An Implementation of the Codelet Model

Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao

University of Delaware, 140 Evans Hall, Newark, DE, USA,
{jodasue,szuckerm,ggao}@capsl.udel.edu,
WWW home page: http://www.capsl.udel.edu

Abstract. Chip architectures are shifting from few, faster, functionally heavy cores to abundant, slower, simpler cores to address pressing physical limitations such as energy consumption and heat expenditure. As architectural trends continue to fluctuate, we propose a novel program execution model, the Codelet model, which is designed for new systems tasked with efficiently managing varying resources. The Codelet model is a fine-grained dataflow inspired model extended to address the cumbersome resources available in new architectures. In the following, we define the Codelet execution model as well as provide an implementation named DARTS. Utilizing DARTS and two predominant kernels, matrix multiplication and the Graph 500's breadth first search, we explore the validity of fine-grain execution as a promising and viable execution model for future and current architectures. We show that our runtime is on par or performs better than AMD's highly-optimized parallel library for matrix multication, outperforming it on average by 1.40× with a speedup up to 4×. Our implementation of the parallel BFS outperforms Graph 500's reference implementation (with or without dynamic scheduling) on average by 1.50× with a speed up of up to 2.38×.

Keywords: Execution model, runtime system, manycore, multicore

1 Introduction

While the advent of many-core chips for mainstream computing is still yet to come, many-core *compute nodes* have become common in new supercomputers. A typical compute node may have 32 to 64 threads (or more), spread across several sockets. In addition, non-uniform memory access (NUMA) has become the new standard for shared-memory nodes. As thread counts increase, memory and even compute resources (such as FPUs) per core are becoming more scarce as seen in the IBM Cyclops-64 [8]. On-chip and off-chip bandwidth must also be seen as scarce resources requiring intelligent allocation among computing units. Furthermore, due to diminishing feature sizes, reducing power consumption has become a predominant obstacle forcing chip manufacturers to simplify the design of individual cores, removing power-hungry branch predictors and cache prefetchers.

The increase in available parallelism found in shared-memory nodes has lead to hard-to-exploit program execution models (PXMs). These models are

still based on the sequential Von Neumann model. The semantics of traditional threads makes it extremely difficult to guarantee correct execution, and race conditions are the dreaded companion of any parallel programmer [17]. However, there are PXMs which emphasize properties such as the isolation of execution and the explicit declaration of producer-consumer relations like the dataflow program execution models [9].

This paper evaluates an implementation of the codelet model [24], a fine-grain PXM inspired by dataflow. We implemented the model using a runtime system, DARTS. We evaluate its usefulness through two case studies comparing DARTS against OpenMP on square matrix multiplication and the Graph500's breadth first search benchmark.

Section 2 provides the necessary background to understand how the codelet model works, and how DARTS implements it. Section 3 presents our two case studies, and describe in details how each problem was decomposed. Section 4 presents the related work. We conclude in Section 5.

2 Background

2.1 The Codelet Model

While most prevalent execution models in their current state are struggling to scale to future machine's peak performance, we propose the codelet execution model. The codelet PXM differs fundamentally from its Von Neumann based competitors, as it draws its roots from the dataflow model.

The Codelet Abstract Machine Model The codelet abstract machine model (AMM) consists of many nodes connected together via an interconnection network. Each node is expected to have several chips containing hundreds of cores. Interconnects with varying latencies will connect components at multiple levels. We envision two types of cores. The first is a simple Computation Unit (CU) which is responsible performing operations. The other is a Synchronization Unit (SU) which is responsible for steering computation. Figure 1 depicts the proposed abstract machine model.

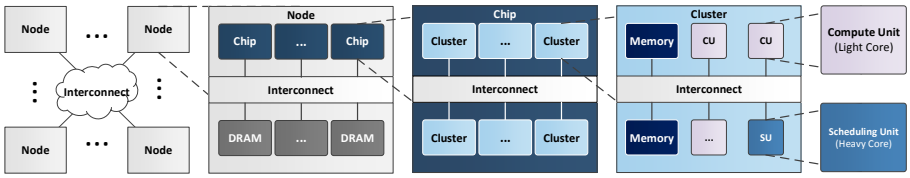


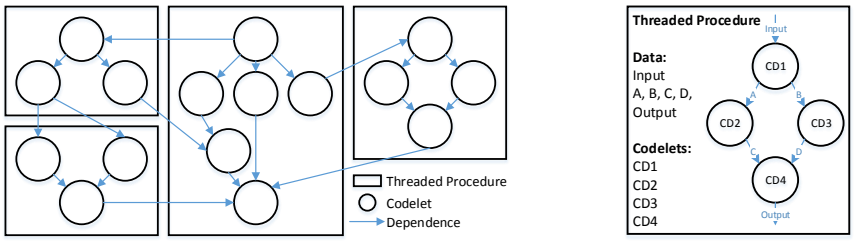
Fig. 1. The codelet abstract machine model

Codelets: Definition and Operational Semantics

Definition A codelet is a collection of machine instructions which are scheduled "atomically" as a non-preemptive, single unit of computation. In the codelet PXM, Codelets are the principal *scheduling quantum*. Codelets are expected (not required) to behave functionally, consuming inputs, working locally, and producing output leaving (ideally) no state behind. A codelet will only fire when all of the resources it requires are available. This entails having the necessary data local prior to execution, eliminating the need to hide latencies due to accessing remote data.

Operational Semantics Codelets differs from a traditional task in their invocation. Similar to a dataflow actor [9], a codelet is fired once all of its dependencies (or *events*) are met. An event primarily consists of the data (i.e. arguments) required by the codelet to perform its operations; however events may include the requirement of any particular shared resource or condition such as bandwidth, power, etc. Each codelet has a requisite number of event which must be satisfied before execution. A Codelet's output is not atomic, meaning a codelet is capable of producing data, signaling other, and continuing execution differing from macro-dataflow actors.

Codelet Graphs Codelets are linked together to form a codelet graph (CDG). In a CDG, each codelet acts as a producer and/or consumer. An initial codelet may fire, producing a result which multiple codelets can consume, giving way for more codelets to execute. Since codelets are linked together based on data dependencies, a CDG may benefit from the same properties as a dataflow graph. This includes the explicit view of parallelism and determinate execution. Figure 2(a) provides an example of various codelet graph instances. Note that codelets can signal other codelets outside of their CDG.



(a) Multiple codelet graphs linked together. (b) A Threaded Procedure

Fig. 2. Examples of codelet graphs (CDGs) and threaded procedures (TPs). TPs are CDG containers, and allocate the space required to hold inputs, outputs, and intermediate results.

Parallel Constructs

Asynchronous Functions Asynchronous functions are called Threaded Procedures (or TPs) in the codelet model. They are closely related to EARTH's [23] TPs. Much like its ancestor, the codelet model's TPs are containers for a codelet graph. A TP also features a *frame*, which holds the inputs passed to it, the resulting output, and values local to the contained CDG, as illustrated in Figure 2(b). A TP is invoked functionally, and exists in memory until all of the codelets in the CDG have finished executing. When a TP is instantiated, it is bound to a single cluster, equally binding the codelets. Prior to instantiation, a TP closure may be load balanced between clusters. In this way we utilize hierarchical parallelism, while providing some form of locality.

Loops Loop parallelism is a crucial form of parallelism and a cornerstone in most useful parallel execution models. As such the codelet model provides a special loop construct enabling a CDG to be executed in successive iterations. Loops without loop-carried dependencies (for all loops) can be executed completely in parallel.

2.2 A Codelet Runtime

An execution model needs to be enforced to be useful. While using a combination of hardware and software is preferable to achieve high-performance [15], it is less time-consuming to implement everything in software that runs on off-the-shelf hardware. This section presents the Delaware Adaptive Run-Time System – DARTS.

Objectives There already exists runtime system implementations of the codelet model currently under development, such as SWARM [16]. While they reuse the codelet object as the central unit of computation, they generally tend to stray from the original specification (see Section 4). Hence, our goal is to build a runtime system which will be true to the codelet model, but also serve as a research vehicle to evaluate and advance the model itself.

Faithfulness DARTS is implemented to be faithful to the base codelet model. Hence, it employs codelets as the base unit of computation, but it also requires the use of threaded procedures as the containers for codelets.

Portability and Modularity DARTS is written in C++. This language is low-level enough to ensure full control of the underlying hardware, while offering an object-oriented model which encourages modularity and component reuse. The latter point is important as we intend to use DARTS to explore and stretch the limit of the codelet PXM.

Implementation

The Codelet Abstract Machine The codelet AMM described in Section 2.1 requires a concrete mapping to a physical machine. We reused the `hwloc` library [2] to obtain the topology of the underlying computation node. Once discovered, the runtime decides how to decompose the hardware resources (processing elements, caches, etc.) according to the user-programmer's selection of preset configurations. For example, one can elect a single socket of an SMP system to act as the AMM's cluster, and a single core on the socket to act as the synchronization unit. New mappings can easily be added to the description of the codelet AMM.

As described in the Section 2.1 each cluster contains two types of cores, one SU and several CUs. Each core runs one of two types of schedulers. Each CU runs a micro-scheduler, responsible primarily for executing codelets. An SU runs a Threaded Procedure scheduler (TP scheduler) which is responsible for load balancing TPs between clusters, instantiating codelets, and distributing codelets within a cluster. Having designed DARTS with modularity as a guiding principle, each scheduler is capable of running several different scheduling algorithms. For the scope of this work, we use a work-stealing policy similar to Cilk [1] to perform load balancing between TP schedulers. Within a cluster, micro-schedulers use a centralized queue to get work.

Codelets The codelet specification is implemented as a `Codelet` class containing a synchronization slot (*sync slot*) and a method called `fire`. The sync slot is used to keep track of the outstanding dependencies. The codelet class must be specialized (i.e. derived) and can be instantiated once the `fire` method is expressed. `fire` is applied on a codelet by a CU's micro-scheduler when the codelet is chosen for execution.

Signaling Each sync slot is initialized with the number of events the codelet requires to run. Codelets within a TP are known statically and can be accessed through the TP frame. The address of a codelet is required to signal codelets outside a TP, and can be provided at runtime. DARTS implements a form of argument fetching dataflow [10], as the act of signaling is dissociated from passing data. For this reason data is written first, and then a codelet is signaled.

Asynchronous Functions DARTS uses TPs as the main way to instantiate portions of the computation graph. They act exactly as explained in Section 2.1. Much like codelets, threaded procedures are implemented as classes that must be derived by the programmer. The `ThreadedProcedure` class embeds an active codelet counter (to know when all the codelets it contains have finished executing), a pointer to a parent TP (the one which invoked it), and a member function to add a new codelet within the TP. The address of the TP frame (in practice, the pointer to the TP instance) is passed along to codelets so that they can access shared variables. Once the last codelet of an instantiated TP has finished running, the TP is deallocated along with all the codelets it contained.

Loops Currently, DARTS implements three types of loops, a serial loop, a TP parallel for loop, and a codelet parallel for loop. Parallel for loops (*forall*) prohibit loop-carried dependencies, conceptually executing all iterations in parallel.

Practically, the iterations are executed when sufficient hardware is available. The TP forall creates a TP for each iteration of the loop, permitting the iterations to run on any cluster. The codelet forall loop adds all the iterations to the invoking TP, pinning them to a single cluster.

Conceptually, a codelet loop requires two codelets, as shown in Figure 3. These "loop controllers" act as a source and sink. The source codelet is signaled normally. Upon execution, the source schedules copies of the enclosed CDG. After the loop body has finished executing, the "leaf" codelets of each iteration signal the sink codelet. Once all iterations have completed, the sink codelet deallocates the copied iterations, and signals the next codelet in the CDG. In practice, the source and sink codelets which control the loop are merged into one, to avoid useless memory allocations. Once it has performed its source action, the loop controller is reset to the number of "leaf" codelets multiplied by the number of iterations prior to scheduling the loop iterations. This approach is sufficient for supporting nested loops.

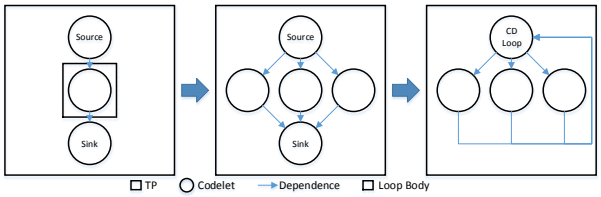


Fig. 3. Loops in the codelet model.

3 Case Studies

We present two case studies, matrix multiplication and Graph 500's breadth first search. The kernels significantly differ from each other. DGEMM is compute-bound and allows for heavy data reuse, while Graph500 is memory-bound and stresses the memory subsystem with random accesses. Together, they provide an ideal base for an initial analysis of both the codelet model and DARTS.

3.1 Experimental Testbed

We evaluate our case studies on a 48-core compute node. It embeds four AMD Opteron 6234 (Interlagos) processors, clocked at 2.4 GHz. The node is equipped with 4×32 GB of DDR3-1333 ECC memory. Each core of the Interlagos have access to a 16 KB L1 data cache. Two cores share a 2 MB L2 unified cache. A 6 MB unified L3 cache is shared by six cores. Hence there are two L3 caches per Interlagos processor. One important architectural aspect of this processor is that one floating-point unit is shared between two cores. It can process up to four double precision operations at once using AVX instructions.

Our testbed runs Scientific Linux 6. Both the DARTS runtime and kernels are compiled with GCC version 4.6.2 with `-O3`. In the following we compare the

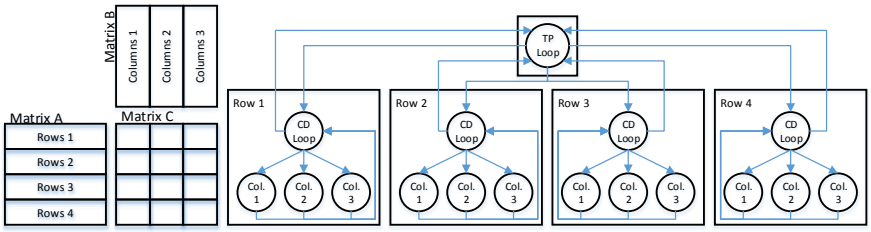


Fig. 4. Codelet representation of Matrix Multiply.

performance obtained with DARTS and OpenMP. All OpenMP programs were run with the threads being pinned as far away from each other as possible, to ensure they had as much cache memory to themselves as possible.

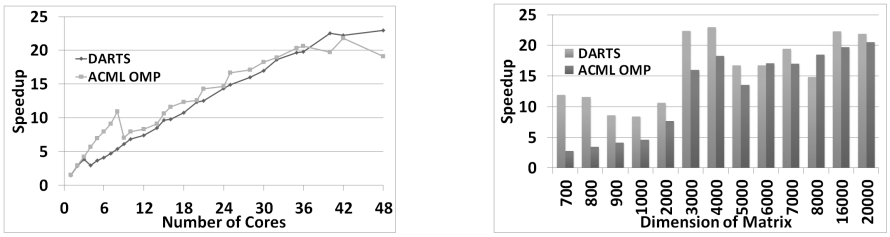
3.2 Matrix Multiplication

We use dense square matrix multiplication (DGEMM) to observe DARTS' performance on a common compute-bound kernel. Regular kernels like DGEMM typically perform well in OpenMP-like environments. In this study, we leverage AMD's Core Math Library (ACML) DGEMM kernel in two ways. First, we use ACML's sequential DGEMM kernel as an optimized building block in our DARTS implementation. Second, we compare our results against ACML's parallelized OpenMP DGEMM.

Figure 4 illustrates our decomposition of the DARTS version of DGEMM. We divide matrix A into rows and matrix B into columns producing a tile of results stored in the C matrix. We leave the "inner tiling" to the sequential ACML kernel. This partitioning is achieved using a TP forall loop to divide matrix A into even groups of rows. We further divide matrix B into columns using a codelet forall loop per spawned TP. Each parallel codelet instantiated will compute a tile in matrix C by calling ACML's DGEMM.

This partitioning translates well to the implementation of the abstract machine. A single group of rows of matrix A will be processed by a single cluster (a group of cores sharing a L3 cache). The cores within a cluster will individually compute a tile of matrix C, using exclusive groups of columns of matrix B, while sharing rows of matrix A.

Figure 5 presents our results. We present DGEMM's strong scaling for 4000×4000 matrices in Figure 5(a). When the number of cores used is small (i.e. less than 12) the OpenMP kernel clearly outperforms our DARTS implementation. As the number of cores grows, the gap becomes much more narrow ($\approx 8\%$ in favor of OpenMP). When the full node is used, DARTS achieves the highest speedup. As the number of cores increases we observe two phenomena: 1) the FPU is more contended, as it is shared between two cores, and 2) contention on the memory banks also increases. Such contention is usually low enough w.r.t. to the number of active cores that there is no visible added latency. However



(a) Strong scalability: N = 4000

(b) Weak scalability: N = 700 to 20,000

Fig. 5. Results for $N \times N$ matrix multiplication. The X axis on Figure 5(a) is the number of cores, and the size of N for Figure 5(b). In both figures, the Y axis shows the speedup w.r.t. ACML's sequential DGEMM.

with such a high number of cores simultaneously active, the delays accumulate in the OpenMP version, having a real impact on the final execution time. This phenomenon is not undocumented [12].

Figure 5(b) shows various results for runs with 48 active cores and dimension sizes ranging from 700 to $20,000$. The DARTS implementation outperforms OpenMP in all but two cases. The average relative speedup between OpenMP and DARTS is 1.40×, while the maximum speedup is 4× when $N = 700$[1].

3.3 The Graph 500 Benchmark

To further evaluate DARTS, we used the Graph 500 parallel breadth-first search (BFS) algorithm[20]. BFS represents a class of irregular applications as the latencies of the memory accesses are dependent on the input data and subjected to NUMA effects. Hassaan et al. [13] present various parallel BFS algorithms in detail.

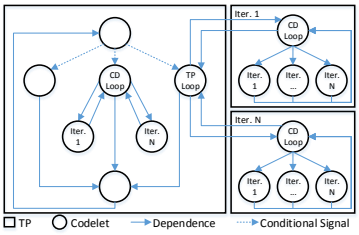
We compare a DARTS implementation of Graph 500's second kernel (BFS) to the OpenMP reference implementation. The kernel performs an in order BFS search. Each iteration traverses through a search frontier, visiting nodes and enqueuing their children for the next iteration's search frontier until all connected nodes have been visited. The kernel's output is a spanning tree.

The OpenMP kernel distributes the nodes in a search frontier using a parallel loop. After exploring a single search frontier, the OpenMP threads enter a barrier before exploring the next frontier. By default, the reference implementation uses static scheduling.

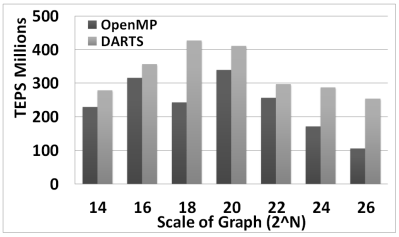
The DARTS implementation uses a similar barrier-like approach. A search frontier is distributed to one or more codelets, and a sink codelet is used upon completion. We however, take advantage of the two-level parallelism inherent in the codelet model. When the frontier is very small, we use a single codelet to process the frontier. As the frontier grows, we scale the parallelism using a

[1] While we present results ranging from $N = 700$ to $N = 20,000$, the overall experiments started with $N = 100$. These numbers are included in our averages.

codelet loop. This limits the parallelism to a single cluster, reducing the overhead of useless parallelism and increasing data locality. Once the frontier is large enough, we partition it into TPs using a TP loop and again into codelets using a codelet loop. Figure 6(a) illustrates our strategy.



(a) Codelet representation of BFS.



(b) Parallel BFS processing RMAT-generated graphs.

Fig. 6. Figure 6(a) illustrates the various strategies used to distribute the search frontier among codelets. Figure 6(b) presents our BFS results. The X axis represents the scale of the graph (2^n); the Y axis represents the harmonic mean of several runs reported in TEPS. Higher is better.

Figure 6(b) presents the number of Traversed Edges Per Second (TEPS), where the greater the number, the faster the implementation. Both implementations were provided identical graphs generated using the RMAT method [5]. Moreover, we use numactl to interleave memory. This approach was not used for DGEMM as it provided no performance gains. We did not present results for OpenMP using dynamic scheduling (creating smaller iteration chunks to increase over-subscription on the machine) as they were significantly worse. For smaller graph sizes, we see DARTS is able to narrowly outperform the reference implementation as work is easily balanced. However as the graph grows, DARTS begins to significantly outperform to the tune of 1.15-2.38×. This is due to the ease in which DARTS can exploit parallelism, scaling with the size of the search frontier. Moreover, DARTS balances the workload hierarchically, first balancing TPs and then codelets. Furthermore by decomposing the machine such that sockets map to AMM clusters, TPs will be balanced between sockets ensuring less contended accesses to DRAM. This result is a natural extension of the Codelet model, where similar approaches are possible, but require much effort.

4 Related Work

The arrival of multicore and manycore systems has rekindled the interest of efficiently running threads on shared-memory systems beyond the classical Pthread [19] and OpenMP [7] models.

Intel Threading Building Blocks [21] is a C++ library which provides several data structures and lock-free constructs to express parallelism. This includs the

recent addition of augmented flow graphs (similar to a dataflow graph). The Codelet model differs as it advocates event-driven fine grained parallelism, while TBB focus on offering various types of parallelism.

Charm++ [14] also uses C++ to provide a parallel, object oriented, programing environment. Charm++'s goals are close to DARTS', with respect to resource management, energy efficiency, etc. However, while DARTS is event-driven, Charm++ is message-driven.

Cilk and its later iterations [1,18] are languages whose underlying execution model is more fine grain than classical shared-memory models. However, they do not implement dataflow features to express computations in terms of data and/or event dependencies.

Habanero Java [4], a "spin-off" of the X10 language [6], extends the initial Cilk syntax rendering it more flexible. Despite its recent additions of data driven constructs [22], its execution model does not rely on dataflow as a foundation unlike the Codelet model (and runtime implementation).

SWARM [16] is another runtime system which implements the codelet execution model. SWARM does not respect the basic semantics of individual codelets as proposed in [11], nor does it implement other advocated features, such as threaded procedures, loop constructs, etc.

The Concurrent Collections (CnC) family of languages [3] is a coordination language which is very much inspired by dynamic dataflow. CnC utilizes a separation of concerns, providing a tuning specification to achieve performance. A stand alone CnC program may not represent an event-driven codelet application, however with a proper tuning specification they could be equivalent.

5 Conclusion

In this paper we have presented an implementation of the fine-grain dataflow inspired codelet execution model. We have tested our implementation on a many-core shared-memory node, using two kernels. Our parallel implementation of DGEMM yields on average a 1.40× speedup over AMD's OpenMP-based implementation for matrix sizes ranging from 100×100 to $20,000 \times 20,000$ with a maximum speedup of 4×. We also compared ourselves to the reference implementation of the Graph500 BFS benchmark. On average, we reached a speedup of 1.50×, with a maximum of 2.38×.

Our future work includes further exploring Graph500 kernels in order to show how the codelet model eases the expression of parallelism and data dependencies between tasks. This includes exploring unordered BFS kernels which discard the barrier found at the end of each forall loop. We also want to further develop DARTS' parallel loop constructs applying software pipelining techniques, and building more general constructs to handle streams. Lastly, we would like to run our experiments on different compute node architectures, such as Intel's Ivy Bridge or other C++-supported general purpose many-core systems.

6 Acknowledgments

This work was partly supported by European FP7 project TERAFLUX, id. 249013. This material is based upon work supported by the Department of Energy (National Nuclear Security Administration) under the Award Number DE-SC0008717. We would also like to acknowledge the help of Tom StJohn for his expertise on the Graph500 benchmark, and Jean-Philippe Halimi for his work on DARTS.

References

1. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, Aug. 1995.
2. F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180 –186, feb. 2010.
3. Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, et al. Concurrent collections. *Scientific Programming*, 18(3):203–217, 2010.
4. V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. In *PPPJ'11: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 2011.
5. D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. *Computer Science Department*, page 541, 2004.
6. P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the Twentieth Annual ACM SIGPLAN Conference on Obj ect-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 519–538, Oct. 2005.
7. L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.
8. M. Denneau and H. S. Warren Jr. 64-bit cyclops principles of operation part i. Technical report, Technical report, IBM Watson Research Center, Yorktown Heights, NY, 2005.
9. J. B. Dennis. First Version of a Data-Flow Procedure Language. In *Colloque sur la Programmation*, number 19, pages 362–376, Paris, Apr. 9–11, 1974.
10. G. Gao, H. Hum, and Y.-B. Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Databases, Parallel Architectures and Their Applications,. PARBASE-90, International Conference on*, pages 112 –116, mar 1990.
11. G. R. Gao, J. Suetterlein, and S. Zuckerman. Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond. Technical Memo – Available on request, April 2011.
12. E. Garcia, D. Orozco, R. Khan, I. Venetis, K. Livingston, and G. R. Gao. Dynamic Percolation: A case of study on the shortcomings of traditional optimization in

Many-core Architectures. In *Proceedings of 2012 ACM International Conference on Computer Frontiers (CF 2012)*, Cagliari, Italy, May 2012. ACM.
13. M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered and unordered algorithms for parallel breadth first search. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 539–540, New York, NY, USA, 2010. ACM.
14. L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, Oct. 1993.
15. R. Knauerhase, R. Cledat, and J. Teller. For extreme parallelism, your OS is sooooo last-millennium. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, pages 3–3. USENIX Association, 2012.
16. C. Lauderdale and R. Khan. Towards a codelet-based runtime for exascale computing: position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.
17. E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.
18. C. E. Leiserson. The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.
19. F. Mueller. Implementing posix threads under unix: Description of work in progress. In *Proceedings of the Second Software Engineering Research Forum*, pages 253–261. Citeseer, 1992.
20. R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.
21. C. Pheatt. Intel Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008.
22. S. Taşırlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *ICPP'11: Proceedings of the International Conference on Parallel Processing*, Sep 2011.
23. K. B. Theobald. *EARTH: an efficient architecture for running threads*. PhD thesis, McGill University, Montreal, Que., Canada, Canada, May 1999. AAINQ50269.
24. S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.