Energy-Efficient Mapping of Streaming Tasks for Crown Scheduling on Many-Core Systems

Nicolas Melot, Christoph Kessler
Linköping University, Sweden
⟨name.surname⟩@liu.se

Jörg Keller
FernUniversität in Hagen, Germany
joerg.keller@FernUni-Hagen.de

ABSTRACT
The allocation, mapping and frequency scaling problem for malleable tasks for multiprocessors with discrete frequency scaling proved to be very challenging. Crown scheduling simplifies the problem by recursively partitioning the cores of a multiprocessor system into groups whose number scales only linearly with the number of cores, and to which parallel tasks can be mapped. It allows to optimize resource allocation, task mapping to processors and discrete task frequency scaling either separately or together in an integrated manner. In this article, we show that the mapping phase of phase-separated crown scheduling benefits from load-balancing tasks over processors, as reducing makespan gives more optimization opportunities to the frequency scaling phase. We provide an optimal load-balancing method based on integer linear programming and we introduce the *Longest Task, Lowest Group* (LTLG) heuristic as a generalization of the Longest Processing Time (LPT) algorithm to achieve load-balancing of parallel tasks. Our experiments indicate that our heuristic produces makespan close to optimal and this benefits the global phase-separated crown scheduling technique.

1. INTRODUCTION
Modern manycore processors exploit parallelism to achieve high computation throughput or low power consumption. Beside using more scalable parallel algorithms, significant performance improvement can be achieved with good techniques for resource allocation, mapping and scheduling. However, computing optimal solutions to maximize throughput or minimize energy consumption is an NP-hard problem. Optimizing resource utilization of processing units, memory or communication channels while taking into account architecture constraints is thus not practical using optimal algorithms. The problem worsens when the number of tasks or resources are increased with growing computational power demand and future processor architectures.

Kessler et al. [2] explore the problem of energy-efficient periodic scheduling of parallel streaming tasks. They describe *crown scheduling* as a task resource allocation, core mapping, scheduling and discrete frequency scaling methodology to optimize off-line a given task collection's energy consumption, given a throughput constraint. It assumes a multiprocessor system whose cores are identical and whose running frequency can be scaled individually for each task they run. It schedules malleable and sequential tasks together, assuming that the overall number of processors is a power of b (we

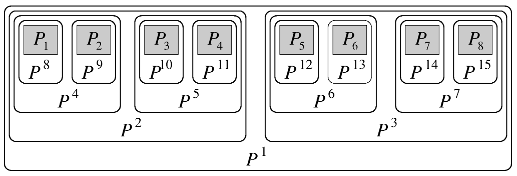


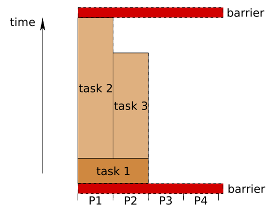
Figure 1: Crown decomposition of 8 cores into 15 recursive groups

use $b = 2$). We call *malleable* a task whose execution time is a function of the number of processors allocated to it [3]. Crown scheduling decomposes recursively a multiprocessor's cores into groups of exponentially decreasing size, and likewise exponentially increasing number of groups of a given size (Fig. 1). Crown scheduling assumes that the tasks it schedules run only on entire groups i.e. power-of-b number of processors[1]. This aims to reduce the mapping problem exploration space, making the problem easier and faster to solve. Crown scheduling can select discrete frequencies for tasks across their assigned core and models the cost of tasks' parallelization through task-individual efficiency functions. This implicitly models communication and synchronization necessary to parallelize a task. However, Kessler et al. [2] do not yet consider the cost of core-to-core communications in their energy consumption and performance model.

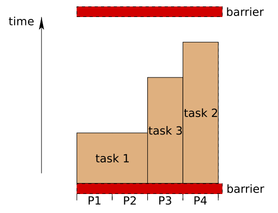
In this paper, we describe an optimal ILP formulation to load-balance tasks mapped to groups at mapping time. We introduce the LTLG heuristic that creates load-balanced mappings for malleable tasks. We demonstrate that our heuristic can accelerate a phase-separated crown scheduler and produces mappings whose load balance is close to optimal. We show further that load-balanced mappings contribute to more energy-efficient crown schedules.

The paper is organized as follow: In Section 2, we summarize crown scheduling. We argue that a good load-balancing strategy can enhance the scheduling solutions it generates and we provide an ILP formulation for an optimal load balancing in the mapping step. Section 3 describes in detail the LTLG heuristic. In Section 4, we evaluate the optimization time and the improved mapping quality resulting from optimal and heuristic-based load balancing, and we observe that our heuristic produces makespans close to optimal. Sec-

[1]This limitation can be relaxed in practice through a crown configuration [2]



(a) Load balanced among groups only.



(b) Load balanced among all cores only.

Figure 2: mapping quality with or without load-balancing

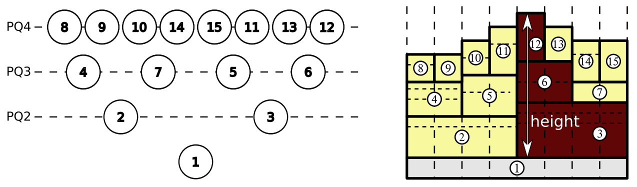


Figure 3: Priority queues of a 8-core system on the left, sorted after the current state of the schedule on the right. The arrow shows the current height of groups 3, 6 and 12.

tion 5 discusses related work and Section 6 concludes and proposes future work.

2. CROWN SCHEDULING

Crown scheduling takes one root group of p processors and divides it in $b > 1$ groups of size p/b. It further divides these groups recursively until obtaining p groups of size 1. A group contains the cores of all groups it has been divided into. We obtain a total $(b \cdot p - 1)/(b - 1)$ groups, and $\log_b(p)$ different group sizes. Crown allocation [2] allocates a number of w_t processors to each of each tasks t (in total n tasks) where w_t is the size of an existing group. At the mapping phase, each task t is mapped to exactly one group of size w_t. A task t has workload $load_t$ and runs for $time_t = load_t / f_t$ time steps at frequency f_t using one processor. A group g has load $load_g$, which is the sum of all tasks' $load_t$ mapped to g (Eq. 1) and time $time_g$ the cumulative $time_t$ of all its tasks. Similarly, a processor p has $load_p$ the sum of the loads $load_g$ of all groups in $G_p = \{g : p \in g\}$ it belongs to (Eq. 2) and $time_p$ the cumulative time of all its groups. We call a group g's height $height_g$ the sum of its load, all its ancestors except group 1, and the maximal height of its b children (Eq. 4).

A crown scheduler must fulfill a makespan constraint, that is, $time_p$ must be lower than a target makespan M for all cores p, and minimize the energy consumption of the schedule. The energy consumption of a crown schedule grows linearly with cores' idle and execution time. Furthermore, the energy of an execution time unit grows cubically with its running frequency. Therefore, a crown scheduler tries to reduce tasks' running frequencies while satisfying the makespan constraint.

Crown schedulers have been implemented using two variants of ILP models [2]. One integrates the allocation, mapping and frequency scaling into a single complex problem. The second relaxes the problem by separating all these phases and sequentially computes an optimal solution for each of them [2]. The integrated variant produces better solutions, but the phases-separated implementation runs faster. This variant minimizes only the groups' load at mapping time; although it benefits from minimizing processors' load (Fig. 2), this considerably increases its runtime. We can achieve an optimal processor load at the mapping step by adding the $maxload$ variable to its ILP formulation, defining it as at least as large as each processors' load (Eq. 3) and minimizing $maxload$. Section 3 describes a polynomial-time heuristic for this problem.

$$\forall g \in [1; \frac{b \cdot p - 1}{b - 1}] : load_g = \sum_{t=1}^{n} load_t \cdot y_{t,g} \quad (1)$$

$$\forall i \in [1; p] : load_p = \sum_{g \in G_p} load_g \quad (2)$$

$$maxload = \max_{p \in [1;p]} load_p \quad (3)$$

3. HEURISTIC CROWN MAPPING WITH LOAD-BALANCING

We describe the *Longest Task, Lowest Group* (LTLG) mapping heuristic for the load balancing problem modeled in Sec. 2. The algorithm is based on three informal intuitions to produce better load-balanced mappings:

1. The assignment of each task t to the group g of least $height_g$ among groups of size w_t limits load imbalance while building a schedule.
2. The insertion of tasks of highest parallel running time first (with w_t processors) lowers the risk of creating load imbalance when adding the last tasks to a schedule.
3. If tasks have the same parallel running time, inserting tasks of highest width first keeps lowest width tasks available to later fill in the crown's holes and better balance it.

The algorithm maintains $\log_b p$ priority queues, one per group size. The priority of groups is defined by their height, or their group number if both groups have the same current height (Eq. 5). Since tasks of size $w_t = p$ do not create load imbalance and as the root group is the only group of size p, we don't use priority queues for it (Fig. 3). The algorithm first sorts the tasks in descending order; the tasks are compared with both their parallel running time and their width using Eq. 6. Considering tasks in this order, we then assign task t to the highest priority group g of size w_t as given by its priority queue. Then we update the load and the height of g as well as all all its children groups, recursively. If the new height of g is the maximum height among all children of group $g' = \lfloor g/b \rfloor$, then the height of g' is also updated, and this is repeated for g' until the root group is reached.

$$\forall g \in [1; \frac{b \cdot p - 1}{b - 1}] : height_g = load_g + \sum_{i=1}^{\lfloor \log_b g \rfloor} load_{\lfloor \frac{g}{b^i} \rfloor} + \max(\{height_{g'} : g' \in [g \cdot b; (g+1) \cdot b]\}) \quad (4)$$

$$cmp(g1, g2) = \begin{cases} height_{g1} > height_{g2} & \text{if } height_{g1} \neq height_{g2}, \\ g1 > g2 & \text{otherwise.} \end{cases} \quad (5)$$

$$cmp(t1, t2) = \begin{cases} load_{t1} > load_{t2} & \text{if } load_{t1} \neq load_{t2}, \\ w_{t1} > w_{t2} & \text{otherwise.} \end{cases} \quad (6)$$

4. EXPERIMENTAL EVALUATION

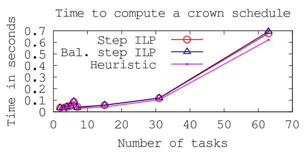
We implement the LTLG heuristic in C++. We substitute the mapping step of the phase-separated variant [2] with our heuristic and we use the same set of randomly-generated synthetic task collections as benchmark. We group synthetic task collections in different categories defined by the number of cores ($p \in \{1, 2, 4, 8, 16, 32\}$), the number of tasks (10, 20, 40 and 80 tasks), tasks' work distribution (triangular or uniform random distributions), and tasks' maximum width W_t: serial ($W_t = 1$ for all t), low ($1 \leq W_t \leq p/2$), average ($p/4 \leq W_t \leq 3p/4$), high ($p/2 \leq W_t \leq p$) and random ($1 \leq W_t \leq p$), always in a uniform distribution. The target makespan of each task collection is defined by Eq. 7.

$$M = \frac{\sum_t task\_load(t)}{2 \cdot min\_freq} + \frac{\sum_t task\_load(t)}{2 \cdot max\_freq} \quad (7)$$

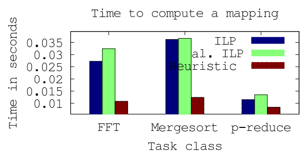
We run both the ILP solver and our heuristic on a quad-core system i7 Sandy Bridge processor at 3GHz and 8GB of main memory. We use Gurobi 5.10 and ILOG AMPL 10.100 with 5 minutes timeout to solve the ILP models and we compile the C++ implementation with gcc 4.4.3 on Ubuntu 10.04.

We measure the overall runtime and schedule quality of the crown scheduler using an optimal load-balanced mapping solver as well as our LTLG heuristic. We compare it to the integrated and the non-balanced, phase-separated ILP implementations [2]. We also measure the runtime of the mapping step alone of non-balanced, optimally balanced and LTLG heuristic-based implementations. We perform the same comparisons using task collections of classic streaming algorithms: FFT, parallel-reduction and mergesort. FFT is characterized by $2p - 1$ parallel tasks of a balanced binary tree. In level $l \in [0; \log_2 p - 1]$ of the tree, there are 2^l tasks of width $p/2^l$ and work $p/2^l$ so all tasks can run in constant time. Parallel reduction involves $\log p$ tasks of maximum width 2^l and work 2^l for $l \in [1; \log_2 p]$; they can also run in constant time. Finally, a mergesort task collection is similar to FFT, but all tasks are sequential.

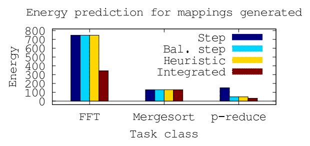
Figures 4 and 5 show mean values for 1,2,4,8,16 and 32 cores, 10, 20, 40 and 80 tasks and across all width classes $\{Serial, Low, Average, High, Random\}$. Figures 4(a) and 4(b) show that our heuristic is considerably faster (3.9× on average) than the phase-separated ILP variant. Both plots are similar, although Fig. 4(a) yields a small offset over Figure 4(b)'s. This shows that the overall computation is mostly dominated by the mapping step.



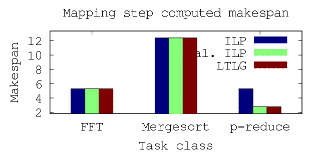
(a) Overall runtime by number of cores



(b) Mapping makespan by task collection class



(c) Energy prediction by task collection class



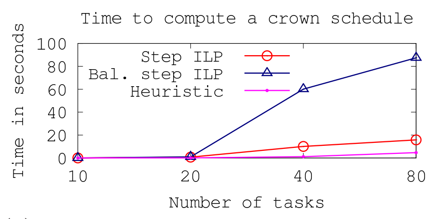
(d) Mapping makespan by task collection class

Figure 5: Overall runtime, mapping runtime, makespan and energy prediction for FFT, mergesort and parallel-reduction

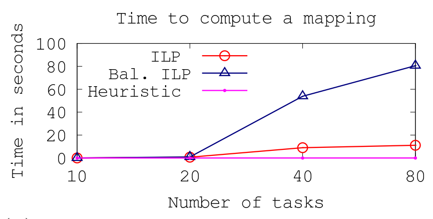
We observe on Fig. 4(c) and 4(d) that the optimal load-balanced mapping does not bring significant energy improvements (average 0.01%) over non-balanced implementation. However, the LTLG-based phase-separated crown scheduler produces schedules of the same quality as the optimal ILP implementation. The integrated solutions are consistently of a higher quality than all other variants. Figure 4(c) shows that the difference seems to grow with the average degree of parallelism of tasks to schedule.

The LTLG mapping heuristic produces makespans very close to optimal (Fig. 4(e)) for sequential tasks and tasks of low and average degrees of parallelism (0.001% worse on average). For highly parallel task collections, the LTLG heuristic produces better solutions than the optimal load-balanced mapping implementation could produce within the 5 minutes timeout. In addition, Fig. 4(f) demonstrates that our heuristic is much faster than ILP non-balanced (8.16× on average) and ILP balanced (580× on average).

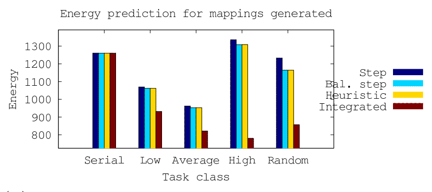
Figure 5(a) shows a slight overall runtime improvement (average 15% faster) to compute the schedule of real tasks collections, whereas Fig. 5(b) shows faster (2.2× average) mapping optimization time for our heuristic. This shows that the overall scheduling runtime is now dominated by the frequency scaling step. We see from Fig. 5(c) and 5(d) that any load-balancing method only improves the mapping makespan and schedule quality of parallel-reduction tasks collections. Parallel-reduction benefits from load balancing, as in the situation shown in Fig. 2. FFT and mergesort do not benefit from additional load balancing.



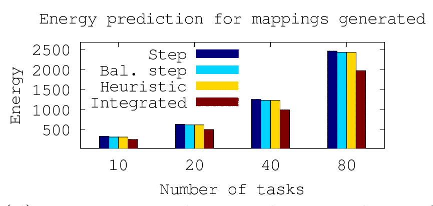
(a) Overall optimization time by number of tasks



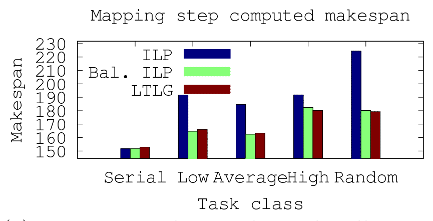
(b) Mapping step runtime by number of tasks



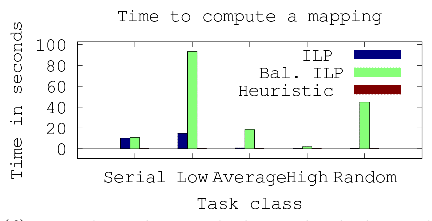
(c) Energy prediction by task collection class



(d) Energy prediction by number of tasks



(e) Mapping makespan by task collection class



(f) ILP-based non-balanced, balanced and LTLG runtime at mapping step

Figure 4: Runtime, mapping runtime, makespan and energy prediction for the synthetic tasks collection

5. RELATED WORK

This load-balancing problem is different from bin packing: bin packing minimizes the number of bins where bins are constrained by the load they admit; instead, the LTLG heuristic minimizes the load of bins and is constrained by the number of bins or groups. Also, bin packing assumes independent bins, but the crown structure makes processor groups dependent on their children and ancestors.

The LTLG heuristic is a generalization of the LPT (Longest Processing Time) algorithm [1], constrained by the crown structure. Considering the mapping of sequential tasks only, priority is given to map tasks running for the longest time, greedily mapping to singleton groups of the least load. As the LPT algorithm is a 4/3-approximation of optimal load-balancing, so is LTLG in the case of sequential tasks. *Home Scheduling* [4] dynamically distributes tasks based on data locality, to the least loaded thread whose cache holds the data the task operates on. Sanders and Speck [5] consider energy-efficient scheduling of n independent *continuously* malleable tasks and *continuous*, unbounded frequency scaling under a deadline constraint. Under these assumptions, they propose a near linear time algorithm that finds an optimal solution.

6. CONCLUSION AND FUTURE WORK

We described a load-balancing method for the mapping step of phase-separated crown scheduling and show how it can improve the quality of its solutions. We presented the *Longest Task, Lowest Group* heuristic (LTLG), a generalization of the LPT-algorithm, to produce load-balanced mapping of malleable tasks. We demonstrate that using this heuristic lowers the overall phase-separated crown scheduling runtime when the ILP runtime is dominated by the mapping step. We show that it contributes to computing more energy efficient schedules. Future work will introduce a heuristic for the frequency scaling step in order to achieve a fast, scalable, efficient crown scheduler. Such heuristic could also be used to dynamically rescale a crown schedule if one or several tasks are not data-ready in a round. Finally, we plan to implement a benchmark set on concrete many-core platforms such as the SCC or MPPA and compare the performance of our crown scheduler to other non-crown schedulers for malleable tasks.

Acknowledgments

Partial funding by Vetenskapsrådet and SeRC, and by the CUGS graduate school at Linköping University.

References

[1] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[2] C. Kessler, N. Melot, P. Eitschberger, and J. Keller. Crown Scheduling: Energy-Efficient Resource Allocation, Mapping and Discrete Frequency Scaling for Collections of Malleable Streaming Tasks. In *Proc. of 23rd Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2013)*, 2013.

[3] W. T. Ludwig. *Algorithms for Scheduling Malleable and Nonmalleable Tasks.* PhD thesis, Univ. Wisconsin-Madison, 1995.

[4] A. Muddukrishna, M. Brorsson, and V. Vlassov. A locality approach to architecture-aware task-scheduling in OpenMP. In *Proc. of Fourth Swedish Workshop on Multicore Computing, Linköping University*, pages 23–25, November 2011.

[5] P. Sanders and J. Speck. Energy efficient frequency scaling and scheduling for malleable tasks. In *Proc. of the 18th Int. Conference on Parallel Processing*, Euro-Par'12, pages 167–178, Berlin, Heidelberg, 2012.