Scheduling Parallel Tasks under Multiple Resources: List vs. Pack

Hongyang Sun (speaker)\textsuperscript{1} Redouane Elghazi\textsuperscript{2}
Ana Gainaru\textsuperscript{1} Guillaume Aupy\textsuperscript{3} Padma Raghavan\textsuperscript{1}

\textsuperscript{1}Vanderbilt University, USA
\textsuperscript{2}École Normale Supérieure de Lyon, France
\textsuperscript{3}Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP, France

This research is supported in part by NSF under award CCF 1719674

IPDPS’18@Vancouver, BC, Canada
May 21, 2018
Introduction

Single-resource scheduling

- Most traditional scheduling problems target a single type of resource (e.g., CPUs)

For example: classic NP-complete problem of makespan minimization on identical machines ($P||C_{max}$). List scheduling is $(2 - \frac{1}{P})$-approx. [Graham 1969]
Introduction

The case for multi-resource scheduling

- HPC systems embrace more heterogeneous components (e.g., CPU, GPU, FPGA, MIC, APU)
- Data-intensive applications drive architecture enhancement for better data-transfer efficiency (e.g., High-Bandwidth Memory, Partitionable Cache, Burst Buffers)

To achieve optimal system/application performance, multiple types of resources (e.g., CPU, GPU, memory, cache, I/O) should be scheduled simultaneously
A multi-resource scheduling model:

- System with $d$ resource types; $i$-th type has $P^{(i)}$ identical resources
- Set $\{1, 2, \cdots, n\}$ of independent, moldable tasks released at time $0$
- Each task $j$'s execution time $t_j(\vec{p}_j)$ depends on its resource allocation vector $\vec{p}_j = (p_j^{(1)}, p_j^{(2)}, \cdots, p_j^{(d)})$
- Assumption: non-increasing execution time

$$\vec{p}_j \preceq \vec{q}_j \quad \text{(or } p_j^{(i)} \leq q_j^{(i)}, \forall i) \quad \implies \quad t_j(\vec{p}_j) \geq t_j(\vec{q}_j)$$

Scheduling objective:

- Find a moldable schedule, i.e., resource allocation vector $\vec{p}_j$ and starting time $s_j$ for each task $j$
  - minimize makespan: $T = \max_j(s_j + t_j(\vec{p}_j))$
  - subject to resource constraint: $\sum_{j \text{ active at time } t} p_j^{(i)} \leq P^{(i)}, \forall i, t$
Focus of This Work

Two scheduling paradigms:

- **List**: greedily schedule tasks in a list on first available resources
- **Pack**: partition tasks in packs to be scheduled one after another

▶ Simple yet efficient schedules favored by practical runtime systems
▶ Easily adopted to online or heterogeneous scheduling environments
Theoretically:

- Approximation ratios that increase linearly with number $d$ of resource types
  - List-scheduling: $2d$-approx.
  - Pack-scheduling: $(2d + 1)$-approx.
- Strategy to transform multi-resource problem to single-resource problem to reduce computational complexity

Empirically:

- Experiments on Intel Xeon Phi Knights Landing (KNL) with 2 resource types (cores + high-bandwidth memory)
- Simulations with up to 4 resource types using synthetic workloads that extend classical speedup profiles
Outline

Introduction

Theoretical Analysis

Experimental Evaluation

Future Work
**Definitions**: for a given resource allocation \( \mathbf{p} = (\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n)^T \)

- Total task area (normalized): \( A(\mathbf{p}) = \sum_{j=1}^{n} \sum_{i=1}^{d} \frac{p^{(i)}_j}{P^{(i)}} \cdot t_j(\vec{p}_j) \)
- Maximum task execution time: \( t_{\text{max}}(\mathbf{p}) = \max_j t_j(\vec{p}_j) \)
**Definitions:** for a given resource allocation $\mathbf{p} = (\vec{p}_1, \vec{p}_2, \ldots, \vec{p}_n)^T$

- Total task area (normalized): $A(\mathbf{p}) = \sum_{j=1}^n \sum_{i=1}^d \frac{p_j^{(i)}}{p^{(i)}} \cdot t_j(\vec{p}_j)$
- Maximum task execution time: $t_{\text{max}}(\mathbf{p}) = \max_j t_j(\vec{p}_j)$

Analogous to *area bound* ($T_1/P$) and *depth bound* ($T_\infty$) in single-resource scheduling.
Preliminaries

**Definitions:** for a given resource allocation \( \mathbf{p} = (\bar{p}_1, \bar{p}_2, \cdots, \bar{p}_n)^T \)

- Total task area (normalized): \( A(\mathbf{p}) = \sum_{j=1}^{n} \sum_{i=1}^{d} \frac{p_j^{(i)}}{P^{(i)}} \cdot t_j(\bar{p}_j) \)
- Maximum task execution time: \( t_{\text{max}}(\mathbf{p}) = \max_j t_j(\bar{p}_j) \)

Analogous to *area bound* \( (T_1/P) \) and *depth bound* \( (T_\infty) \) in single-resource scheduling

**Lower bound** (on makespan): \( L(\mathbf{p}, d) = \max \left( \frac{A(\mathbf{p})}{d}, t_{\text{max}}(\mathbf{p}) \right) \)

**Proposition**

*The optimal makespan satisfies*

\[
T_{\text{OPT}} \geq L_{\text{min}}(d) = \min_{\mathbf{p}} L(\mathbf{p}, d)
\]
Moldable Scheduling

Two-phase approach [Turek et al. 1992]:

▶ *Phase 1*: Determines a resource allocation for each moldable task

▶ *Phase 2*: Constructs a rigid schedule based on the fixed resource allocations of all tasks
Phase 1: Resource Allocation

**Goal:** find allocation $p_{min}^d$ matching lower bound $L_{min}(d) = \min_p L(p, d)$

**Resource Allocation ($RA_d$)**

- **Step (1).** For each task $j$:
  - Linearize all $P = \prod_{i=1}^{d}(P^{(i)} + 1)$ allocations
  - Remove ones with both higher execution time and larger area
  - Sort in order of increasing execution time and decreasing area

- **Step (2).** For all $n$ tasks:
  - Traverse the $n$ lists in $\leq nP$ steps by tracing $t_{max}(p)$ at each step until dominated by $\frac{A(p)}{d}$ (v.s. exhaustive search in $P^n$ time)

**Complexity:** $O(nP(\log P + \log n + d))$
Phase 1: Resource Allocation

**Goal:** find allocation $p_{\text{min}}^d$ matching lower bound $L_{\text{min}}(d) = \min_p L(p, d)$

Resource Allocation ($RA_d$)

- **Step (1).** For each task $j$:
  - Linearize all $P = \prod_{i=1}^{d} (P^{(i)} + 1)$ allocations
  - Remove ones with both higher execution time and larger area
  - Sort in order of increasing execution time and decreasing area

- **Step (2).** For all $n$ tasks:
  - Traverse the $n$ lists in $\leq nP$ steps by tracing $t_{\text{max}}(p)$ at each step until dominated by $\frac{A(p)}{d}$ (v.s. exhaustive search in $P^n$ time)

**Complexity:** $O(nP(\log P + \log n + d))$

**Proposition**

*If a rigid scheduling algorithm $R_d$ that uses $p_{\text{min}}^d$ produces a makespan $T_{R_d}(p_{\text{min}}^d) \leq c \cdot L_{\text{min}}(d)$ then the two-phase algorithm $RA_d + R_d$ is $c$-approximation.*
Phase 2: Rigid Scheduling

For a fixed resource allocation:

- **List Scheduling** ($LS_d$): 2-approx. for $d = 1$
  - Arrange all tasks in a list. Whenever an existing task completes, scan the list and schedule first task that fits (i.e., with sufficient resources in all dimensions)

- **Pack Scheduling** ($PS_d$): 3-approx. for $d = 1$
  - Sort all tasks in decreasing order of exec. time. Assign each task in sequence to last pack if fits (i.e., with sufficient resources in all dimensions). Otherwise, create a new pack.

Proposition

For a set of rigid tasks with fixed resource allocation $p$, we have

- **List Scheduling** ($T_{LS_d}(p)$) $\leq 2^d \cdot L(p, s)$
- **Pack Scheduling** ($T_{PS_d}(p)$) $\leq (2^d + 1) \cdot L(p, s)$

⇒ $RA_d + LS_d$ is 2$^d$-approx. and $RA_d + PS_d$ is (2$^d$ + 1)-approx.

Moreover, the bounds are tight for the two algorithms.
Phase 2: Rigid Scheduling

For a fixed resource allocation:

- **List Scheduling** ($LS_d$): 2-approx. for $d = 1$
  - Arrange all tasks in a list. Whenever an existing task completes, scan the list and schedule first task that fits (i.e., with sufficient resources in all dimensions)

- **Pack Scheduling** ($PS_d$): 3-approx. for $d = 1$
  - Sort all tasks in decreasing order of exec. time. Assign each task in sequence to last pack if fits (i.e., with sufficient resources in all dimensions). Otherwise, create a new pack.

**Proposition**

For a set of rigid tasks with fixed resource allocation $p$, we have

- **List Scheduling**: $T_{LS_d}(p) \leq 2d \cdot L(p, s)$
- **Pack Scheduling**: $T_{PS_d}(p) \leq (2d + 1) \cdot L(p, s)$

$\Rightarrow RA_d + LS_d$ is $2d$-approx. and $RA_d + PS_d$ is $(2d + 1)$-approx.

Moreover, the bounds are tight for the two algorithms.
Transformation (TF):

- **Step (1).** *d-resource instance* $I \mapsto 1$-*resource instance* $I'$
  - $I'$ has same number $n$ of tasks and total resource $Q = \text{lcm}_{i=1}^{d} P^{(i)}$.
  - For any task $j'$ in $I'$: execution time $t_{j'}(q) = t_{j}(\lfloor \frac{q \cdot P^{(i)}}{Q} \rfloor)_{i=1 \ldots d}$ for all $q$.

- **Step (2).** Solve the *1-resource instance* $I'$

- **Step (3).** *1-resource solution* $S'$ $\mapsto$ *d-resource solution* $S$
  - For any task $j$ in $I$: starting time is same $s_{j} = s_{j'}$.
  - Resource allocation is $\bar{p}_{j} = \lfloor \frac{q_{j'} \cdot P^{(i)}}{Q} \rfloor$ for all $i=1 \ldots d$.
Transformation (TF):

- **Step (1).** \( d \)-resource instance \( I \) \( \rightarrow \) 1-resource instance \( I' \)
  - \( I' \) has same number \( n \) of tasks and total resource \( Q = \text{lcm}_{i=1}^{d} P(i) \)
  - For any task \( j' \) in \( I' \): execution time \( t_{j'}(q) = t_j\left(\left\lfloor \frac{q \cdot P(i)}{Q} \right\rfloor_{i=1}^{d} \right) \forall q \)

- **Step (2).** Solve the 1-resource instance \( I' \)

- **Step (3).** 1-resource solution \( S' \) \( \rightarrow \) \( d \)-resource solution \( S \)
  - For any task \( j \) in \( I \): starting time is same \( s_j = s_{j'} \)
  - resource allocation is \( \tilde{p}_j = \left(\left\lfloor \frac{q_{j'} \cdot P(i)}{Q} \right\rfloor_{i=1}^{d} \right) \)

**Performance:** \( TF + RA_1 + LS_1 \) is \( 2d \)-approx.
\( TF + RA_1 + PS_1 \) is \( (2d + 1) \)-approx.

**Complexity:** Transform \( Q = \text{lcm}_i P(i) \) v.s. Direct \( P = \prod_i (P(i) + 1) \)
If \( P(i) = p \ \forall i \) \( \Rightarrow \) \( O(p) \) v.s. \( O(p^d) \)
Outline

Introduction

Theoretical Analysis

Experimental Evaluation

Future Work
Experimental Setup

**Platform:** Intel Xeon Phi 7230 Knights Landing (KNL)

- 64 cores
- 96GB slow memory (DDR)
- 16GB fast memory (MCDRAM)
  - 4-5x the bandwidth
  - 3 configuration modes

In flat mode, consider fast memory (like cores) as a type of limited resource shared by competing tasks

![Diagram showing memory configurations](image-url)
Experimental Setup

**Platform**: Intel Xeon Phi 7230 Knights Landing (KNL)

- 64 cores
- 96GB slow memory (DDR)
- 16GB fast memory (MCDRAM)
  - 4-5x the bandwidth
  - 3 configuration modes

In flat mode, consider fast memory (like cores) as a type of limited resource shared by competing tasks

**Benchmarks**: STREAM (*triad, write, ddot*)

- Create tasks of different sizes by varying array length and thus memory footprint as % of MCDRAM size
Experimental Results

Comparing different algorithms:

- Comparable performance for list- and pack-based solutions
- LPT (list) and FF (pack) perform generally better
- Transform-based solutions perform just as well
Experimental Results

Flat mode vs. cache mode:

- Managing fast memory directly as a resource (in flat mode) result in better performance than treating it as a cache for co-scheduled applications (due to possible interference).
Simulation Setup

Resources:

- Up to four different types (e.g., CPU, GPU, cache, memory, I/O)
- Amount of resources for each type: \((64, 32, 16, 8)\)

Workload (synthetic):

- **Extended Amdahl’s law**: \(s_0 \sim \mathcal{U}(0, 0.2)\)

  (i) \(1 / \left( s_0 + \sum_{i=1}^{d} \frac{s_i}{p(i)} \right) \); (ii) \(1 / \left( s_0 + \frac{1-s_0}{\prod_{i=1}^{d} p(i)} \right) \); (iii) \(1 / \left( s_0 + \max_{i=1..d} \frac{s_i}{p(i)} \right) \)

- **Extended power law**: \(\alpha_i \sim \mathcal{U}(0.3, 1)\)

  (i) \(1 / \left( \sum_{i=1}^{d} \frac{s_i}{(p(i))^{\alpha_i}} \right) \); (ii) \(\prod_{i=1}^{d} (p(i))^{\alpha_i} \); (iii) \(1 / \left( \max_{i=1..d} \frac{s_i}{(p(i))^{\alpha_i}} \right) \)

![Different colors indicate different resources](image)

(i) sequential  (ii) collaborative  (iii) concurrent
Simulation Results

Performance (makespan normalized w.r.t lower bound):

- Ratios increase with $d$, but far below theoretical bounds
- List algorithms perform better, but gap reduces as $d$ increases
- Transform-based solutions perform slightly better
Simulation Results

**Complexity (running time of algorithms):**

- Pack algorithms run slightly faster than list algorithms
- Direct solutions increase drastically with $d$
- Transform-based solutions orders of magnitude faster (esp. $d \geq 3$)
Simulation Results

Transform-based pack scheduling offers fast, efficient, and easy-to-implement solutions when managing a large number of resources.

**Complexity (running time of algorithms):**

- Pack algorithms run slightly faster than list algorithms.
- Direct solutions increase drastically with $d$.
- Transform-based solutions orders of magnitude faster (esp. $d \geq 3$).
Outline

Introduction

Theoretical Analysis

Experimental Evaluation

Future Work
Open Questions

Performance of list-scheduling under multi-resources

- Rigid jobs: \((d + 1)\)-approx. [Garey and Graham, 1975]
- Moldable jobs: \(2d\)-approx. [This work, with algo. lower bound]
- Malleable jobs: \((d + 1)\)-approx. [He et al. 2007]
  (Represented as DAGs containing unit-size tasks of different types)

- Can we achieve \((d + 1)\)-approx. for moldable jobs (possibly with a more coupled design/analysis of resource allocation and rigid scheduling), or is it inherently harder?
Open Questions

Performance of list-scheduling under multi-resources

- Rigid jobs: \((d + 1)\)-approx. [Garey and Graham, 1975]
- Moldable jobs: \(2d\)-approx. [This work, with algo. lower bound]
- Malleable jobs: \((d + 1)\)-approx. [He et al. 2007]
  (Represented as DAGs containing unit-size tasks of different types)

- Can we achieve \((d + 1)\)-approx. for moldable jobs (possibly with a more coupled design/analysis of resource allocation and rigid scheduling), or is it inherently harder?

Performance of general models for moldable task scheduling

- 2-Pack Sol.: \((1.5 + \epsilon)\)-approx. [Mounié et al. 2004, Jansen 2012]
- Precedence constraints: e.g., \((3 + \sqrt{5})\)-approx. [Lepère et al. 2001]

- Could these results be extended to multi-resource scheduling?
Open Questions

Performance of list-scheduling under multi-resources

- Rigid jobs: \((d + 1)\)-approx. [Garey and Graham, 1975]
- Moldable jobs: \(2d\)-approx. [This work, with algo. lower bound]
- Malleable jobs: \((d + 1)\)-approx. [He et al. 2007]
  (Represented as DAGs containing unit-size tasks of different types)

- Can we achieve \((d + 1)\)-approx. for moldable jobs (possibly with a more coupled design/analysis of resource allocation and rigid scheduling), or is it inherently harder?

Performance of general models for moldable task scheduling

- 2-Pack Sol.: \((1.5 + \epsilon)\)-approx. [Mounié et al. 2004, Jansen 2012]
- Precedence constraints: e.g., \((3 + \sqrt{5})\)-approx. [Lepère et al. 2001]

- Could these results be extended to multi-resource scheduling?

Other practical applications of multi-resource scheduling
- e.g., cache partitioning, bandwidth allocation, burst buffer sharing?