Parallelizing Games for Heterogeneous Computing Environments

Wessam AlBahnassi

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Software Engineering) at

Concordia University

Montreal, Quebec, Canada

October 2012

CONCORDIA UNIVERSITY

Department of Computer Science and Software Engineering

This is to certify that the thesis prepared

By: **Wessam AlBahnassi**

Entitled: **Parallelizing Games for Heterogeneous Computing Environments**

and submitted in partial fulfillment of the requirement for the degree of

Master of Applied Science (Software Engineering)

Complied with the regulations of the University and meets with the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____Chair
Dr. Peter Rigby

_____Examiner
Dr. Bipin C. Desai

_____Examiner
Dr. Hovhannes Harutyunyan

_____Co-Supervisor
Dr. Dhrubajyoti Goswami

_____Co-Supervisor
Dr. Sudhir P. Mudur

Approved by _____
Chair of Department or Graduate Program Director

_____20____ _____
Robin A.L. Drew, Ph.D.,ing., Dean
Faculty of Engineering and Computer Science

ABSTRACT

Parallelizing Games for Heterogeneous Computing Environments

Wessam AlBahnassi

Game applications nowadays run on platforms that contain multiple and different types of processors. Benefiting from the available processing power requires: (1) Code to be designed to run in parallel; and (2) Efficient mapping techniques to schedule tasks optimally on appropriate processors for increasing performance. The work in this thesis is directed towards improving the execution time performance of game applications on host platforms. The methodology adopted consisted of the following different stages: (1) Studying the main features of game applications; (2) Investigating parallelization opportunities within these applications, (3) Identifying/designing the programming structures needed to realize those opportunities; and (4) Designing, implementing, verifying and testing the parallel programming structures developed to support heterogeneous processing and maximize use of available processing power.

The research reported in this thesis presents a new framework for parallel programming of games. This framework consists of two major components which form the two main contributions of our work. These are (1) Sayl, a new design pattern to enable parallel programming of game applications and to reduce the amount of programmer work when writing parallel code; and (2) Arbiter Work Stealing, a new task scheduler capable of efficiently and automatically scheduling dynamically generated tasks to processors in a heterogeneous processing and memory environment. The framework was used to successfully parallelize the serial code of a sample game application to work in a

heterogeneous processing environment. The parallel version shows far superior performance as compared to the serial version.

ACKNOWLEDGEMENTS

First and foremost, I wish to express my profound gratitude to my supervisors Dr. Sudhir P. Mudur and Dr. Dhrubajyoti Goswami for their continuous commitment, encouragement, and support. I feel privileged for having the opportunity to work with them and share passion for research. I would like also to extend my appreciation to my committee members and to all professors who provided me with their valuable knowledge during my studies at Concordia. I would like to pass my special thanks and gratitude to all my family members and friends:

- My parents. There is nothing I can do to return even an iota of favor for you.
- My wife for her pure love, support and patience and my son Omar for being the most wonderful thing in life.
- My brother Homam who supported me throughout the entire process with guidance and experience, and his wife and son 'Aws' whose name was used in this thesis.
- My friend Abdulrahman Al-lahham for his valuable help during my course work.

Additionally, I would like to thank my colleagues at Electronic Arts Montreal for their support and understanding during pressure times of my thesis.

Finally I would like to dedicate this work to all heroes of the Syrian Revolution and all the humanitarian values it is carrying. Names like Hamza Al-Khateeb and Husein Harmoosh will remain engraved in my heart forever. Victory is eventual, god willing.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

2D	Two-dimensional
3D	Three-dimensional
AI	Artificial Intelligence
API	Application Programming Interface
AWS	Arbiter Work Stealing
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
GPU	Graphics Processing Unit
PPE	Parallel Programming Environment
PPU	Power Processing Unit
PS3	Sony Playstation3®
SDK	Software Development Kit
SLOC	Source Lines of Code
SPU	Synergetic Processing Unit
UI	User Interface

CHAPTER 1 INTRODUCTION

1.1 GENERAL BACKGROUND

The domain of game applications continues to grow in complexity and size. With the ubiquity of multi-core processors, it is important that game programs be designed to make effective use of this commodity parallel processor architecture. Most applications in this domain have originally been written for a single processor environment. At their core, these programs usually include a game/simulation loop. In any iteration of this loop, all tasks required for each frame are carried out. In recent years, some of the existing games have started transitioning to support multiple processors. This is usually done by converting selected sections of their code to run in parallel. Tasks within a single iteration (frame) of the game loop are usually parallelized but not the tasks across frames. Newly written games should be able to maximize use of the available multiple processors. Detecting parallel tasks throughout the application rather than confining to a single iteration of the game-loop is therefore important. Whether the programmer is parallelizing existing serial game code or writing new parallel game code from scratch, there is a need for appropriate parallelization models which can make this parallel program development simple and straight forward. Unfortunately, there is not much published work found on this topic.

The interest in utilizing the graphics processor unit (GPU) for general-purpose computations (commonly known as GPGPU) has increased after those processors have shown significant computational powers in addition to being wide-spread among commodity machines. A large portion of GPGPU work has mostly been focused on

programming specific algorithms to work efficiently on the GPU (e.g., matrix operations, image filters, and sort algorithms). A game application can include a much wider variety of computations (e.g., artificial intelligence, physics simulation, character animation, and 3D rendering), and can also benefit from the GPU as a general purpose processor. The environments in which games are usually run include at least one CPU and one GPU on the same machine, in addition to, possibly, other processor types (e.g., the SPUs on the Sony Playstation3 game console). Some of the computations in such applications map better to the capabilities of one processor over another. It is thus important to be able to execute computations in parallel on the different processors in a way that execution time is minimized, and hence overall performance is maximized. Supporting heterogeneous processing environments is important but adds considerable complexity to the scheduling problem.

1.2 THE RESEARCH PROBLEM

Based on the above our research problem can be stated as follows:

What are the parallelization models which take into account the specific characteristics of game applications and make parallel program development on heterogeneous computing environments simple and straight forward, and also result in performance improvements as compared to their serial code versions?

1.3 RESEARCH OBJECTIVES AND METHODOLOGY

The main objective of this research is to investigate the process of parallelization of game applications from both code design and run-time perspectives and to develop suitable

structures to minimize programmer effort and maximize application performance while considering the heterogeneous processing capabilities of game host environments.

The research methodology followed has the following stages:

1. Identify parallelization opportunities in game applications by studying their common structure and dependencies, and the distinctive capabilities of the different computing components in a heterogeneous computing environment.
2. Propose a new framework for parallelization of game programming on such heterogeneous computing environments. The framework should simplify the job of writing parallel game code or parallelize existing serial game code and minimize the amount of programmer effort involved in the process. It should exploit parallelization opportunities in the game applications. Further, it should automate the job of optimally mapping and scheduling subtasks in the game application to appropriate computing elements in the environment so as to be able to achieve the best application performance.
3. Implement, test and verify the performance of the proposed framework on sample game applications running in an environment with heterogeneous computing elements.

1.4 CONTRIBUTIONS

From this research our main thesis is that game applications have specialized parallelization requirements in the form dynamic tasks which need to be executed in different frames of the game and that these requirements can be met with a game

parallelization framework consisting of a parallel design pattern in the front end and a task scheduler at the back end which maps and assigns tasks to processing elements based on minimizing a cost function taking into account processor type, processor loading, data transfer times and data locality. The framework not only simplifies the parallelization task in terms of programmer effort but also helps improve the performance of games significantly by better utilization of the processing elements in the heterogeneous computing environment.

The main goal is to improve performance of game applications by better utilization of the heterogeneous parallel processing capabilities available in their host environments. In order to do so, first the game code must be written to express game tasks in parallel. Then, these tasks must be efficiently scheduled to processing elements in such a way that attempts to achieve optimal overall performance. We propose a new game parallelization framework for achieving these goals.

To achieve the first goal, the framework includes a new parallel design pattern that – when applied- will build a dynamic task graph of all game tasks, honoring their dependencies. The second goal is achieved by including a task scheduler that can take tasks from the dynamic task graph and schedule them to processing elements in the host environments, depending on their loading, their capabilities, data transfer costs and data locality.

By applying this framework, game applications can scale their performance with the processing capabilities of their host environments.

Our specific contributions in this thesis are:

- Design and implementation of the Sayl parallel design pattern for game applications and
- Design and implementation of the Arbiter Work Stealing scheduler, a task scheduler that supports efficient dynamic scheduling of tasks to processing elements in heterogeneous environments.

1.5 THESIS STRUCTURE

The rest of this thesis is organized as follows. In the next chapters, a comprehensive literature review is first presented in Chapter 2 including game applications, game hardware, parallel design patterns and task scheduling algorithms. In Chapter 3, we present the new game parallelization framework consisting of a design pattern for parallel game programming and a task scheduling system for heterogeneous processing environments. In Chapter 4, the case studies involving parallelization of a sample game application using this framework are presented, including a summary of the framework implementation details, and a discussion of the test and results. The research contributions and future work are described in Chapter 5.

CHAPTER 2 LITERATURE REVIEW

2.1 INTRODUCTION

This chapter focuses on related topics from game applications structure, parallel design patterns and task scheduling algorithms. The purpose is to investigate the trends in research for parallelization and scheduling techniques that can support the parallelization process of game applications.

2.2 GAME APPLICATIONS

The process of programming game applications has been growing in complexity due to the growth and expansion of the various aspects in games. Nowadays games may include a combination of artificial intelligence algorithms, 3D rendering techniques, audio processing algorithms, multiplayer and online gaming support and input device processing in addition to game logic (Blow, 2004). Those tasks generate data and update the state of the game and output graphics and audio to the player. The tasks are usually executed in a certain order in a continuous loop commonly called the *game loop*, which is a standard structure most game applications are built around. The game loop is iterated in a certain frequency, usually at 30 hertz or 60 hertz that matches the display device's refresh rate. In each loop iteration or *frame*, the game gathers user input and uses it to update the logic and state of the game before finally outputting results to the player (e.g., graphics display, audio output and input device feedback) (Valente, Conci, & Feijó, 2005). Figure 2.1 shows a simplified graph of a game loop.

Figure 2.1: A simple game loop (Valente, Conci, & Feijó, 2005)

Each iteration of the game loop must finish execution within 16.6 milliseconds or 33.3 milliseconds depending on whether the game targets 60 hertz or 30 hertz display refresh rates respectively. However, the continuous growth in size and complexity of game computations makes it challenging to execute all tasks in a single game loop iteration within the time limit. Thus, optimization techniques became necessary to increase efficiency of calculations. Such optimizations include:

- Space partitioning techniques such as Binary Space Partitioning Trees (BSP Trees) and Octrees to speed up queries of game entities (Ericson, 2005).
- Frustum view culling to reduce the number of displayed game entities.
- Vectorization of calculations using Single-Instruction-Multiple-Data (SIMD) compiler intrinsics (Gschwind, 2007) such as SSE (Intel Corporation, 2007) and AltiVec (Diefendorff, Dubey, Hochsprung, & Scales, 2000). Vector and matrix operations are commonly optimized with this technique.

Even though these optimizations help improve performance, they are not always enough to bring down the time taken to complete a frame to fit within the limit. Additionally, it is not possible anymore to rely on faster processors to be able to run the game fast enough. The growth of processing power by increasing processor clock speed has stagnated and instead processing power now grows by employing additional processing cores in the machine. Games written using the classical game loop structure cannot automatically benefit from such kind of processing power growth without changing the code to be able to run in parallel on the available processing cores. The work of (Rhalibi, Merabti, & Shen, 2006) offers a framework with which games can be modeled as cyclic-task-dependency graphs by analyzing the dependencies between the different tasks in the game. The tasks in the graph are then executed on a multithreaded system in a manner that preserves task dependency, thus benefiting from parallel processing power in a shared memory system with more than one processor, but of the same kind.

The graphics processing unit (GPU) has become nowadays an integral part of commodity personal computers as well as game consoles. This is yet another processor available for game applications that is targeted mainly towards accelerating 3D graphics rendering and display. The work in (Joselli, Clua, Montenegro, Conci, & Pagliosa, 2008) and (Joselli M. , et al., 2009) offloads some of the game tasks such as physics to the GPU such that these tasks run in parallel to other tasks in the game loop. It is thus clear that performance scalability of games on modern hardware considerably depends on the amount of parallelization the game code supports.

2.3 GAME HARDWARE

This section reviews the major characteristics of hardware on which games are hosted in recent years.

2.3.1 The Graphics Processing Unit

Unlike the general purpose central processing unit (CPU), the GPU is a specialized processor which holds a large collection of fixed-function and software-programmable processing resources (Fatahalian & Houston, 2008). It contains an arithmetic logic unit (ALU) with comparatively high peak floating-point rates. It is originally designed to process 3D graphics, which is often described as an *embarrassingly parallel* task (Hansen, Crockett, & Whitman, 1994). This explains the decision behind the current architecture of the GPU (Kilgariff & Fernando, 2005), which is built around a series of connected components matching the pipeline steps used to process 3D graphics from input mesh and texture data set until outputting pixels on the screen (Figure 2.2).

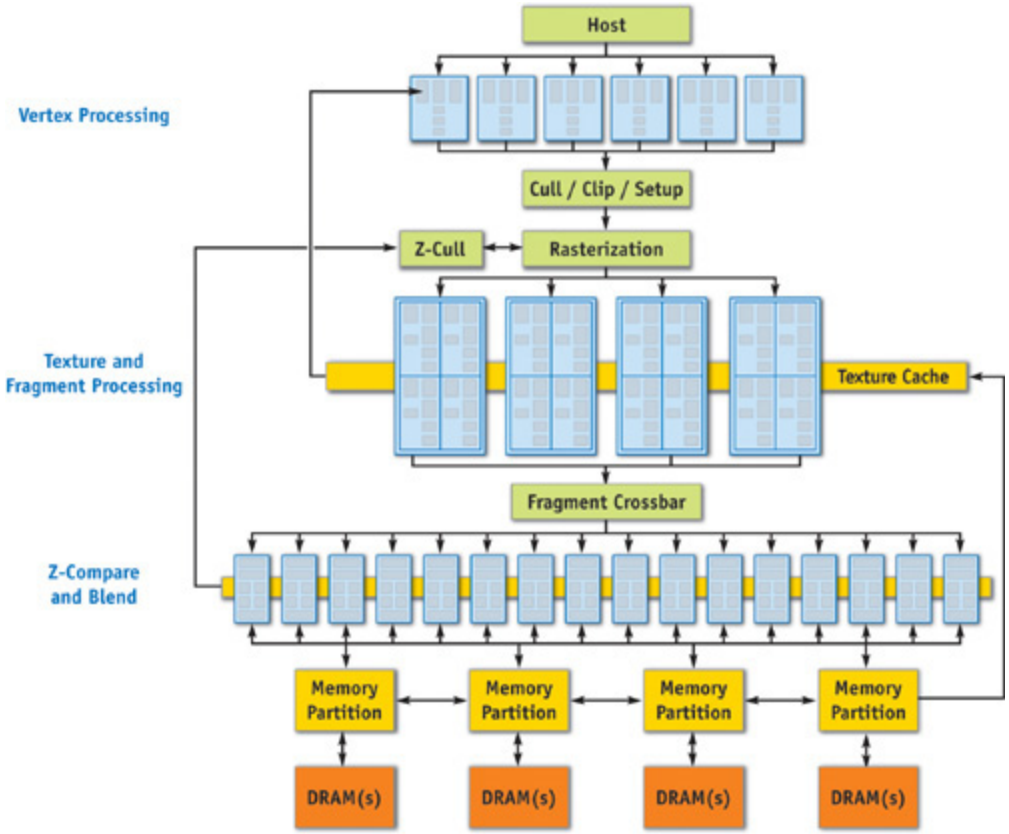


Figure 2.2: A block diagram of a GPU (Kilgariff & Fernando, 2005)

The most notable feature of these components is the presence of multiple *vertex processing units* (VPU) and *fragment processing units* (FPU). Each group of these processors can be configured to run a *shader program* which is set by the programmer through APIs like OpenGL (Khronos Group, 1997) and Direct3D (Microsoft Corp., 2012). This processing power has attracted researchers as well as game programmers to utilize the GPU for more than just graphics output tasks. The interest in utilizing the GPU for general-purpose computations (commonly known as GPGPU) has increased. A number of APIs have been developed to facilitate GPGPU computations such as CUDA (NVIDIA Corp., 2007), OpenCL (The OpenCL Specification, 2010) and DirectCompute (Boyd, 2008). In addition, C++ AMP (Microsoft Corp., 2012) is a set of language

extensions that allows running computational kernels written in C++ on the GPU and the CPU. This has considerably simplified program development for these processors.

A large portion of GPGPU work has mostly been focused on the programming of specific algorithms to work efficiently on the GPU (e.g., matrix operations, image filters, and sort algorithms). Game applications have a much wider variety of computations and can also benefit from the GPU as a general purpose processor.

2.3.2 Personal Computer Architecture

Most personal computers (PCs) are built around an architecture that contains one or more main CPUs -where each CPU could be a multi-core processor of its own- in addition to one or more GPUs residing on graphics cards (see Figure 2.3). In such systems, the CPUs have direct access to the system's main memory (RAM), while the GPU may have its own local memory that is built in the graphics card (VRAM). In this architecture, the GPU has no access to RAM, but the CPU has read/write access to the VRAM (Hovland, 2008). In most cases, write speed from CPU to VRAM is multitudes faster than read speed (Gregg & Hazelwood, 2011).

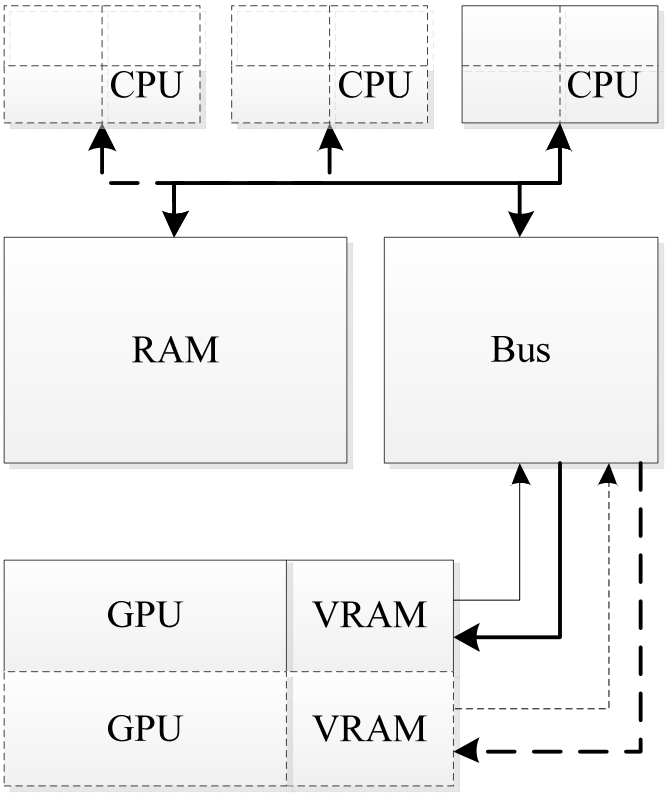


Figure 2.3: An abstract high-level architecture graph for common PC platform systems focusing on processor, memory and bus components. Lines represent memory access between components. Arrows represent direction allowed for data transfer. Thick lines represent faster data transfer compared to thin lines. In this architecture, the number of CPUs and GPUs is flexible. Moreoever, each CPU might have a number of cores.

2.3.3 Microsoft Xbox360 Architecture

The Microsoft Xbox360 is one among recent common game consoles. From a high-level architectural point-of-view, the Xbox360 shares some similarities with the PC's CPU/GPU high-level architecture. The Xbox360 (Andrews & Baker, 2006) has a CPU made of three hyper-threaded cores, in addition to a GPU chip. One major difference to the PC architecture is that the Xbox360 is built around unified memory architecture (UMA), and the GPU itself is the memory controller of the system. In this case, data

transfer times between the CPU and the GPU are similar, and the GPU has full access to the system's RAM (Figure 2.4).

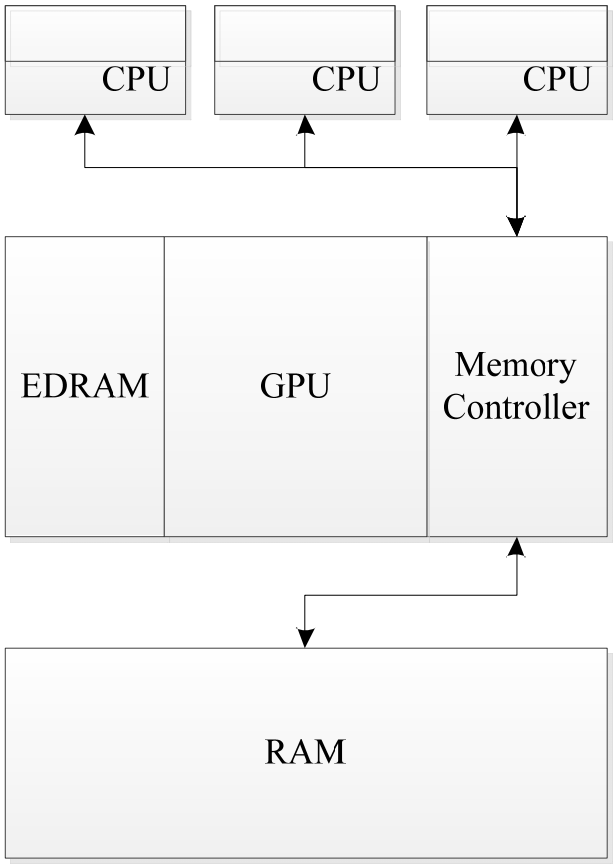


Figure 2.4: An abstract high-level architecture graph for the Microsoft Xbox 360 platform focusing on processor, memory and bus components. Lines represent memory access between components. Arrows represent direction allowed for data transfer.

2.3.4 Sony Playstation 3 Architecture

Another recent game console is the Sony Playstation 3 (PS3). Its architecture is special among others in that it utilizes IBM's CELL processor (Kahle, et al., 2005), which can be described as one dual-threaded CPU and 6 Synergetic Processing Elements (SPUs) capable of fast vector computations exposed to the programmer. Each SPU has 256kB local memory in addition to full access to RAM via DMA requests. The PS3's GPU

(similar to the PC) has its own VRAM and has restricted access to RAM. Transfer speeds between the CPU and the GPU are also similar to that of the PC architecture (Figure 2.5).

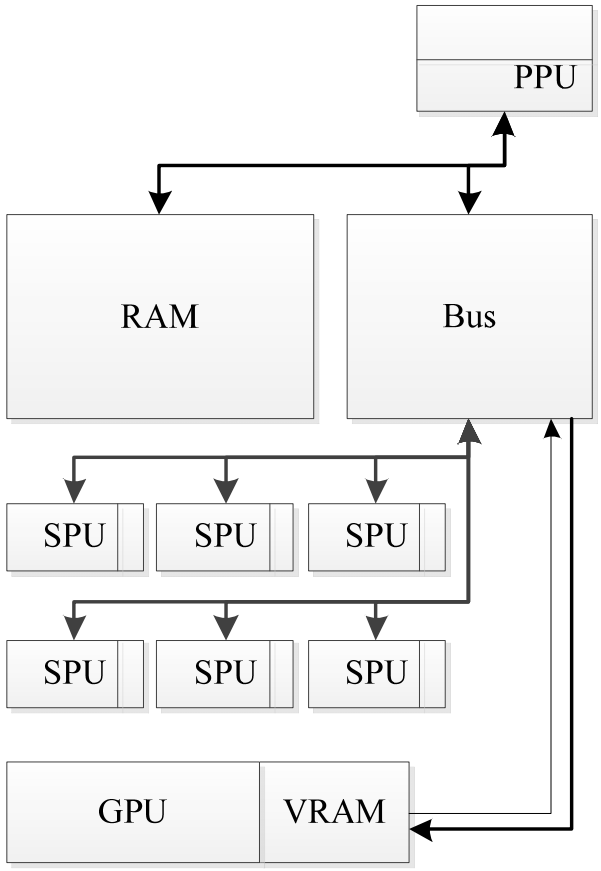


Figure 2.5: An abstract high-level architecture graph for the Sony Playstation3 platform focusing on processor, memory and bus components. Lines represent memory access between components. Arrows represent direction allowed for data transfer. Thick lines represent faster data transfer compared to thin lines. Note that each SPU has a small local memory block attached to it.

2.4 PARALLEL DESIGN PATTERNS

Design patterns are general reusable solutions to commonly occurring problems within a given context in software design. Game applications, having a set of common characteristics and could benefit from design patterns that help solve problems often

occurring in these applications. One such problem that is the focus of this thesis is parallelization of tasks in a game application. Thus, it is worth looking into existing work to solve this problem for the specific domain of games.

Task Parallelism (Mattson, Sanders, & Massingill, 2004) is a design pattern that describes a solution strategy for task-based parallelism expressed in the form of a task graph. It assumes that the problem can be decomposed into a collection of tasks that can execute concurrently. The tasks can have dependencies among them. They may be known at the beginning of the computation or arise dynamically as the computation unfolds. The task parallelism pattern can be considered a generic pattern, as it does not specify how to express tasks and their different types of dependencies (e.g., the data-flow type dependencies). Moreover it does not specify how to execute these tasks on parallel computing architectures. The pattern thus could serve as a basis for specialization into a domain-specific design pattern.

Fork-Join (Kim & Agrawala, 1989) is a classic pattern that specifies a queue where incoming tasks are split for execution by threads and joined before departure (Figure 2.6).

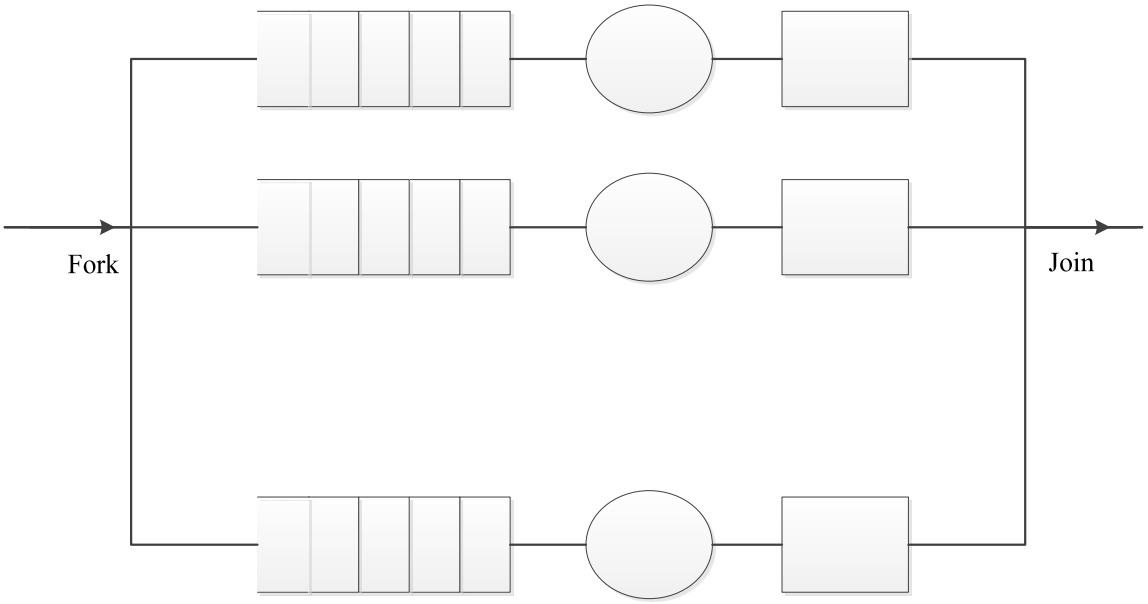


Figure 2.6: Fork-Join pattern (Kim & Agrawala, 1989)

The pattern only has support for specific types of data-flow dependencies (fork-join type) and it only supports parent-child synchronization via join. It cannot be used to model more irregular dependency types.

The data-flow pattern (Manolescu, September, 1997) is based on the data-flow paradigm introduced in the late 1960s (Adams, 1969) (Rodrigues & Bezos, 1969). The pattern models parallel programs as modules of execution which take input and produce output (Figure 2.7). Those modules can only run when all their inputs have been evaluated. When the module finishes executing, it sends outputs that then become available to other modules. Output generated from a data-flow module is sent as input to other modules only when the module calculating the output has finished.

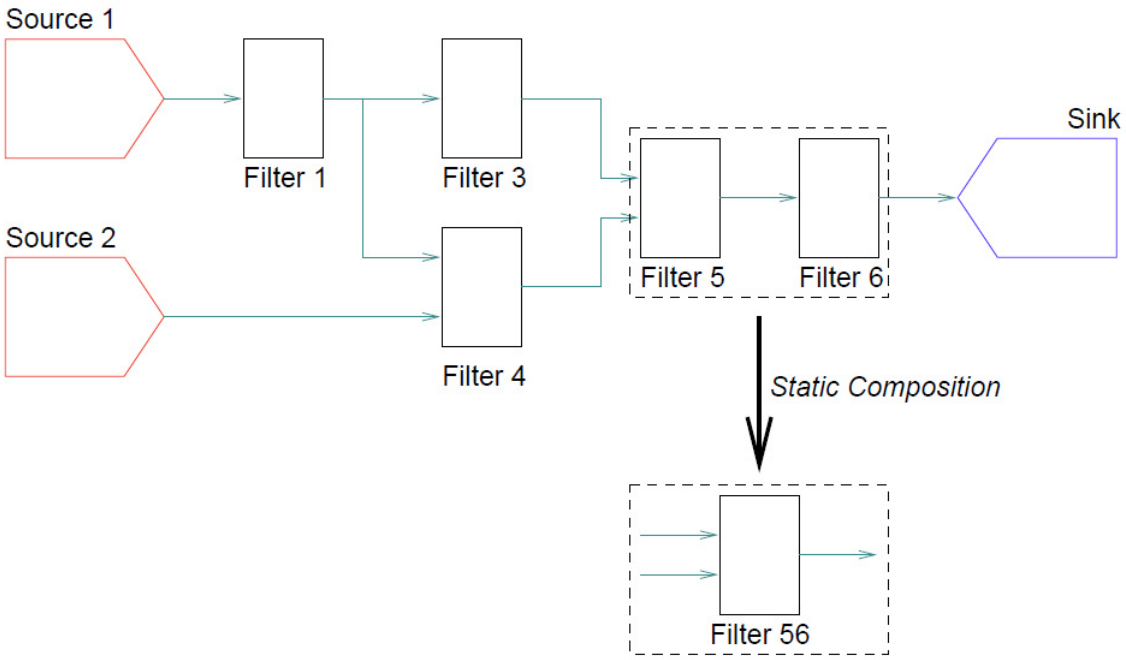


Figure 2.7: The data-flow pattern (Manolescu, September, 1997)

The Task Graph pattern (Miller, 2010) is a task-parallel pattern that specifies breaking down the computation into a series of atomic tasks with clearly defined dependencies to form a directed acyclic graph. Tasks in the task graph are specified at compile time, but they are scheduled at run-time to run on a pool of processing elements. This allows the graph to adapt to the run-time environment setup and thus scale with the number of processing elements available. Once the tasks and their dependencies are specified by the programmer then the task graph can be executed. Dynamically spawned tasks are not supported directly by the pattern, but a suggestion is made as to emulate the dynamic spawning behavior by using conditional tasks. Such tasks may be present in the graph but the task and the conditions under which it executes are defined by the program prior to execution of the graph.

The Dynamic Task Graph (DTG) pattern (Johnson, Davis, & Hadfield, 1996) describes the same task graph concepts as the previously mentioned works, but adds the capability to dynamically spawn tasks in the task graph. The DTG pattern requires each task in the program to be uniquely identified by a string name. This name is a key piece of information for referencing tasks in the graph. It is important to note that processors in the DTG pattern block execution when a task requests the output of another task that has not become eligible for execution.

Cilk (Blumofe, et al., 1995) is a language extension to the C language. It is based on a run-time system for multithreaded parallel programming which uses a work-stealing scheduler. Programs built with Cilk dynamically unfold as a directed acyclic graph (Figure 2.8). The program consists of a collection of "procedures". A procedure is broken into a sequence of "threads". A thread is effectively a non-blocking C function. Such functions will run as soon as they are called without blocking the caller. A function can itself spawn other threads. A thread may have data dependencies which effectively prevent its execution until they are fulfilled. Code written in Cilk language is translated to C to interface with application code.

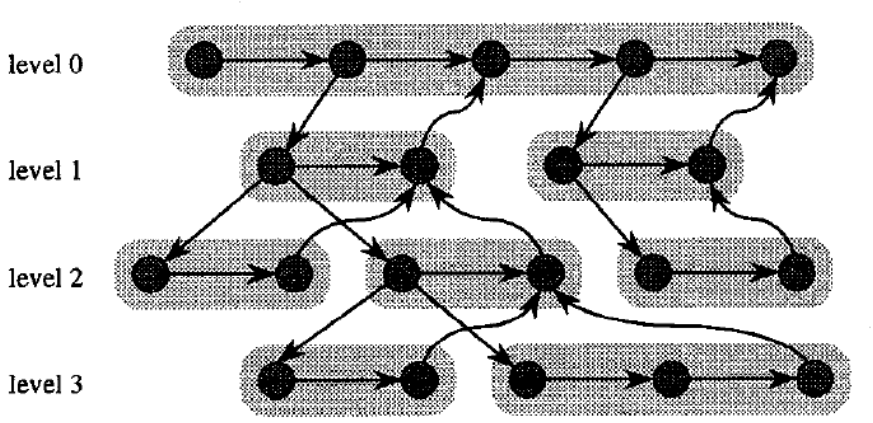


Figure 2.8: The Cilk model of multithreaded computation. Circles represent threads which are grouped into procedures. Straight lines represent spawns and curved lines represent data dependencies (Blumofe, et al., 1995).

The work of (Best, et al., 2009) describes an effort made to parallelize a game engine. This effort led to the creation of Cascade, which is a parallel programming environment (PPE) specifically addressing the games domain. Cascade allows the programmer to organize game tasks in a graph with dependencies in a fashion similar to the Task Graph pattern by (Miller, 2010). It supports an efficient parallel implementation of the producer/consumer pattern. It also supports multi-instance tasks, that is, spawning the same task multiple times simultaneously but with different data sets to work on for each of the task instances (Figure 2.9 and Figure 2.10). Similar to (Miller, 2010), Cascade does not allow for dynamically generated tasks in the task graph.

```
// Create task objects.
// Task A: 1 instance
A a( 1 );
// Task B: batch size 2, 1 instance
B b( 2, 1 );
// Task C: batch size 1, 4 instances
C c( 1, 4 );
// Task D: batch size 1, 3 instances
D d( 1, 3 );
a->addDependent( b );
b->connectOutput( c );
c->connectOutput( d );
```

Figure 2.9: Declaring a task graph using Cascade (Best, et al., 2009).

```
class C : public Task<int,int>
{
public:
   C(int batchSize, int numThreads) :
    Task<int,int>(batchSize, numThreads) {}

   void work_kernel(TaskInstance<int,int>* tI) {
    for(int i=0; i < tI->receivedBatch->size(); i++)
     tI->sendOutput(tI->receivedBatch->at(i)+1);
   }
};
```

Figure 2.10: A task with work kernel for Cascade (Best, et al., 2009).

2.5 TASK SCHEDULING

The problem of task scheduling on parallel processing environments has been thoroughly researched in general and many different scheduling algorithms have been developed. Some works support heterogeneous processing, while others only work in homogeneous processing environments. In both cases, only a few works exist specifically for game applications. Throughout the rest of this section, we use the terms ‘mapping’ and ‘scheduling’ interchangeably for the same meaning, that is, assigning a task to a processor in the system.

The works of (Joselli, Clua, Montenegro, Conci, & Pagliosa, 2008), (Joselli M. , et al., 2008), (Joselli M. , et al., 2009) and (Joselli M. , et al., 2010) are among those. The focus is on a few select types of tasks in games like physics simulation. In these works, a number of strategies for automatically distributing tasks between the CPU and GPU are proposed as follows:

1. **Starting**: this strategy executes the task and computes the time taken to finish a number of frames on the GPU and then the same number of frames on the CPU. With these times, it selects the fastest processor to process all the frames of the application.
2. **Cycle**: this strategy repeats the previous strategy every period of time during the application's lifetime.
3. **Starting full test**: this strategy repeats the starting strategy for the same task multiple times. In each iteration the value of one of the parameters of the task is increased and the timings are stored. Based on those computed times, the simulation can determine which processor is faster for a determined parameter value of the task.
4. **Adaptive**: this strategy runs with the starting strategy at the beginning, then after a period of time it starts checking how different the task's parameter values are since the start of the application. If they differ above a certain threshold then the "starting" strategy is repeated again.

Those strategies all revolve around executing and sampling the tasks on each of the processors for a certain period of time, and then continuing to use whichever processor is generally faster or has less work load (as measured by operating system API functions).

Qilin (Luk, Hong, & Kim, 2009) is a programming system that supports automatic adaptive mapping to map computations to processing elements on heterogeneous multiprocessors. The decision behind mapping a task to a certain processor is directed by a database that provides execution-time projection for all the programs Qilin has ever executed. The first time that a program is run under Qilin, it is used as the training run, where timings are measured under the different processors. The timings are then analyzed with curve fitting to build a projection function that can be used at run-time to determine the best processor type to execute a task depending on the task's input parameters size (Figure 2.11).

Harmony (Diamos & Yalamanchili, 2008) is a runtime system that relies on compute kernels that can be executed either in each kernel's local memory or in global memory. The former case is used to sample kernel performance on processors without affecting program data, while the second case is used for actual kernel execution. The kernel's performance is sampled with its data to make a decision on the best processor to use for execution. The sampled data is cached to avoid continuous kernel evaluation.

In all of the works above, data transfer times between processors is not taken into account. The Hybrid Remapper (Maheswaran & Siegel, 1998), the MATEHa algorithm (Giusti, Chichizola, Naiouf, & Giusti, 2008) and AMTHA algorithm (Giusti, Luque, Chichizola, Naiouf, & Giusti, 2009) on the other hand support a communication matrix that specifies data transfer times between the processors (read and write speeds). However, they do not support dynamically generated tasks, and require that conditional tasks and loops be embedded in larger tasks.

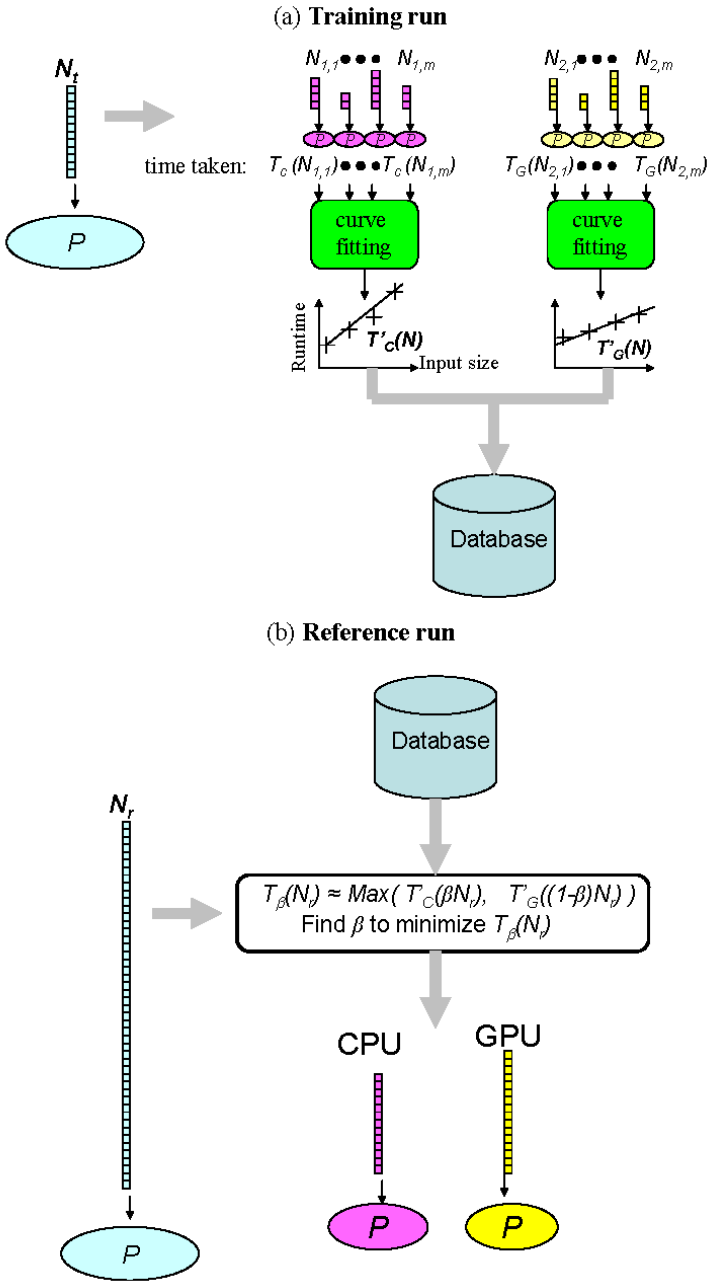


Figure 2.11: Qilin's adaptive mapping technique (Luk, Hong, & Kim, 2009)

The Hybrid Remapper (Maheswaran & Siegel, 1998) is a dynamic mapping algorithm that uses the run-time values that become available for task completion times and

machine availabilities during application execution time to conduct mapping decisions. It uses some results based on an initial static mapping in conjunction with information available only at execution time (hence the name).

MATEHa (short for Mapping Algorithm based on Task dependencies in Heterogeneous Architecture) (Giusti, Chichizola, Naiouf, & Giusti, 2008) is a mapping algorithm that works on DAGs representing tasks and their dependencies. It sections the DAG into "levels" that are then further processed to map tasks in each level to processors. The algorithm works in two steps: The first step determines the level of each graph node, and the second step consists in evaluating to which processor each graph task should be allocated (Figure 2.12).

AMTHA (short for Automatic Mapping Task on Heterogeneous Architectures) (Giusti, Luque, Chichizola, Naiouf, & Giusti, 2009) is an iterative static mapping algorithm that is applied to DAGs describing tasks and their dependencies. The mapping supports heterogeneous processing environments with heterogeneous communication times between the processors. Tasks in the DAG must carry timing estimations for each of the supported processor types. AMTHA then maps the tasks in the DAG one by one until none is left.

Procedure MATEHa (G(V,E))
{
  Calculate the level of each node corresponding to V of G.
  Allocate each task to a processor.
}

Procedure CalculateLevel (G)
{
  Given a graph G, the level of an LN(T) node is defined as the
    minimal number of tasks that should have started its
    execution for the tasks corresponding to node T to be
    started as well. The following formula expresses all that
    has been stated above:
  $LN(T) = \min_{Tin \in S} d(T_{in}, T)$
  where:
    S is the set of initial nodes (tasks that do not depend on any
        other task to start its execution), and
    $d(T_{in},T)$ corresponds to the minimal number of arches that
    have to be crossed from T_{in} to T.
}

Procedure AllocateTaskToProcessor (G)
{
  This algorithm starts from an initial list where tasks are
  ordered from lowest to highest LN(Ti)value, and carries out
  the following steps in order to obtain task allocation in the
  processors:
    a. Select the first level n without allocation.
    b. Calculate the maximal max_gain(Ti) gain for those
       tasks at level n that have not yet been allocated, together
       with the proc_opt(Ti processor to which the task should
       be allocated. Order the tasks in a decreasing manner
       according to max_gain(Ti).
    c. Allocate the first task Ti of the list generated in step b to
       the proc_opt(Ti) processor.
    d. If there are still tasks without allocation at level n, one
       should follow with step b.
    e. If there still remain levels without processing, one
      should go back to step a.
}

Figure 2.12: Procedures of the MATEHa algorithm (Giusti, Chichizola, Naiouf, & Giusti, 2008)

Tasks are first ranked according to a function that takes into account the time of the task and how many of its dependents are done, and the number of processors of each type in the environment. The task with the highest rank is selected for mapping in each iteration of the algorithm. The selection finds the processor that will result with the minimum time after executing the task in addition to its previous work load. Figure 2.13 lists the general algorithm steps.

Calculate rank for each task.
Whereas (not all tasks have been assigned)
1. Select the next task t to assign.
2. Chose the processor p to which task t should be assigned.
3. Assign task t (selected in step 1) to processor p (selected in step 2).
4. Update the rank of the tasks involved in step 3.

Figure 2.13: Pseudo-code with the basic steps of the AMTHA algorithm.

The SASH algorithm (short for Self-Adjusting Scheduling for Heterogeneous Systems) (Hamidzadeh, Lilja, & Atif, 1995) is a dynamic scheduling algorithm supporting tasks that are dynamically generated at arbitrary times during program execution. It utilizes a dedicated processor to execute the scheduling algorithm. SASH performs repeated scheduling phases in which it generates partial schedules. At the end of each scheduling phase, the scheduling processor places the tasks scheduled in that phase on to the working processors' local queues. Scheduling uses an on-line optimization technique that minimizes an execution cost function which takes into account task execution times as well as communication times in heterogeneous environments, and it shares some similarities with the selection function in AMTHA. The algorithm searches through a space of all possible partial and complete schedules represented by a tree. In each iteration the node with the lowest cost is expanded by extending the partial schedule with

an additional edge (a scheduled task). The expansions continue until all of the tasks have been scheduled or the scheduling phase uses its allowed time.

The work stealing algorithm (Blumofe & Leiserson, 1999) is a randomized scheduling algorithm that has been shown to give optimal results when run on homogeneous environments with negligible communication times, which is the case for multi-core systems with shared memory. The algorithm is simple. Each processor is first associated with a local work deque onto which tasks can be pushed. Tasks (or work) are initially assigned at random to any processor. A processor pops tasks from its deque one by one and executes them. When the processor's deque becomes empty, it enters work stealing mode and operates as follows. The processor becomes a *thief* and attempts to steal work from a *victim* processor chosen uniformly at random. The thief queries the deque of the victim, and if it is nonempty the thief removes and begins work on the top task. If the victim's ready deque is empty then the thief tries again by picking another victim at random. The algorithm is capable of scheduling dynamically generated task graphs. However, it does not support heterogeneous processing environments.

CellCilk (Werth, Schreier, & Philippsen, 2011) extends the Cilk language (Blumofe, et al., 1995) to work on the IBM CELL processor (PPU and SPUs) (Kahle, et al., 2005) by utilizing different work stealing strategies applied on different scheduling levels:

1. Nano: The scheduling level where the worker adds and pulls tasks from its local work queue and executes them.
2. Micro: The scheduling level used when stealing from another work of the same processor type.

3. Macro: The scheduling level at which tasks are transferred from one processor type to another.

CellCilk adds 5 new keywords to the Cilk language, which include `spu_spawn` in addition to the Cilk `spawn` keyword. The latter being used to spawn tasks for the SPU explicitly, as the original `spawn` keyword does not hide this detail from the programmer.

StarPU (Augonnet, Thibault, Namyst, & Wacrenier, 2011) offers a flexible plug-in based run-time system that supports scheduling dynamically generated tasks in heterogeneous processing environments. The system is targeted towards numerical kernel designers. It offers a high level library for transparently performing data movements between the different memory locations of system, in addition to a unified task execution model. StarPU can use one of multiple scheduling policies available to determine the best processor to be used for dynamically generated tasks. The paper lists 4 different scheduling policies (the Work Stealing algorithm being one of them). However, none of those policies seem to take into consideration data transfer times between processors in their decisions.

2.6 SUMMARY AND CONCLUSIONS

A number of works in the literature describing game application design and game hardware was reviewed. In addition, previous work on parallel design patterns and task scheduling algorithms and run-time systems for multi-processor environments was also discussed. In spite of the extensive work, it is clear that there is need for a specialized framework for parallelizing games which takes into account dynamic tasks, processor capabilities, data transfer and data locality.

CHAPTER 3 A NEW GAME PARALLELIZATION FRAMEWORK

3.1 INTRODUCTION

As discussed in Chapter 2, the growing complexity of game applications requires that these applications be able to make efficient use of their host environments. Since these environments offer parallel processing capabilities in the form of heterogeneous processors, it is important that games be able to execute their tasks on those processors to maximize performance. This cannot be automatically obtained from the classical code structure that games are usually built upon. Many games nowadays are built upon a *game engine*, which is a reusable collection of utilities and functionality common to a number of games. Many game engines exist with different capabilities and supporting different game types. Most of those engines have been written for serial execution. Converting a game engine to work in parallel is a difficult task that often requires refactoring code and data structures. It is thus important to consider reducing this effort as much as possible when suggestion a solution to parallelizing games. Of equal importance, if the solution was easy to use for programmers, then newly written games will also benefit from this feature. A design pattern is by definition a solution strategy that can be applied to problems of similar type. Since games share a similar code structure, it is desirable to find a design pattern that can address parallelizing this type of applications. Using this design pattern, the programmer should be able to express game tasks in such a way that maximizes parallelism between these tasks. Unfortunately, very little of the previous work on design patterns targets the game domain. Other works exist but mostly fail to

match the special requirements of game applications. Without fulfilling these requirements the game application cannot realize its full parallelization potential, and thus will not result in optimal performance.

Once the parallel game tasks have been generated, a run-time system must be able to schedule them to be executed on one of the processing elements available in the host environment. As has already been shown in section 2.3, the host environments onto which games are run nowadays offer heterogeneous processing capabilities. It is thus important to find a strategy for efficiently scheduling game tasks in a heterogeneous processing environment consisting of different processor types with different memory access policies and speeds. Again, research that targets the games domain in this area is limited. Other works fall short in one or more detail that would render the application of the solution inefficient.

In this research we propose a new game parallelization framework consisting of a parallel design pattern as the front end to simplify the game parallelization task for programmers and a game task scheduler at the back which automatically maps and assigns tasks to processors using a cost function that takes specific aspects of game applications into account and results in improved performance with better utilization of the heterogeneous computing resources available in the host environment. We describe these two major components of our framework in the rest of this chapter.

We split this chapter into two main sections. The first section presents a parallel design pattern for the game applications. It starts by highlighting properties of game applications that lead to deriving a set of requirements needed to maximize parallelization in this type

of applications. Then, a new design pattern fulfilling these requirements is developed. By applying the design pattern, a game application will have generated at run-time a set of tasks to be scheduled. The second section focuses on the design and implementation of a dynamic scheduler which can take tasks generated from the work in the first section and map them to processing elements in a heterogeneous environment in a manner that maximizes overall application performance.

3.2 SAYL DESIGN PATTERN

In this section, we first describe the major characteristics of computations in the games application domain. A directed acyclic graph (DAG), with a few additional properties to capture specifics of this domain, is used to represent these computations as tasks with dependencies among them. Possible solution strategies are discussed and then Sayl[1] is presented, a new domain-specific design pattern for parallel programming of games and simulations, with support for creation of dynamic task graphs. We describe a sample implementation which is then compared with some other works in this domain.

3.2.1 The Problem Domain

Games usually involve a variety of computations, including:

1. **Functional calculations**: Calculations which transform input data to output data of possibly different size and type. Tasks of this type can be functional (i.e., they have no side effects). However, it is also common in game applications that tasks of this

1 The name Sayl is the pronunciation of the Arabic word meaning “stream”. The name is inspired from the properties of a river stream, that is, a group of parallel threads of water flowing endlessly next to each other. This is similar to what results when using the Sayl design pattern: a stream of continuously-generating tasks executing in parallel threads.

type use additional acceleration structures to increase performance (e.g., cache). This invalidates the status of these tasks as having no side effects, and hence requires additional care when parallelizing such tasks. A typical example is world culling, which requires determining of game entities that fall within the view range of the 3D world camera. Another example is character bone update, which requires calculation of bone transforms for animated characters.

2. **State-based calculations**: Calculations which depend on continuous tracking and update of state. This is commonly associated with game entity updates (e.g., enemies, cars, etc.) which occur at every frame of the game. Data accessed in such tasks usually involves other objects (e.g., a character querying a nearby object for a certain value). An example is game AI, which is frequently implemented using finite state machines. Such tasks rely on evaluating current state to make a decision and possibly update that state.
3. **Data-retrieval**: Almost all game applications rely on graphics and sound as a medium of output to the user. This requires loading, generating, or gathering data from input sources for further processing. Such tasks usually do not involve complex computations, but rather a lot of interfacing with the operating system to request the data. One example is load resources - resources such as textures, 3D geometry, sounds, images, etc., are usually stored on a storage device, and must be loaded into memory before being used. Another example is read device - games require user interaction usually through a controller device connected to the computer.

From the above, it can be seen that a typical game application has wide diversity in the types of tasks that need to be carried out. Some of the tasks are data-dependent on other

tasks in order to progress. For example, game input is required to determine how to update the player's state, which in turn affects other aspects of the game. Generally, as a task executes it may produce multiple pieces of data at different times during its execution. These data may be required by other tasks. Ideally, for increased parallelism, the data should be available to the dependent tasks as soon as they are produced even if the originating task has not finished execution.

Furthermore, the set of tasks that a game executes might change depending on various conditions and logic. For example, consider a game that draws an explosion effect made of particles on each bullet hit. Here the number of explosions and number of particles change depending on how many bullets have been shot and what materials did they hit. If no bullets were shot, then no tasks for particle simulation need to be executed.

Based on the previous discussion, following is a list of the major characteristics of the tasks in a typical game application:

1. **Heterogeneity of tasks**: Game tasks perform different operations (e.g., particle simulation, character animation, path-finding, etc.). Additionally, there could be considerable variations in their completion times and functionalities.
2. **Data-flow type dependency**: Some tasks depend on data from other tasks before they can execute. This dependency between tasks can be represented by a connected directed acyclic graph (DAG), where each edge represents a dependency in the form of a data-flow (Figure 3.1).
3. **Dynamic set of active tasks**: The task graph repeats itself in each frame of the game for the entire life-time of the application. However a different subset of tasks may run

in different iterations, which is represented by the dashed arrows and nodes with crossed patterns in the task graph (Figure 3.1). In other words, the task graph is dynamic.

Our goal is to create a solution for specification and parallel execution of such a dynamic task graph with the game domain-specific characteristics listed above.

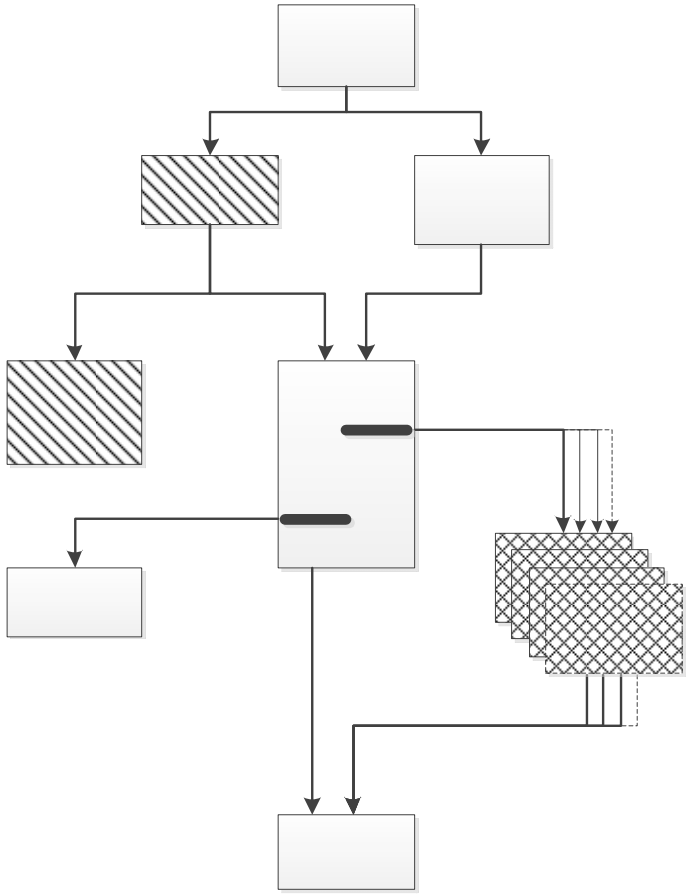


Figure 3.1: A directed acyclic graph representing tasks and their dependencies in a game application (boxes represent tasks, and arrows represent data flow dependency between tasks). Tasks hatched with a cross pattern illustrate that the actual number of tasks of that type is determined at runtime. The thick lines inside some task boxes represent points in time at which a certain piece of data is calculated and is ready to be sent to other tasks in the graph. Boxes that do not have outgoing arrows are output tasks (display to screen, audio output, etc.).

3.2.2 Solution Strategy

For the purpose of separation of concerns, we can identify three main components in the given problem: tasks, their dependencies, and scheduling. The game designer/programmer deals with the first two components. Thus, we will refer to these two components as the front-end, which is detailed in section 3.2.3. The scheduler component is the back-end (detailed in section 3.2.4) and the game designer/programmer is not directly involved in its design or programming. In essence, the front-end is concerned with generating tasks, while the back-end is concerned with how to execute them.

It may be noted that the work described in this thesis does not at present address automatic parallelization of existing sequential code. Hence, it is assumed that designers/programmers will manually decompose the given sequential algorithm into tasks of suitable granularity, and will identify their dependencies. As described next, their work is considerably simplified since creation of the dynamic task graph is implicitly embedded in the front-end pattern. A few of the computations may be more difficult to decompose than others. Data retrieval tasks for example may involve two-way communication between threads, which cannot be directly expressed using a DAG. Section 3.2.6 presents a method of addressing this situation.

3.2.3 Sayl Front-end

The Sayl Front-end component is based on the concept of invoking methods with eager parameter evaluation in imperative programming languages. This pattern lets the programmer model the tasks and their dependencies as method calls with parameters. By

doing so, the programmer has implicitly defined the tasks and their dependencies in a data-flow fashion. Parameters can be evaluated in parallel if no dependencies exist between them and then passed to the method requiring them. Once a task has received all of its parameters its execution is started as in eager parameter evaluation.

The above programming methodology implicitly results in a dynamic task graph. Execution of a task may result in generation of other tasks, as shown in a following subsection. The back-end component discussed next gives strategies for executing these tasks efficiently.

3.2.4 Sayl Back-end

It is the job of the back-end component to take the tasks generated by Sayl front-end and schedule them for execution as soon as their parameters become available. Any task with two or more parameters is stored in a prepare container until all its inputs have been fulfilled, after which the task is moved to the ready container. Tasks with at most one parameter are stored directly in the ready container. Thus, the ready container contains only tasks which are ready for execution (Figure 3.2 and Figure 3.3) in an order independent fashion. The dependency requirement is maintained by the fact that only methods which have their parameters fulfilled are scheduled for execution.

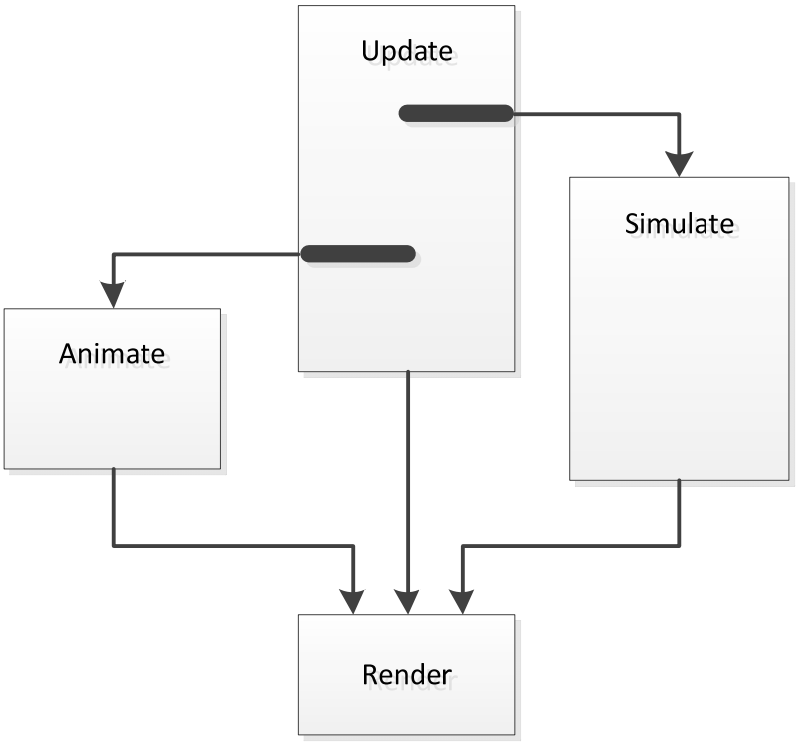


Figure 3.2: A graph representing methods called in a hypothetical game application.

In addition to these two containers, the back end also has a pool of continuously running threads, called worker threads. The number of worker threads should match the number of available processing elements so that each thread runs on a processing element without context switching overhead. A worker thread removes a task from the ready container, executes it to completion, and then looks for another task. Any tasks generated by the executing task are put back into appropriate containers.

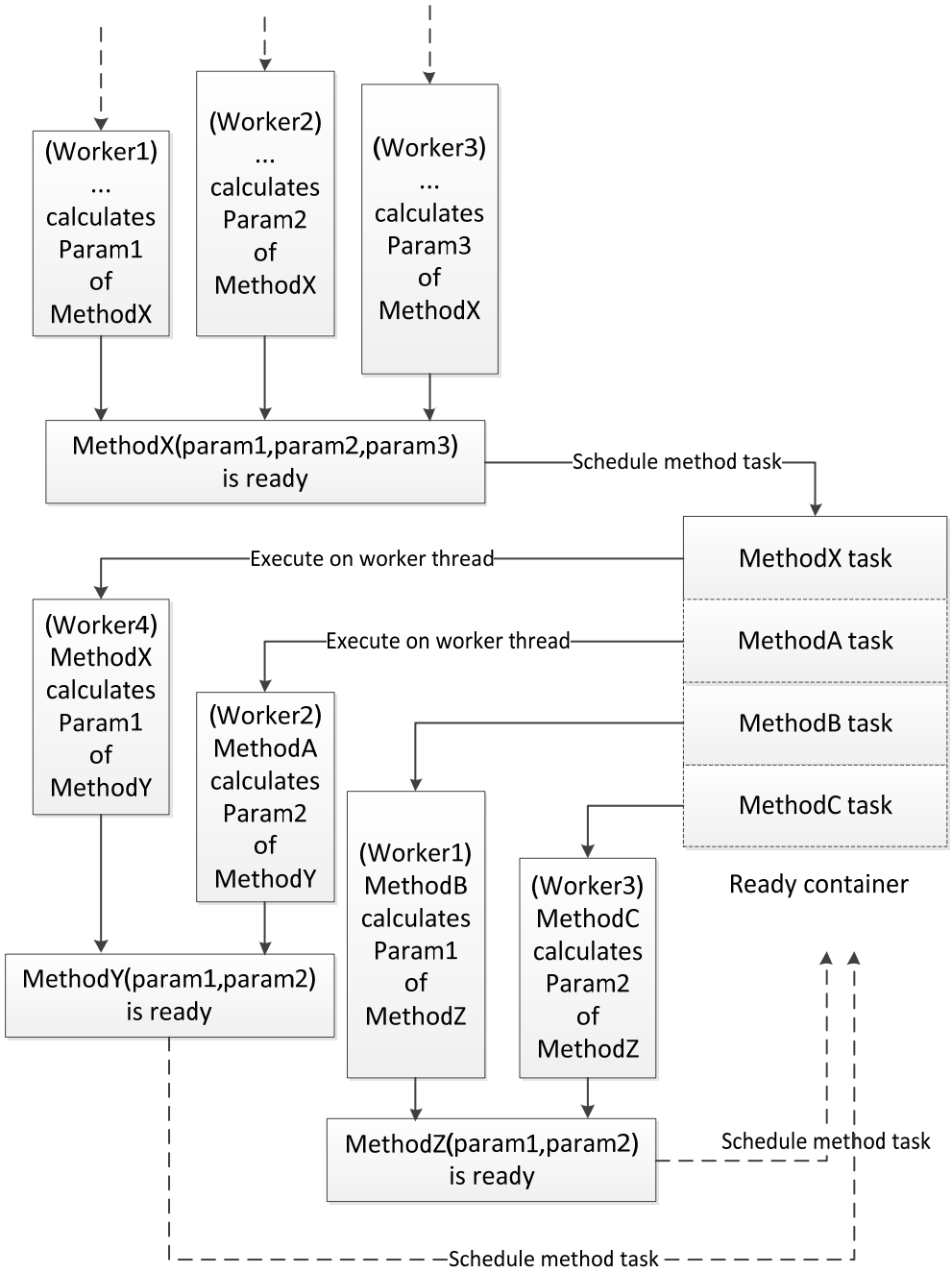


Figure 3.3: Run-time interaction diagram in the back-end. A number of worker threads consume tasks from the ready container. Upon execution, the task (method) calculates the value of one (or more) parameters of another method. When the second method's parameters are fulfilled, it is inserted into the ready container to be picked up for execution by a worker thread.

The order of queuing or executing tasks in the ready container is irrelevant to the correctness of the program. This flexibility allows the implementation to utilize any container design which is found efficient to achieve high performance in task scheduling or to fulfill additional task requirements. For example: if the tasks in the ready container have priorities, then an implementation using binary heap based priority queues may be quite efficient. Figure 3.3 illustrates the run-time interaction in the back-end of the Sayl pattern.

3.2.5 Implementation

A sample implementation of the Sayl pattern in C++ is described in the following. The front-end interfaces with the back-end via APIs (see Figure 3.4).

Starting with the back-end, each `WorkerThread` object has two member fields: an event named `Wake`, and a Boolean variable named `Exit`. The `ReadyTasks` container holds tasks pending execution. The `PreparingTasks` container holds tasks that are not ready as some of their parameters are not yet available. `IdleWorkers` is a container storing idle worker threads at any time during execution.

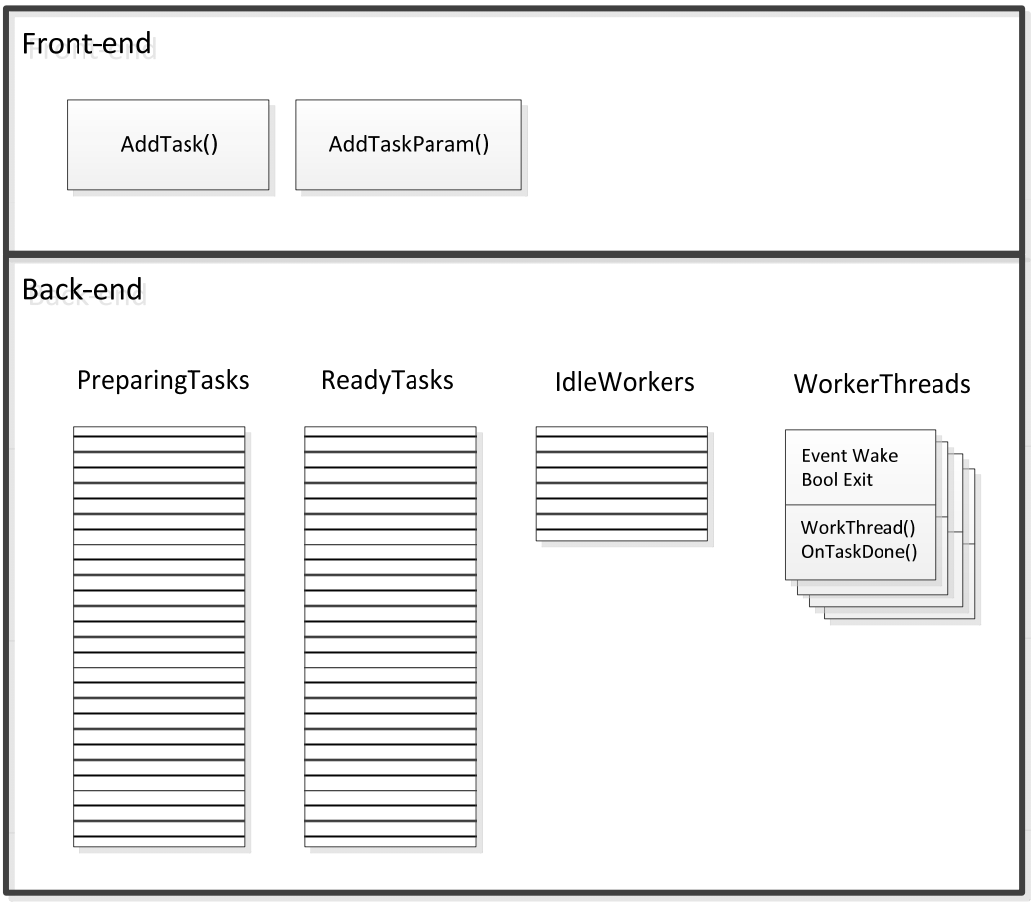


Figure 3.4: Pattern interface APIs and implementation components.

The code in Figure 3.5 below shows the two main methods driving the back-end component of the pattern.

The `WorkerThread` is a thread running the internal method `WorkThread`. This method is a simple loop that continues while the method `OnTaskDone` returns `true`. `OnTaskDone` checks the `ReadyTasks` container to see if the thread can pick a new task to execute. If no tasks are found in the container, then the thread adds itself to the `IdleWorkers` container and sleeps waiting for its `Wake` event to be signaled. The method returns `false` when thread termination is determined.

```
internal method WorkThread()
{
  while (OnTaskDone()) {};
}

internal method OnTaskDone()
{
  Task TaskToExecute = ReadyTasks.PopAny();
  if (TaskToExecute != null)
  {
    TaskToExecute.Execute();
    return this.Exit;
  }
  int IdleWorkersCount = IdleWorkers.Push(this);
  if (IdleWorkersCount == WorkerThreadsCount)
  {
    while (true)
    {
      Thread IdleWorker = IdleWorkers.PopAny();
      if (IdleWorker == null)
        break;
      IdleWorker.Exit = true;
      SignalEvent(IdleWorker.Wake);
    }
    return false;
  }
  else SleepUntilSignalled(this.Wake);
  return this.Exit;
}
```

Figure 3.5: Sayl methods that drive the back-end.

The front-end of the pattern is implemented by exposing the only two public methods the user needs to deal with, namely `AddTask` and `AddTaskParam` (Figure 3.4). Figure 3.6 below shows the implementation for both.

```
public method AddTask(TaskMethod, optional Param)
{
  ReadyTasks.Push(TaskMethod,Param);
  Thread WorkerThreadObj = IdleWorkers.PopAny();
  if (WorkerThreadObj != null)
    SignalEvent(WorkerThreadObj.Wake);
}

public method AddTaskParam(
    TaskMethod,
    TaskSubID,ParamsCount,
    ParamID,ParamValue)
{
  EnterCriticalSection(PreparingTasks)
  TaskParamsEntry TaskParams =
      PreparingTasks.Find(TaskMethod,TaskSubID);

  if (TaskParams == null)
    TaskParams = PreparingTasks.AddNew(
                      TaskMethod,ParamsCount);

  Assert(TaskParams.ParamsCount == ParamsCount);
  TaskParams.SetParam(ParamID,ParamValue)
  bool TaskReady =
      (TaskParams.Count == ParamsCount);

  if (TaskReady == true)
    PreparingTasks.Remove(TaskParams);

  ExitCriticalSection(PreparingTasks)

  ReadyTasks.Push(TaskMethod,
                  TaskParams.ParamsList);
  Thread WorkerThreadObj = IdleWorkers.PopAny();
  if (WorkerThreadObj != null)
    SignalEvent(WorkerThreadObj.Wake);
}
```

Figure 3.6: Pseudo-code implementing front-end methods of Sayl.

`AddTask` adds a new task to the `ReadyTasks` container and wakes an idle thread to execute it if available. `AddTaskParam` is a more powerful version of `AddTask` that allows passing parameters to a task that takes two or more parameters, which could be

generated by different tasks. It requires the user to specify the total number of parameters the task needs, and an ID for the task to uniquely identify it from other instances if they may exist. Such tasks are added to the `PreparingTasks` container. Each call to `AddTaskParam` passes the value of one of the parameters to the task. When all parameters have been provided, the task is added to the `ReadyTasks` container. Until then the task remains in the `PreparingTasks` container.

Using just these two API calls, Sayl users can express tasks and their dependencies with the guarantee that these dependencies will be correctly preserved.

The code in Figure 3.7 below shows an example of how Sayl-front-end can be used to express an execution graph for a small part of a hypothetical game application as shown in Figure 3.2.

From the previous example, it can be seen that using the Sayl pattern, spawning tasks to execute methods in parallel is a simple transition from writing sequential code. This quality is very important as it contributes significantly to the ease-of-use aspect of this pattern. At the same time, it shows the fundamental differences with traditional fork-join and similar approaches (e.g., the `Render` method is data-flow dependent on the other methods).

```
public method Update()
{
  ... /* Update code */
  AddTask(Simulate,Player)
  ... /* Update code */
  AddTask(Animate,PlayerModel)
  ... /* Update code */
  AddTaskParam(
      Render,FrameIndex,3,Param_Camera,Camera);
}

public method Simulate(PlayerToSimulate)
{
  ... /* Simulation code */
  AddTaskParam(Render,FrameIndex,3,
               Param_SimPlayer,PlayerToSimulate);
}

public method Animate(PlayerToAnimate)
{
  ... /* Animation code */
  AddTaskParam(Render,FrameIndex,3,
               Param_AnimModel,PlayerToAnimate);
}

public method Render(Camera,
                     SimPlayer,PlayerModel)
{
  ... /* Rendering code */
}
```

Figure 3.7: Pseudo-code using Sayl to represent an execution graph for a small part of a hypothetical game application.

One of the commonly found tasks in games is data retrieval, which could require two-way communication with other tasks. Games often run a secondary resource loading thread in addition to the main game loop thread. The loading thread receives commands to load particular resources and executes them without blocking the game loop while the data is being transferred. In each game loop iteration (represented by a colored block in

Figure 3.8 left), the requested resources are polled for completion. Using Sayl, it is possible to achieve the same functionality without a secondary resource loading thread as follows: when a task determines that a resource needs to be loaded, it spawns a new task and passes it the resource identifier and the identifier of the task that requires the resource. The new task (represented by a hashed box in Figure 3.8 right) then loads the resource when it runs. When done, it simply passes the loaded resource as a parameter to the specified task.

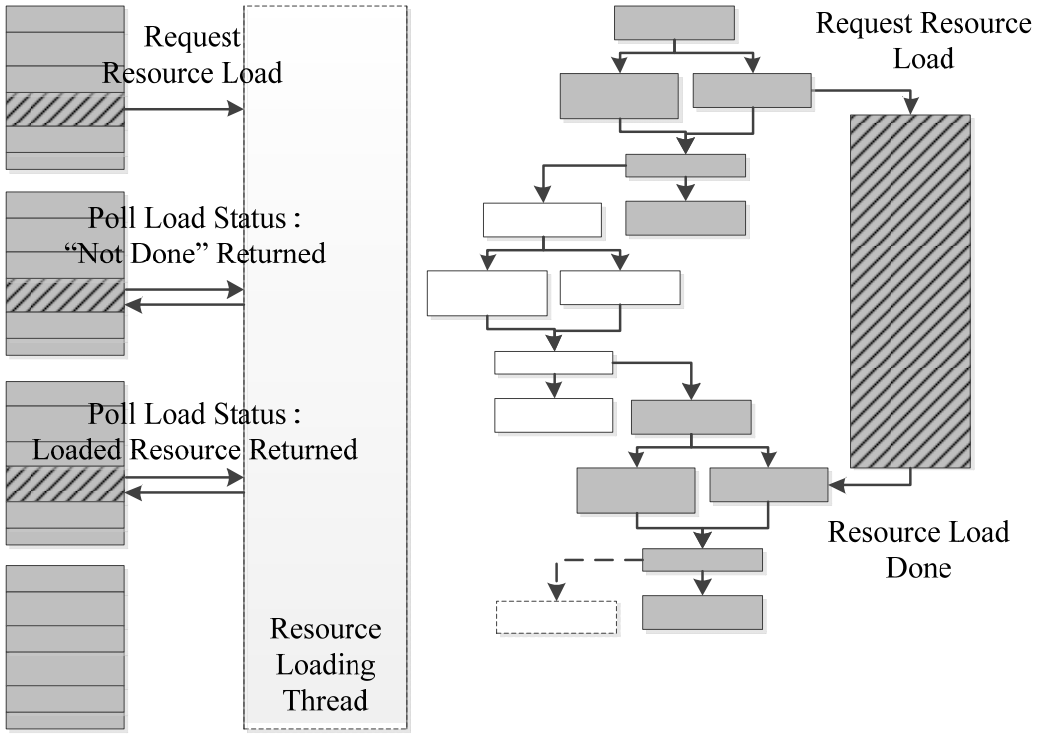


Figure 3.8: Left: Multiple iterations of a hypothetical game loop, in which there is a resource load request being communicated with a resource loading thread. Right: The same game loop after applying the Sayl design pattern. The resource load request becomes spawning a task that loads the resource and passes it to the task that needs it. Each group of solid colored tasks represent tasks belonging to a single game loop iteration.

3.2.6 Discussion

The Sayl pattern has a number of advantages:

1. A task is only spawned when its dependencies are fulfilled. For game applications, this is more efficient than spawning tasks that keep polling to check their dependencies for completion. This way precious processing power is not wasted.
2. Tasks in Sayl are only allocated in memory when they are preparing to be executed (i.e., on demand); whereas in some other patterns (e.g., Cascade and Task Graph) tasks are allocated up-front and they wait until their dependencies are fulfilled before they become ready to execute. Thus, Sayl implementations should be able to utilize memory for storing tasks at run-time more efficiently than some of the other patterns.
3. In Sayl, the programmer has implicitly defined the tasks and their dependencies by writing the program using methods and parameters. Consequently the programmer need not explicitly declare dependencies for each task as in the case of Task Graph and Cascade. This is what makes programming using this pattern rather simple.
4. Sayl can also help convert existing game code bases into parallel versions with greater ease, thus motivating increased code reuse.
5. Not mandating any order of execution for the tasks in the back-end opens more opportunities for choosing the appropriate type of task container.

For game applications, supporting dynamic task graphs makes a difference versus statically defined task graphs. For example, Cascade is a parallel programming environment (PPE) specifically addressing the games domain. While it has support for task heterogeneity and task dependencies, it lacks the support for dynamic task graphs.

The Sayl pattern, on the other hand, enables the programmer to build the task graph dynamically by spawning tasks only on demand by the executing task(s).

In any static task graph-based approach, the programmer must define all tasks and their dependencies beforehand and the graph has to be fully constructed prior to its execution. Typically, it only represents a portion of the application's lifetime (e.g., one frame because subsequent frames may require modifications to the task graph). The program therefore has to wait for the current static task graph to finish before executing the next one, even if it were possible to execute some tasks from the current graph in parallel with tasks from the next graph (e.g., pure output tasks). On the other hand, the dynamic task graph is capable of implicitly representing all the frames in the entire lifetime of a game.

Figure 3.9 illustrates how overlapped executions among frames are possible using a dynamic task graph. As shown, tasks from the next frame are invoked by tasks from the current frame as the execution proceeds, and hence task executions are not bounded by frame boundaries. The same is not possible when static task graphs are used.

Moreover, the realization of the static task graph requires allocating it entirely in memory prior to executing it. More specifically, using Cascade, the programmer builds the task graph in a three-step process as follows:

1. Define a class for each type of task. The class contains the implementation of the task's method, in addition to implementation of an interface to define the inputs and outputs of the task.
2. Define instances of the tasks in the task graph.
3. Specify dependencies between the tasks instances.

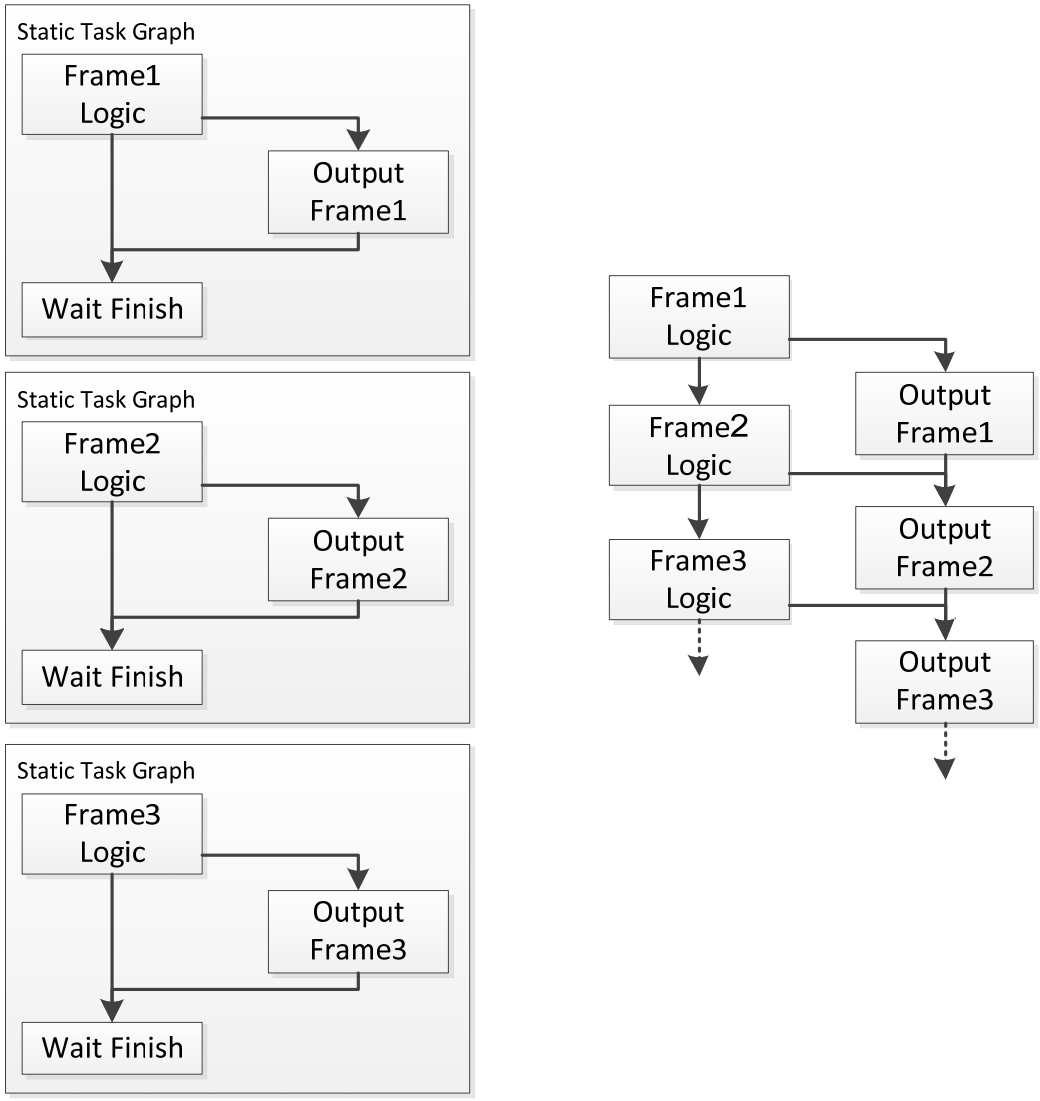


Figure 3.9: Task graphs representing methods called in a hypothetical game application. Left: Multiple static task graphs, where each instance computes one frame of the game. Right: The dynamic task graph is capable of running the logic for the next frame while output of the current frame is being processed in parallel.

The above 3 steps incur significant overhead to the programmer in terms of programming efforts required. On the other hand, defining the task graph using Sayl involves writing the task's method along with specifying its parameter list. Within the method, parameters are passed to other task(s) (method(s)) as soon as they are computed. Sayl extracts

dependency information directly from methods and their parameters. The net outcome is that Sayl involves less programmer work in writing parallel game applications.

In practice, applying a methodology like Cascade or Task Graph involves even writing more code to build the static task graph. Building the graph involves decisions made by different modules of code (e.g., input, AI, rendering). If the graph is built by a single method then that method must have access to all game modules, which can lead to software engineering problems such as reduced modularity or encapsulation. An alternative way is to have each game module expose a method which assesses the module's internal state, generates the tasks needed and adds them to the task graph. This would solve the software engineering problem mentioned before, but adds more programmer work to write these methods. These problems do not exist in Sayl. Task methods do their work and pass parameters only to methods requiring them. Thus they do not require global visibility to all game modules.

So far, the pattern has not made any specifications particular to heterogeneous processing. We leave this detail for the actual implementation of the pattern for a particular game application. As has been highlighted, the back-end of the pattern should be flexible enough to support an implementation that schedules tasks in a heterogeneous processing environment. Thus, applying Sayl for heterogeneous processing should not require any major modifications to the design pattern. This will be shown in the next section.

3.3 ARBITER WORK STEALING SCHEDULER

In this section we describe the Arbiter Work Stealing (AWS) scheduler, a back-end implementation of the Sayl design pattern. The scheduler makes use of the basic work stealing algorithm and adds new concepts to it to support heterogeneous processing environments efficiently.

3.3.1 The Problem Domain

From section 2.3 it can be noted that varied processing capability and varied data transfer times between processors are standard characteristics of environments in which game applications are executed today.

Efficient use of the different types of processors available in the hardware environment requires knowledge of the capabilities of each processor and the kinds of computations it excels. For example, the CPU is better than the GPU and the SPU in integer math and branching logic, while the GPU is efficient at large floating-point data-parallel computations. The SPU is also the best at floating-point vector computations on streamed data. A task scheduler must have this knowledge in order to make appropriate decisions when assigning tasks to processors. Tasks might be able to run on a particular processor type only, or might run on several types but with different performance characteristics. It is also important to note that task data might reside in a certain memory location that cannot be efficiently accessed by another processor on which the task has been scheduled, in which case the time of data transfer must also be taken into account when making the decision.

Our goal is to design a scheduler that can map dynamically generated task graphs to processors in a heterogeneous environment by taking into account the capabilities, speed, and memory access specifications of each processor in the system in order to minimize overall execution time of a game application. Arbiter Work Stealing is our scheduler for performing this job. First, we describe the high level concepts introduced in the scheduler, followed by details of the new aspects in its method.

3.3.2 AWS scheduler basic concepts

The processors in the heterogeneous system are first categorized into distinct homogeneous groups, where a homogeneous group contains processors only of the same type, and which can communicate among each other relatively faster than with processors in other groups (for example, via shared memory access or a fast data bus).

Processors in a single homogeneous group run the classical work stealing algorithm as-is. That is, each processor has a work queue. Once tasks from the processor's work queue are done, it attempts to steal tasks from other processors in the same homogeneous group. The only exception being that newly generated tasks are not added to a local processor's work queue directly, but instead they are sent to an Arbiter module.

The Arbiter module takes on the job of mapping each new task to a homogeneous group by considering the conditions for the task and its data. When the arbiter decides on the "best" homogeneous group for the task, it adds the task to the local task queue of a randomly chosen processor in that homogeneous group. The Arbiter's decision is based on several important considerations including data-transfer and communication times between processors as will be detailed in the following sub-section.

3.3.3 AWS scheduler details

There are three important aspects of how AWS works. These are the *Homogeneous Grouping*, *Tasks* and the *Arbiter*.

3.3.3.1 Homogeneous Grouping

The concept of homogeneous groups allows AWS to leverage the simplicity and efficiency benefits of the work stealing algorithm without major modifications. Since all processors in the homogeneous group are of the same type and have low communication costs, the two assumptions made in the original work stealing algorithm are satisfied. Whenever one of those two assumptions is violated then the processors involved should be separated into different homogeneous groups. The effect of this separation will be evident in the following subsections.

Each homogeneous group keeps track of its *total estimated work amount*. The total estimated work amount is the sum of *estimated work amount of each task* in the task queues of the processors in the homogeneous group. This estimation is described in the following. As tasks are added or executed in the group, the total estimated value is updated accordingly.

3.3.3.2 The Tasks

Tasks are executable units of code that perform operations on optional input data to produce new data or modify system state. AWS requires that with each task, three pieces of information be supplied. These will be used in decision making by the algorithm.

First, the processor types on which a task is able to run must be specified. This implies that an implementation for each of those processor types is available. In some cases, an implementation written only once can run on different processor types if the programming language supports it (e.g., C++ AMP code can be compiled to run on the CPU as well as the GPU on PC systems).

Second, data locality is to be taken into consideration. For this, the memory location of each piece of input data and its size are required. As we have shown in section 2, different processors have access to different memory locations even if those processors reside on the same machine. If a task is to be run on a processor different from where the task data resides, then data transfer might be first required. AWS takes into account the cost of this data-transfer operation.

The third important piece of information is the time taken by a particular task to finish on each of the supported processor types. This information can be either calculated at run-time by an estimation function or gathered via a run-time sampling session and then used in later runs. In both cases, the estimation must be given using a uniform time unit (e.g., milliseconds) because the Arbiter uses this information to compare estimates of different tasks.

3.3.3.3 The Arbiter

The Arbiter module takes the largest role in decision making in AWS. The Arbiter periodically samples the total estimated work amount from each of the homogeneous groups. When the Arbiter receives a new task that has become ready for execution, it invokes a decision function that takes into account the following information:

1. The sampled total estimated work load for each homogeneous group.
2. The estimated time of the task at hand on each of the supported processor types.
3. The time required to transfer task input data to a processor in each of the homogeneous groups.
4. The number of processors in each homogeneous group.

The decision function then selects a homogeneous group as a target for running the task. The selection attempts to minimize the overall running time in a greedy fashion by utilizing the above mentioned information to rank the available homogeneous groups depending on:

1. How fast a processor of that homogeneous group can finish the task.
2. How much additional time is needed to transfer input data for access by a processor of that homogeneous group.
3. Availability of processing capacity in the homogeneous group. This is a relation between the current work load and the number of processors in the homogeneous group.

The following function is used to estimate the running time of each homogeneous group when adding the task to that group:

Equation 3.1

$$run\ time_p = task\ time_p + input\ data\ transfer\ time_p + \frac{group\ total\ work\ time_p}{processor\ count\ in\ group\ p}$$

Where:

- $run\ time_p$ is the total estimated run time for a homogeneous group p after adding the task to it.
- $task\ time_p$ is the estimated run time for the task on a single processor in the homogeneous group p.
- $input\ data\ transfer\ time_p$ is the total time required to transfer input data from its current location to a location accessible by a processors in the homogeneous group p.
- $group\ total\ work\ time_p$ is the total estimated work time on homogeneous group p before adding the task to the group.

The Arbiter then picks the homogeneous group p with the minimum total run time value calculated and assigns the task to a randomly chosen processor in that group. Subsequently, the work stealing algorithm will pick up the task and execute it on one of the processors in the group. Figure 3.10 illustrates the components involved in AWS.

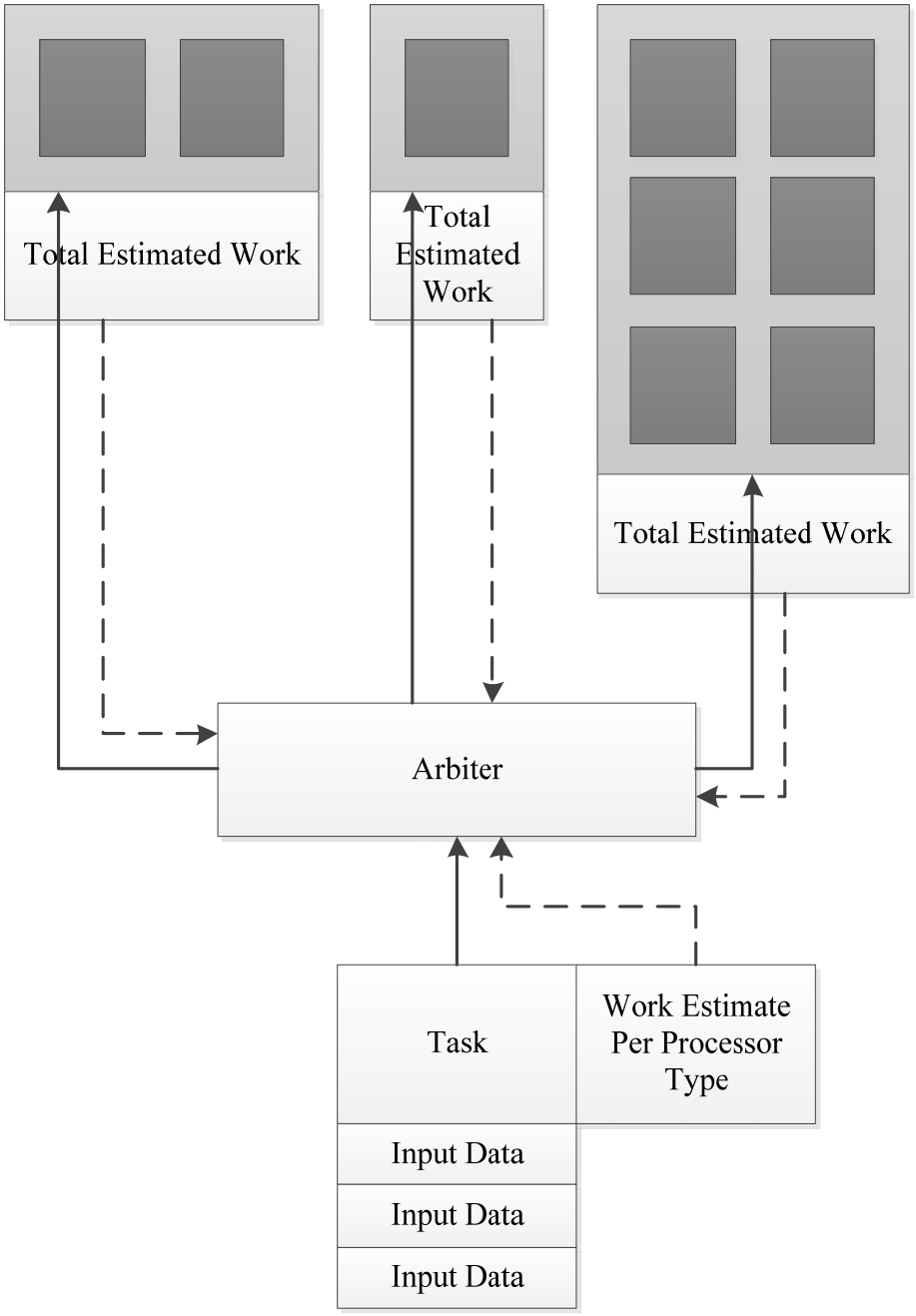


Figure 3.10: An illustration of the conceptual components involved in AWS algorithm. The light gray blocks are processor groups, which contain one or more processors depicted by dark gray blocks. Solid arrows represent "submission" communication, and dashed arrows represent information flowing towards the Arbiter for decision making.

3.3.4 Interface

As it has been discussed in Section 3.3.3, for each task to be executed by AWS there is some small required information that must be supplied by the programmer in addition to the actual task code. This information is supplied to AWS through a programmer-scheduler interface. The code in Figure 3.11 the following illustrates such an interface:

```
public struct TaskData
{
   byte[] data;
   int dataSize;
   int dataLocation;
}

public struct Task
{
   function taskCode;
   function estimator;
   TaskData[] inputData;
   int inputDataCount;
   bit_field supportedProcessors;
}
```

Figure 3.11: Data structures representing the interface to AWS.

The `dataLocation` field in the `TaskData` structure specifies the memory location of the data. For example, a value of 0 indicates CPU RAM and 1 indicates GPU VRAM. The `supportedProcessors` field in the `Task` structure is a bit field where each bit represents a processor type (e.g., bit 0 represents the CPU, bit 1 represents the GPU and bit 2 represents the SPU, etc.). The bit respective to each supported processor type must be raised.

After filling the `Task` structure, the programmer hands it to the Arbiter, after which no further input from the programmer is needed.

Since AWS is used as a back-end implementation for Sayl, then the interface information mentioned above will be prepared by Sayl when all task data becomes ready for execution. The information is then submitted to AWS for execution. Some of this information must be given by the programmer for each task, thus it must be exposed through the front-end of Sayl. This data is:

- Methods that implements the task for each processor type supported by the task.
- The memory location of task parameters.
- A method that provides an estimation of the task's execution time on each of the supported processors.

This information can be easily passed to Sayl through the same front-end APIs described in section 3.2.5 by extending the parameters received by these APIs to take the additional information required for heterogeneous processing support.

3.3.5 Discussion

The AWS algorithm builds on the classical work stealing algorithm by adding one level above it that "manages" multiple running instances of the work stealing algorithm. The Arbiter only handles high level processor group assignment, and leaves the details of scheduling the task to a particular processor in the group to the base level work stealing algorithm. The Arbiter takes some time to make its decision. This time grows linearly with the number of processor groups in the system. Thus, it is important to avoid overly dividing the system's processors. For example, creating a processor group for each core of a CPU can result in poor Arbiter performance as task mapping overhead becomes significant. On the other hand, merging processors of heterogeneous types or

communication costs under one processor group will result in bad performance in the base work stealing algorithm, and an overall poor application performance as time is wasted in randomly stealing works to processors of probably weaker capabilities for executing a particular task.

One example of good grouping on the Sony Playstation3 platform is to have one processor group for the PPU, one processor group for all 6 SPUs and one processor group for the GPU. Another example of good grouping on a multicore PC with 2 graphics cards is to have all CPU cores in one processor group, and each GPU in its own processor group. This is because each GPU has its own local VRAM which cannot be directly accessed by the other GPU. Thus, the Arbiter must be aware of this case so that it takes data transfer times into account properly.

Note that some processor group might include only one processor in it. In this case, it is pointless to run the base work stealing algorithm on this processor. Instead, only a simple queue consumption algorithm is needed in that case.

Task run time estimation is one important aspect of AWS. Through this model, a programmer can “skew” execution of tasks to certain processor groups. Accurate timing is not really required however. In the general case, it is enough to just maintain proper timing relationships in task estimations. If the estimation is artificially skewed towards a certain processor group, then the general performance may degrade even though the skewing was done intentionally to occupy idle processors in the system. The goal of AWS is not to keep all processors in the system occupied, but to maximize performance. If this strategy resulted in processors idling for a considerable amount of time, then this

can be seen as an opportunity to expand the game application with more features or higher quality computations that can be executed on those idle processors, and thus make efficient use of the full potential of the system.

3.4 SUMMARY AND CONCLUSIONS

In this chapter, we presented a new game parallelization framework consisting of a parallel design pattern in the front end and a dynamic task scheduler at the back end. In the first section we presented a domain specific parallel design pattern called Sayl for game applications. By highlighting properties of game applications it was possible to derive a set of requirements needed to maximize parallelization for game applications. The proposed design pattern was built to fulfill these requirements. Additionally, it was proposed that Sayl makes it easy for programmers to parallelize existing code or write new code. Further, through its support for dynamic task graphs, it provides improved run-time performance as compared to competing parallel programming approaches for this specific domain. By applying the Sayl design pattern, a game application will have generated at run-time a set of tasks to be scheduled.

The second section focuses on the design and implementation of a domain-specific dynamic scheduler which can take tasks generated from the work in the first section and map them to processing elements in a heterogeneous environment in a manner that maximizes overall application performance. The claim was that the Arbiter Work Stealing (AWS) scheduler is capable of efficiently scheduling tasks on a heterogeneous environment because it takes not only processor capabilities into account, but also the data locality and communication costs. Using this information, AWS manages to

schedule tasks to different processor types in the system, which were arranged into homogeneous groups, with the goal of maximizing application performance. The scheduler requires specifying an estimated cost model for tasks, which was used to guide the decisions of the Arbiter.

CHAPTER 4 CASE STUDY

4.1 INTRODUCTION

The work in this chapter takes up a case study of converting the serial code of a simple game into a parallel version by using the game parallelization framework in Chapter 3 and analyzes the effect of this on the game's code and performance aspects.

This chapter is arranged as the following: Section 4.2 introduces the sample game application and the host environment details, and lists the development tools and technologies that are used to apply the framework's methodology. Section 4.3 conducts the various tests on the sample game, including modifications following applying Sayl and AWS to the game code to leverage heterogeneous processing capabilities. Section 4.4 carries a discussion and analysis of the different tests' results. Finally, Section 4.5 presents summary and conclusions.

4.2 TEST SETUP DETAILS

The sample game is a space war game which involves the player controlling a space ship that fires shots against a large number of particles. The particles attempt to group together in a certain form, and the player must prevent them from doing so by shooting them and pushing them away from converging (Figure 4.1 and Figure 4.2).

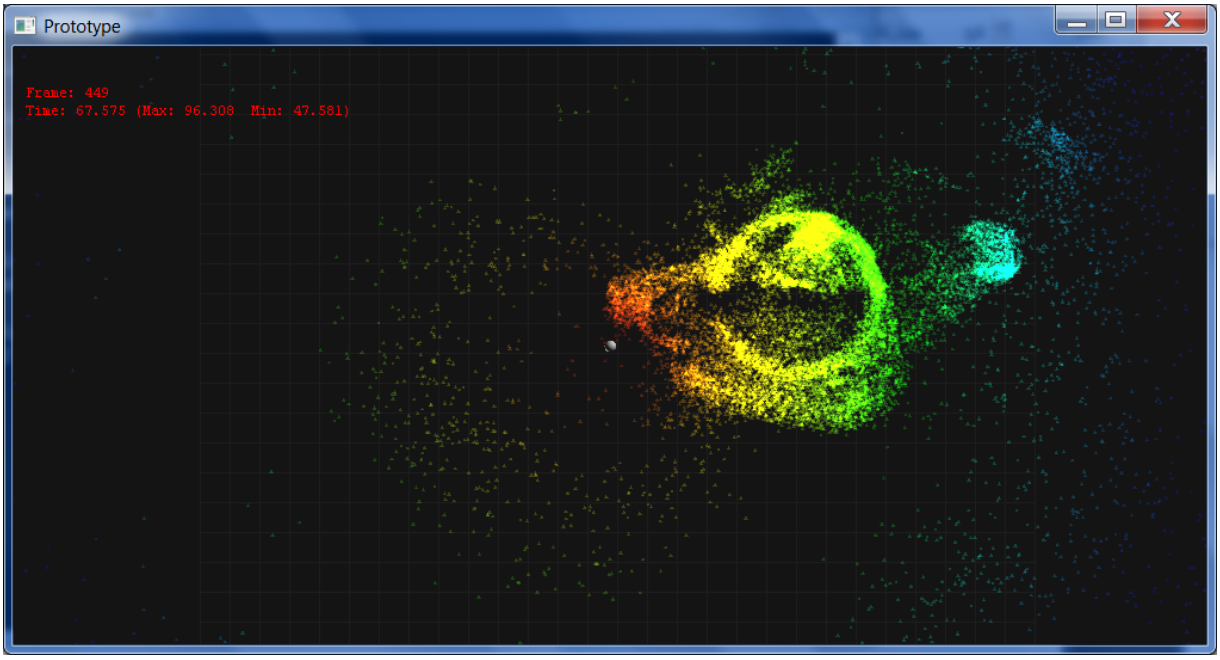


Figure 4.1: Screen shot from the sample game application.

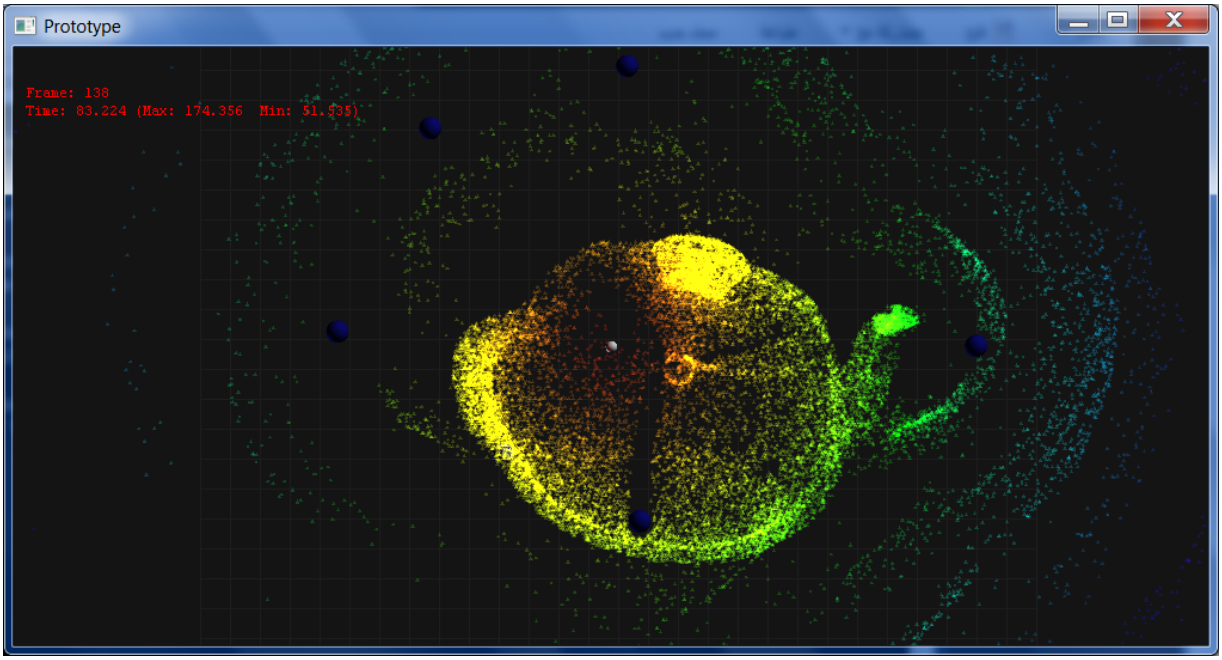


Figure 4.2: Screen shot from the sample game application, showing the player shooting to distort particles.

The game code is written with the C++ language (Stroustrup, 1997) and is compiled to an executable binary using Microsoft Visual Studio® 2010 C++ compiler. To save time, the code relies on a few libraries to do certain tasks.

The Microsoft Direct3D® 9 API (Microsoft Corp., 2012) is used for communicating with the GPU and executing rendering commands. Direct3D 9 exposes the third generation of programmable vertex shaders and pixel shaders. These offer flexible rendering capabilities in addition to being able to execute a certain class of GPGPU computations thanks to the support of floating-point render target textures.

The XInput Game Controller API (Microsoft Corp., 2012) enables applications to receive input from the Xbox 360 Controller for Windows. It is used to gather input from the player in addition to the keyboard device, which is sampled via the `GetAsyncKeyState` API from the Win32 SDK to support two methods of input for the player.

Sayl's C++ implementation is in the form of a library called Sayl-Lib. The library provides its users with a class which implements the back-end part of Sayl, and exposes the two public methods `AddTask` and `AddTaskParam` to implement the front-end part of Sayl. It also provides a few pre-processor macros that further facilitate interacting with the library (see Appendix A). Embedded within Sayl-Lib is an implementation of the Arbiter Work Stealing algorithm as a scheduling back-end to the library.

The sequential code of the sample game is written in a classical way, i.e., a game loop, which contains calls to the different game components in a serial manner; feeding the output of one component to the next. The major components in the game are:

- *Reading user input*: sampling a few keys from the keyboard, and the state of an Xbox 360 Controller, and converting the values into game actions.
- *Updating the player's state*: taking the game actions from the previous component and applying it to modify the player's speed and position, and shooting new bullets and updating previous ones' speed and position.
- *Simulating the particles*: calculating position and color of a large number of particles in response to bullets and other forces.
- *Rendering the 3D scene*: uploading simulation data to the GPU for rendering, and issuing draw commands to present the scene.

The presence of these major game components in the sample game is important to make the test representative of how applying the methodology would affect a real-world game application. The type of tasks and the dependencies between them in this sample game highlight the capabilities of Sayl in comparison to other works. Also, not all of the tasks above can run on all processor types. For example, reading user input and updating the player's state can only be done on the CPU due to system API calls. However, particle simulation can run on the CPU and the GPU, while 3D scene rendering can only run on the GPU. Such setup represents the cases which the Arbiter Work Stealing scheduler might face in a game application.

For parallelization of this game, some algorithmic changes were first required in the particle simulation code. These were modifications to compute separate chunks of particles in each task (see Figure 4.3).

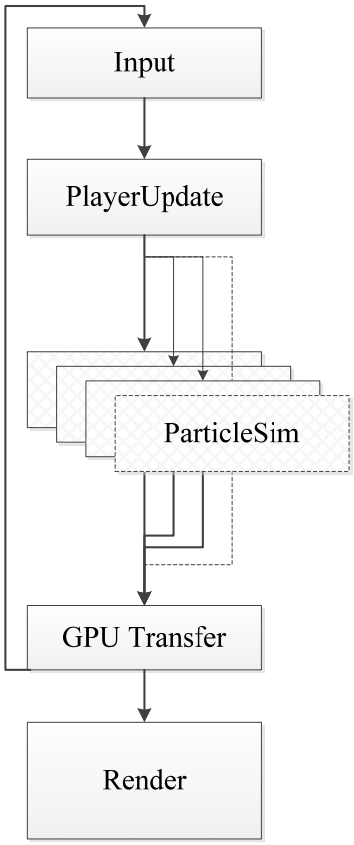


Figure 4.3: Dependency graph of the sample game application.

After simulation of each of those chunks, the data must be sent to the GPU VRAM for rendering. This is the GPU Transfer task. The particle simulation task from the next frame depends on the finishing of the GPU Transfer task from the previous frame because it reuses the same memory for updating and storing particle simulation information – hence the dependency arrow in the graph from the bottom of the GPU Transfer task back to the Input task. After a copy of the data is transferred to the GPU, the rendering task can occur in parallel with other tasks.

It must be noted that even though the game is written using sequential code, the game still exhibits some parallelism that occurs in each frame after the rendering task issues draw commands to the GPU, since the actual drawing and rasterization of the 3D scene is done on the GPU in parallel to the CPU. While this is not strict sequential execution, it would be unfair to force the sequential execution of the game even on the GPU (by forcing the CPU to wait until the GPU finishes rendering the frame) as this is not what sequentially written games do. Our tests aim to show the benefits of applying the proposed methodology in a practical comparative environment more than in a theoretical one. Thus, we accept this parallelism to be part of the sequential game code. Likewise, all tests we conduct that claim to utilize the CPU only will exhibit the same CPU/GPU parallelism described above.

All tests were conducted on a Microsoft Windows 7 64-bit operating system running on a machine with a quad-core Intel® Xeon® 2.80Ghz processor. The machine has two graphics cards installed. The first is an AMD Radeon HD 6900 and the second is NVidia Geforce GTX 260. The two graphics cards are not connected using Crossfire or similar technologies. Their GPUs have no direct access to each other's VRAM.

In all the run-time performance tests we conducted, we measure performance as the time taken to execute one complete frame of the game in milliseconds. A frame is measured as the time difference between the ends of two consecutive render tasks. The measurement is done using the `QueryPerformanceCounter` API (Microsoft Corp., 2007). The frame time value is taken as the average of maximum frame time across multiple game runs. In each run, the game is left to run for 30 seconds.

4.3 TESTS AND RESULTS

To bring up the differences when applying the methodology to the sample game application, we have conducted three sets of tests, each highlighting a certain aspect of the differences.

4.3.1 Set 1: Sayl versus Cascade

The first set of tests compares the original serial game code with:

- 4CPU Cascade: A parallel version by applying our implementation of the Cascade design pattern.
- 4CPU Sayl: A parallel version by applying Sayl design pattern.

Since Cascade does not support heterogeneous processing, Sayl was limited in this set of tests to run on the CPU only by using the standard work stealing algorithm only instead of using the Arbiter Work Stealing scheduler back end.

In addition to run-time performance, we measure programmer effort involved in writing game code using each of Sayl and Cascade. For this, we used the source-lines of code (SLOC) metric as a measure to compare code size changes in the parallel versions as compared to the serial version. The CLOC (Danial, 2009) measure was used to count only the files containing game code, ignoring pattern implementation files and other libraries used.

The results gathered from the test are summarized Figure 4.4. Figure 4.5 shows the speed ups associated with the timings in calculated in Figure 4.4.

The SLOC test was run on each version of the sample game. The serial version counted 631 lines, the Sayl parallel version counted 702 lines and the Cascade parallel version counted 868 lines of code.

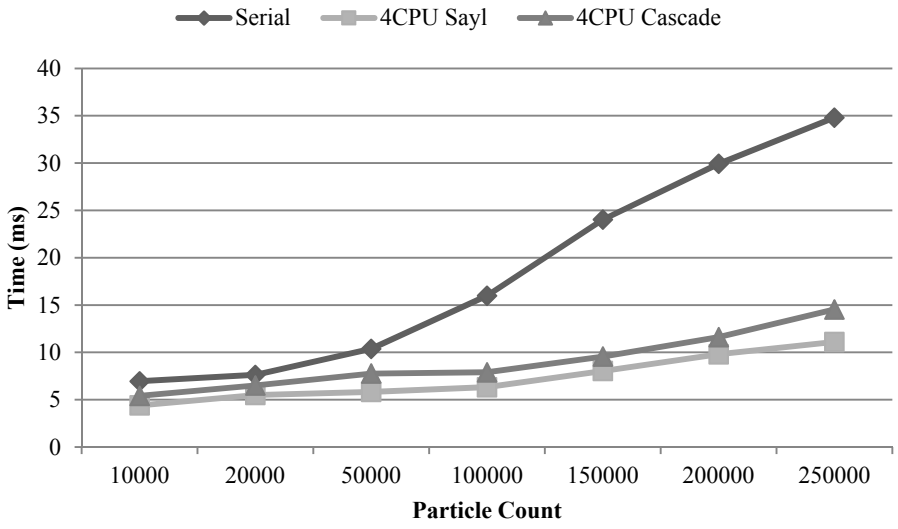


Figure 4.4: Run-time performance results for the serial, 4CPU Sayl and 4CPU Cascade versions of the sample game and their relationship to the number of simulated particles in the game.

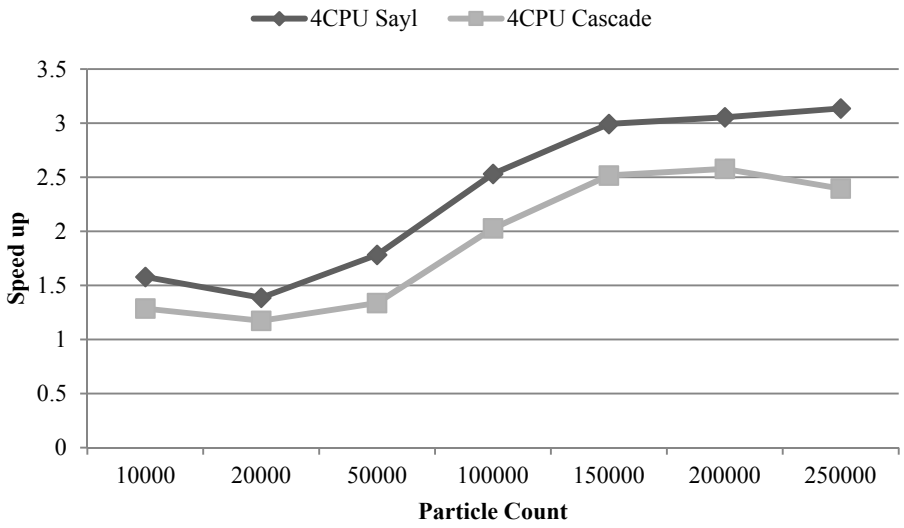


Figure 4.5: Speed up values for the run-time performance tests in Figure 4.4.

4.3.2 Set 2: Homogeneous processing versus heterogeneous processing

The second set of tests compares the serial game code with:

- 4CPU Sayl: A parallel version running on the CPU only, with the GPU being used only for standard 3D rendering and display tasks.
- 4CPU+1GPU: A parallel version that uses heterogeneous processing to execute game tasks on 4 CPUs in addition to 1 GPU.
- 4CPU+2GPU: A parallel version that uses heterogeneous processing to execute game tasks on 4 CPUs in addition to 2 different GPUs.

The results gathered from the test are summarized Figure 4.6. Figure 4.7 shows the speed ups associated with the timings in calculated in Figure 4.6.

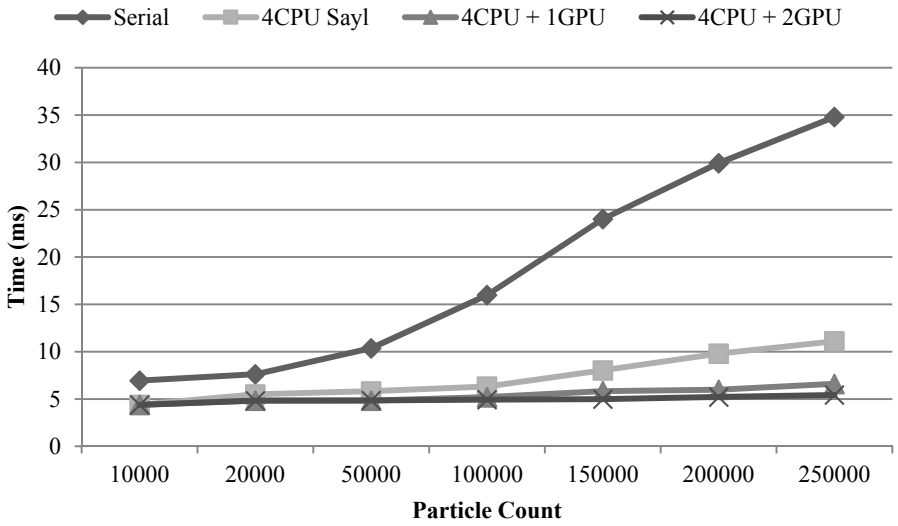


Figure 4.6: Run-time performance results for the serial, 4CPU Sayl, 4CPU+1GPU and 4CPU+2GPU versions of the sample game and their relationship to the number of simulated particles in the game.

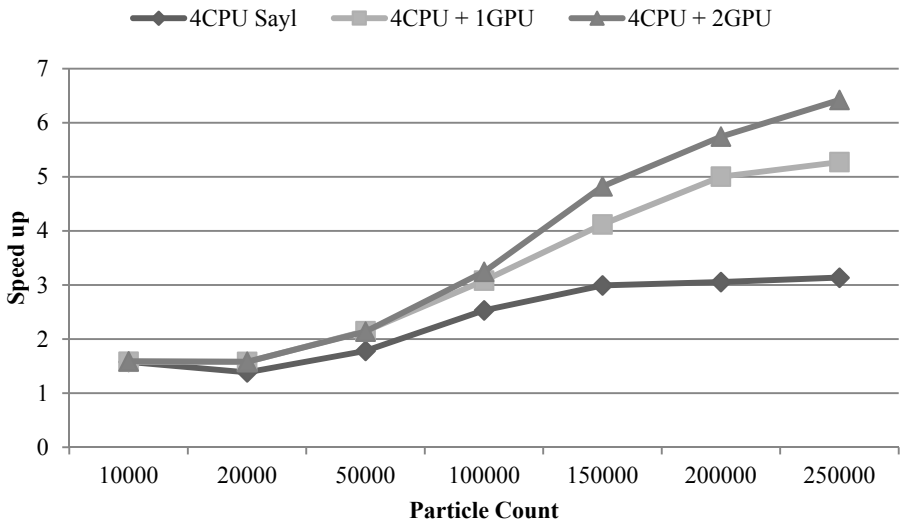


Figure 4.7: Speed up values for the run-time performance tests in Figure 4.6.

4.3.3 Set 3: Heterogeneous work stealing with the Arbiter versus without the Arbiter

The last set of tests aims to highlight the role of the Arbiter and proper processor grouping in the Arbiter Work Stealing scheduler. The versions tested are:

- 4CPU+2GPU: A parallel version that uses heterogeneous processing to execute game tasks on 4 CPUs in addition to 2 different GPUs using Arbiter Work Stealing.
- No Arbiter: A parallel version that runs on 4 CPUs and 2 GPUs using work stealing with random processor group assignment instead of using the Arbiter.
- Bad Grouping: A parallel version that uses heterogeneous processing to execute game tasks on 4 CPUs in addition to 2 different GPUs using Arbiter Work Stealing where each CPU is put in a separate processor group.

The results gathered from the test are summarized Figure 4.8. Figure 4.9 shows the speed ups associated with the timings in calculated in Figure 4.8.

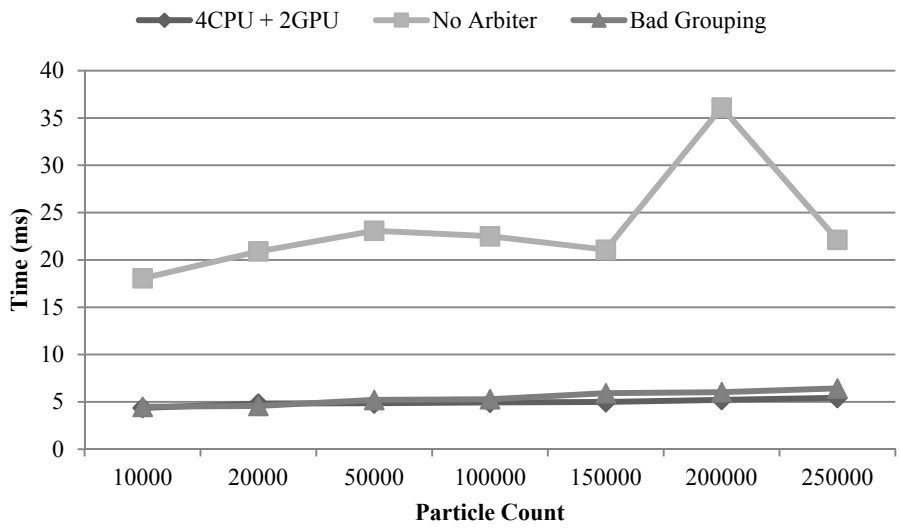


Figure 4.8: Run-time performance results for the 4CPU+2GPU, No Arbiter and Bad Grouping versions of the sample game and their relationship to the number of simulated particles in the game.

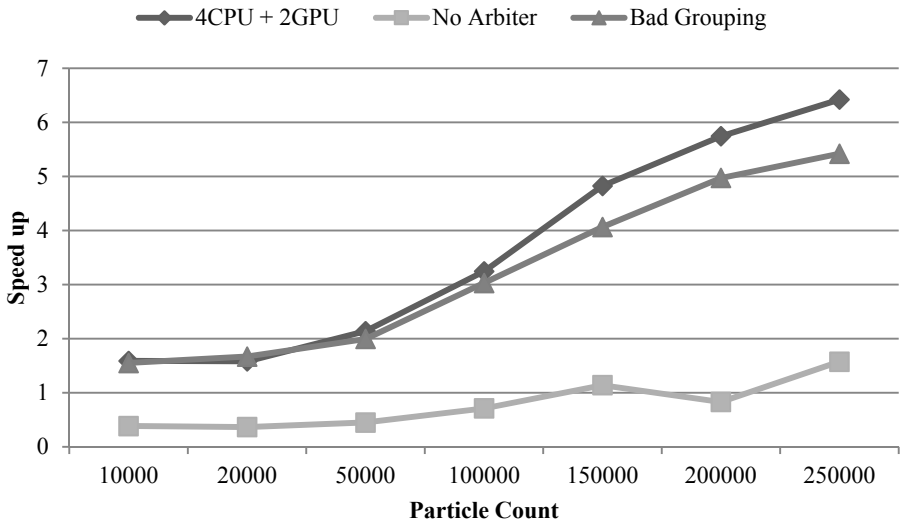


Figure 4.9: Speed up values for the run-time performance tests in Figure 4.8.

4.4 DISCUSSION AND ANALYISIS

The SLOC test shows that applying the Sayl pattern adds a relatively small amount of change to the code of the serial version. Using Sayl-Lib, converting a C++ serial function call to a parallel function call required only two changes:

1. Replacing the function's declaration by a single preprocessor macro that takes the function's name plus a list of its parameters.
2. Replacing each call to the function by a single preprocessor macro that takes the function's name plus parameter values.

Both of the changes above did not cause any increase in the number of lines in the code. The 71 lines of code difference include Sayl-Lib initialization code and the particle simulation algorithm changes mentioned in Section 4.2.

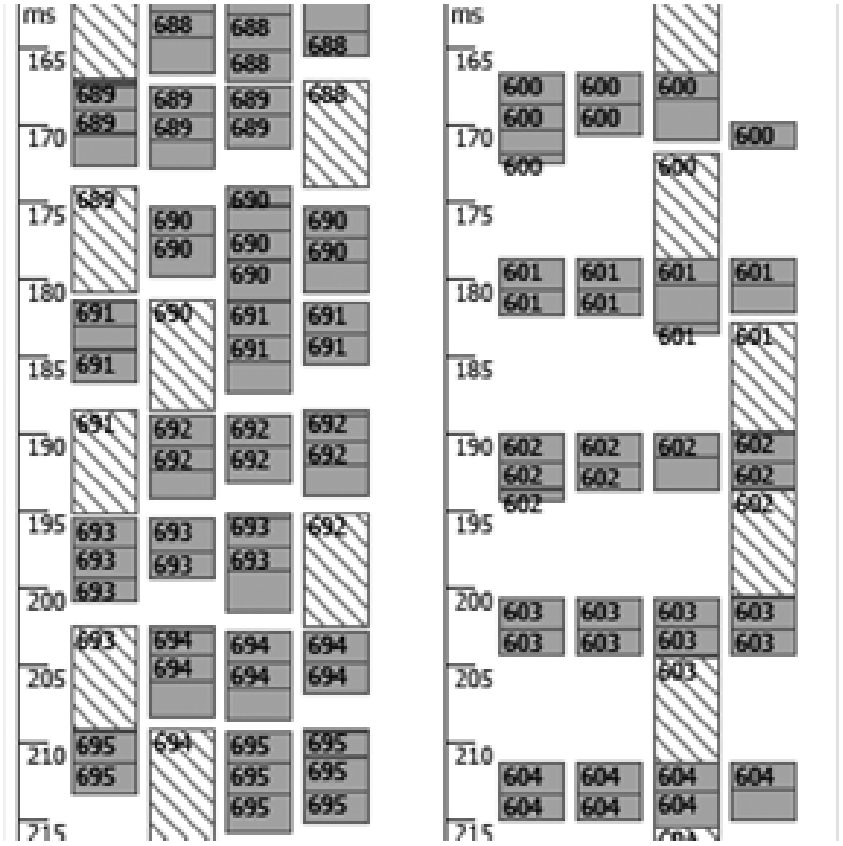
Applying Cascade to the serial version required replacing functions with task classes implementing a common interface. These classes are then used in declaring task instances for building the static task graph. The changes amounted to 237 lines of code, including initialization code and the same algorithmic changes. The majority of the increase was due to the requirements of declaring task classes and building the static task graph.

The results in Figure 4.4 and Figure 4.5 show that Sayl-based implementation performs much better than Cascade-based implementation. This is mainly due to Sayl's support for dynamic task graphs. For example, the rendering task of the game has shown to take a considerable amount of time to execute; but Sayl allowed running of computations for the next frame while the rendering task of the current frame is still ongoing. This overlap in

execution of tasks required for the generation of multiple frames was effective in increasing the overall performance.

Figure 4.10 shows a snapshot from the runtime of each of the two parallel versions of the first set of tests: the Sayl-based and the Cascade-based versions. The figure shows the allocation of executing tasks on the processing cores. Each column represents a single processing core. The test program was executed on a quad-core processor, hence there are 4 columns. Each block represents a task. The number in the block represents the frame index with which the task is associated. The size of the block reflects the task's CPU burst in milliseconds. Blocks with a hatched fill represent the rendering task, while blocks with solid fill represent the remaining tasks (i.e., input update, player update and particle simulation). It clearly shows that in the Cascade-based version all tasks in a frame must finish before tasks from the next frame can start execution.

Figure 4.5 shows that as the number of particles decreases, the speed up for both parallel versions decreases. When the number of particles becomes too small, then the game utilizes only two processing elements. The game's parallelism becomes limited to running the rendering task in parallel to all other tasks in the game.



Sayl-based Cascade-based

Figure 4.10: Snapshot from the test program runtime showing the allocation of executing tasks on the processing cores. The left figure is captured from the Sayl-based version, while the right figure is captured from the Cascade-based version.

The 4CPU Sayl parallel version achieved sub-linear speedup. The reason was mainly due to the large amount of data transfer from the CPU to the GPU after particle simulation is done, whereas the 4CPU+1GPU and 4CPU+2GPU parallel versions were able to get better speedups because the simulation of many particle chunks was done on the GPU and their data remained in VRAM, which decreased data transfer time. Figure 4.11 shows a snapshot from the runtime of each of the two parallel versions of the test program: the 4CPU Sayl and the 4CPU+1GPU versions.

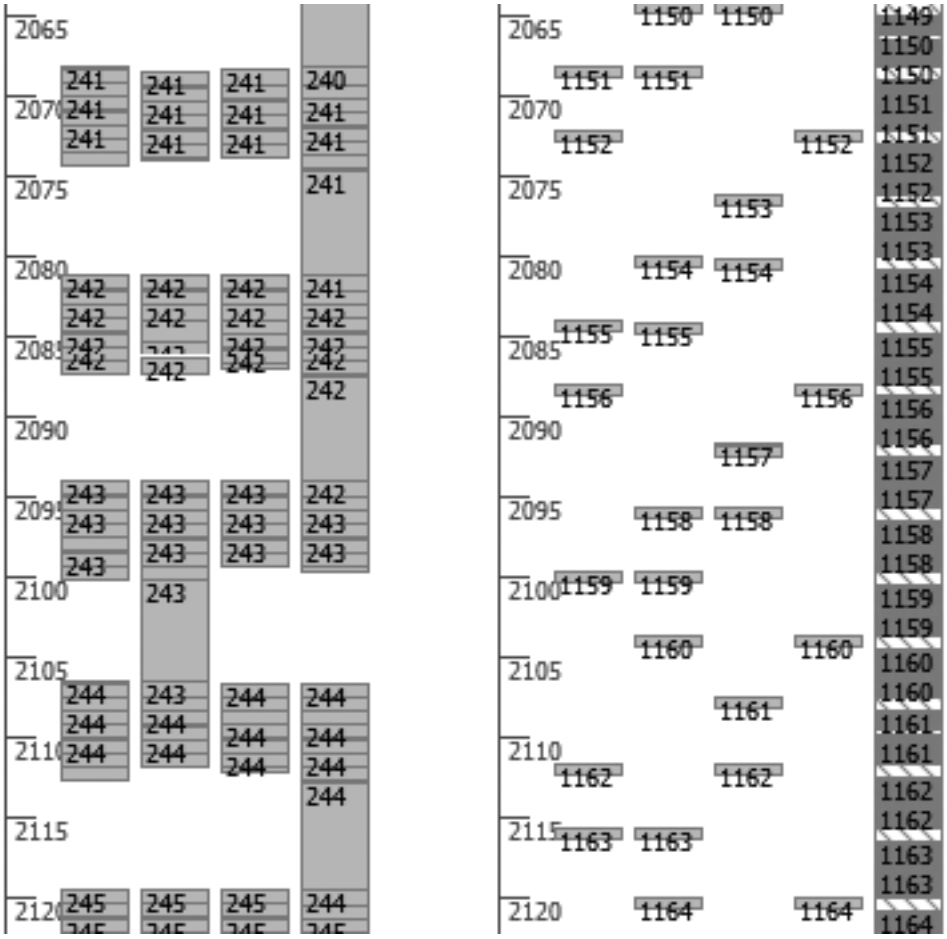


Figure 4.11: Snapshot from the test program runtime showing the allocation of executing tasks on the processing cores. The left figure is captured from the 4CPU Sayl parallel version, while the right figure is captured from the 4CPU+1GPU parallel version, where the right-most column represents tasks executed on the GPU.

The figure shows the allocation of executing tasks on the processing cores. The figure reveals the data transfer bottleneck in the 4CPU Sayl version. For the 4CPU+1GPU case, it shows that the Arbiter preferred to run the majority of particle simulation tasks on the GPU except for a very few in every frame. This is because the CPU is not only slower than the GPU for such a task, but the cost of data transfer is considerable too. The Arbiter

decides to execute particle simulation tasks on the CPU only when it finds that the GPU will finish later if it handled all particle simulation tasks.

The results from the last set of tests indicate the importance of the Arbiter's role in achieving better overall application performance by using well-guided decisions. The performance in the No Arbiter version suffered from two problems: data transfer overhead for moving parameter data between different memory locations, plus bottlenecking the performance by executing tasks on less-efficient processor types. The Bad Grouping version also exhibits slightly reduced performance in comparison to the proper 4CPU+2GPU version. This is because the bad grouping of processors resulted in deactivating the original work stealing algorithm and relying on the Arbiter completely to map tasks to individual processors, which takes more time than the original work stealing algorithm to do the mapping due to the various considerations it takes.

4.5 SUMMARY AND CONCLUSIONS

This chapter conducted three sets of tests on a case study sample game application that was originally written in a sequential manner using classical game code structures. The tests demonstrate the benefits of applying the framework to game applications. The programming tools and languages used were listed, and a description of the sample game application is given. The tests were done in three sets, where each set focused on certain aspects of the framework. The results have shown good indicators for the benefits of applying the framework's methodology in game applications on the run-time performance front in addition to the programmer effort front. Performance was analyzed under different environment setups with varying work size. Several comparison tests

were conducted with Cascade, in which Sayl demonstrated better results in both SLOC and run-time performance metrics.

In conclusion, the use of the proposed new framework of the parallel design pattern, Sayl in conjunction with Arbiter Work Stealing is useful in helping improve the performance of game applications running on heterogeneous processing environments.

CHAPTER 5 SUMMARY, CONCLUSIONS AND FUTURE WORK

5.1 SUMMARY

This thesis investigated several design patterns and scheduling algorithms from existing literature in addition to analyzing the structure of game applications in order to develop a new game parallelization framework consisting of a parallel design pattern and a heterogeneous processing task scheduler to allow parallelizing game applications with the goal of improving their performance on heterogeneous processing environments. Such environments already exist in commodity personal computers as well as game consoles. The benefits of the proposed framework were demonstrated through a case study of parallelizing a sample game application that was originally written using sequential code.

Because previous work targeting game applications is scarce, there was little work that takes into account the particularities of games in order to maximize parallelization and hence improve performance. Sayl, a new parallel design pattern has been developed and implemented to allow parallelization of games with little programmer effort in addition to allowing better parallelization opportunities by supporting dynamically generated task graphs. The design pattern has two parts: a front-end through which programmers spawn tasks, and a back-end which schedules tasks on processing elements. The back-end specification was made flexible enough to handle different implementations.

This research also developed and implemented a task scheduler capable of efficiently scheduling tasks in heterogeneous environments. The Arbiter Work Stealing scheduler is

based on the classic work stealing algorithm and has been integrated as the back-end to allow game tasks to benefit from heterogeneous processing capabilities available in their host environments. The scheduler groups processors in the environments into separate groups based on capability and communication criteria. Each group runs the classical work stealing algorithm. The Arbiter is a new component that monitors work load in each processor group to make guided decisions for each task to map it the best processor group in such a way that results in overall improved application performance.

A case study of a sample game was developed to demonstrate the feasibility and results of applying the framework. The sequential code of the game was converted to work in parallel by following different approaches and comparing the results to validate and evaluate the performance of each approach. The tests were done in three sets to highlight certain aspects of the work. The results of these tests indicated good performance gains by applying the framework's methodology to games without requiring large programmer effort.

5.2 CONTRIBUTIONS

The main contribution of this research is a new game parallelization framework consisting of the following:

(1) A parallel design pattern for game applications has been designed. The pattern aims to reduce the amount of programmer effort involved in writing parallel game code, as well as maximize parallelization of game tasks to achieve better performance.

(2) A dynamic task scheduler for heterogeneous environment has been developed. The scheduler integrates with the parallel design pattern proposed in the thesis, and allows game applications to execute their tasks on the different processors available in their host environments in an efficient manner.

(3) A sample game application was developed and converted to run in parallel on heterogeneous processing environments consisting. Several comparison tests were made to show the programmer effort and performance benefits of applying the proposed work versus previous work.

5.3 LIMITATIONS AND FUTURE WORK

While fulfilling the objectives of this research, there are certain areas that require further research including the following:

(1) Although the proposed design pattern simplifies programmer work in the aspect of task graph generation and execution, it does not provide facilities for automatic data management and movement in heterogeneous processing environments.

(2) Because the host environments have processors of different types, with a possibly different programming language each, it would be beneficial to design a language that allows writing the task code once and can target all of those processor types automatically by translating the code into processor-specific code (such as shader programs for the GPU).

(3) Not all work done in games can be expressed as game tasks. The example of background resource loading was given in the thesis, but there are other tasks that

require further thought before they can be structured following the Sayl design pattern. Network communication and audio mixing are examples of such tasks.

(4) For the future, we plan to investigate the benefits of providing the scheduler with information about future task input data locations, which could be used to guide the Arbiter into even better decisions.

(5) The simplicity of the serial-to-parallel code conversion process when applying Sayl opens the opportunity for considering the development of an automatic game parallelization system which can convert existing serial code bases to run in parallel.

REFERENCES

Adams, D. A. (1969). *A Computation Model with Data-Flow Sequencing.* Tech. Report TR CS 177, Stanford Univ., Computer Science Dept., Stanford, Calif.

Agrawal, K., He, Y., & Leiserson, C. E. (2007). Adaptive work stealing with parallelism feedback. *The 12th ACM SIGPLAN symposium on Principles and practice of parallel programming.* San Jose, California, USA: ACM. doi:10.1145/1229428.1229448

Andrews, J., & Baker, N. (2006, March). Xbox 360 System Architecture. *IEEE Micro, 26*(2), pp. 25-37. doi:10.1109/MM.2006.45

Augonnet, C., Thibault, S., Namyst, R., & Wacrenier, P.-A. (2011, February). StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper., Special Issue: Euro-Par 2009*, pp. 187–198.

Best, M. J., Fedorova1, A., Dickie, R., Tagliasacchi, A., Couture-Beil, A., Mustard, C., . . . Brownsword, A. (2009). Searching for Concurrent Design Patterns in Video Games. *Euro-Par '09 Proceedings of the 15th International Euro-Par Conference on Parallel Processing.* London: Springer-Verlag London.

Blow, J. (2004, February). Game Development: Harder Than You Think. *Queue - Game Development, 1*(10). doi:10.1145/971564.971590

Blumofe, R. D., & Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM), 46*(5), 720-748. doi:10.1145/324133.324234

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., & Zhou, Y. (1995). Cilk: an efficient multithreaded runtime system. *Proceedings of the*

fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, (pp. 207-216). Santa Barbara, California, United States. doi:10.1145/209936.209958

Boyd, C. (2008). The DirectX 11 compute shader. *SIGGRAPH 2008*. Retrieved from http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf

Cosnard, M., & Jeannot, E. (1999, September). Compact DAG representation and its dynamic scheduling. *Journal of Parallel and Distributed Computing, 58*(3).

Danial, A. (2009). CLOC: Count lines of code. Retrieved December 2011, from http://cloc.sourceforge.net/

Diamos, G. F., & Yalamanchili, S. (2008). Harmony: an execution model and runtime for heterogeneous many core systems. *Proceedings of the 17th international symposium on High performance distributed computing*, (pp. 197-200). doi:10.1145/1383422.1383447

Diefendorff, K., Dubey, P. K., Hochsprung, R., & Scales, H. (2000). AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro, 20*(2), 85-95. doi:10.1109/40.848475

Ericson, C. (2005). *Real-Time Collision Detection.* Morgan Kaufmann.

Fatahalian, K., & Houston, M. (2008). GPUs: A Closer Look. *ACM Queue*, 18-28.

Giusti, L. D., Chichizola, F., Naiouf, M., & Giusti, A. D. (2008). Mapping Tasks to Processors in Heterogeneous Multiprocessor Architectures: The MATEHa Algorithm. *SCCC '08 Proceedings of the 2008 International Conference of the Chilean Computer Science Society*, (pp. 85-91). doi:10.1109/SCCC.2008.11

Giusti, L. D., Luque, E., Chichizola, F., Naiouf, M., & Giusti, A. D. (2009). AMTHA: An Algorithm for Automatically Mapping Tasks to Processors in Heterogeneous Multiprocessor Architectures. *Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering*, (pp. 481-485). doi:10.1109/CSIE.2009.175

Goetz, B. (2002, July 1). *Java theory and practice: Thread pools and work queues.* Retrieved from IBM: http://www.ibm.com/developerworks/java/library/j-jtp0730/index.html

Gonina, E., & Chong, J. (2008). *Task Queue Implementation Pattern.* Retrieved from UC Berkeley ParLab: http://parlab.eecs.berkeley.edu/wiki/_media/patterns/taskqueue.pdf

Gregg, C., & Hazelwood, K. (2011). Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (pp. 134-144). IEEE. doi:10.1109/ISPASS.2011.5762730

Gschwind, M. (2007, August 22). *Using data-parallel SIMD architecture in video games and supercomputers.* Retrieved from IBM: http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.simd.html

Hamidzadeh, B., Lilja, D. J., & Atif, Y. (1995, October). Dynamic Scheduling Techniques for Heterogeneous Computing Systems. *Concurrency: Practice and Experience, 7*(7), pp. 633-652.

Hansen, C., Crockett, T., & Whitman, S. (1994). Guest Editors' Introduction: Parallel Rendering. *IEEE Concurrency, 2*(2), 7. doi:10.1109/MCC.1994.10014

Hovland, R. J. (2008). *Latency and Bandwidth Impact on GPU-systems Projet report.* Norwegian University of Science and Technology, Department of Computer and Information Science. Retrieved from http://www.idi.ntnu.no/elster/master-studs/runejoho/ms-proj-gpgpu-latency-bandwidth.pdf

Intel Corporation. (2007, April). *Intel SSE4 Programming Reference.* Retrieved from Intel: http://software.intel.com/sites/default/files/m/9/4/2/d/5/17971-intel_20sse4_20programming_20reference.pdf

Johnson, T., Davis, T. A., & Hadfield, S. M. (1996, February). A concurrent dynamic task graph. *Parallel Computing, 22*(2), 327-333. doi:10.1016/0167-8191(95)00061-5

Joselli, M., Clua, E., Montenegro, A., Conci, A., & Pagliosa, P. (2008). A new physics engine with automatic process distribution between CPU-GPU. *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games.* Los Angeles, California: ACM. doi:10.1145/1401843.1401871

Joselli, M., Zamith, M., Clua, E. W., Montenegro, A., Leal-Toledo, R. C., Valente, L., & Feijó, B. (2010). An Architecture with Automatic Load Balancing and Distribution for Digital Games. *2010 Brazilian Symposium on Games and Digital Entertainment.*

Joselli, M., Zamith, M., Clua, E., Montenegro, A., Conci, A., Leal-Toledo, R., . . . Pozzer, C. (2008). Automatic Dynamic Task Distribution between CPU and GPU for Real-Time Systems. *11th IEEE International Conference on Computational Science and Engineering, CSE '08* (pp. 48-55). IEEE. doi:10.1109/CSE.2008.38

Joselli, M., Zamith, M., Clua, E., Montenegro, A., Leal-Toledo, R., Conci, A., . . . Feijó, B. (2009, December). An adaptative game loop architecture with automatic distribution of tasks between CPU and GPU. *Computers in Entertainment (CIE), 7*(4). doi:10.1145/1658866.1658869

Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., & Shippy, D. (2005, July). Introduction to the cell multiprocessor. *IBM Journal of Research and Development, 49*(4/5), 589-604.

Khronos Group. (1997). Retrieved from OpenGL: http://www.opengl.org/

Kilgariff, E., & Fernando, R. (2005). The GeForce 6 Series GPU Architecture. In M. Pharr, & R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley Professional.

Kim, C., & Agrawala, A. K. (1989, February). Analysis of the Fork-Join Queue. *IEEE Transactions on Computers, 38*(2), pp. 250-255. doi:10.1109/12.16501

Lavender, R. G., & Schmidt, D. C. (1996). Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2.* Boston, MA: Addison-Wesley Longman Publishing Co. Inc.

Luk, C.-K., Hong, S., & Kim, H. (2009). Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. *MICRO 42 Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 45-55). New York, NY, USA: ACM. doi:10.1145/1669112.1669121

Maheswaran, M., & Siegel, H. J. (1998). A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems. *Proceedings of the Seventh Heterogeneous Computing Workshop*, (p. 57).

Manolescu, D.-A. (September, 1997). A Data Flow Pattern Language. *Proc. 4th Pattern Languages of Programming.* Monticello, IL.

Mattson, T., Sanders, B., & Massingill, B. (2004). *Patterns for Parallel Programming.* Addison-Wesley Professional.

Microsoft Corp. (2007, November). *How to use the QueryPerformanceCounter function to time code in Visual C++*. Retrieved 12 2011, from Microsoft: http://support.microsoft.com/kb/815668/en-us/

Microsoft Corp. (2012). *C++ AMP (C++ Accelerated Massive Parallelism).* Retrieved from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/hh265137.aspx

Microsoft Corp. (2012). *Direct3D*. Retrieved from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/windows/desktop/hh309466%28v=vs.85%29.aspx

Microsoft Corp. (2012, 6 21). *XInput Game Controller APIs*. Retrieved from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/windows/desktop/hh405053%28v=vs.85%29.aspx

Miller, A. (2010). The task graph pattern. *ParaPLoP '10: Proceedings of the 2010 Workshop on Parallel Programming Patterns.* ACM. doi:10.1145/1953611.1953619

NVIDIA Corp. (2007). NVIDIA CUDA: Compute Unified Device Architecture.

Rhalibi, A. E., Merabti, M., & Shen, Y. (2006). Improving game processing in multithreading and multiprocessor architecture. *Edutainment'06 Proceedings of the First international conference on Technologies for E-Learning and Digital Entertainment* (pp. 669-679). Springer-Verlag Berlin, Heidelberg. doi:10.1007/11736639_81

Rodrigues, J. E., & Bezos, J. E. (1969). *A GRAPH MODEL FOR PARALLEL COMPUTATIONS.* Cambridge, MA: Massachusetts Institute of Technology.

Stroustrup, B. (1997). *The C++ Programming Language Third Edition.* Massachusetts: Addison Wesley, Reading.

The OpenCL Specification. (2010). Retrieved from http://www.khronos.org/registry/cl/

Valente, L., Conci, A., & Feijó, B. (2005). Real time game loop models for single-player computer games. *Proceedings of the Fourth Brazilian Symposium on Computer Games and Digital Entertainment*, (pp. 89-99).

Werth, T., Schreier, S., & Philippsen, M. (2011). CellCilk: Extending Cilk for heterogeneous multicore platforms. *The 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2011).* Fort Collins, Colorado, USA.

APPENDICES

APPENDIX A: Part of the Sayl C++ preprocessor macros for declaring parallel C++ functions.

```
//#define SAYL_GLOBAL_ACCESSOR g_Sayl->

// The macros SAYL_MEMBER_FUNCTIONx are used to declare
// tasks that do not use asynchronous parameters, instead
// they require that all of its x parameters are specified
// at once (not asynchronous).

#define SAYL_MEMBER_FUNCTION(classType,funcName,procType) \
struct funcName ##SaylParams { classType& m_obj;\
  static const ProcessorType kProcessorType = procType; \
  void operator =(const funcName ##SaylParams&) {}; \
  funcName ##SaylParams(classType& obj) : m_obj(obj) {} \
  static void Fn(void **prm,const TaskContext& ctx) \
  { var& p = *((funcName ##SaylParams*)prm[0]); \
  p.m_obj.funcName(ctx); } \
  static long Est(void **prm,const TaskContext& ctx) \
  { var& p = *((funcName ##SaylParams*)prm[0]); \
  return p.m_obj.funcName ##Estimate(ctx); } }; \
  long funcName ##Estimate(const TaskContext& ctx); \
  void funcName(const TaskContext& ctx)

#define
SAYL_MEMBER_FUNCTION1(classType,funcName,procType,T1,N1) \
struct funcName ##SaylParams { classType& m_obj; T1 p1; \
  static const ProcessorType kProcessorType = procType; \
  void operator =(const funcName ##SaylParams&) {}; \
  funcName ##SaylParams(classType& obj,T1 v1) : \
  m_obj(obj),p1(v1) {} \
  static void Fn(void **prm,const TaskContext& ctx) \
  { var& p = *((funcName ##SaylParams*)prm[0]); \
  p.m_obj.funcName(p.p1,ctx); } \
  static long Est(void **prm,const TaskContext& ctx) \
  { var& p = *((funcName ##SaylParams*)prm[0]); \
  return p.m_obj.funcName ##Estimate(p.p1,ctx); } }; \
  long funcName ##Estimate(T1 N1,const TaskContext& ctx); \
  void funcName(T1 N1,const TaskContext& ctx)
```

```
#define
SAYL_MEMBER_FUNCTION2(classType,funcName,procType,T1,N1,T2,
N2) \
struct funcName ##SaylParams { classType& m_obj; T1 p1;T2
p2; \
  static const ProcessorType kProcessorType = procType; \
  void operator =(const funcName ##SaylParams&) {}; \
  funcName ##SaylParams(classType& obj,T1 v1,T2 v2) : \
  m_obj(obj),p1(v1),p2(v2) {} \
  static void Fn(void **prm,const TaskContext& ctx) \
  { var& p = *((funcName ##SaylParams*)prm[0]); \
  p.m_obj.funcName(p.p1,p.p2,ctx); } \
  static long Est(void **prm,const TaskContext& ctx) \
  { var& p = *((funcName ##SaylParams*)prm[0]); \
  return p.m_obj.funcName ##Estimate(p.p1,p.p2,ctx); } }; \
  long funcName ##Estimate(T1 N1,T2 N2,const TaskContext&
ctx); \
  void funcName(T1 N1,T2 N2,const TaskContext& ctx)


#define SAYL_MEMBER_CALL(classType,funcName,obj) \
SAYL_GLOBAL_ACCESSOR AddTask(\
     classType::funcName ##SaylParams::Fn,\
     classType::funcName ##SaylParams::Est,\
     classType::funcName
##SaylParams::kProcessorType,classType::funcName
##SaylParams(obj));

#define SAYL_MEMBER_CALL1(classType,funcName,obj,v1) \
SAYL_GLOBAL_ACCESSOR AddTask(\
     classType::funcName ##SaylParams::Fn,\
     classType::funcName ##SaylParams::Est,\
     classType::funcName
##SaylParams::kProcessorType,classType::funcName
##SaylParams(obj,v1));

#define SAYL_MEMBER_CALL2(classType,funcName,obj,v1,v2) \
SAYL_GLOBAL_ACCESSOR AddTask(\
     classType::funcName ##SaylParams::Fn,\
     classType::funcName ##SaylParams::Est,\
     classType::funcName
##SaylParams::kProcessorType,classType::funcName
##SaylParams(obj,v1,v2));

// The macros SAYL_MEMBER_APFUNCTIONx are used to declare
// tasks that use asynchronous parameters. Each parameter
// can be supplied independently using the
```

```
// SAYL_MEMBER_PASS macro.
// The SAYL_MEMBER_APFUNCTIONN macro is used for
// tasks that require a variable number of asynchronous
// parameters. For such cases, the task itself has to
// extract the parameters.
// Use SAYL_MEMBER_PASSN to pass parameters to such tasks.

#define
SAYL_MEMBER_APFUNCTION2(funcName,obj,procType,T1,N1,T2,N2)
\
     struct funcName ##SaylParams { static const int
kParamCount=2; \
     static const ProcessorType kProcessorType = procType;
\
     static void Fn(void **prm,const TaskContext& ctx) { \
       var& N1 = *((T1*)prm[0]); \
       var& N2 = *((T2*)prm[1]); \
       (obj)->funcName(N1,N2,ctx); } \
     static long Est(void **prm,const TaskContext& ctx) { \
       var& N1 = *((T1*)prm[0]); \
       var& N2 = *((T2*)prm[1]); \
       return (obj)->funcName ##Estimate(N1,N2,ctx); } \
     }; \
  long funcName ##Estimate(T1 N1,T2 N2,const TaskContext&
ctx); \
  void funcName(T1 N1,T2 N2,const TaskContext& ctx)

#define
SAYL_MEMBER_PASS(classType,funcName,taskSubID,paramID,val)
\
SAYL_GLOBAL_ACCESSOR AddTaskParam(\
  classType::funcName ##SaylParams::Fn,\
  classType::funcName ##SaylParams::Est,\
  classType::funcName ##SaylParams::kProcessorType,\
  taskSubID,classType::funcName
##SaylParams::kParamCount,paramID,val);

#define SAYL_MEMBER_APFUNCTIONN(funcName,obj,procType) \
     struct funcName ##SaylParams { \
     static const ProcessorType kProcessorType = procType;
\
     static void Fn(void **prm,const TaskContext& ctx) {
(obj)->funcName(prm,ctx); } \
     static long Est(void **prm,const TaskContext& ctx) {
return (obj)->funcName ##Estimate(prm,ctx); } \
     }; \
```

```
  long funcName ##Estimate(void **pParams,const
TaskContext& ctx); \
  void funcName(void **pParams,const TaskContext& ctx)

#define
SAYL_MEMBER_PASSN(classType,funcName,taskSubID,paramsCount,
paramID,val) \
SAYL_GLOBAL_ACCESSOR AddTaskParam(\
     classType::funcName ##SaylParams::Fn,\
     classType::funcName ##SaylParams::Est,\
     classType::funcName
##SaylParams::kProcessorType,taskSubID,paramsCount,paramID,
val);
```

APPENDIX B: List of publications.

Articles accepted / submitted to referred conferences

Wessam AlBahnassi, Sudhir P. Mudur and Dhrubajyoti Goswami, A Design Pattern for Parallel Programming of Games, High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on , vol., no., pp.1007-1014, 25-27 June 2012. doi: 10.1109/HPCC.2012.147.

Wessam AlBahnassi, Sudhir P. Mudur, and Dhrubajyoti Goswami. Arbiter Work Stealing for Parallelizing Games on Heterogeneous Computing Environments, Submitted to IPDPS-2013 27th IEEE International Parallel & Distributed Processing Symposium, 20-24 May, 2013, Boston, Massachusetts, USA.