University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Locality-Driven Scheduling of Tasks for Data-Dependent Multithreading

Jaime Arteaga, Stephane Zuckerman, Elkin Garcia, Robert Pavel, and Guang Gao

CAPSL Technical Memo 125
January, 2014

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Contents

Abstract

The amount of data movement in an application has a direct impact on both its execution time and power consumption. One way to reduce this, is the implementation of locality-aware scheduling algorithms to maximize the reuse of data during the assignment of work to hardware threads. Locality-Driven Code Scheduling (LDCS), an example of such algorithms, groups the tasks that process a common data block as phases of a single coarse-grain construct named super-task, with each phase being fired according to dataflow semantics. LDCS reduces the number of long latency operations by executing all the phases of a super-task with the same hardware thread and by reading and writing the data block from and to main memory only with the first and last phases of the super-task, while the others rely on the presence of the block in the upper levels of the memory hierarchy. This paper analyzes the impact that LDCS can have on the execution time and the power consumption of an application, and presents experimental results performed on two systems: one with a software-managed memory hierarchy and another one with hardware data caches, showing that LDCS can improve the power efficiency of an application up to 72% and by 28% on average, respectively, for weak scaling.

1 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 exploiting data locality as much as possible.

Scheduling techniques based on data locality have been studied for multicore architectures using common multithreaded 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].

This document presents an analysis and experimental evaluation such improvements with the following main contributions:

(1) An analysis of the reduction in the amount of data movement of an application obtained with LDCS;

(2) an analysis of the power savings achieved with such reduction; and

(3) experimental results on a architecture with a software-managed memory hierarchy and another one with hardware data caches that confirms the LDCS' ability to improve the power efficiency of an application.

To that end, this paper is organized as follows: A description of LDCS and its implementation using a linear algebra application is presented in Section 2, outlining its advantages in terms of performance and energy consumption; experimental results on two architectures are presented in Section 3, followed by a review of related work in Section 4. Finally, Section 5 presents concluding remarks and an outline of future work.

2 Locality-Driven Code Scheduling

2.1 Definition

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
```

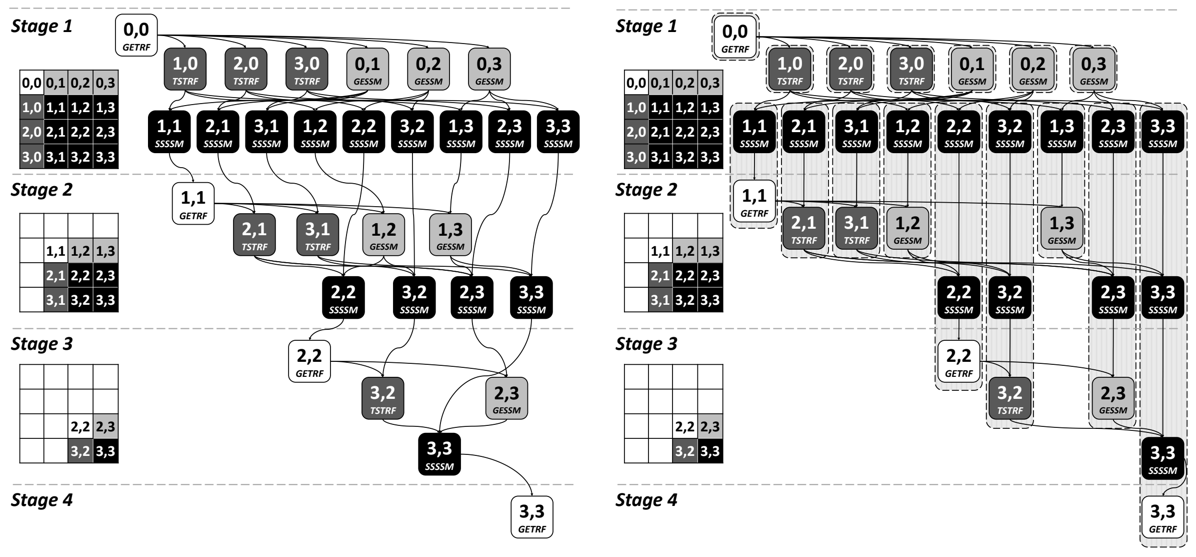
The implementation of an application using LDCS can effectively improve its performance since the number of long latency operations is reduced. LDCS may also improve its power efficiency due to the cost (in terms of energy) that must be paid when moving data across the different memory levels of a multi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

2.2 LU Factorization using LDCS

As an example of an application that can benefit from LDCS, Figure 1a presents the data dependency graph (DDG) for the LU factorization of a matrix $A_{M,M}$, which has been divided in blocks of $T \times T$ elements for an efficient blocked implementation. LU factorization is a popular linear algebra application used for the performance evaluation of modern supercomputers [11, 12, 13].

Since the number of blocks to process on each stage (and therefore to write back to memory) decreases by one in each dimension (see Figure 1a) and the number of store operations per block is T^2, the total number of store operations in the LU factorization can be defined as in Eq. 1:

$$Total\ ST = T^2 \times \left(\sum_{k=0}^{S-1} (S-\ k)^2 \right) = T^2 \times \left(\frac{1}{3}S^3 + \frac{1}{2}S^2 + \frac{1}{6}S \right) \quad (1)$$



(a) Classical Blocked Implementation (b) LDCS Implementation

Figure 1: DDG for the LU factorization of a matrix $A_{M,M}$ with 4×4 blocks: *GETRF* tasks are white, *TSTRF* are dark gray, *GESSM* are light gray, and *SSSSM* are black; dashed boxes enclose tasks grouped as a single super-task and computed by the same hardware thread.

Where S represents the total number of stages of the algorithm, defined by Eq. 2.

$$S = \frac{M}{T} \tag{2}$$

Once determined the DDG of the application, LDCS can be applied by grouping and inlining in a single super-task the tasks processing a common data block, as shown in Figure 1b. For instance, the super-task on the far right is composed of four phases (three *SSSSM* phases and one *GETRF* phase), which will process data block (3,3) using the same hardware thread. Of these four, only the first *SSSSM* phase will read the data block from main memory, while the last phase (*GETRF*) will be the only one writing it back.

Consequently, each data block is written in main memory only once. In this case, the total number of store operations with LDCS can be defined as in Eq. 3, which is significantly lower in comparison with the one calculated for the typical blocked implementation (Eq. 1).

$$Total\ ST\ _{LDCS} = (T \times S)^2 \tag{3}$$

The number of load operations is also reduced since each data block is loaded only once to be processed. After being written in main memory, a block may be read by another super-task's phase, but only to be used as an input during the computation of other block(s), not to be processed again and, therefore, would not require additional store operations.

3 Experiments

3.1 Experimental Platforms

The experimental evaluation of LDCS was performed on two platforms. The first was C64, a homogeneous many-core architecture with a software-managed memory hierarchy completely controlled by the programmer. This hierarchy is composed of three levels: A 16 KB on-chip scratch-pad memory, a 2.5 MB on-chip global SRAM memory, and a 1 GB external DRAM memory. One C64 chip contains 160 single-issue hardware threads, clocked at 500 MHz. Applications for C64 were simulated with FAST, a highly accurate C64 simulator [14].

The second platform was a 2-socket Intel Xeon E5-2670 system running at 2.6 GHz. Each processor features 8 cores and 16 hardware threads (for a total of 32 threads of execution), with a 32 kB L1 data cache, a unified 256 kB L2 cache, and a 20MB shared unified L3 cache. Experiments for this system were written in C using Pthreads and the Intel's Math Kernel Library (MKL) in sequential mode. Intel's *icc* was also used with flags `-mavx` and `-O3`.

3.2 Design of Experiments

Figure 2 presents the different versions of the LU factorization developed for the two experimental platforms. A variant of the algorithm without pivoting was used as our benchmark. The assignment of super-tasks to hardware threads in the LDCS versions was made by giving more priority to super-tasks containing tasks in the critical path of the algorithm's DDG (see Figure 1).

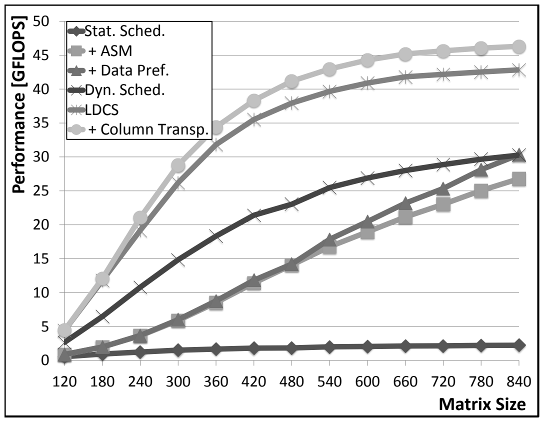
C64		**Intel Xeon**	
Version	***Features***	***Version***	***Features***
Stat. Sched.	Static Scheduling of Tasks in C	***MKL – DGETRF***	With CBLAS' DGETRF
+ ASM	Super-tasks written in assembly	***Stat. Sched.***	Pthreads with CBLAS' DAXPY and DGEMM using 100x100 tiles
+ Data Pref.	With software pipelining and loop unrolling	***Dyn. Sched.***	With dynamic scheduling of tasks
Dyn. Sched.	With dynamic scheduling of tasks using 6x6 tiles		
LDCS	With LDCS following Figure 2	***LDCS***	With LDCS following Figure 2
+ Column Transp.	With transposed storing of data to exploit C64 features		

Figure 2: Versions of LU factorization developed for the experimental evaluation of LDCS.

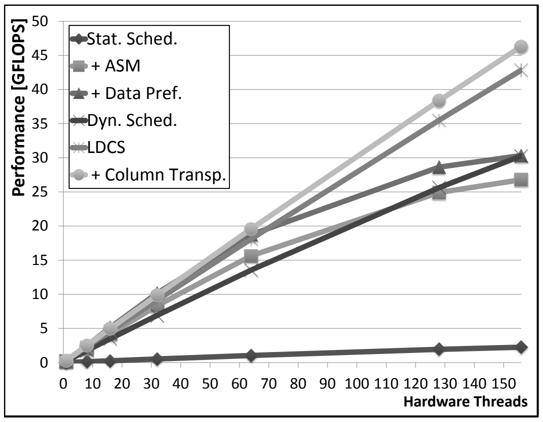
3.3 Experimental Results using a Software-Managed Memory Hierarchy

3.3.1 Performance

Figure 3 presents the performance results for C64. The most important observations to be made about these results are the improvements in performance obtained with LDCS with respect to dynamic scheduling and the better scalability of the LDCS versions, particularly for strong scaling.



(a) Weak Scaling using 156 Hardware Threads.



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

Figure 3: Performance of LU factorization on C64: Higher is better.

3.3.2 Energy Consumption

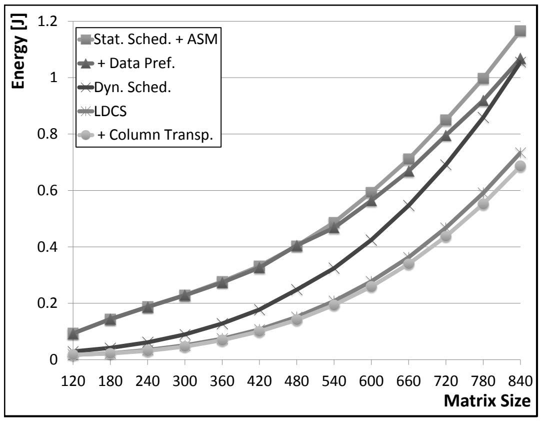
Energy consumption on C64 was measured by counting the total number of instructions executed by the application and using static energy coefficients, following the method proposed by Garcia et al. [15].

As can be seen in Figure 4, LDCS versions exhibit the smallest energy consumption. The difference between LDCS and the other versions is more noticeable in weak scaling than in strong scaling, where the improvements in energy consumption are more significant after using 128 hardware threads. The static scheduling version (not shown in Figure 4 in order to have a better view of the charts for the other versions) exhibited the highest energy consumption for weak and strong scaling.

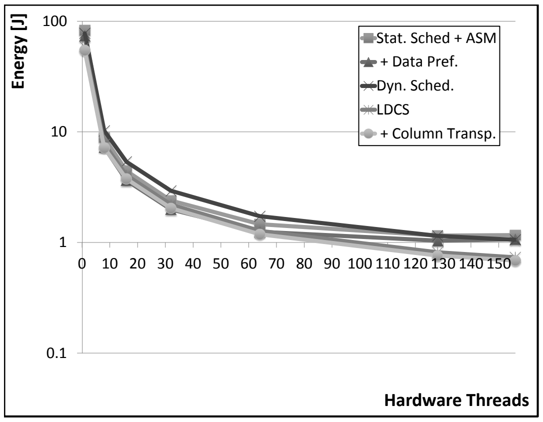
3.3.3 Power Efficiency

Power efficiency was calculated by dividing the performance of the application (measured in FLOPS and presented in Figure 3) by its power (measured in Watts and presented in Figure 4).

The power efficiency results for C64 are presented in Figure 5. LDCS with column transposition has the best power efficiency, being superior by 72% in weak scaling and by 46% in strong scaling, on average, with respect to the dynamic scheduling version. Even though the charts for performance and power efficiency on C64 look similar, a careful analysis reveals that the performance charts for LDCS with strong scaling (Figure 3b) are more linear than those for power efficiency (Figure 5b). This means that as the number of hardware threads increases, it becomes more expensive to reach a given performance value.

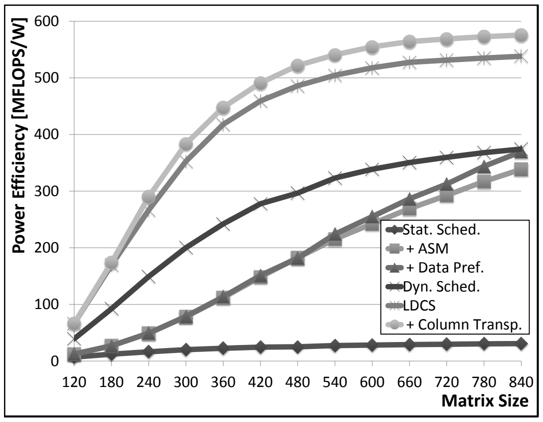


(a) Weak Scaling using 156 Hardware Threads.

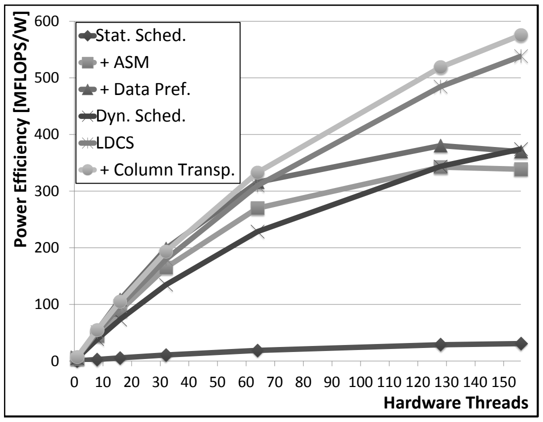


(b) Strong Scaling using an 840×840 Matrix (logarithmic scale).

Figure 4: Total Energy Consumption of LU factorization on C64: Lower is better.



(a) Weak Scaling using 156 Hardware Threads.



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

Figure 5: Power Efficiency of LU factorization on C64: Higher is better.

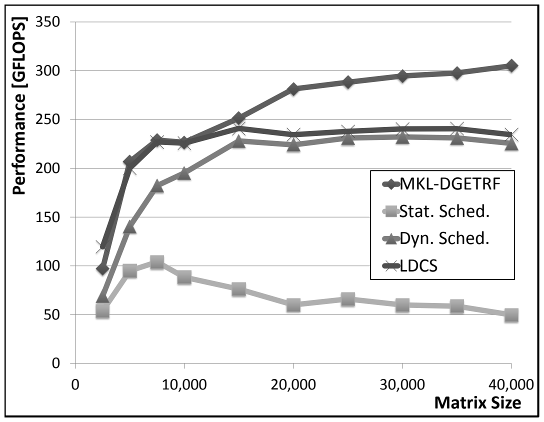
3.4 Experimental Results using Hardware Data Caches

3.4.1 Performance

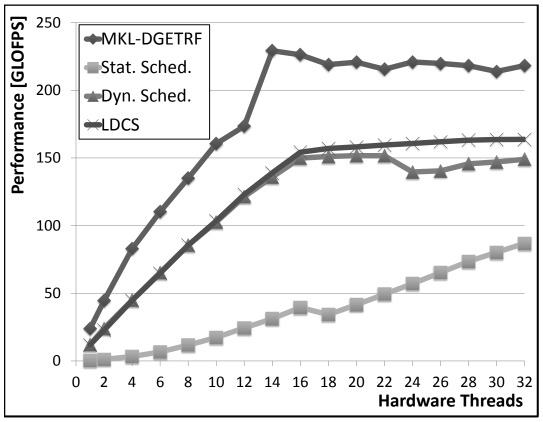
Figure 6 presents the performance results for Intel Xeon. For both weak and strong scaling, LDCS follows an optimized implementation of the application using Intel's MKL. For strong scaling, it is important to notice how both versions start to stagnate when using 16 or more hardware threads, when some L1 and L2 caches are shared by two hardware threads.

3.4.2 DRAM Power Consumption

Power results on Intel Xeon were obtained by reading the RAPL counters available in Sandy Bridge architectures using the *likwid* suite v.3.1 [16, 17]. Figure 7 presents the average power



(a) Weak Scaling using 32 Hardware Threads.

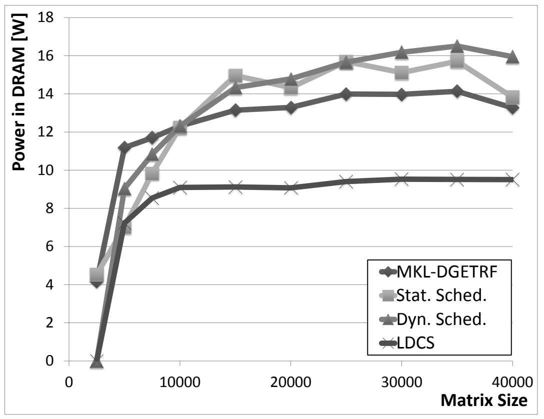


(b) Strong Scaling using a $10k \times 10k$ Matrix.

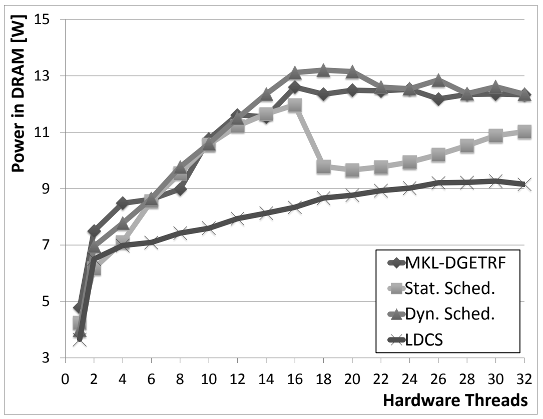
Figure 6: Performance of LU factorization on Intel Xeon: Higher is better.

consumption in the DRAMs of the two NUMA nodes of the Intel Xeon experimental platform described in subsection 3.1.

For weak and strong scaling, LDCS exhibits the smallest power consumption in DRAM. Intel's MKL follows LDCS for weak scaling, while for strong scaling the static scheduling version is the second best. These results were expected since LDCS main objective is the reduction of the amount of data movement (and its associated energy cost) by accessing main memory only with the first and last phases of the super-tasks, which is directly translated into improvements in the power consumption of the application [10, 18]. Energy results on DRAM, not presented for lack of space, showed that LDCS exhibits the smallest consumption, being followed closely by Intel's MKL.



(a) Weak Scaling using 32 Hardware Threads.



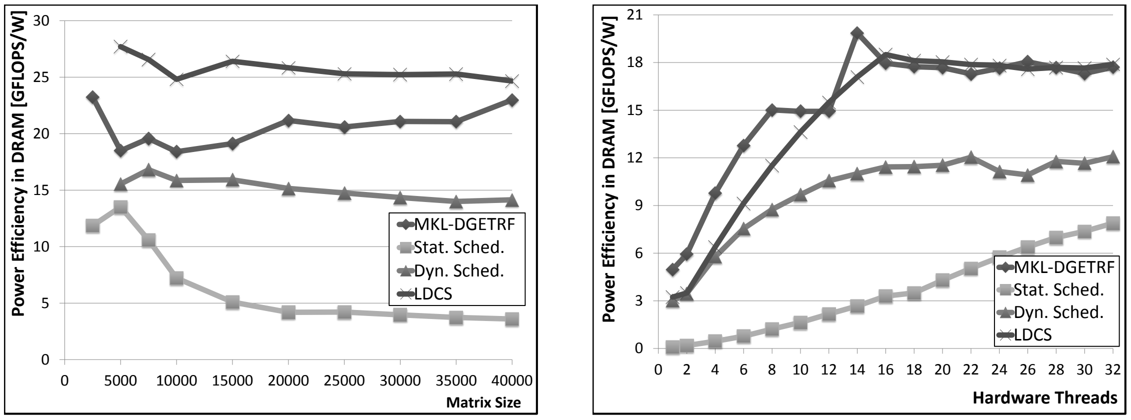
(b) Strong Scaling using a $10k \times 10k$ Matrix.

Figure 7: Average DRAM Power Consumption of LU factorization on Intel Xeon: Lower is better.

3.4.3 DRAM Power Efficiency

DRAM power efficiency for Intel Xeon was calculated as the ratio between the performance of the application (measured in GFLOPS and presented in Figure 6) and the power consumption in DRAM (measured in Watts and presented in Figure 7).

For weak scaling, LDCS's power efficiency is the best by 28% on average with respect to Intel's MKL, which comes second. For strong scaling, LDCS surpasses Intel's MKL by 1.3% on average when the total number of hardware threads is more than 16.



(a) Weak Scaling using 32 Hardware Threads. (b) Strong Scaling using a $10k \times 10k$ Matrix.

Figure 8: DRAM Power Efficiency of LU factorization on Intel Xeon: Higher is better.

4 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 [19]. 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 [20] 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.

5 Conclusions and Ongoing and Future Work

An analysis and experimental evaluation of the impact that LDCS has in the power efficiency of an application has been presented. LDCS is a locality-aware scheduling technique that groups under a single super-task all the tasks processing the same data block. With LDCS, only the first and last phases of the super-task are required to access main memory to read and write, respectively, the data block processed by all the phases of the super-task, reducing this way the amount of data movement made by the application and its associated energy cost.

Experimental evaluation performed on two architectures showed that the reduction in the amount of data movement achieved with LDCS can effectively improve the power efficiency of an application. On an architecture with software-managed memory hierarchy, an improvement of 72% on average in weak scaling was obtained in comparison with a dynamic scheduling version of the application. Performance is also greatly increased thanks to the complete control the programmer has on the content of all the memory levels.

On an architecture with hardware data caches, LDCS improves the DRAM power efficiency of the application by 28% on average, also in weak scaling, versus a highly optimized version of the application using Intel's MKL. On this architecture, LDCS' performance is competitive due to the increase in data locality obtained by executing with the same hardware thread all the tasks that process a common data block and by inlining such tasks in a single super-task.

Future work will focus in the optimization of LDCS in order to obtain further improvements in performance in architectures with hardware data cache and in the implementation of other applications such as Cholesky and QR Decomposition using LDCS. Furthermore, we intend to analyze the advantages that LDCS could offer in heterogeneous systems using accelerators.

6 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. Mller, 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.

[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 Inter-*

national Parallel & Distributed Processing Symposium (IPDPS), Boston, Massachusetts, États-Unis, May 2013.

[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] M. Horton, S. Tomov, and J. Dongarra, "A class of hybrid lapack algorithms for multicore and gpu architectures," in *Symposium on Application Accelerators in High-Performance Computing*, ser. SAAHPC '11, 2011.

[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] 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.

[14] 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.

[15] E. Garcia, D. Orozco, and G. Gao, "Energy efficient tiling on a Many-Core Architecture," in *Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2011)*, Heraklion, Greece, Jan. 2011, pp. 53–66.

[16] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ser. ICPPW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 207–216.

[17] (Last Visit: 2014) likwid - Lightweight performance tools. http://code.google.com/p/likwid/.

[18] G. Kestor, R. Gioiosa, D. Kerbyson, and A. Hoisie, “Quantifying the energy cost of data movement in scientific applications,” in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013, pp. 56–65.

[19] 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.

[20] 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.