Optimal Greedy Algorithm for Many-Core Scheduling

Anuj Pathania, Vanchinathan Venkatramani, Muhammad Shafique, Tulika Mitra, Jörg Henkel

Abstract—In this work, we propose an optimal greedy algorithm for the problem of run-time many-core scheduling. The previously best known centralized optimal algorithm proposed for the problem is based on dynamic programming. A dynamic programming based scheduler has high overheads which grow fast with increase in both the number of cores in the many-cores and number of tasks executing on them. We show in this work that the inherent concavity of extractable Instructions Per Cycle in tasks with increase in number of allocated cores allows for an alternative greedy algorithm. The proposed algorithm significantly reduces the run-time scheduling overheads, while maintaining optimality. In practice, it reduces the problem solving time 10,000x to provide near-optimal solutions.

Index Terms—Scheduling, Throughput Maximization, Many-Cores, Greedy Algorithm.

I. INTRODUCTION

Many-cores are hundred core (or even thousand core) processors designed to execute several multi-threaded tasks in parallel [9]. One of the most crucial scheduling decision to be made on a many-core is the distribution of limited processing cores amongst the executing tasks in such a manner so that the processor always operates at its peak performance. In this work, we use adaptive many-cores [12], [16] which allows acceleration of not just multi-threaded tasks by Thread Level Parallelism (TLP) exploitation but also acceleration of single-threaded tasks by exploitation of Instruction Level Parallelism (ILP).

To reduce context-switching overheads, many-cores prefer to operate with one thread per-core model [5]; keeping the problem of many-core scheduling mathematically discrete. A task goes through different phases during its execution. These phases result in a high variance within the speedup a task can derive from a many-core as shown in Figure 1. N-core speedup of a task is defined as the ratio of its Instructions Per Cycle (IPC) when assigned N cores with its IPC when assigned only to a single core.

Malleable tasks [6] by design allow their core allocations to increase or decrease in size at any time during their execution with negligible performance penalties. When operating with malleable tasks, schedulers can exploit their speedup variance by performing run-time scheduling (core redistributions) to achieve maximum throughput (total IPC) from a many-core at all times [15].

The problem of run-time scheduling on many-core (throughput maximization) can be solved optimally in polynomial time using dynamic programming [8]. A scheduler based on dynamic programming has high overheads which grow fast as both the number of cores in a many-core and the number of tasks they can execute in parallel increase. Dynamic programming given its overheads is not suitable for scheduling at run-time due to frequent invocations. Therefore, a search for alternative low overhead algorithms for run-time many-core scheduling is mandated. In past, researchers have resorted to proposing heuristics with no guarantees on the obtained results [17]. We instead propose a low overhead solution in this work, which preserves the theoretical optimality of results.
This search is aided by the fact that extractable IPC in tasks executing on a many-core is in general concave with increase in number of allocated cores as shown in Figure 2. The observed concavity is due to the saturation of exploitable ILP/TLP in the tasks.

Our Novel Contributions: In this work, we present a greedy algorithm for the problem of run-time many-core scheduling based on the inherent concavity in extractable IPC in executing tasks. We show that a scheduler based on the proposed greedy algorithm significantly reduces the processing- and space overhead in comparison to a scheduler based on dynamic programming without sacrificing on the theoretical optimality. In practice where concavity property does not always hold, the algorithm results in significantly near-optimal solutions.

II. RELATED WORK

Multi/Many-core scheduling is a well-studied problem in research [17]. Though, major focus in past for performance-oriented schedulers remained on design-time makespan minimization instead of run-time throughput maximization.

Makespan minimization involves determining a fixed schedule that finishes a given workload on a multi/many-core processor in the shortest time [7]. Makespan minimizing schedulers often assume all information about the workload such as execution time and arrival time of constituent tasks is known beforehand. They also often assume executing tasks lack any phases and thereby always progress at a fixed rate. When operating with malleable tasks with concave speedups/IPCs, authors in [4] proposed a scheduler that can optimally minimize makespan with \( O(T \max(C, T \log^2 C)) \) processing overhead, where \( T \) is the number of tasks and \( C \) is the number of processing cores.

Throughput maximization on the other hand involves determining a schedule at discrete intervals for a workload, where often neither arrival time nor the departure time of its constituent tasks are known to the scheduler [7]. A run-time scheduler when running unprofiled tasks can rely only on information available to it from online observations or predictions that can be made from them.

Authors in [8] proposed a dynamic programming based algorithm with \( O(CT^2) \) processing overhead for optimal throughput maximization. Given the high-overhead for obtaining optimal solutions through dynamic programming, alternative approaches to reduce overhead of run-time many-core scheduling was proposed in form of distributed algorithms. Authors in [1] and [13] introduced best-effort heuristic Multi-Agents Systems (MAS) for run-time many-core scheduling with malleable tasks. We ourselves designed a MAS in [15] that solved the problem optimally using a distributed consensus algorithm based on cooperative game theory. Though promising in reducing the per-core processing overhead by disbursement of scheduling calculations across multiple/all cores, the reduction is accompanied by concurrent increase in the communication overhead.

We instead in this work, improve upon best-known centralized algorithm for throughput maximization [8] by introducing another optimal centralized algorithm that decreases both the processing as well as space overhead without changing the communication overhead. The proposed algorithm when operating with malleable tasks brings the efficiency of throughput maximizing schedulers to the same level as that of makespan minimizing schedulers.

III. GREEDY ALGORITHM FOR RUN-TIME MANY-CORE SCHEDULING

System Model: We begin by presenting the notations used in this work to model a many-core system. Let \( T \) and \( C \) be the number of tasks and cores in the system, respectively. \( C_i \) be the number of cores currently allocated to Task \( i \). Let \( I_{C_i} \) denote the IPC of Task \( i \) when allocated \( C_i \) cores.

Using the above notations, the problem of throughput maximization on many-core can be mathematically stated as below.

\[
\text{Maximize } \sum_{i=1}^{T} I_{C_i}, \text{ under constraint } \sum_{i=1}^{T} |C_i| \leq C
\]

In the context of run-time many-core scheduling, the above equation needs to be solved by the scheduler nearly in every scheduling epoch. Scheduling epoch is the frequency at which a scheduler is invoked by an Operating System (OS). In this work, its value is assumed to be 10 ms; same as the default scheduling epoch used by Linux kernel [14].

Furthermore to operate, a scheduler also needs to know the value of IPC \( I_{C_i} \) of Task \( i \) from a possible core allocation \( C_i \) from its current observable IPC \( I_{C_i} \) when \( C_i \neq C \). For this, we employ regression-based performance-prediction models for adaptive many-cores developed in [18].

The concavity in IPC extraction as shown in Figure 2 is mathematically captured by following equation.

\[
I_{C_i} + n - I_{C_i} \geq I_{C_i+n} - I_{C_i} \text{ if } C_i \leq C_i
\] (1)

Greedy Algorithm: We now propose a greedy algorithm for the problem of run-time many-core scheduling. The proposed algorithm is composed of following sequential steps performed by the greedy scheduler before every scheduling epoch.

1) Assume \( C_i = 0 \) \( \forall i \in T \).
2) Sort all tasks in \( T \) in ascending order by using comparator \( |I_{C_i} + 1 - I_{C_i}| \) and store in a queue.
3) Virtually assign a core to Task \( j \) in front of the queue and update the corresponding \( I_{C_j} \) using performance-prediction models from [18].

Fig. 2: Observed average IPC of different benchmarks when allocated different number of cores.
4) Reposition the Task $j$ according to the updated $I_{C_j}$ in the sorted queue using a binary search insertion.
5) Repeat Step 3 and Step 4 till all cores are allocated.
6) Re-adjust real core allocations from last scheduling epoch to reflect the new optimal core allocations.
7) Execute tasks with the optimal core allocations.

The proposed greedy scheduler, like all greedy algorithms is minimalistic in its approach and is quite easy to implement. Nevertheless, its real strength come from its ability to provide optimal results. We now proceed to prove the theoretical optimality of our algorithm.

**Theorem 1.** The greedy core allocations are optimal.

**Proof.** Let $(C_1, C_2, ..., C_k)$ be the core allocations chosen by our proposed greedy algorithm. We prove our theorem using proof by induction.

**Base Case:** Suppose $(C_1, C_2, C_{x_2} - 1, ..., C_y + 1, ..., C_k)$ instead be the optimal core allocations in which Task $x$ has one less core allocated to it, which instead is allocated to Task $y$.

Now for the optimal core allocations to be better than greedy core allocations following equation must hold.

$$I_{C_y+1} - I_{C_y} > I_{C_x} - I_{C_{x-1}}$$ (2)

The above equation says that the benefit (in terms of increased IPC) of assigning additional core to Task $y$ must outweigh the loss of taking away that core (in terms of decreased IPC) from Task $x$ under optimal core allocations.

Our greedy algorithm does not reconsider core allocation decisions. So, the suboptimal decision of allocating an additional core to Task $x$ when already allocated $C_x - 1$ cores happens when either $C_y < C_y'$ or $C_y' < C_y$ cores were allocated to Task $y$. We consider both cases below.

If suboptimal core allocation happens when Task $y$ had $C_y$ cores, then it implies following relation.

$$I_{C_x} - I_{C_{x-1}} \geq I_{C_{y+1}} - I_{C_y}$$

The above equation is in contradiction to Equation (2).

On the other hand, if the suboptimal core allocation happens when Task $y$ had $C_y' < C_y$ cores, then

$$I_{C_x} - I_{C_{x-1}} \geq I_{C_{y'+1}} - I_{C_y'}$$

But, we know from concavity Equation (1)

$$I_{C_{y'+1}} - I_{C_{y'}} \geq I_{C_{y+1}} - I_{C_{y}}$$

$$\Rightarrow I_{C_x} - I_{C_{x-1}} \geq I_{C_{y+1}} - I_{C_{y}}$$ (3)

Again a contradiction to Equation (2). Hence, we prove our base case is optimal.

**Step Case:** Suppose $(C_1, C_2, C_{x_2} - 2, ..., C_y + 1, C_z + 1, ..., C_k)$ instead be the optimal core allocations in which Task $x$ has two less cores allocated to it, which instead are allocated to Task $y$ and Task $z$; one each. Without loss of generality the proof will also hold if both cores from Task $x$ were allocated to Task $y$ or Task $z$ exclusively instead.

Now for above optimal core allocations to be better than greedy core allocations following equation must hold.

$$I_{C_{y+1}} - I_{C_y} + I_{C_{z+1}} - I_{C_z} > I_{C_x} - I_{C_{x-2}}$$ (4)

As argued in base case, suboptimal decision of allocating the first additional core to Task $x$ when allocated $C_x - 2$ happens when either $C_y' \leq C_y$ or $C_z' \leq C_z$ or both. Therefore, by design of the greedy algorithm following relations hold.

$$I_{C_{z+1}} - I_{C_{z-2}} \geq I_{C_{y'+1}} - I_{C_y'} \geq I_{C_{y}+1} - I_{C_y}$$ (5)

Similarly, suboptimal decision of allocating the second additional core to Task $x$ with $C_x - 1$ cores already allocated happens when either $C_y' \leq C_y$ or $C_z' \leq C_z$ or both. Therefore, by design following relations also hold.

$$I_{C_x} - I_{C_{x-1}} \geq I_{C_{y+1}} - I_{C_y} \geq I_{C_{z}+1} - I_{C_z}$$ (6)

Adding Equation (5) and (6) we get,

$$I_{C_x} - I_{C_{x-2}} \geq I_{C_{y+1}} - I_{C_y} + I_{C_{z}+1} - I_{C_z}$$

But, we know from concavity Equation (1)

$$I_{C_{y+1}} - I_{C_y} \geq I_{C_{y+1}} - I_{C_y}$$

$$I_{C_{z}+1} - I_{C_z} \geq I_{C_{z}+1} - I_{C_z}$$

Therefore,

$$I_{C_x} - I_{C_{x-2}} \geq I_{C_{y+1}} - I_{C_y} + I_{C_{z}+1} - I_{C_z}$$

Above equation is in contradiction to Equation (4). Hence, we prove our step case to be optimal.

**Assumption Case:** We assume greedy allocations are optimal till $n$ cores are removed from Task $x$ and distributed among remaining tasks in any combination. Mathematically following relationship is assumed to be true.

$$I_{C_x} - I_{C_{x-n}} \geq I_{C_y+\alpha_y} - I_{C_y} + ... + I_{C_z+\alpha_z} - I_{C_z}$$ (7)

where $\alpha_y + ... + \alpha_z = n$.

**Induction Case:** Now we assume in optimal allocation $(n+1)^{th}$ core is removed from Task $x$ and without loss of generality given to Task $y$, while previously removed $n$ cores are distributed in same combination as in assumption case.

Now for the optimal core allocations to be better than greedy core allocations following equation must hold.

$$I_{C_{y+\alpha_y+1}} - I_{C_y} + ... + I_{C_{z+\alpha_z}} - I_{C_z} \geq I_{C_x} - I_{C_{x-n-1}}$$ (8)

Since greedy algorithm choose to allocate core to Task $x$ with $C_x - n - 1$ cores allocated instead of Task $y$ with $C_y+\alpha_y$ cores allocated, by the design of the greedy algorithm following relationship holds.

$$I_{C_{x-n}} - I_{C_{x-n-1}} \geq I_{C_y+\alpha_y+1} - I_{C_y+\alpha_y}$$

Adding Equation (7) to above equation we get,

$$I_{C_x} - I_{C_{x-n-1}} \geq I_{C_{y+\alpha_y+1}} - I_{C_y} + ... + I_{C_{z+\alpha_z}} - I_{C_z}$$

Above equation is in contradiction to Equation (8), proving our induction step is optimal; hence proved.

**Complexity:** Given that dynamic programming already provides the optimal solution for the problem of run-time many-core scheduling, the primary reason for developing a greedy algorithm is to reduce the scheduling overheads.

---

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2016.2618880, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
These reductions will directly translate into improvement in performance in real-world systems.

In the proposed algorithm, one time sorting in Step 2 has a processing overhead of $O(T \lg T)$. Additionally, binary search in Step 4 has a processing overhead of $O(\lg T)$. Since Step 4 is repeated $C$ times, the total processing overhead of our algorithm is $O(T \lg T + C \lg T)$ or $O(\max(C, T) \lg T)$. This overhead is substantially less than $O(CT^2)$ processing overhead of dynamic programming.

Furthermore, the proposed algorithm requires maintenance of only one queue data structure with space overhead of $O(T)$. On the other hand, dynamic programming has a significantly higher space overhead of $O(CT)$.

IV. EXPERIMENTAL EVALUATIONS

Setup: We use a two-stage adaptive many-core simulator as shown in Figure 3 for our proof-of-concept empirical evaluations. First stage is based on gem5 [3] cycle-accurate simulator with Bahurupi [16] adaptive cores implementing ARM v7 ISA and running at 1GHz. Cores have 2-way out-of-order pipeline and have separated 4-way associative L1 instruction- and data cache of size 64KB each. An 8-way associative 2MB unified L2 cache is shared by all cores.

Unfortunately, cycle-accurate many-core simulations are not time-wise feasible and hence our first stage is limited to maximum of eight cores. To bypass this limitation, we take isolated execution traces of different tasks from the cycle-accurate simulator with up to eight cores allocated and extrapolate them using an in-house trace-simulator for many-core simulations. Our extrapolated trace simulations lack the modeling depth of cycle-accurate simulations but we believe them to be sufficient as an initial testbed for many-core algorithms.

On software side, we took 36 benchmarks from SPEC [10], [11], SD-VBS [19], PARSEC [2] and SPLASH [20] benchmark suites as listed in Table I. All benchmarks were executed in Syscall Emulation (SE) mode. SPEC, SD-VBS and PARSEC/SPLASH benchmarks were executed with “ref”, “full-hd” and “sim-small” inputs, respectively.

Performance Evaluation: We chose to evaluate our algorithm on a closed many-core system. In a closed system, a workload is fixed at the start and constituent tasks of the workload restart execution immediately after completion. Our workload comprises of a random mix of all the available benchmarks with uniform distribution. Throughput measured in terms of average total Instructions per Cycle (IPC) of the system over two billion cycles is selected as the metric for measuring performance.

Figure 4 shows the performance of a closed many-core system with 256 cores under different size workloads when operated with a dynamic programming- and a greedy algorithm based schedulers. In our experiments, we observed that the performance under the greedy scheduler can differ from dynamic programming based scheduler by up to 1.24% even though both are proven theoretically optimal.

This discrepancy is attributed to the fact that the greedy algorithm requires IPC concavity to be observed by all benchmarks at all times, while no such requirement is mandated by dynamic programming. In practice, IPC concavity is exhibited by all benchmarks most of the time, but there are few rare exceptions. For example, benchmarks like h264 and sft sometime exhibit super-linearity as shown in Figure 1 where speedup of more than two can be observed on allocations of only two cores. Hence, a slight degradation in performance is observed when the concavity assumption is violated.

Figure 5 shows the performance of closed many-core systems of various sizes under different schedulers when a given workload is executed on them. We observed that the greedy scheduler results in near-optimal solutions in under all cases; similar to observations made in Figure 4.

Scalability: To get a measure of real-world benefits from a greedy scheduler against a dynamic programming based scheduler, we ran both schedulers cycle-accurately on gem5 within a single core in our simulated many-core with rep...
TABLE II: Comparison of time taken by the greedy scheduler and the dynamic programming based scheduler to solve run-time many-core scheduling problem of different sizes.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Tasks</th>
<th>Dynamic Programming (ms)</th>
<th>Greedy (ms)</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>32</td>
<td>7.985</td>
<td>0.061</td>
<td>130.90x</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>16.203</td>
<td>0.077</td>
<td>210.42x</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>32.910</td>
<td>0.108</td>
<td>304.72x</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>71.005</td>
<td>0.113</td>
<td>628.36x</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>144.214</td>
<td>0.144</td>
<td>1,001.48x</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>285.617</td>
<td>0.223</td>
<td>1,280.79x</td>
</tr>
<tr>
<td>256</td>
<td>128</td>
<td>586.276</td>
<td>0.230</td>
<td>2,345.11x</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>1,167.108</td>
<td>0.318</td>
<td>3,657.57x</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>2,314.798</td>
<td>0.555</td>
<td>4,170.80x</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>4,553.491</td>
<td>0.635</td>
<td>7,170.85x</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>9,056.819</td>
<td>0.866</td>
<td>10,458.22x</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>18,031.729</td>
<td>1.578</td>
<td>11,426.95x</td>
</tr>
</tbody>
</table>

representative varisized inputs. We then recorded the simulated system-time it took for both schedulers to reach a solution in Table II on average for a scheduling epoch. Empirical evaluations show that dynamic programming based scheduler requires 7.985 ms to solve the problem of runtime many-core scheduling for a 64-core many-core running 32 tasks. For a 10 ms scheduling epoch at which a multicore OS operates, this will result in impractical overhead of 79.85%. The greedy scheduler in comparison will have an overhead of only 0.61% for the same size problem. Furthermore, the overhead of dynamic programming based scheduler grows exponentially with both increase in the number of cores and the tasks executing on them. On the other hand, overhead of greedy scheduler grows much more slowly. Even on a 512-core running 1024 tasks, overhead of greedy algorithm stands at a somewhat acceptable 15.78%. To reduce overhead further, it seems a many-core OS will need to operate with a much higher values of scheduling epoch.

It is important to note that beside direct reduction in processing overhead, a reduction in space overhead also plays critical role in reducing the total problem solving time. When operating with large data-structures, a scheduler is forced to perform page-swaps with main memory when the last-level cache is saturated. Since main memory accesses have several times the latencies of cache accesses, the performance suffers enormously. Given the fact that space-overhead for our greedy scheduler is much smaller than dynamic programming based scheduler, cache-saturation will happen in the former for problem of much larger size than the latter.

V. CONCLUSION

In this work, we proposed a theoretically optimal greedy algorithm for the problem of run-time many-core scheduling. Given the large optimization search-space, the problem requires light-weight algorithms that can be applied online. A dynamic programming based scheduler can solve the problem optimally but has high scheduling overheads associated with it. As an alternative, we proposed a scheduler based on a greedy algorithm in this work. The proposed greedy scheduler exploits the concavity in IPC extraction inherent in many-core workloads to maintain theoretical optimality.

In practice, it requires 10,000x less time than a dynamic programming based scheduler to reach a solution, while providing near-optimal performance.

ACKNOWLEDGMENT

This work was supported in parts by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR 89) and in parts by Huawei International Pte. Ltd. in Singapore.

REFERENCES