, the line does not fall rapidly in speedup after passing four threads. Nor does it gradually sink, but moves upward instead, at the point of eight threads. However, this is not because of an uneven spread of work described above, because in a parallel pipeline tasks are generated for parallel execution when the first serial step for any token has executed. The results may even be the outcome of the randomness of the encounters within the boid flock. The differences in speedup are not very large and the general tendency across the graph is a falling one.

Otherwise, there were no difficulties when implementing a pipeline for the game algorithms separately, but a challenge arose when both buffers and their operations had to coexist within the same pipeline, for the sake making them operate in parallel. This required additional logic, i.e. overhead. The solution was to skip the step involving the algorithm that had already taken it’s toll of operations. In short, an addition of statements to check if one or the other set of operations had been executed was made, despite of which the speedup increased in a fashion similar to when testing the algorithms
separately, which proves that a parallel pipeline can be used for two series of elements and still achieve an increase in speedup.

Figure 7: The shared parallel pipeline for the flocking and animation algorithms where each parallel step is responsive for executing operations corresponding to one of the two algorithms.

7. Discussion

Here lies the comparative results in the form of graphs and their matching descriptions. What follows is the knowledge derived from the experiences of testing different types of parallel design patterns for a variety of test cases and how that knowledge matters in the larger context of parallelism in game development.
Figure 8: When parallelizing our flocking algorithm, the pipeline template function is clearly more effective when it comes to achieving speedup.

Figure 9: The differences displayed in figure 8 also apply to the animation solution, but the differences in speedup and efficiency aren’t as big. However, when increasing the amount of threads for the pipeline function, a solid speedup is nonetheless obtained, in difference to the fork-join template.

Figure 10: As in figure 9 and 8, the pipeline template generates the more performant result, although the gap in speedup between the two functions is significantly smaller.
What all these tests amount to is a stage free of dependencies, in a typical game loop. Aspects such as artificial intelligence and animation are seldom dependent upon one another and were therefore chosen as the branches of game logic to be parallelized. Knowing how the areas of a game loop are compatible with each other for parallelization is key in order to efficiently parallelize. However, the uncertainty regarding the independent blocks of logic remains. Since in reality, the workload of segments may differ within such a stage or a superfluous amount of threads may exist in relation to physical cores or the options for iterating upon buffers can span across such a variety of parallel design patterns that it might put the user in a state of uncertainty.

What our final results display then, is the most efficient way to parallelize a particular case, which resembles what game developers encounter when parallelizing their software. When parallelizing buffers, the easiest and most efficient template function and parallel design pattern to use is the parallel pipeline. This also applies to multiple buffers, even if the number of elements between the containers don’t match perfectly.

However, the parallel pipeline is not as efficient where buffers are not concerned. It would therefore be an option to mix the use of design patterns, such that a pipeline would iterate upon a buffer in parallel with a task consisting of a larger sequence of operations, through the use of the fork-join design pattern. The amount of possible cases for which different design patterns can be applied and combined is of an undetermined scale which will not be covered in this paper. Nevertheless, the results may function as a general guideline for when to use the fork-join as well as the pipeline design pattern and how to combine the two, to generate the strongest possible push in speedup.

Another lesson to be derived from the results is that the general tendency across all results shows that the gained speedup drops in accordance with the overhead generated by the abundance of threads in relation to the amount of available physical cores, which confirms Gustafson-Barsis’ Law. In other words, it is important to focus the parallelism efficiently across the available amount of physical cores.

But not only are there lessons to derive from test results where an optimal level of performance is articulated. In fact, the opposite can be just as useful because it articulates what not to do. Recall iteration 1: The sequential loop outperformed the parallel versions which was the consequence of an arbitrary approach with no consideration for overheads, rendering the parallelism inefficient.
8. Further research

To gain an even larger understanding of how parallelism can be utilized for performance optimizations in games, more parallel design patterns can be implemented in different combinations for larger sets of cases. One way would be to use parallel design patterns for game loop branches which differ from the ones used in our study or for the same types of branches, i.e. AI and animation, but with different structures in logic. In short, the variations in which algorithms can be constructed with regards to parallel design patterns is a field worthy of exploration. It is only through the use of testing more patterns in relation to a larger variety of parallelizable operations that the particulars of how to most efficiently parallelize can be mapped. Another relevant course of action would be to compare differences in implementations of parallel design patterns. Algorithmic alterations of the same design pattern from different libraries could be compared as a way of clarifying this subject matter. Yet another fundamental aspect of managing parallelism is the manner in which operating systems vary and generate a scope of results, which make them an obvious choice for scrutiny. Then there is the low-level approach of observing the relationship between hardware and design patterns through comparing results from a range of machines differing in clock rate, cache memory, bus width or RAM for instance.

A hardware approach could also be useful in carrying through the same tests but with a greater amount of tasks, since the current amount is relatively low (19 bones and 16 cells), to determine if the results scale well on more efficient systems.

9. Conclusion

Since there exists a lack of knowledge regarding parallel design patterns for specific cases of parallelizable steps in a game loop, this study was conducted. Two parallel design patterns were implemented and compared in speedup and efficiency for cases which in all likelihood resemble what a large group of game developers encounter when parallelizing their loops. The results present a general guideline for what parallel design pattern to use at what point in the different stages of the game loop, by showing that the pipeline design pattern is preferable when parallelizing the iteration of buffers. Furthermore, the results confirm that multithreading outperforms the single-threaded architecture and especially so when matching the amount of available threads with the amount of physical cores.
References


