Position Paper: Using a “Codelet” Program Execution Model for Exascale Machines[*]

Stéphane Zuckerman
Joshua Suetterlein
University of Delaware
szuckerm@capsl.udel.edu
jodasue@capsl.udel.edu

Rob Knauerhase
Intel Labs
knauer@intel.com

Guang R. Gao
University of Delaware
ggao@capsl.udel.edu

ABSTRACT

As computing has moved relentlessly through giga-, tera-, and peta-scale systems, exa-scale (a million trillion operations/sec.) computing is currently under active research. DARPA has recently sponsored the “UHPC” [1] — ubiquitous high-performance computing — program, encouraging partnership with academia and industry to explore such systems. Among the requirements are the development of novel techniques in “self-awareness”[1]in support of performance, energy-efficiency, and resiliency.

Trends in processor and system architecture, driven by power and complexity, point us toward very high-core-count designs and extreme software parallelism to solve exascale-class problems. Our research is exploring a *fine-grain, event-driven* model in support of adaptive operation of these machines. We are developing a Codelet Program Execution Model which breaks applications into *codelets* (small bits of functionality) and *dependencies* (control and data) between these objects. It then uses this decomposition to accomplish advanced scheduling, to accommodate code and data motion within the system, and to permit flexible exploitation of parallelism in support of goals for performance and power.

Categories and Subject Descriptors

CR-number [**subcategory**]: third-level

[*]This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.
[1]Fans of science fiction will note that Department of Defense attempts to build self-aware systems have not always been successful (cf. *War Games*, *Terminator*...).

Keywords

exascale, manycore, dataflow, program execution model

Acknowledgements

Government Purpose Rights
Purchase Order Number: N/A
Contractor Name: Intel Corporation
Contractor Address: 2111 NE 25th Ave M/S JF2-60, Hillsboro, OR 97124
Expiration Date: None

The Government’s rights to use, modify, reproduce, release, perform, display, or disclose this technical data are restricted by paragraphs B (1),(3) and (6) of Article VIII as incorporated within the above purchase order and Agreement. No restrictions apply after the expiration date shown above. Any reproduction of the material or portions thereof marked with this legend must also reproduce the markings. The following entities, their respective successors and assigns, shall possess the right to exercise said property rights, as if they were the Government, on behalf of the Government: Intel Corporation (www.intel.com); ET International (www.etinternational.com); Reservoir Labs (www.reservoir.com); University of California San Diego (www.ucsd.edu); University of Delaware (www.udel.edu); University of Illinois at Urbana-Champaign (www.uiuc.edu).

1. INTRODUCTION

1.1 Motivation

Contemporary systems include multi- and many-core processors. However, few current implementations include more than a small number of cores per chip — IBM’s Cyclops-64, or Intel’s Single-chip Cloud machines being (for now) the exceptions rather than the rule. As the number of cores inevitably increases in the coming years, traditional ways of performing large scale computations will need to evolve from legacy models such as OpenMP[17] and MPI[14]. Indeed, such models are very much wedded to a control-flow vision of parallel programs, thus making it more difficult to express asynchrony in programs and requiring a rather coarse-grain parallelism. This in turn reduces the ability for the system to insert points at which adaption decisions can be made, as well as requiring expensive whole-context copies if adaptation requires moving code to another core, or replicating a large context when reliability mandates redundant calculation.

We believe that to efficiently exploit the computing power of the future many-core exascale machines, we need to express programs in a finer-grained manner. Our work indicates that by decomposing a program into snippets of code (codelets), we can more efficiently gain beneficial attributes for solving problems in this space. The natural breaks between codelets provide a plethora of opportunities to observe system state and adapt as needed. For example, we can run as many (or as few) simultaneous codelets as required or allowed by the machine's environment, nimbly changing the amount of parallel execution we exploit. Reliability problems — e.g. from transient core failures — are quickly detected, and cores can be put into a different frequency/voltage state or taken "out of service" entirely with minimal effect to the program as a whole.

Our Codelet Program Execution Environment (Codelet PXE) provides a runtime and system software in which the adaptability benefits of codelets can be realized. It relies on existing work in dataflow theory [8] to provide strong theoretical results to guarantee forward progress of a parallel program. Although we do not address them (for lack of space), languages such as Concurrent Collections [4], X10 [6] or Chapel [5] can perfectly fit on top of our Codelet PXE[2], as long as there is a codelet-aware compiler which can decompose a (say) Chapel program into a collection of codelets.

The remainder of this paper is organized as follows. Section 2 describes our fine-grain codelet-based execution model; section 3 tackles issues induced by exascale machines, such as scalability, energy efficiency, and resiliency. Section 4 skims through related research; we conclude in Section 5 and sketch our future research.

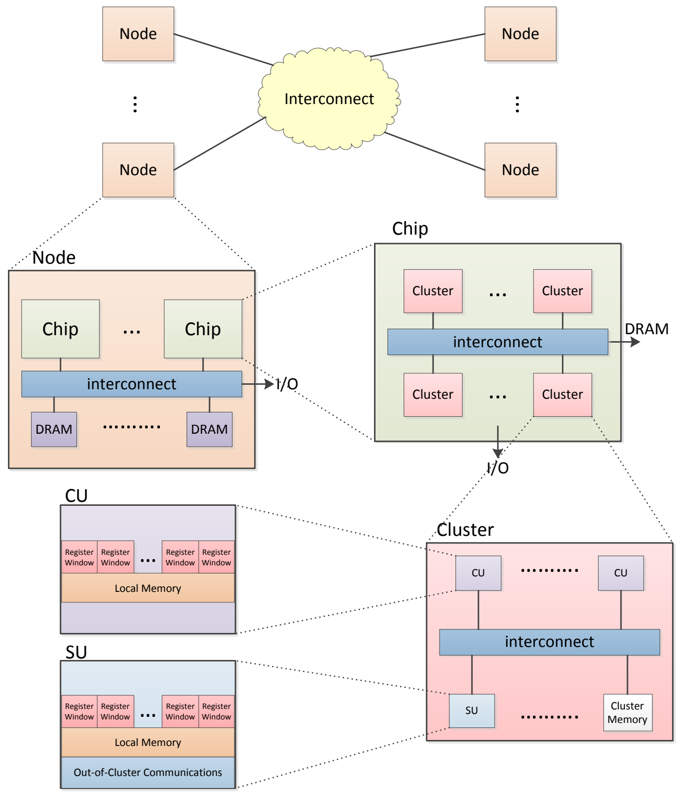

Figure 1: Abstract machine

2. THE CODELET PROGRAM EXECUTION MODEL

2.1 Abstract Machine Model

As an established practice in the field, our proposed program execution model is explained in the context of a corresponding abstract (parallel) machine architecture model. In this section, we present an outline of an abstract machine that can be viewed as an abstract model for the class of parallel computer hardware and software system that realize the proposed codelet design goals.

The abstract machine consists of many nodes that are connected together via an interconnection network as shown as in Figure 1. Each node contains a many-core chip. The chip may consist of up to 1-2 thousand processing cores being organized into groups (clusters) linked together via a chip interconnect. In the abstract machine presented, each group contains a collection of computing units (CU) and at least one scheduling unit (SU), all being linked together by their own on-chip interconnect. A node may also contain other resources, most notably additional memory which will likely be DRAMs or other external storage.

2.2 The Codelet Model

2.2.1 *Goals*

The codelet execution model is designed to leverage previous knowledge of parallelism, and to develop a methodology for exploiting parallelism for a much larger scale machine. In doing so, we must take into account the (application dependent) sharing of both machine and data resources. As the number of processing elements will continue to increase, it is likely that future performance bottlenecks will emanate from this sharing. The codelet model aims to effectively represent the sharing of both data and other vital machine resources. To represent shared data while exploiting massive parallelism, the codelet model imposes a hybrid dataflow approach utilizing two levels of parallelism. In addition, our model provides an event interface to address machine constraints exacerbated by an application. Moreover, the codelet model seeks to provide a means for running exascale applications within certain power budget.

2.2.2 *Definition*

The finest granularity of parallelism in the codelet execution model is the *codelet*. A codelet is a collection of machine instructions that can be scheduled "atomically" as a unit of computation. In other words, codelets are the principal *scheduling quantum* found in our execution model. Codelets are more fine grained than a traditional thread, and can be seen as a small chunk of work to accomplish a larger task or procedure. As such, when a codelet has been allocated and scheduled to a computation unit, will be kept usefully busy until its completion. Many traditional systems utilize preemption to overlap long latency operations with another coarse grain thread incurring a context

[2]In fact, in the case of Concurrent Collections, many concepts are very similar to those we expose in the remainder of this paper.

switch. Since codelets are more fine grain, we believe the overhead of context switching (saving and restoring registers) is too great. Therefore the underlined abstract machine model and system software (e.g. compiler, etc.) must be optimized to ensure such non-preemption features can be productively utilized. With a large amount of codelets and a greater amount of processing power, our work is leading us to believe that it may be more power efficient to clock-gate a processing element while waiting for long latency operations than to multitask. In efforts to reduce power consumption and to decrease latency of memory operations, codelets presuppose that all data and code be local to its execution. A codelet's primary means of communication with the remaining procedure is through its inputs and outputs.

2.3 The Codelet Graph Model

The codelet graph (CDG) to be introduced below has its origin in *dataflow graphs.*

A CDG is a directed graph $G = (V, E)$ where V is a set of nodes and E is a set of directed arcs and tokens. In a CDG nodes in G may be connected by arcs from E. A node (also called a codelet actor) denotes a codelet.

An arc (V_1, V_2) in the CDG is an abstraction of a precedence relation between nodes V_1 and V_2. Such a relation may be due to a data dependence between codelet V_1 and V_2, but we allow other types of relations. For example, it can simply specifies a precedence relation: the execution of the codelet actor V_2 must be following the execution of V_1.

Arcs may contain event tokens. These tokens correspond to an event, the most prominent being the presence of data or a control signal. These arcs correspond to the inputs and outputs of a codelet. Tokens travel from node to node, enabling codelets to execute. When the execution of a codelet is completed, it places tokens (its resulting values) on its output arcs, and the state of the abstract machine (represented by the entire graph) is updated. The interested reader can find examples of programs expressed as codelet graphs in a previously released technical report [12].

Operational Semantics.

The execution model of a codelet graph is specified by its operational semantics called *firing rules.* A codelet can be in one of three states. A codelet begins as *dormant.* When all its events are satisfied it becomes *enabled.* Once a processing element becomes available, a codelet can be *fired.*

Codelet Firing Rule.

A codelet becomes enabled once tokens are present on each of its input arcs. An enabled codelet actor can be fired if it has acquired all required resources and is scheduled for execution. A codelet actor fires by consuming tokens on its input arcs, performing the operations within the codelet, and producing a token on each of its output arcs upon completion.

While codelet actors are more coarse than traditional dataflow, codelet actors still require glue to permit conditional execution (i.e. a loop containing multiple codelet actors). In order to provide greater functionality, we include several actors including a decider, conditional merge, and T and F-gates which follow directly from the dataflow literature [7].

Threaded Procedures.

Viewing a CDG in conjunction with the firing rule paints a picture of how a computation is expressed and executed. Codelets are connected by dependencies through their inputs and outputs to form a CDG. The formation of these CDGs gives rise to our second level of parallelism, *threaded procedures.*

A threaded procedure (TP) is an asynchronous function which acts as a codelet graph container. A TP serves two purposes, first it provides a naming convention to invoke a CDG, and secondly it acts as a scope of which codelets can efficiently operate within.

Figure 2 shows three threaded procedure invocations.

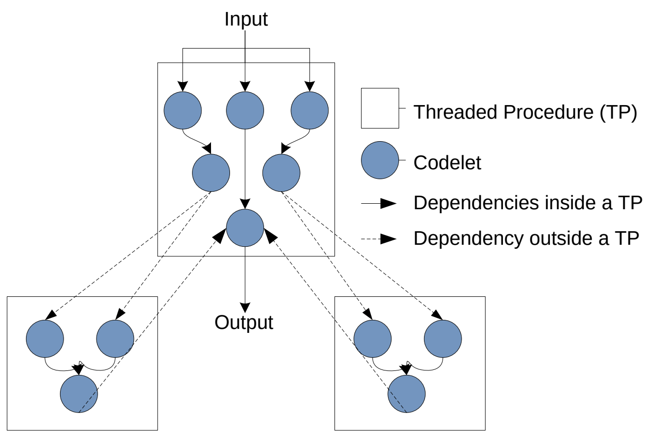

Figure 2: An examples of three threaded procedures

Codelets and TPs are the building block of an exascale application. Codelets act as a more fine grain building block exposing more parallelism. TPs conveniently group codelets providing composability for the programmer (or compiler) while increasing the locality of shared data. TP's also act as a convenient means for mapping computation to a hierarchical machine.

2.4 Well-Behaved Codelet Graphs

A CDG is well-behaved if its operations terminate following each presentation of values on its input arcs, and places a token on each of its output arcs. Furthermore, a CDG must not have any circular dependencies and be self-cleaning (i.e.the graph must return to its original state after it is executed).

In well-behaved graphs, as tokens flow through the CDG, forward progress in the application is guaranteed. This is an important property for applications composed with fine grain actors sharing resources. Furthermore, a well-behaved graph ensures determinate results regardless of the asynchronous execution of codelets and TPs. This property holds when all the data dependencies between codelets are expressed.

CDGs adhere to the same construction rules as a dataflow graph which is cited in [7]. In particular, well-behaved components form a well-behaved graph. Control actors are not well-behaved actors; however they may still be part of a well-behaved CDG because they are considered a well-formed schema [7].

A framework which provides the default of being well-behaved and determinate aids in reducing development effort (by easing the debugging process) and enabling the "self-aware" adaptability we desire.

3. SMART ADAPTATION IN AN EXASCALE CXM

So far, we have described our codelet model of computation, without explaining how it fits into a live system, e.g. how the runtime system (RTS) will deal with codelet scheduling for scalability, energy-efficiency, or resiliency. This section addresses these issues, and exposes various mechanisms which we believe will enable programs using the codelet PXE to run efficiently.

3.1 Achieving Exascale Performance

Loop parallelism and Codelet Pipelining.

Loop parallelism is an important form of parallelism in most useful parallel execution models. Since the codelet graph model has evolved from dataflow graphs, naturally we intend to exploit the progress made on loop parallelism in the dataflow model. One of the most successful techniques to exploit loop parallelism is at the instruction-level using software pipelining (SWP) [16].

Based on the work of the past decade on loop code mapping, scheduling and optimization, various ideas of loop scheduling have now converged to a novel loop nest SWP framework for multi- and many-core systems. Among them, we count Single-dimension Software Pipelining (SSP) [18], which is a unique resource-constrained SWP framework that overlaps the iterations of an n-dimensional loop. SSP has been implemented with success on the IBM Cyclops-64 many-core chip [10].

As previously stated, the goal of the codelet model is to better utilize a larger machine with more fine grain computation. Using SWP techniques to permit more codelets aids in achieving this goal, however other mechanisms are required to saturate the machine.

Split-Phase Continuations.

The "futures" concept and their continuation models have been investigated or are under investigation in a number of projects (e.g. Cilk, Java, C#, ...). Under the codelet model, each time a future-like asynchronous procedure invocation is launched, it will be considered as a long (and hard to predict) latency operation, and the invoking codelet will not be blocked. In fact, the codelet should only contain operations that will not depend on the results of the launched procedure invocation. Therefore, once the codelet has finished its remaining operations, it will quit (die) naturally.

Conceptually, the remaining codelets of the calling procedure will continue, hence the term "continuation." Also, since the launching codelet(s) have already all quit, the continuation could be considered as starting as another "phase" of the computation, hence the term split-phase continuation. We anticipate that such split-phase actions will be very effective in tolerating long and predictable latencies.

Another important benefit from split-phase continuation is that it provides the opportunity for the scheduler to clock-gate (e.g. saving power while preserving state) a given computation unit which issued one such continuation, and is waiting for its results to continue its execution. It can then simply wait for the scheduler to send a signal to wake up once the data dependence is finally resolved.

Meeting Locality Requirements.

The codelet model presupposes that codelet have their inputs available locally. However, some inputs may very well simply be a pointer referencing a remote memory location. While the pointer itself is locally available, the data it references may not be available locally when the codelet fires. One way of dealing with this is to perform split-phase continuation (cf. paragraph 3.1). However, it may be more efficient to simply make sure the data referenced is indeed locally available to the running codelet.

In current systems, such a task is mostly performed through the use of code or data prefetching. However, classical code and data prefetching only concerns themselves with bringing *blocks* of code or data into local memories (usually caches) in an efficient way. While an efficient technique, prefetching is done in a systematic manner at the hardware and software level, hence increasing the power demands on the overall system. Moreover, relatively complex data structures (say, linked lists or graphs) usually tend to defeat prefetchers, as locality is not guaranteed. However, on a many-core architecture, knowing how data will be allocated, in which cluster, etc., will be extremely difficult. Furthermore It is unlikely that classical hardware data-prefetching will be very efficient for particular challenge problems like graph traversals.

In addition, too aggressive prefetching can pollute the local memory of a computation unit, by using too big a prefetch distance and thus copying in useless data. This is mainly due to the fact that prefetching is "just" a technique to load blocks of data, regardless of their meaning for the current computation. To deal with more clever ways of moving data, additional techniques have been devised.

3.2 Achieving Power and Energy Efficiency

Energy efficiency is one of the major challenges for exascale supercomputers. Extrapolation of current (naïve) technologies into exascale domains quickly becomes unsupportable. Indeed, some UHPC approaches explicitly state that running a whole system at full power will introduce quick failure. To this end, future systems will have to build on dynamic voltage/frequency scaling along with system-wide metrics to adapt consumption according to available power, user-defined system goals, and other factors.

Therefore, exascale systems will in all probability require ways to quickly and efficiently shut down parts of one or more computation nodes when they are not useful. There are two approaches to this problem. One is to provide mechanisms which will proactively aim at reducing energy-consumption, which is partially achieved through the use of percolation hint (see below). The other one is to react to events as they happen in real time and decide what to do. This is where our event-driven Codelet PXE is going to help.

Code and Data Percolation for Energy Efficiency.

The HTMT program execution model [13] introduced the *percolation* mechanism [19, 2], to deal with memory hierarchies with very different latencies (going from a few cycles to several hundreds). It is different from classical data-prefetching in that it fetches "intelligently" the next block of data according to its type. For example, consider an element in a linked list. There is usually a field, say `next`, which points to the next element in the list. Here, `next` could be marked (through a hint) as needing to be followed, hence giving the compiler and runtime system enough information to know how important it is to have data available locally. In a naïve version of a graph node data structure, a given node would have an array of such pointers to reference this node's neighbors.

Data percolation is a useful tool to hide latencies and increase the overall performance of a system without incurring the drawbacks of general prefetching. However, moving data around can be extremely costly in terms of energy-consumption. Our CXE and runtime percolator applies percolation to code itself. Depending on the relative sizes of code (remember, codelets are small) and data, it will sometimes make more sense to move only the code, and let the data stay where it was last modified. Users of many-core architectures will need to think in a data-centric way rather than a code-centric one; however, our system helps by adapting based on data-access patterns [15] from both hardware performance monitoring units and compiler hints/instrumentation.

While some of these factors can be determined at compile-time, others are strictly a function of dynamic runtime state. Our hardware designs (outside the scope of this paper) include deliberate features to monitor memory access and to ease the ability to relocate code and data as needed throughout the system.

How deep percolation should go (e.g., how many links or neighbouring nodes should be retrieved in our linked list and graph examples) depends on the runtime knowledge of available local memory for a given codelet, what other memory requests should be issued next to the ones currently be processed, and so forth. This will remain an important area for ongoing research.

Self-Aware Power Management.

The runtime system (RTS) used in our prototype Codelet PXE is continuously fed system-monitoring events which reflect the "health" of the system. In particular, probes are expected to warn the scheduling units when (parts of) the computation node they are running on is going over a given power threshold. Thresholds can be determined by the user (power "goals" input directly to the system, or hints coded into the application) or by the system itself (hard limits based on heat, or availability/quality of power at a given moment).

The scheduler decides what to do with threaded procedures (and the CDG attached to them) that are currently running on the parts programmed to be shut down. If the hysteresis (or "heat history") indicates a negative trend, it can decide to enact energy-saving measures (clock-gating, frequency-scaling, or — perversely — increasing performance to finish the CDG after which it can shut down that area of the chip for some period. If none of these measures is sufficient, the scheduler may simply turn off that part of the chip (or memory) and migrate or restart one or mode codelets elsewhere. Resiliency checkpointing techniques (described in a following section) will of course help this migration.

3.3 Achieving Resiliency

There are a number of factors that may cause an exascale system to exhibit less than optimal reliability. For example, even with a very high mean time between failures (MTBF) of each core, a system with a million cores will see transient failures. Likewise, as part of the manufacturing process, different cores will have different tolerance of voltages; the aggressive power-scaling that is required for energy efficiency may cause some cores to fail under certain threshholds. Lastly, environmental factors (heat generated by use of certain "areas" of the system, or external cooling availability) may also introduce intermittent problems. To ensure correctness of ongoing computations, several adaptation mechanisms must be provided.

At the most basic level, our CXE can provide resiliency by duplicating computation in various parts of the machine. For mission-critical applications, the value of this may be worth the costs of increased energy usage or reduced capacity in the system.

Our related research includes observation and introspection systems which will help detect failures. The runtime system uses this information as part of its codelet scheduling. It can, for example, change the voltage and/or frequency settings of individual cores or blocks to ameliorate errors, or it can shut down areas of the system and relocate computation elsewhere to allow failing cores to return to a less error-prone temperature.

Lastly, because our CXE includes dependency information, and because codelets are small and make very localized changes to their data, many opportunities for traditional resiliency — checkpointing and the like — can be quickly accommodated. Indeed, the runtime can perform incremental state versioning even for computation that isn't necessarily "transactional" in the original application. We expect to prototype heuristics to determine when to speculatively create versions of codelet state based on machine state and predictions of localized failures.

4. RELATED WORK

Dataflow (DF) was first proposed in its static definition by Jack Dennis[8, 9]. DF was later extended by Agerwala and Arvind creating dynamic DF. Dynamic DF is considered to exploit the finest granularity of parallelism, but suffers large overheads due to token matching and lack of locality.

The EARTH system [20] was our second inspiration for our current work. It is a hybrid dataflow model which runs on off-the-shelf hardware. EARTH took an evolutionary approach to exploring multithreading, and its runtime is not suited to scale to massive many-core machines.

The Cilk programming language [3] is an extension to the C language which provides a way to simply express parallelism in a divide-and-conquer manner.

Other programming models also provide asynchronous execution, such as StarSs [11], which provide additional pragmas or directives to C/C++ and Fortran programs. StarSs requires to explicitely state which are the inputs and outputs of a given computation, and is basically an extension to OpenMP-like frameworks.

Chapel [5] and X10 [6] are part of the Partitioned Global Address Space (PGAS) languages. Both have specific features which differentiate them from each other, but provide an abstration of the memory addressing scheme to the programmer. All the while, they give him/her the ability to express critical sections and parallel regions with ease through adapted constructs.

Our work is clearly inspired by previous work on dataflow, as well as the research done on the EARTH system. However, contrary to EARTH's fibers, our codelet model is specifically designed to run codelets on multiple cores. In addition, we address power and resiliency issues, which none of the previously described programming models do. Cilk's algorithmic approach to scheduling can also be used in our Codelet PXE, but we believe we can go a step further, while still retaining some important theoretical properties offered by Cilk.

Moreover, the work described in this section is in no way in opposition with what we are proposing. Codelets should be seen as the "assembly language" of parallel frameworks. Programs written in the previous languages can be decomposed into codelets.

5. CONCLUSION AND FUTURE WORK

This paper presents a novel program execution model aimed at future many-core systems. It emphasizes the use of finer grain parallelism to better utilize the available hardware. Our event-driven system can dynamically manage runtime resource constraints beyond pure performance, including energy efficiency and resiliency.

Future work includes developing a codelet-aware software stack (starting with a compiler and a runtime system) which will implement our ideas related to scalability, energy efficiency and resiliency. Further research will also be needed to take security as another component of our model.

6. REFERENCES

[1] Ubiquitous high performance computing (uhpc).

[2] J. N. Amaral, G. R. Gao, P. Merkey, T. Sterling, Z. Ruiz, and S. Ryan. An htmt performance prediction case study: Implementing cannon's dense matrix multiply algorithm. Technical report, 1999.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. pages 207–216.

[4] Z. Budimlic, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. M. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[5] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

[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 R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005.

[7] J. Dennis, J. Fosseen, and J. Linderman. Data flow schemas. In A. Ershov and V. A. Nepomniaschy, editors, *International Symposium on Theoretical Programming*, volume 5 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1974.

[8] J. B. Dennis. First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, number 19 in Lecture Notes in Computer Science. Springer-Verlag, April 9–11, 1974.

[9] J. B. Dennis and J. B. Fossen. Introduction to data flow schemas. Technical Report 81-1, September 1973.

[10] A. Douillet and G. R. Gao. Register pressure in software-pipelined loop nests: Fast computation and impact on architecture design. In *LCPC*, pages 17–31, 2005.

[11] A. Duran, J. Perez, E. Ayguadé, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. In R. Eigenmann and B. de Supinski, editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 111–122. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-79561-2_10.

[12] G. R. Gao, J. Suetterlein, and S. Zuckerman. Toward an execution model for extreme-scale systems — runnemede and beyond. CAPSL Technical Memo 104, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, May 2011.

[13] G. R. Gao, K. B. Theobald, A. Márquez, and T. Sterling. The HTMT program execution model. Technical Report 09, 1997.

[14] R. L. Graham. The mpi 2.2 standard and the emerging mpi 3 standard. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28:54–66, May 2008.

[16] M. S. Lam. Software pipelining: an effective scheduling technique for vliw machines (with retrospective). In *Best of PLDI*, pages 244–256, 1988.

[17] L. Meadows. Openmp 3.0 — a preview of the upcoming standard. In *Proceedings of the 3rd international conference on High Performance Computing and Communications*, pages 4–4, Berlin, Heidelberg, 2007. Springer-Verlag.

[18] H. Rong, A. Douillet, R. Govindarajan, and G. R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *Proc. of the 2004 Intl. Symp. on Code Generation and Optimization (CGO)*, March 2004.

[19] G. Tan, V. Sreedhar, and G. Gao. Just-in-time locality and percolation for optimizing irregular applications on a manycore architecture. In J. Amaral, editor, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008.

[20] K. B. Theobald. *Earth: an efficient architecture for running threads*. PhD thesis, Montreal, Que., Canada, Canada, 1999. AAINQ50269.