Collection-focused Parallelism

Micah J Best
Nicholas Vining
University of British Columbia
{mjbest, nvining}@cs.ubc.ca

Daniel Jacobsen
Gaslamp Games
daniel.jacobsen@gaslampgames.com

Alexandra Fedorova
Simon Fraser University
fedorova@cs.sfu.ca

Abstract

Constructing parallel software is, in essence, the process of associating 'work' with computational units. The definition of work is dependent upon the model of parallelism used, and our choice of model can have profound effects on both programmer productivity and run-time efficiency. Given that the movement of data is responsible for the majority of parallelism overhead, and accessing data is responsible for the majority of parallelism errors, data items should be the basis for describing parallel work. As data items rarely exist in isolation and are instead parts of larger collections, we argue that subsets of collections should be the basic unit of parallelism. This requires a semantically rich method of referring to these sub-collections. Sub-collections are not guaranteed to be disjoint, and so an efficient run-time mechanism is required to maintain correctness. With a focus on complex systems, we present some of the challenges inherent in this approach and describe how we are extending Synchronization via Scheduling (SvS) and other techniques to overcome these difficulties. We discuss our experiences incorporating these techniques into a modern video game engine used in an in-development title.

1. Introduction

Parallel programming is the new reality for an ever increasing number of software developers. All the problems inherent in conventional software development are still present, but with a new series of challenges. Abstractions to allow programmers to easily construct serial software have been explored for several decades, but complementary abstractions for parallelism are still in their infancy. Fundamental questions about the structure of parallel programs that go beyond programming languages and caching strategies are still up for debate. Nearly every domain of software has been affected in some way and those attempting to wring parallel performance out of complex systems such as web browsers or video games face the steepest challenge.

By examining many proposed parallelism models, it is clear that many in the field, both researchers and practitioners, conceive of parallelism as a subdivision process applied to code. In this traditional model, code is broken into chunks and assigned to processors. This misses one core aspect of parallelism: the movement of data creates overhead on the processor and across the bus, and it is this accessing of data that causes errors.

In this paper, we expand on and refine the argument that data items should be the basic unit of parallelism (Section 2). Data items, in the vast majority of software, are not isolated entities; rather, they are part of larger *collections* within some kind of structure, be it a simple array or a complex graph. Given that the goal of parallelism is to make computation as separable as possible, we argue that the basic unit of parallel programming should be (potentially overlapping) subsets of collections, or *sub-collections* (Section 3). We describe a commonly occurring parallel pattern in this model (Section 4) and give details of how we are realizing a portion of this concept by extending Synchronization via Scheduling[8] (Section 5).

Most parallelism frameworks focus on either single-algorithm applications, such as scientific concerns, or provide tools for the parallelization of existing conventional serial programs. Frameworks like OpenMP[3] and Intel's Thread Building Blocks [23] provide programmers with constructs to 'chop up' blocks of code, generally loop iterations, for parallel execution. Facilities for the manual protection of data accesses are provided. Languages such as Jade[24] and Cilk[15] are more sophisticated in their approach to parallelism, but are again based on traditional serial programming styles with augmentations to facilitate splitting the flow of execution onto multiple processing contexts. There exist several alternate programming models such as the popular MapReduce[14] which are not based on existing serial models, but are limited to problems with a very specific set of properties and not feasibly useful for parallelizing complex systems.

We focus our attention on video game development, as it represents one of the major areas in consumer software where efficient solutions for rapid parallelism are required. Video games form a complicated collection of interactive, highly responsible complex systems with heterogeneous workloads, and their creation involves multiple complex algorithmic patterns and a team of individuals with varying skill levels and familiarities with parallel programming. As such, traditional methods often do not scale to the entire software program. Our work is focused on solving these problems, and in making Synchronization via Scheduling a practical tool for video game development on upcoming hardware such as the Playstation 4 and next-generation personal computers.

2. Focus On Data, Not Code

Numerous proposals have been offered on how to organize parallel programs. A complete survey of the existing literature is beyond the scope of this work. Complex consumer software, such as video games, commonly use one of two models: the threading model[16], in which parallel streams of execution communicate through shared memory, or the task-graph model, also called the job model[5], which breaks the program code into discrete 'tasks' that are organized into a partial order respecting computational dependencies.

However, we argue that this focus on how to divide the *code* is asking the wrong question. It is not the execution of code that causes the problems, or yields the rewards, in parallel programming. For example, consider the costs of moving data on the 16-core, 4 NUMA domain, AMD Barcelona processor[12]. Accessing the L1 cache takes a mere 3 cycles. Accessing a remote L1 cache increases the cost to 121 cycles, and accessing a remote memory node can be as high as 327 cycles. These potential hobbling penalties have, of course, not gone unaddressed by the research community and many optimizations have been suggested [4][22]. However, we advocate that the programing model be aligned as closely as possible with an execution model that minimizes costs and maximizes throughput so that programmers with no particular expertise will tend to write automatically optimizable software.

In the context of a video game, the accumulation of cache miss penalties can significantly impact performance. Furthermore, the vast majority of errors in parallel programming are caused when data is accessed in an order that the programmer did not foresee. These errors induce deadlock, livelock, corruption by simultaneous writes, and many other forms of undesirable behavior.

It follows that correct, efficient parallelism requires a schedule that organizes not the distribution of code across multiple processors, but the movement of data. Code and data have a symbiotic relationship, but most parallelism models explicitly parallelize the execution of code; this in turn implicitly organizes the movement of data. We argue that this should be inverted in that the first consideration is the movement of data and the applicable code is subordinate to this.

The dataflow model[17], as the name suggests, incorporates notions of the movement of data through the system as a sequence of steps. Each step modifies the data in a particular way until the operation is complete. The dataflow model, while compatible with the task-graph model, is a bad fit for complex systems that rarely exhibit the type of homogenous, pipelined execution that dataflow processing thrives on. When parallelizing complex systems on shared memory architectures, 'sending' data from stage to stage is considered to be too expensive; the programmer then resorts to pointers or similar indirect methods in order to describe the work that must be performed. This indirection can create multiple problems, including state conflicts.

Data-Informed Scheduling [7] has been proposed to address the movement of data. We will expand on these ideas, and suggest (in the next section) a method for increasing the semantic value of indirect references for safer, easier parallelism.

3. Collections and Sub-Collections

The term *data* implies a collection of individual elements, which we typically view as a collection. We then speak of running data through an algorithm, in which we apply the same process to different instances of a problem. Additionally, items in a collection of data often have non-trivial relationships with each other; the processing of one datum, in anything but the most embarrassingly parallel of algorithms, often involves the state of other pieces of data.

Almost all implementations of the stencil pattern[21], for example fluid simulation[9], involve evaluating multiple cells in a grid simultaneously. Some cells are written, and some cells are read. For these algorithms to be executed correctly, each subset of the grid must be evaluated atomically; that is to say, no grid members can be changed between the start of the evaluation and its completion.

Often the subset of concern is not so easily described. In mesh refinement[11], the collection of data to be refined takes the form of a connected, triangular graph. The refinement processes deals with 'bad' triangles that have undesirable angles. In order to fix these bad triangles, it must not only atomically deal with the three vertices in the triangle, but the neighborhood surrounding it as well.

Of course, there are many algorithms, the so-called embarrassingly parallel ones, that only examine one item at a time. In raytracing, for instance, each pixel can be processed independently of any other pixels. This can be seen as a degenerate form of subset processing.

Working with subsets of collections is so common that it shows the necessity for support for *first class collections* in a parallelism framework. By this, we mean that the concept of a collection should be understood by the language, the compiler/interpreter, and the runtime. As the above discussion indicates, collections are not the solution by themselves. To fully capitalize on the observed patterns of parallelism, support is needed in the framework for *describing* and indirectly referring to subsets of a collection that is understood by all levels of the framework.

It should be noted that most lower level languages, C++ for example, do not support first class collections. The concept is reflected in their standard libraries, but their compilers understand only arrays and pointers/references. All semantic information contained in the statement `myVector.push_back(foo)` is lost at compilation time as it is expanded into a series of variable modifications and pointer assignments.

Consider the problem of spatial data structures in video games. Different data structures (BSP trees, octrees, quadtrees, etc.) may be used for different tasks, but all data structures share the common properties of being a hierarchical, tree-like structure whose area is the aggregate area of its children and whose leafs are actual objects. We focus on one of these tree-like structures where one walks the tree to find objects that are descendants of the node representing a given space.

A programmer using one of these structures thinks *I want to deal with all objects in this specific area*, but writes statements like `if( curnode.child[i].x < target.x ) { curnode = curnode.child[i] }`. Even if the tree walk is part of a library, all static analysis can determine is that a number of pointers/references are read and eventually that a reference of a different type is modified. Even while the program is being written, semantic information is lost. Obviously, barring the invention of purely natural language programming, some loss of this type is going to occur, but when patterns are common and important they should be reflected in the language.

Consider instead the following pseudo-code: `targetObjs = octree.subcollection(bBox)`. If the collection `octree` was a first class language entity, the compiler/runtime would be able to perform a number of optimizations and implement a number of safety measures, including separating the derivation of `targetObjs` from modification to its members, ensuring the derivation of `targetObjs` was atomic without the programmer having to worry about complicated lock discipline, using static analysis to optimize the filtering and protection of `octree`, easily associating changes to `targetObjs` with changes to `octree`, and scheduling based around sub-collections such that no overlapping sub-collections are co-scheduled.

Technically, subset parallelism support could be implemented as a library for a lower level language for some benefit. However, this library must be part of the parallel scheduling system to be effective and the majority of opportunities for static analysis are lost. Additionally, its use would almost certainly require the programmer to follow a set of conventions and protocols. Low level languages, such as C++, are notorious for allowing a programmers to 'cheat around' such systems without realizing it and creating numerous headaches in the process.

The existing work that most closely embodies the principles we've discussed is Concurrent Collections[13] which, as the name suggests, bases their parallelism model on collections and requires the programmer to add semantic information in the form of tags on data items. However, the focus of this work is on the single items in a collection, as opposed to sub-collections, and its dataflow based model makes parallelizing a complex system impractical.

In the next section, we illustrate some uses of sub-collections in a real-world complex system and specify the particular subset of this pattern we are implementing. In Section 5 we will detail how we intend to perform some of the above optimizations. ,l.

4. Isolate-Modify-Release

Clockwork Empires is an in-development video game title in the pre-alpha state of development. Similar to *Dwarf Fortress*[1] and the *SimCity*franchise [2], *Clockwork Empires* is a sandbox simulation featuring hundreds of non-player character agents (NPCs) with complex behaviors and interactions. Other complex entities such as monsters, buildings and animals may exist in the world, each with their own internal logic. While standard parallelization primitives, such as a task-graph model, may be suitable for other in-engine processes such as rendering, parallelizing the highly heterogeneous and interconnected simulation is a task that exceeds the bounds of most models. There is a plethora of data describing the world from the attributes of agents to numerical representations of geography. Any data item can be modified by the logic associated with any agent at anytime. In order to be responsive, one game simulation frame must be computed in tens of milliseconds.

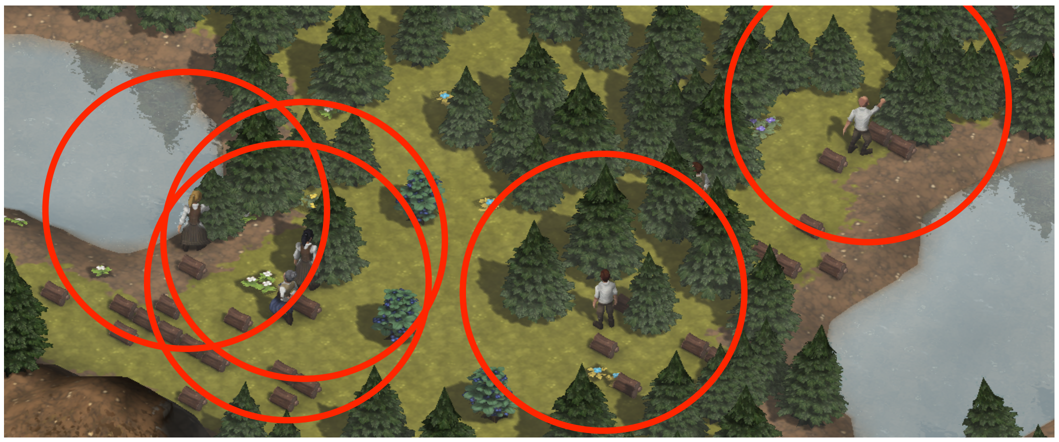

Figure 1. Clockwork Empires screenshot with each NPC's area of concern indicated

Given that very few members of the development team are parallelism experts, the core parallelism constructs in the engine must be easy, automatic, fine grained and very, very efficient.

The set of data comprising the working set for the game is organized into collections, all of which are shared. We will highlight two collections to illustrate the specific type of collection-focused parallelism we are implementing.

The first example is that of the 'Spatial Dictionary', which is responsible for tracking the locations of all physical objects. Since the game simulation occurs on a grid, the dictionary is organized as two-dimensional set of cells. Each of these cells is associated with a list of physical items. Additionally, each cell has a number of mutable attributes indicating whether a cell is passable, dangerous or contains useful resources. Agents are responsible for moving themselves between grid cells, and the spatial dictionary is the primary conduit for an agent to discover and interact with other entities in the world.

At a high level, the process of an agent interacting with the Spatial Dictionary is as follows:

1. The agent asks *What objects are in this area?*
2. The dictionary returns a reference to the objects in the given area.
3. The agent selects an entity within this reference and interacts with it, either modifying it directly or marking it as the target or destination for a longer process.

Figure 1 shows a screenshot of a number of agents. The circles represent the aproximate area that they will query looking for the particular type of object they are interested in interacting with. Notice that while many of these areas overlap, several are also completely disjoint.

For correctness and program stability, this process must be atomic; while the NPC is making its selection the objects in the given area must remain unchanged. However, any object outside of the specified area can be modified without violating the process. This fact, and the vast number of requests made to the Spatial Dictionary, make an efficient coscheduling strategy that respects the need for atomicity of paramount importance.

The second key collection in Clockwork Empires is the Job Blackboard. The behavior of agents is not directly controlled by the player; instead the player directs modifications to the world by designating the floor plans for buildings and other high level construction activities. The NPCs will then react to these instructions and, based on their individual characteristics and logic, attempt to carry them out. Additionally, the game itself will create situations for the NPCs to react to. Every opportunity for an agent to modify the world, itself, or another NPC, is a 'job' - not to be confused with the parallel decomposition of computational work with the same name. Each job is posted to the Job Blackboard which every NPC has access to.

At a high level, an agent's interaction with the Job Blackboard is as follows:

1. The agent asks the Job Blackboard for all jobs that could apply to it.
2. The Job Blackboard returns all the jobs applicable to the agent.
3. Using its individual characteristic to rank the items in this reduced list, the agent selects a Job.
4. The agent modifies the job to indicate its participation in the job, or removes the job from the list.

As with the Spatial Dictionary, this process must be atomic. Any parallelization strategy must make good use of the fact that the list of applicable jobs may be different for different NPCs.

The process involved in accessing the Spatial Dictionary and the Job Blackboard have a common underlying structure. In both cases, the the collection is first filtered for a particular set of results based on criteria specific to the collection and its contents. In effect, the sub-collection is *isolated*. In the second step, the initiating entity then *modifies* the resulting sub-collection, independent of the original collection. There is a third, implicit, step which satisfies the need for atomicity. The data in the sub-collection is 'held' unchanged for the length of the process and so finally it must be *released*.

Analyzing these processes leads us to believe a parallelization strategy to satisfy the needs of this simulation should directly support this *isolate-modify-release* pattern. An analysis of other collection-based accesses in the game supports this assertion. In very few cases does a process need to view an entire collection or retain references for use outside of the process.

Upon further analysis, we see that many common algorithms that operate on collections of data also employ this template. In fact, both of the examples at the beginning of Section 3 employ this pattern. A stencil is an isolation of the grid, where only the cells in the stencil are modified. Once these modifications are complete, that step in the algorithm is complete and the cells can be released for use by other stencil instances. In mesh refinement, the neighborhood containing a bad triangle is isolated and its contents are modified. Once that is complete, the algorithm has no further use for that particular neighborhood.

In the next section we will discuss how we are implementing the isolate-modify-release pattern for use in *Clockwork Empires*.

5. Implementation

Clockwork Empires is written, as with most games, in C++. The majority of NPC/entity logic is written in Lua[18], a language specifically designed for embedding and popular with video game developers. The parallelism component is a modified and extended version of the Cascade[6] runtime engine.

Cascade was originally a task-graph based parallelism framework using dataflow primitives. Static analysis is used to determine the dependencies of the task graph, and to supply information to the runtime scheduler in order to determine if these dependencies were necessary. The runtime component would examine tasks which, according to the static analysis, may have accessed the same data. If the execution-determined parameters to these tasks indicated that their access were actually disjoint, it would allow them to be run in parallel. This is a process called Synchronization via Scheduling, and we extend it for our uses in implementing *isolate-modify-release*.

In addition to adding support for collections we have made two major changes to the way Cascade works. We have changed it from being task-graph based, to being data-centric. Additionally, we have replaced dataflow with message passing using constructs similar to the Actor Model[19]. However, our goal is not to advocate for Cascade as there are many excellent parallel programming frameworks. We provide these details to illustrate the context in which we are implementing the concepts we have presented.

```
name "chicken"

state
<<
    string name
    GridPosition gridPosition
    int renderHandle
>>

receive Create()
<<
    name = "Eggers McFowlbeak"
    ready()
>>
```

Figure 2. An example entity defintion

As argued in Section 2, the focus of parallelism should be on the data items themselves. Instead of defining tasks, we define data items and attach a set of behaviors to them in a manner reminiscent of object-oriented programming. Collections are represented as objects to be parallelized with their own set of behaviors which provide facilities for data-parallelism. In the same section, we discussed some of the pitfalls in using dataflow in complex systems such as video games.

Message passing[25] can be seen as a generalization of dataflow that has higher semantic value and can directly support the manner of indirection that programmers tend use when applying dataflow to these kind of complex systems. In this system, an entity has access to other data only through the message passing mechanism and this provides an abundance of opportunities for static analysis and parallel optimizations. A simplified definition of an entity can be seen in Figure 2 which shows the values associated with entity and a simple message-receipt body.

The majority of these changes tended to be more 'programmer facing', in terms of changes to language, while the runtime engine has remained largely intact. The task mechanism in Cascade has been repurposed for the new data-focus by constructing an executable unit consisting of a data item and a received message. A description of the conceptual replacement for the task-graph is beyond the scope of this work and is a key component of our future research.

A great deal of similar work on maintaining collections in parallel can be found in the exploration of in-memory databases[20]. However, at least for video game this work is not immediately usable. Many features of these databases, such as rollback support are not required and the responsiveness requirements for games, and other systems with constant user interaction , are much higher. Games have an approximate time budget of 16ms in order to construct and draw a frame.

Consider the following example call to the Spatial Dictionary, which employs the isolate-modify-release pattern:

```
result = query(gameSpatialDictionary,
               allObjectsInRadiusRequest,
               state.gridPosition, 5);
```

The following description details the aspects of the implementation that we are building to support parallelizing this call.

The first major aspect to discuss is the variable `result`. This variable, a collection, cannot be stored and has a lifespan only consisting of the local scope. When the local scope is complete, the release step is performed automatically. This was a major request of the programmers, and is in keeping with our desire to make parallelism as automatic and transparent as possible. The variable itself represents not a list, but access to the collection itself to avoid unnecessarily copying. The interface to the sub-collection prevents access to data items that do not fall under the given query.

In terms of scheduling, the call to `query` causes the currently running code to yield to the scheduler. The scheduler makes a record and forwards the message to the Spatial Dictionary collection.

At this point, a modified version of Synchronization via Scheduling is used to resolve the queries and restart the yielded behavior. The Spatial Dictionary, like all collections in the system, maintains a *signature* representing the items in the collection that are currently in use. This signature, essentially a Bloom filter[10] with a single hash function, is a fixed-sized bit string where each bit represents a certain subset of the grid. A hashing function is used to determine which subset a particular cell is a part of.

The collection first assembles a signature for the query. For each cell applicable under the query the bit of its subset is set to 1 and is left as 0 otherwise. The collection now has a compact representation of the sub-collection represented by the query. This query signature is then compared, via logical *and* to the signature of the collection. If the test is negative, in that all the resulting bits are zero, then the sub-query does not overlap with any queries not yet released. In this case the query signature is added to the collection signature with logical *or* and the sub-collection representation is prepared for returning to caller. Additionally, a counter for the number of unreleased sub-collections is incremented.

If the test for signature collision was positive, the query is queued. Modifications to a Bloom filter are one-way operations, in that once a signature has been added this operation cannot be reversed. When a query is released, the internal counter is decremented. When this counter reaches zero, the collection stops accepting queries temporarily and resets each bit in its signature to zero. At this point the queued queries are evaluated. Queries whose signatures test negatively are satisfied and returned to the sender, and queries that test positively are requeued for future evaluation.

Note that with use of atomic operations on the signatures this taking of sub-collections can be done in parallel. The only point at which serial execution is required is the clearing of signatures when the internal counter drops to zero.

In this way, we are building support for safely executing the isolate-modify-release pattern in a manner that is both friendly to programmers and efficient to execute.

6. Conclusion and Future Work

In this work, we have outlined our argument for collection focused parallelism, given a high level overview of what this entails, and documented our efforts to build support for these patterns into current real-world software.

Full support for collections beyond isolate-modify-release will involve a number of different algorithms and techniques. The algorithm detailed in Section 5 still does not meet all the requirements for production use. It currently has no facility for determinism, which is of paramount importance for networked gameplay, and yielding during the call to query breaks the guarantee of atomicity for the entire local scope that was one of main virtues of Cascade. We are also expanding on the syntax for taking sub-collections to include the ability to specify queries using collection specific atoms such as `max` and `nearest` in a fashion reminiscent of SQL. At the time of this writing we have a prototype using C++11 lambda expressions, but there are many details left in order to ensure the safety of these primitives and their applicability for static analysis.

References

[1] Bay 12 Games: Dwarf Fortress `www.bay12games.com/dwarves`.

[2] SimCity `www.simcity.com`.

[3] The OpenMP specification for parallel programming `http://www.openmp.org`.

[4] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 126–138, New York, NY, USA, 1993. ACM.

[5] Johan Andersson. Parallel futures of a game engine `http://repi.blogspot.com/2009/11/parallel-futures-of-game-engine.html`.

[6] Micah Best, Alexandra Fedorova, Ryan Dickie, Andrea Tagliasacchi, Alex Couture-Beil, Craig Mustard, Shane Mottishaw, Aron Brown, Zhi Huang, Xiaoyuan Xu, Nasser Ghazali, and Andrew Brownsword. Searching for concurrent design patterns in video games. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 912–923. Springer Berlin / Heidelberg, 2009.

[7] Micah J Best, Shane Mottishaw, Craig Mustard, Mark Roth, Parsiad Azimzadeh, Alexandra Fedorova, and Andrew Brownsword. Schedule data not code. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism*, HotPar'11, Berkeley, CA, USA, 2011. USENIX Association.

[8] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 640–652, New York, NY, USA, 2011. ACM.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[10] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[11] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Br uno Levy. *Polygon Mesh Processing*. AK Peters, 2010.

[12] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, pages 43–57, 2008.

[13] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasirlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, August 2010.

[14] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 523–534, New York, NY, USA, 2010. ACM.

[15] Blumofe et al. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[16] J. Gregory. *Game Engine Architecture*. Ak Peters Series. A K Peters, Limited, 2009.

[17] J. Herath, Y. Yamaguchi, N. Saito, and T. Yuba. Dataflow computing models, languages, and machines for intelligence computations. *IEEE Trans. Softw. Eng.*, 14(12):1805–1828, December 1988.

[18] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Luaan extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652, June 1996.

[19] Shams M. Imam and Vivek Sarkar. Integrating task parallelism with actors. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 753–772, New York, NY, USA, 2012. ACM.

[20] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011.

[21] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[22] Guilherme Ottoni and David I. August. Communication optimizations for global multi-threaded instruction scheduling. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 222–232, New York, NY, USA, 2008. ACM.

[23] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[24] Martin. Rinard and Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[25] Nicholas Vining. The threads that bind us. *Game Developer Magazine*, April 2012.