Position Paper: Locality-Driven Scheduling of Tasks for Data-Dependent Multithreading

Jaime Arteaga, Stephane Zuckerman, Elkin Garcia, and Guang Gao
Department of Electrical and Computer Engineering
University of Delaware
Newark DE, 19716, USA
Email: {jaime@, szuckerm@, egarcia@, ggao@capsl.}udel.edu

***Abstract*—Implementing locality-aware scheduling algorithms using fine-programming models may generate scheduling overheads due to the potential elevated number of tasks. In order to reduce such overhead, while increasing at the same time data locality in multithreaded applications, this paper proposes a new technique named *Locality-Driven Code Scheduling (LDCS)*. LDCS uses the data dependency graph of an application to identify the tasks writing to a common chunk of data and groups them into a single coarse-grain construct called *super-task*. LDCS uses fine-grain synchronization to start the execution of a super-task, but relaxes the constraints of classical macro-dataflow models by signaling a super-task in the middle of its execution to fire each of its internal *phases*. Since all the phases of a super-task process the same block of data and the scheduling of work to hardware threads is made in terms of super-tasks, long latency operations are significantly reduced. Preliminary results show that LDCS can improve the performance of a linear algebra kernel by 72% on average for weak scaling in comparison with a dynamic scheduling version of the kernel when using an architecture with software-managed memory hierarchy.**

***Keywords*-Dynamic Scheduling; Locality-Aware Scheduling; Many-core Architecture; Multithreading; Super-tasks.**

I. Introduction

Over the last decade, computer architects and programmers have relied mainly on multi and many-core architectures to boost the performance of scientific and everyday applications. Traditional numerical methods (such as linear algebra kernels and applications) have already been decomposed to be executed concurrently and to utilize these machines to the fullest. However, new algorithms must be proposed to better take advantage of these massively parallel chips and to reduce the latency of applications running on such systems. Given good prior knowledge of a regular application, it is possible to determine its data access patterns and thus try to reduce the amount of data movement while executing—exploiting data locality as much as possible.

Scheduling techniques based on data locality have been studied for multicore architectures using common multi-threaded programming models and different levels of task granularity [1], [2], [3], [4].

For applications requiring fine-grain parallelism, other works have used the dataflow and the codelet model for the implementation of such techniques [5], [6], [7]. Even though these models are preferred for applications that need to adapt dynamically during runtime, overheads could be produced in the scheduling of the many small tasks generated by the model.

We believe that an application can benefit more from a locality-aware scheduling technique in a fine-grain programming model by grouping tasks that process a common block of data in a single coarse-grain construct called *super-task*, which requires dependence satisfaction in the middle of its execution. We call this technique *Locality-Driven Code Scheduling (LDCS)*.

Operational semantics of super-tasks are derived from dataflow semantics [8], and in particular macro-dataflow [9]. A super-task is comprised of several phases that execute in sequence. Each phase is tied to a set of dependence signals and is triggered when external data it depends on has been fully updated. Super-tasks provide several advantages: they improve data reuse, drastically reduce scheduling overheads, and, as phases are inlined within a super-task, they make the economy of function calls. This reduction in the amount of data movement is directly translated into improvements in the execution time of the application [10].

LDCS is presented in this paper with the following main contributions:

1) The description of the technique.
2) An analysis of the impact that LDCS has on an application's performance.
3) Preliminary experimental evaluation of the technique in an architecture with software-managed memory hierarchy.

The paper is organized as follows: First, we introduce LDCS using the LU factorization as example in Section II, followed by some preliminary experimental results in a software-managed memory hierarchy in Section III. Then, we review some of the related work and analyze their differences with LDCS in Section IV. Finally, we outline our conclusions and ongoing and future work in Section V.

II. Locality-Driven Code Scheduling

Locality-Driven Code Scheduling (LDCS) is a locality-aware scheduling algorithm based on the use of coarse-grain constructs named *super-tasks*. If the data dependency graph (DDG) of the target application is known by the programmer, then the tasks writing the same block of data can be grouped in a single super-task, which is computed by the same hardware thread. Each task of the super-task becomes a *phase* of computation of the super-task and starts execution only when its corresponding dependencies have been satisfied. In this sense, LDCS relaxes dataflow models specifications by allowing super-tasks to be signaled in the middle of their execution, in order to fire each of their internal phases. Moreover, LDCS decreases the number of long latency operations because only the first and last phases of the super-task are required to access main memory to read and write, respectively, the block of data processed by all the tasks of the super-task. In order to achieve this, the programmer must select an appropriate size for the block of data so this one can fit in one of the upper levels of the memory hierarchy of the target platform, along with any other data required by the super-task for its processing.

If the DDG of an application is known, the steps a programmer needs to follow to implement LDCS are:

1) Determine the number of blocks of data to be produced by the application and their associated tasks.
2) For each block, create a super-task with all the corresponding tasks.
3) Assign dynamically super-tasks to available hardware threads. Prioritize super-tasks containing tasks in the critical path of the DDG.
4) Execute each super-task following Algorithm 1.
5) Repeat steps 3 and 4 until all super-tasks have been assigned and processed.

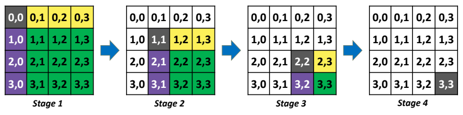
Algorithm 1 LDCS: Execution of a Super-Task by a Hardware Thread

```
 1: procedure ProcDataBlock(NP = Number of Phases, ND[NP] = Number of Dependencies for each Phase)
 2:     for (p=0; p < NP; p++) do
 3:         Wait for ND[p]'s dependencies to be satisfied
 4:         if p==0 then
 5:             Read the block of data of the super-task.
 6:         end if
 7:         Read any other data required.
 8:         Process the block of data with phase p
 9:     end for
10:     Write the block of data back into main memory
11:     Signal any thread(s) waiting for this block of data
12:     Make hardware thread available
13: end procedure
```

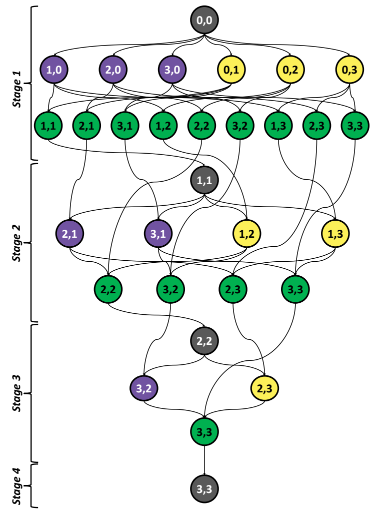
The implementation of an application using LDCS can effectively improve its performance since the number of long latency operations is reduced. LDCS may also improve its power efficiency due to the cost (in terms of energy) that must be paid when moving data across the different memory levels of a multi or many-core architecture [10].

LDCS is especially suitable for blocked linear algebra kernels such as LU factorization, which is commonly used for the benchmarking of high-performance systems([11], [12], [13]). Figure 1 presents the DDG of a typical blocked implementation of this algorithm. As can be seen, several blocks of data are processed more than once using different tasks (defined as *GETRF*, *GESSM*, *TSTRF*, and *SSSSM* [14]). If a typical fine-programming model is used for the implementation of this kernel, then each block of data must be read and written by each task that processes it. In this implementation, the number of units of work to be scheduled to hardware threads is equal to the number of tasks.

In order to reduce both the number of long latency operations performed by the kernel and the number of units of work to be mapped to hardware threads, LDCS can be applied to the DDG of LU factorization as shown in Figure 2. In this figure, different tasks that process the same block of data have been grouped in the same super-task. Block $(3, 3)$, for instance, is processed by three *SSSSM* tasks and one *GETRF* task, which are now phases of the same super-task. Only the first *SSSM* and the *GETRF* tasks must access to main memory to read and write this block of data, while the other phases rely on its presence in the upper levels of the memory hierarchy.



(a) Processing Stages



(b) Data Dependence Graph

Figure 1. Classical Blocked LU factorization: *GETRF* tasks are dark gray, *TSTRF* tasks are purple, *GESSM* tasks are yellow, *SSSSM* tasks are green.

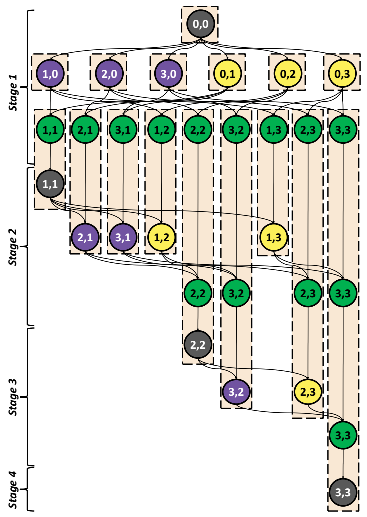


Figure 2. DDG of the LU factorization algorithm using LDCS: *GETRF* tasks are dark gray, *TSTRF* tasks are purple, *GESSM* tasks are yellow, *SSSSM* tasks are green, and light-orange dashed boxes enclose tasks computed by the same hardware thread and conforming a super-thread.

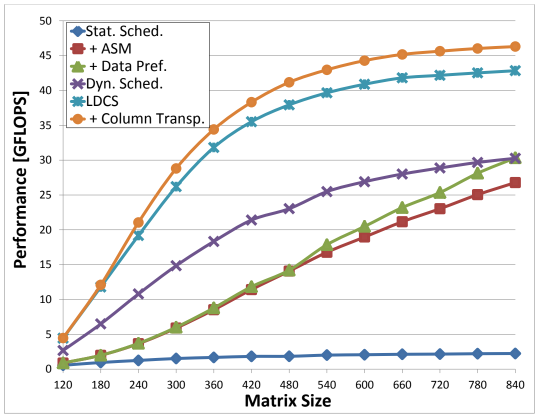
III. Experimental Evaluation

This section presents a preliminary experimental evaluation of LDCS using the IBM Cyclops-64 (C64) [15]. C64 is a homogeneous many-core architecture featuring 160 single-issue thread units (TU) per chip running at 500 MHz and a peak performance of 80 GFLOPS. C64 features a three-level software-managed memory hierarchy completely visible to the programmer. For the experiments, a highly accurate C64 simulator was used [16].

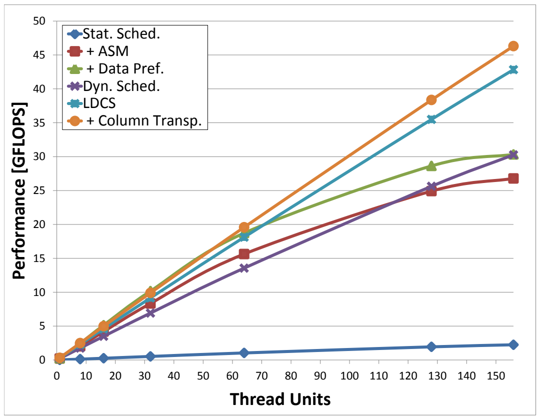
Several versions of LU factorization were designed in order to evaluate the performance of LDCS, some of which are based on those presented in [17]:

1) *Stat. Sched.:* Static scheduling version using C-Tiny-Threads [16].
2) *+ASM:* Optimized with *GETRF*, *TSTRF*, *GESSM*, and *SSSSM* tasks written in assembly.
3) *+Data Pref.:* Optimized with software pipelining and loop unrolling.
4) *Dyn. Sched.:* As the previous one but using dynamic scheduling of tasks.
5) *LDCS:* Optimized with LDCS.
6) *+Column Transp.:* Optimized with column transposition to take advantage of the instructions available on C64's ISA for the reading and writing of several data elements in a single transaction.

A tile of 6×6 was used so the block of data to be processed and any other data required by the super-task could fit in the highest level of the C64's memory hierarchy. On the LDCS versions, the first super-tasks to be scheduled to hardware threads were those containing tasks in the critical path of LU factorization, favoring *GETRF* tasks which are the bottleneck of the algorithm (see Figure 2).



(a) Weak Scaling using 156 Thread Units.



(b) Strong Scaling using an 840×840 Matrix.

Figure 3. Performance of LU factorization using LDCS on C64.

Figure 3 presents the performance results for the LU factorization running on C64. LDCS with column transposition delivers the best performance in weak and strong scaling, achieving a maximum of 46.3 GFLOPS when a matrix of 840×840 is processed with 156 TUs. This is followed by the LU factorization version using only LDCS with a maximum performance of 42.8 GFLOPS and the dynamic scheduling version with 30.62 GFLOPS.

These results also surpass those reported on [17], where a maximum performance of 39 GFLOPS was obtained with a dynamic scheduling of tasks and a diagonal tile of 12×12.

The two variants of LDCS (with and without column transposition) exhibit an almost linear scalability with respect to the number of TUs. Moreover, the only versions exhibiting a good scalability in Figure 3b are those using dynamic scheduling.

IV. Related Work

The combination of tasks in a super-task is similar to the concept of task aggregation used by the QUARK scheduler in PLASMA [18]. Task aggregation's objectives are to reduce the overhead of scheduling fine-grain tasks and to group several kernels to be computed by GPUs. The main difference of this concept with LDCS' super-tasks is that the kernels must not have dependencies among them in order to take full advantage of the GPU, whereas in LDCS dependencies among the tasks grouped in a super-task exist, mainly because all the tasks process the same block of data.

As LDCS, the technical report by Chan [6] proposes an approach to schedule to the same hardware thread the tasks overwriting a particular block of data. However, his work is bound to the specifications of the dataflow model with dataflow actors as unit of scheduling. This means that each task must read and write the block of data that processes and must perform the mandatory register book-keeping on each function call. LDCS, on the other hand, uses super-tasks as unit of scheduling, which can be signaled in the middle of their execution. Moreover, LDCS reduces the number of register book-keeping operations by inlining tasks inside a super-task. Additionally, Chan's work implements software caches (that follow the content of the hardware caches and which are analyzed in terms of hits and misses) to determine the assignment of a task to a specific hardware thread. LDCS does not need this mechanism since it uses an a priori knowledge of the DDG of the application to map super-tasks to hardware threads.

The work of Gautier et. al [5] proposes a locality-aware work stealing algorithm for dataflow programming that computes dependencies between tasks before assigning a new task to a hardware thread. Recursive applications can benefit from this approach, at the expense of producing a larger critical path. LDCS instead, computes the dependencies among tasks from the beginning if the DDG is previously known, reducing the overhead of the application during runtime and keeping the DDG of the application (and its critical path) unchanged.

The work of Chen et. al [7] in the codelet model [19] uses a codelet graph to find the best scheduling algorithm of an application targeting a many core architecture. In the graph, edges are annotated with the amount of data shared by a source and a sink codelet and codelets are grouped in order to minimize the sum of intergroup weights. As stated by the authors, the algorithm could schedule to the same hardware thread two codelets with no dependencies among them, forcing the creation of new dependencies to maintain the total order of codelets and reducing the parallelism of the application. LDCS guarantees that tasks scheduled to the same hardware thread have dependencies between them (since they overwrite a common block of data), so no extra dependencies are needed.

The works of Muddukrishna et. al [1], [2] propose a locality-aware scheduling algorithm that uses data footprint information to increase the locality of data. Unlike LDCS that uses one single queue for the scheduling of super-tasks to hardware threads, Muddukrishna et. al work's implements one queue per NUMA node and schedules each new task to the queue with the least total memory latency between the node's DRAM and last level of cache. The work of Yoo et. al [3] studies locality-aware scheduling techniques for unstructured programs (i.e. tasks have no explicit dependencies among them), while LDCS targets applications with explicit dependency information. Finally, Ding et. al [4] propose a scheduling strategy to schedule iterations of dependence-free loop nests to cores based on the reduction of data reuse distances and cache hierarchy.

V. Conclusions and Ongoing and Future Work

The efficient implementation of a locality-aware scheduling algorithm in a fine-grain programming model has been proposed in the form a new technique named Locality-Driven Code Scheduling (LDCS). This technique relaxes dataflow model specifications by using coarse-grain constructs named *super-tasks* that may be signaled in the middle of their execution. Since these constructs are composed of tasks that process a common block of data, the number of long latency memory accesses is reduced, which is translated into a reduction of the execution time of an application.

Experiments on an architecture with software-managed memory hierarchy show that LDCS can effectively improve the performance of a linear algebra kernel by 72% on average for weak scaling when compared with a dynamic scheduling version.

Current work is focused on the implementation of the technique in an architecture with hardware data caches and on its experimental evaluation in terms of energy consumption and power efficiency for LU factorization and other linear algebra kernels such as Cholesky and QR decomposition. Future work will focus on the analysis of the potential benefits that LDCS can have in heterogeneous systems using accelerators and in the possible addition of the technique to a compiler, given the systematic analysis performed by LDCS on the DDG of an application.

VI. Acknowledgements

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717. This work was partly supported by European FP7 project TERAFLUX, id. 249013.

References

[1] A. Muddukrishna, M. Brorsson, and V. Vlassov, "A locality approach to architecture-aware task-scheduling in openmp," in *OpenMP in the Era of Low Power Devices and Accelerators*, ser. Fourth Swedish Workshop on Multi-core Computing, 2011.

[2] A. Muddukrishna, P. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on numa systems," in *OpenMP in the Era of Low Power Devices and Accelerators*, ser. Lecture Notes in Computer Science, A. Rendell, B. Chapman, and M. MÃijller, Eds. Springer Berlin Heidelberg, 2013, vol. 8122, pp. 156–170.

[3] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, “Locality-aware task management for unstructured parallelism: A quantitative limit study,” in *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’13. New York, NY, USA: ACM, 2013, pp. 315–325. [Online]. Available: http://doi.acm.org/10.1145/2486159.2486175

[4] W. Ding, Y. Zhang, M. Kandemir, J. Srinivas, and P. Yedlapalli, “Locality-aware mapping and scheduling for multicores,” in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, 2013, pp. 1–12.

[5] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin, “XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures,” in *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, États-Unis, May 2013. [Online]. Available: http://hal.inria.fr/hal-00799904

[6] E. Chan, “Runtime data flow scheduling of matrix computations. FLAME Working Note #39,” The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-09-22, Aug. 2009.

[7] C. Chen, Y. Wu, J. Sutterlein, L. Zheng, M. Guo, and G. R. Gao, “Automatic locality exploitation in the codelet model,” in *11th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA-13)*, 2013.

[8] J. B. Dennis, “First version of a data-flow procedure language,” in *Proceedings of the Colloque sur la Programmation*, ser. number 19 in Lecture Notes in Computer Science, Apr. 1974.

[9] J.-L. Gaudiot and M. D. Ercegovac, “Performance analysis of a data-flow computer with variable resolution actors,” in *ICDCS*, 1984, pp. 2–9.

[10] US Department of Energy, “ASCR Programming Challenges for Exascale Computing,” in *2011 Workshop on Exascale Programming Challenges*, Jul. 2011.

[11] J. J. Dongarra and D. W. Walker, “Software libraries for linear algebra computations on high performance computers,” *SIAM Rev.*, vol. 37, no. 2, pp. 151–180, Jun. 1995. [Online]. Available: http://dx.doi.org/10.1137/1037042

[12] J. Dongarra, P. Luszczek, and A. Petitet, “The linpack benchmark: past, present and future,” *Concurrency and Computation: Practice and Experience*, pp. 803–820, 2003.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: characterization and methodological considerations,” *SIGARCH Comput. Archit. News*, vol. 23, no. 2, pp. 24–36, May 1995. [Online]. Available: http://doi.acm.org/10.1145/225830.223990

[14] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov, “LU factorization for accelerator-based systems,” in *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*, 2011, pp. 217–224.

[15] M. Denneau, “Cyclops,” in *Encyclopedia of Parallel Computing: SpringerReference (www.springerreference.com)*, D. Padua, Ed. Springer-Verlag Berlin Heidelberg, 2011.

[16] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, “FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture,” in *Workshop on Modeling, Benchmarking, and Simulation (MoBS ’05). In conjunction with the 32nd Annual International Symposium on Computer Architecture (ISCA ’05)*, 2005, pp. 11–20.

[17] E. Garcia, J. Arteaga, R. Pavel, and G. R. Gao, “Optimizing the LU Factorization for Energy Efficiency on a Many-Core Architecture,” in *Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2013)*, Santa Clara, CA, USA, Sep. 2013.

[18] J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra, “Multithreading in the PLASMA Library,” in *Multi and Many-Core Processing: Architecture, Programming, Algorithms, & Applications*. Ahmed, M., Ammar, R., Rajasekaran, S. eds. Taylor & Francis, 2013.

[19] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, “Position paper: Using a "codelet" program execution model for exascale machines,” in *1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT 2011)*, 2011.