Searching for Concurrent Design Patterns in Video Games

Micah J Best[1], Alexandra Fedorova[1], Ryan Dickie[1], Andrea Tagliasacchi[1], Alex Couture-Beil[1], Craig Mustard[1], Shane Mottishaw[1], Aron Brown[1], Zhi Feng Huang[1], Xiaoyuan Xu[1], Nasser Ghazali[1], and Andrew Brownsword[2]

[1] Simon Fraser University
[2] Electronic Arts Blackbox

Abstract. The transition to multicore architectures has dramatically underscored the necessity for parallelism in software. In particular, while new gaming consoles are by and large multicore, most existing video game engines are essentially sequential and thus cannot easily take advantage of this hardware. In this paper we describe techniques derived from our experience parallelizing an open-source video game Cube 2. We analyze the structure and unique requirements of this complex application domain, drawing conclusions about parallelization tools and techniques applicable therein. Our experience and analysis convinced us that while existing tools and techniques can be used to solve parts of this problem, none of them constitutes a comprehensive solution. As a result we were inspired to design a new parallel programming environment (PPE) targeted specifically at video game engines and other complex soft real-time systems. The initial implementation of this PPE, Cascade, and its performance analysis are also presented.

1 Introduction

A video game engine is the core software component that provides the skeleton on which games are built and represents the majority of their computational complexity. Most game engines were originally written to be executed on machines with no facility for truly parallel execution. This has become a major problem in this performance hungry domain. In addition to the near ubiquity of multicore processors in consumer PCs the latest generation of gaming consoles have followed this trend as well. Microsoft's Xbox 360 and Sony's PlayStation 3 both feature multicore processors. To address this problem, video game engines are being restructured to take advantage of multiple cores.

Restructuring for parallelization has begun[1], but much of the potential performance gains have yet to be realized as parallelizing a video game engine is a daunting task. Game engines are extremely complex systems consisting of multiple interacting modules that modify global shared state in non-trivial ways. This paper describes our experience of parallelizing a video game engine, our preliminary accomplishments, the lessons we learned and the insight for future research that emerged from this experience.

For our research, we use an open-source video game engine, Cube 2, which we enhanced by adding extra logic to AI and Physics modules so that it more closely resembles a commercial engine – we refer to this extended engine as Cube 2-ext. Careful inspection of this game engine, as well as our knowledge of commercial game engines, convinced us that hand-coding the engine to use threads and synchronization primitives would not only be difficult for an average programmer, but introduce a level of complexity that would limit even experts in extracting parallelism. Instead, we felt that relying on a parallel library that facilitates the expression of parallel patterns in sequential code and then parallelizes the code automatically would be more efficient. An evaluation of existing parallel libraries showed that none of them offered the support that we needed to express all the computation patterns present in game engines and so we created Cascade, our own library to fill this gap.

In its first incarnation, Cascade allowed a dependency-graph style of programming, where the computation is broken down into tasks organized in a graph according to their sequential dependencies and, most importantly, supported an efficient parallel implementation of a *producer/consumer* pattern. The producer/consumer pattern, pervasive in video games, consists of two tasks where one task (the producer) generates data for another (the consumer). While the pattern itself is not new, the parallel implementation and its semantic expression described in this paper are unique to Cascade. By applying this producer/consumer pattern we were able to achieve, using eight cores, a 51% decrease in the computation time necessary for the non-rendering or 'simulation' phase of Cube 2-ext.

While this first parallelization attempt was fruitful, we also learned about the limitations of the producer/consumer pattern and about the difficulty of trying to overcome these limitations in C++. In particular, we learned that whenever a computation had a *side-effect*, i.e., where the modification of some global state was required, the application of the producer/consumer pattern was very difficult without significant restructuring of the code. We also learned that the producer/consumer pattern did not allow for in-place transformations of data as a producer always generates a new copy of data. This was limiting in terms of performance and memory demands. At the same time, we were unable to find the required semantic constructs for expressing parallelism with side-effects and in-place transformations in existing C++ libraries. This has inspired us to design the Cascade Data Management Language (CDML). CDML, a work in progress, allows expression of parallel constructs and yet avoids these limitations. CDML is translated into C++, the language required by the game industry. We give a preliminary sketch of parts of CDML and describe its design philosophy.

In the rest of the paper we provide an overview of video game engines and the challenges involved in their parallelization (Section 2), we describe how we parallelized parts of Cube 2-ext using the new implementation of the producer/consumer pattern, briefly introduce Cascade and provide experimental analysis (Section 3). We describe the lessons learned and introduce CDML in Section 4. We describe related work and conclude in Sections 5 and 6.

2 Challenges in Parallelizing Video Game Engines

A game engine performs a repeated series of game state updates where a new state is generated for each video frame. Each part of this state is the responsibility of one or more distinct subsystem in the engine. The AI subsystem, for example, is responsible for dictating the behaviour of artificial agents in the game world while the Rendering subsystem is responsible for combining texture and geometry data and transferring it to the GPU. While the nature of data and computations in a particular subsystem can be quite different from another they are tightly coupled and the product of one subsystem may be required by several others.

A trivial attempt at parallelizing an engine could amount to running each of these subsystems in its own thread. This is far from ideal as the degree of parallelization would be limited by the number of subsystems and the differences between computational loads. Furthermore, this solution would be difficult to implement efficiently because the subsystems modify shared global state and interact in non-trivial ways. For instance, the AI subsystem updates the behaviours of the AI agents and then passes the control to Physics which simulates new positions and poses for characters' skeletons based on their behaviour, which are then used by the Renderer for display. This creates a series of *data dependencies* among these subsystems. If these subsystems run in parallel the global shared state must be protected, significantly limiting concurrency.

One alternative to using threads is to use parallel constructs such as parallel-for, found in OpenMP [2] and other libraries. Parallel-for is used for loop parallelization where the compiler relies on programmer-inserted directives to generate threaded code. While such tools are appropriate for some cases they are not sufficient to solve the entire problem. Data dependencies among different subsystems do not easily map to these simple parallel constructs making it difficult to express fine-grained and inter-subsystem parallelism. These approaches also exhibit a tendency for execution to alternate between periods of high parallelism and serial execution, leaving the serial phases as a critical bottleneck.

Furthermore, any parallelization tools must preserve the determinism of results and guarantee consistent performance to assure a seamless experience and correctness in multiplayer games. These constraints imply that using dynamic compilation or other kinds of dynamic optimizations are often a poor choice for this domain. In fact, some video game platforms simply forbid dynamic compilation.

In summary, video game engines are structured as multiple interacting subsystems with complex data dependency patterns making explicit threading and simple parallel constructs a poor option. A better solution would be to employ a PPE that allows the integration of diverse styles of parallelization methods.

3 Parallelizing Cube 2-ext

We identified the open source project Cube 2 as a test bed for our research. Despite having many attractive properties as an experimental test bed, Cube

2 lacked many modern game features such as fully realized physics simulation or character skinning. We addressed these limitations by adding some of these features to Cube 2. In particular, we added a more robust physics layer in the form of the popular Open Dynamics Engine (ODE) [3] and replaced the existing AI routines with more sophisticated versions that simulated 'pack behaviour'. In the rest of the text we refer to this extended version as Cube 2-ext.

3.1 Parallelizing the Producer/Consumer Pattern

Our analysis of Cube 2-ext showed that in many places the computation was structured as a chain of interacting modules, or subsystems. Similar analysis applies to commercial video game engines with which we are familiar. For instance, the AI subsystem updates the behaviours of the AI agents and then calls Physics in order to assess the effect of the agent's actions on the world. Further, within the Physics subsystem itself, the simulation first determines which game objects collide and then passes this data to the next stage which determines how to react to collisions.

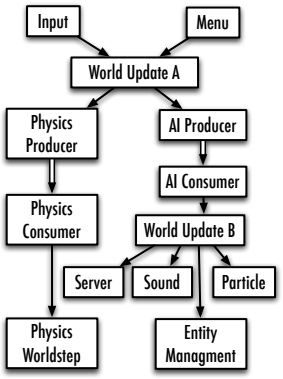

Fig. 1. The Cube 2-ext Dependency Graph

We observed that a dependency-based decomposition of interrelated subsystems maps nicely to the well known dependency graph model [4][5][6][7]. In this model, the program is represented as a directed acyclic graph where each node represents a computational task and directed edges between nodes represent dependencies or data flow. We can execute tasks in parallel by mapping tasks to threads while respecting these dependencies. In our rendition of this model dependencies are one of two types, either logical or dataflow. In a logical dependency the parent must complete before the child commences execution. A dataflow dependency defines a relationship in which data comprising a shared state between parent and child is passed from the parent to the child when it is safe for the child to process it. Figure 1 shows the dependency graph of our preliminary parallelization of Cube 2-ext. In our graphical representation an edge from parent to child represents a dependency, hollow arrow denoting dataflow and thin arrow indicating logical dependency.

Additional parallelism is possible in the case of a dataflow dependency. If the parent has produced enough data that the child can begin processing the child can begin execution before the parent completes. We exploit this possibility when parallelizing with the producer/consumer pattern.

In Cube 2-ext we applied this pattern in the ODE Physics and AI subsystems. The Physics subsystem determines whether there is a collision between objects in the world, boxes bouncing off floors and walls in our case, and packs the resulting collision data in an array. This process comprises the *producer* part. Then, the *consumer* processes each object to determine how to respond to the collision.

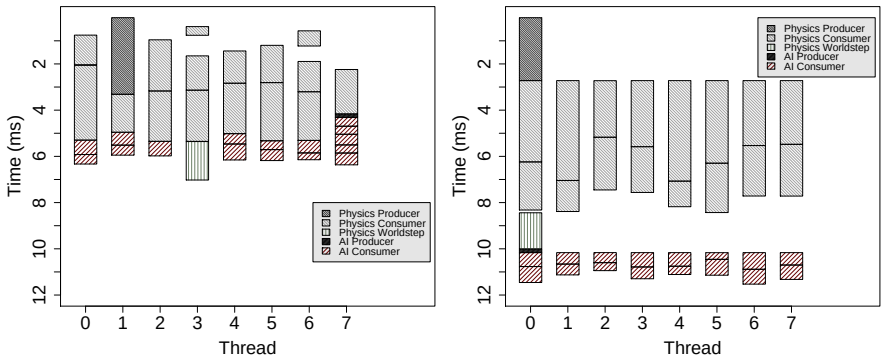

Fig. 2. Producers/consumers in Cube 2-ext(left) and Simulating parallel-for(right)

In the AI subsystem, the producer has been applied to the 'monster control' algorithm. The producer determines whether the monsters can 'see' each other and then makes this visibility data available to the consumer who processes it to determine the monster's next action. Since the updates performed on each object in the consumer steps are independent the work of the task can be divided over multiple threads without synchronization.

The parts of a task that are mapped to threads at runtime are referred to as its instances. Note that although this producer/consumer pattern with multi-instance tasks is very similar to a data-parallel pattern using a parallel-for or parallel do-all, there is an important difference. In parallel-for or parallel do-all the parallel computation cannot begin before all the required data is ready as all parallel tasks commence simultaneously. However, with the producer/consumer pattern the first child instance begins as soon as the parent readies the first batch of data.

Figure 2(left) illustrates a typical execution for a single frame of the above task graph on 8 cores, showing the work done by each of the eight threads (x-axis) over time (y-axis). The AI and Physics represent the vast majority of calculations performed so we have eliminated the others from analysis for clarity. The experiments that follow were done on a Mac Pro system with two Quad-Core Intel Xeon CPUs. The $7ms$ execution time presented in this figure is typical based on averages of thousands of executions. Observe that the Physics consumer begins well before the producer has finished.

To demonstrate the difference between the producer/consumer pattern and parallel-for, we emulated parallel-for by adding restrictions to the task graph such that consumers did not begin processing data until the producer completed. Figure 2(right) demonstrates a typical execution of the above task graph on 8 cores when these restrictions are introduced. Notice that it takes $12ms$ to execute the task graph, longer than the $7ms$ in the unrestricted case.

```
// 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 );
```

Fig. 3. Declaring a task graph

```
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);
 }
};
```

Fig. 4. A task with work kernel

3.2 Cascade

Although several existing PPEs compatible with C++ supported a dependency-graph pattern, such as Intel TBB [6], Cilk [4] and Microsoft PCP [7], none of them supported the concept of multi-instance tasks, which was key in our design. Dryad [5], a PPE that does support multi-instance tasks is targeted largely for database-style queries over computing clusters so it did not suit our needs either.

To fill this void we designed a new PPE, Cascade. Cascade began as a simple C++ PPE enabling the parallelization of producer/consumer and conventional dependency graph patterns with support for multi-instance tasks. In this section we describe the implementation of Cascade used for this paper and Section 4 details its future.

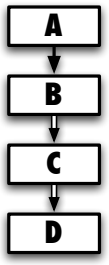

Fig. 5. Task graph from Figure 3

The key constructs in Cascade are *tasks*, *instances*, the *task graph* and the *task manager*. A Cascade task is an object that includes a work kernel and input/output channels. A work kernel is a function that encapsulates the computation performed by the task and an input or output channel is a memory buffer to facilitate communication. Tasks are organized into a computation graph according to their dependencies and the programmer specifies the producer/consumer semantics by connecting the tasks' input and output channels. Figure 5 shows an example dependency graph.

Figure 3 illustrates the C++ syntax used to map this dependency graph onto a Cascade graph abstraction. The programmer instantiates task objects, inheriting from one of Cascade's generic classes, specifying the number of instances to run in each task and the size of the batch that must be made ready by the parent before the child instance is launched. An example implementation of the class for task C from the example is shown in Figure 4.

As Figure 4 demonstrates, complexity of the code for creating parallel child instances is hidden from the programmer. The simplicity of semantics for creating producer/consumer relationships means that very few changes were required

to the code in Cube 2-ext. Producer and consumer work functions were wrapped in work kernels and made to exchange data via input/output channels.

3.3 Cascade Performance Evaluation

For experiments in this section we used microbenchmarks, similar to that shown in Figure 5. The work kernel was almost identical to that in Figure 4, except for it was configured to perform some amount of 'work' after receiving the batch and before sending it out. The amount of work was varied depending on the experiment. In all cases, we report average performance based on at least 20,000 trials. The standard deviation in all cases was under 5% with the exception of experiments where the work size was very small, taking only a few microseconds to execute, or the number of threads was large, more than six – in these cases operating system activity, interrupts, and non-determinism in scheduling contributed to the noise.

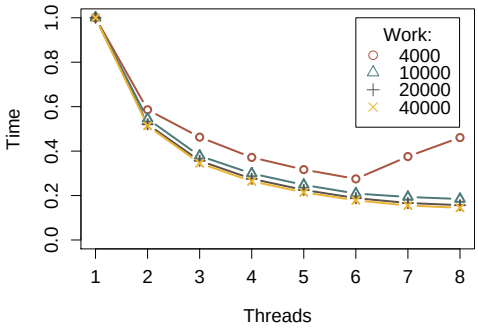

Fig. 6. Scalability with varying work.

3.4 Summary of Evaluation

In the first set of experiments we evaluated the scalability of Cascade. Figure 6 demonstrates the results. The x-axis shows the number of threads and the y-axis shows the time normalized to the single-threaded runtime. As the number of threads increases, we expect the running time to decrease. Scalability in Cascade may be limited due to contention on the *runqueue*, the queue of tasks whose dependencies have been satisfied and that are ready to be assigned to a thread. The degree of runqueue contention depends on the amount of work done in the work kernel. Contention varies inversely proportional to the amount of work done. To demonstrate this effect, we show several curves for different work sizes, displayed in the number of instructions. With a small work size, 4000 instructions, the overhead of synchronizing on the runqueue hurts scalability and so beyond six threads we see no benefit. In fact, we witness performance degradation. With larger work sizes, however, the workload achieves nearly linear scalability, the running time with eight threads being 0.14 at 40K instructions. This demonstrates that the

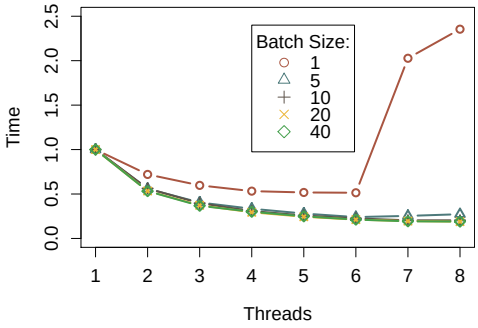

Fig. 7. Effect of batch size

choice of work size is important for applying Cascade. In Cube 2-ext the work size in producers and consumers was on the order of millions of instructions.

The batch size is a measurement of the amount of data the producer accumulates before sending it to a consumer. We next examined the performance effects of varying this parameter. The smaller the batch the more work needs to be done to process it and the more synchronization there is on the batch queue. Figure 7 shows that with a very small batch size scalability is poor, but it quickly improves as the batch size is increased and we see reasonable scaling with batch sizes as small as 5 or 10.

Our analysis of Cascade demonstrates that scalability depends on the amount of work done by the task and the size of the batches. With work size of 10K instructions and a batch size of 10 scalability approaches linear. Using a lock-free queue as opposed to a mutex queue improves performance and scalability dramatically when the work size is small.

Applying the producer/consumer parallelization style to the Physics and AI subsystems of Cube2-ext reduced the average time per frame from $15.5ms$ using a single thread to $7.8ms$ using 8 threads. This gives the 51% speedup that we have claimed. Despite these improvements, there is still vast numbers of opportunities for parallelization that we have yet to exploit.

4 Cascade Data Management Language: Inspiration from the Lessons Learned

Our experience with Cascade and Cube 2-ext underscored the fact that there is no universal remedy that address all the complexities of this domain. Discussions from previous sections promote the dataflow approach expressed through the producer/consumer pattern as a good model for organizing programs and managing their interactions efficiently in parallel. While this model was effective for many cases, in our experiments we discovered that it has inherent limitations.

Problems parallelizing the AI subsystem illustrates these limitations. In this subsystem, each agent performs multiple checks to ascertain if another agent is visible. These tests were implemented using Cube's original routines to walk the octree structure that held the world data. In theory, walking an octree is a read-only algorithm, but as an optimization collision results were being cached in static buffers. Concurrent writes to these cache structures caused corruption and resulted in crashes. These unexpected side-effects were not readily apparent in the public interface and were further obscured by the complexity of the system. Efficiently managing this complexity in parallel is a primary motivation for Cascade.

This situation is a perfect example of how changes to the system's state are outside of the dataflow model. Transformations of the system state or *side effects* are not exceptions in the video game domain, but generally represent the only feasible solution to many problems. The problems with AI could have been prevented with a more sophisticated scheduling mechanism that is aware of side effects and schedules accordingly.

Our scheduler, built on dataflow principles, had only limited facility for incorporating state change information as dataflow is essentially stateless outside of the contents of the communication channels. Logical dependencies serve as very coarse grained constraints, indicating that the state changes a parent makes must precede the child's execution. However, there is no direct association between a particular state change and the logical dependency and this dependency is only available between tasks and not between instances of the same task. Making specific information about these side effects available to the scheduler would allow it to correctly manage data accesses while promoting concurrency. Explicit and highly granular data dependencies allow for a number of optimizations particular to parallelizations such as an automatic Read-Copy Update [8] style solution to the multiple reader/one writer problem.

We have come to the conclusion that any effective parallelization strategy in this domain must provide rich and compact semantics for describing side effects in terms of transformations to the program state and provide first class support for specifying data dependencies. We believe that C++ is a poor choice to express these concepts, even mediated through a library. C++ pointers are of particular concern. Giving the programmer direct access to memory invalidates most guarantees of any system attempting to manage data. However, we acknowledge that C/C++ can be used to generate highly efficient serial code and is the industry standard language, especially in console development.

These observations have indicated the need for a system that requires the explicit expression of data interactions, contains primitives that reflect common data access patterns, allows for the decomposition of programs into dataflow tasks without an arduous amount of programmer effort and acknowledges the dominance of C++ in the domain. With this in mind we have begun work on CDML (*Cascade Data Management Language*) which addresses the problem of side effects inherent in procedural programming. State changes will be performed by composable transformations described in a declarative manner. Changes to system state are explicitly given as part of both the input and output to these function-like constructs. Larger units can be created by binding together these transformational constructs in a familiar procedural style. A program is created by joining these units together into an explicit dataflow graph. In this way all dependencies are expressible and many parallel optimizations are possible. This blending of styles allows the programmer to put emphasis on those best suited to the problem at hand.

CDML code will not actually be compiled, but instead translated to C++. To support the transition to CDML the embedding of C++ code will be supported. Unlike CDML, where stated dependencies are verified at translation time, the onus will fall on the programmer to ensure that all of the effects of C++ code on global state are declared.

While space limitations prevent us from giving a comprehensive example, we will illustrate how a programmer would take advantage of these concepts when implementing a single transformation. In AI systems it is common to pick a number of arbitrary members of a group as pack leaders. Expensive calcu-

lations, such as the line of sight testing in the AI subsystem discussed above, are performed only for pack leaders and the others inherit the results. In this case the player's actions may attract the attention of monsters by causing some kind of disturbance. If a pack leader can see the player then that disturbance increases the alertness level of it and its followers. A list of any sufficiently affected monsters is sent to another part of the AI subsystem.

In order to implement this process, the programmer will need to iterate over a list of entities and apply a transformation to each one. The following CDML example provides the code for this transformation.

```
iterator distributeAlertness ( Entity[] enemies as agent )
    views:  enemies, disturbanceLevel, playerPos < Input.check
    modifies: agent
    sends: hunters => AI.initiateHunt
{
    if ( agent.packLeader == true ) {
        if ( lineOfSight( playerPos, agent.position ) ) {
            agent.alertness += disturbanceLevel;
        }
    } else {
        agent.alertness = enemies[ agent.packLeader ].alertness;
    }
    if ( agent.alertness > HUNTINGTHRESHOLD ) {
        hunters << agent;
    }
}
```

CDML makes explicit the natural decision making process a programmer goes through when implementing an algorithm. First the scope of the transformation must be determined. Does it affect a single data item or collection? If the later, is only a subset affected or is the entire collection modified? Finally, is there an ordering to the modifications? In our example, we would like to process a collection structured as an index set, or array, and the ordering of updates is unimportant. The construct selected for this operation is the `iterator`, which corresponds in CDML to this data pattern. The signature of the transform defines the type accepted by this transformation (`Entity`) and defines a name (`agent`) for the individual element of the collection (`enemies`) being modified.

CDML will contain constructs adaptable to all the permutations of scope mentioned above and more. For example, if we only wanted to process a subset of the index set we could have specified a filter on the input. Filters are very important in CDML as they provide more information about the exact data affected by a transformation.

Following the signature is the *constraint block* which explicitly states the data dependencies of the transformation. For pure CDML code these dependencies are verified at translation time. There are several possible categories in the constraint block, but only `views`, `modifies` and `sends` are needed so the others are omitted.

The input required to perform the desired computation is specified in the `views` list. In this case we would like to read the other elements of the `enemies` array, the current `disturbanceLevel` and the players current position, `playerPos`.

In the case of `playerPos`, we would like to use it's value after it has been calculated by the input checking subsystem. Thus we use the '<' operator, to

denote that we would like `playerPos` 'as modified by' `Input.check`. This is one way of specifying logical dependencies in CDML.

Output is either in the form of state changes as specified by the `modifies` list or directly output through named data channels as specified by the `sends` list. In the example, we would like to change the current agent's `alertness` value and produce a list of agents that reach the threshold to be further processed by `AI.initiateHunt`. Sending through a channel is done using the '<<' operator.

The information required for the constraint block represents considerations a programmer already makes. These characteristics of a process are simple for a programmer to specify, but can be extremely difficult to derive computationally. Most computations in this domain can be expressed as operations on structured collections and since constraints are inherited when transformations are composed, any given transformation will have a manageable number of constraints.

To further ease specification of data constraints, the constraint block will be refined in translation to eliminate, if possible, over-constraint. For example, the programmer could have added `enemies` to the `modifies` list, instead of `agent`. This would be detected and reduced to `agent` alone. The requirement to list all constraints in the code itself, as opposed to deriving them during translation, is partially to facilitate the embedding C++ and other languages and to encourage a parallelization-friendly programming style.

5 Related Work

Cascade shares many similarities with other PPEs such as Intel's TBB [6], Microsoft's Parallel Computing Platform (PCP) [7], and Cilk [4]. Those PPEs are also built on top of general-purpose languages (C++, C#, C) and also support parallelization of computations expressed as a dependency graph. Due to the reasons detailed in Section 3.2, we could not use any of these PPEs for our project. Dryad [5], a PPE designed largely for database-style queries over computing clusters, provides rich support for parallel patterns and does include functionality similar to multi-instance tasks. Although we have not used any of the existing PPEs in our work so far, we envision using them in our future work to parallelize concurrent patterns that are not supported in Cascade. OpenCL [9], a new open standard for C-based parallel environments supporting both CPU and GPU programming, seems well suited to this.

Cascade shares similarity with domain-specific PPEs, in that it was designed with application specific needs in mind. Thies et. al. described a library targeted for streaming applications [10]. Galois [11] is a library targeted at irregular data structures. While these domain-specific libraries target a particular computation pattern, Cascade has a broader focus in considering an *application* domain, and so it must support multiple computation patterns and consider domain-specific performance needs.

Finally, in our work we rely on research into algorithmic patterns. Although we do not cite all work in this area due to space limitations, we note a relatively recent report from UC Berkeley [12] that categorized computation patterns ac-

cording to thirteen classes, or 'dwarfs'. Many of these 'dwarfs' are present in video game engines and so this classification system will be instructive in our further search for concurrent patterns in video games.

6 Summary

While parallelization of video game engines is a serious challenge faced by major video game companies [1], very little information describing details of this problem exists in the public domain. Our work sheds light on this subject and to the best of our knowledge this is the first detailed account of this problem in the public domain. We identified computations that could be parallelized by applying the producer/consumer pattern; this allowed us to achieve as much as 51% speedup on eight cores in AI and Physics subsystems. This process of parallelization and our analysis of existing tools informed the design of our PPE Cascade and more importantly shed light on the requirements for a set of tools that would truly encompass the patterns of this domain. Our experiences using Cascade showed that an even more general solution, one based around explicit data transformation and the expectation of side-effects is needed. To address this fact we are designing CDML, the Cascade Data Management Language, to leverage our constantly expanded and refined library and improve programmer performance while increasing program efficiency and correctness.

References

1. Leonard, T.: Dragged Kicking and Screaming: Source Multicore. Game Developers Conference (2007)
2. Chandra, R.: Parallel Programming in OpenMP. Morgan Kaufmann (2001)
3. Open Dynamics Engine. (http://www.ode.org/)
4. Blumofe, R.D., et al: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing **37**(1) (1996) 55–69
5. Isard, M., et al: Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. **41**(3) (2007) 59–72
6. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly (2007)
7. Microsoft Parallel Computing Platform. (http://msdn.microsoft.com/en-ca/concurrency/)
8. McKenney, P.E., et al: Read-copy update. In: Ottawa Linux Symposium. (2002) 338–367
9. Munshi, A.: OpenCL: Parallel Computing on the GPU and CPU. (2008)
10. Thies, W., et al: A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In: MICRO (2007)
11. Kulkarni, M., et al: Optimistic parallelism requires abstractions. (2007) 211–222
12. Asanovic, K., et al: The Landscape of Parallel Computing Research: A View from Berkeley. Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December **18**(2006-183) (2006) 19