Task Scheduling in High-Performance Computing

Thomas McSweeney
School of Mathematics
The University of Manchester

thomas.mcsweeney@postgrad.manchester.ac.uk

Numerical Linear Algebra Group Meeting
October 16, 2018

Outline

Current hardware trends in HPC

HPC systems have long been highly parallel and modern machines are increasingly likely to also be **heterogeneous**.

Summit

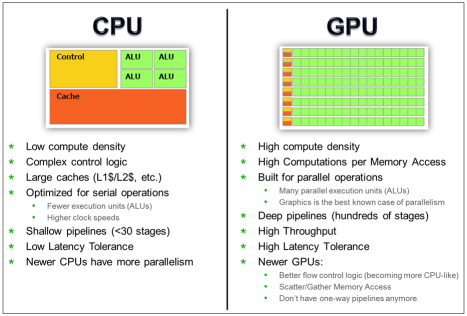
4,356 nodes, each with:

- Two **22-core Power9 CPUs**
- Six **NVIDIA Tesla V100 GPUs**

Altogether, there are **2,282,544** cores!

Exploiting these changes

Different types of processors have different attributes—and ultimately different kinds of tasks that they are good at.



CPU

* Low compute density
* Complex control logic
* Large caches (L1$/L2$, etc.)
* Optimized for serial operations
  * Fewer execution units (ALUs)
  * Higher clock speeds
* Shallow pipelines (<30 stages)
* Low Latency Tolerance
* Newer CPUs have more parallelism

GPU

* High compute density
* High Computations per Memory Access
* Built for parallel operations
  * Many parallel execution units (ALUs)
  * Graphics is the best known case of parallelism
* Deep pipelines (hundreds of stages)
* High Throughput
* High Latency Tolerance
* Newer GPUs:
  * Better flow control logic (becoming more CPU-like)
  * Scatter/Gather Memory Access
  * Don't have one-way pipelines anymore

Roughly: CPUs are better at small serial tasks and GPUs at large parallel ones.

How do we make the most effective use of the diverse computing resources of modern HPC systems?

Task-based programming

A popular parallel programming paradigm in recent years is based on the concept of a **task**—essentially, a discrete unit of work.

The main idea: specify jobs as collections of tasks and the **dependencies** (i.e., dataflow) between them.

- Very portable—just need to think at task level.
- Relatively easy to code.
- Well-suited to numerical linear algebra.

Runtime systems based on this model include **StarPU** and **PaRSEC**.

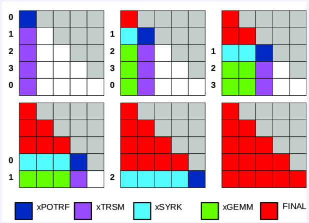
Task-based NLA

In numerical linear algebra, perhaps the most natural way to define tasks is as BLAS calls on individual tiles of the matrix.

Tiled Cholesky factorization

Suppose $A = \begin{bmatrix} A_{00} & \cdots & A_{N0} \\ \vdots & \ddots & \vdots \\ A_{N0} & \cdots & A_{NN} \end{bmatrix}$ is symmetric positive definite.

```
for $k = 0, 1, \dots, N$ do
    $A_{kk} := \texttt{POTRF}(A_{kk})$
    for $m = k+1, \dots, N-1$ do
        $A_{mk} := \texttt{TRSM}(A_{kk}, A_{mk})$
    for $n = k+1, \dots, N-1$ do
        $A_{nn} := \texttt{SYRK}(A_{nk}, A_{nn})$
        for $m = n+1, \dots, N-1$ do
            $A_{mn} := \texttt{GEMM}(A_{mk}, A_{nk}, A_{mn})$
```



Source: [4, Tomov et al., 2012].

From tasks to DAGs

We can view applications as **D**irected **A**cyclic **G**raphs (**DAGs**), with nodes representing the tasks and edges the dependencies between them.

If all values were set, we could solve this through any standard approach.

But weights of the DAG depend on the system it is to be executed on.

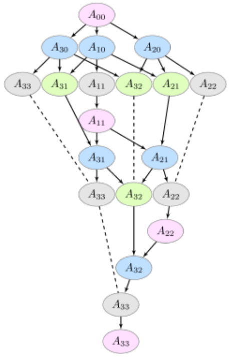


Image: Task graph of a 4×4 block Cholesky factorization of a matrix A. Colours represent BLAS routines: green = `GEMM`, **pink =** `POTRF`, **grey =** `SYRK`, **blue =** `TRSM`.

Source: [5, Zafari, Larsson, and Tillenius, 2016].

The task scheduling problem

Given a HPC system and an application/DAG, what is the optimal way to assign the tasks to the processors? In other words, how do we find an optimal **schedule**?

This is really just a classic optimization problem—**job shop scheduling**.

Unfortunately, this is known to be **NP-complete**.
$\Longrightarrow$ heuristics and approximate solutions.

What is currently done?

Listing heuristics are currently the most popular approach:

1. Rank all tasks according to some attribute.
2. Schedule all tasks according to their rank.

Many use the **critical path**, the longest path through the DAG—and a lower bound on the total execution time of the whole DAG in parallel.

There are two fundamental types of scheduling algorithms:

- **Static**: schedule is fixed before execution.
- **Dynamic**: schedule may change during execution.

HEFT

Heterogeneous Earliest Finish Time

Define the **upward rank** of a task to be the length of the critical path starting from that task (inclusive).

1. Set all weights in the DAG by averaging over all processors.
2. Calculate the upward rank of each task in the DAG.
3. List all tasks in descending order of upward rank.
4. Schedule each task in list on the processor estimated to complete it at the earliest time.

Basic idea: use mean values to set the DAG and solve with dynamic programming.

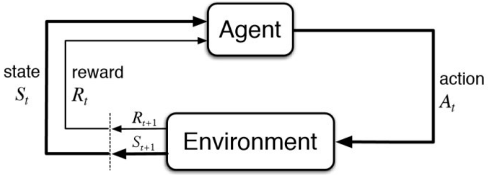
Usually pretty good, and lots of variants (e.g., dynamic) exist—but still room for improvement.

Outline

Reinforcement learning

One approach we are investigating is the application of **reinforcement learning** (**RL**) to the problem.

RL is a kind of **machine learning** that takes inspiration from trial-and-error, "behaviorist" theories of animal learning. The basic set up is:



We have an **agent** that interacts with an **environment** by taking **actions** for which it receives some **reward** that it seeks to **maximize**, which is the only feedback it receives.

The RL framework

We have a **value function** that tells the agent how "good" it is in the long run to take an action when the environment is in a particular **state**,

$$V(s) = r, \qquad Q(s, a) = r.$$

This lets us weigh immediate rewards against potential long-term losses.

A **policy** π defines the agent's behavior given the current state of the system, $\pi(s) = a$.

- Can be deterministic or stochastic.

The goal is to find the **optimal** policy π^*, which maximizes the total reward received.

Dynamic programming

More formally, RL attempts to solve a **Markov decision process** (**MDP**) by using value functions to guide decisions.

Thus RL is in some sense equivalent to **dynamic programming** (**DP**)—but the dynamics of the MDP are unknown and must be learned from experience.

DP methods find optimal value functions—and thus policies—by solving the **Bellman equation**

$$V^*(s) = \max_a \left\{ R(s,a) + \gamma \mathbb{E}\Big[V^*(s') \Big] \right\}$$

or

$$Q^*(s,a) = \mathbb{E}\Big[R(s,a) + \gamma \max_{a'} Q(s',a') \Big].$$

Generalized policy iteration

Many RL algorithms can be expressed in a simple framework called **generalized policy iteration**.

Start with a policy. Repeat until convergence:

1. Evaluate it.
2. Improve it.

We have a lot of scope:

- **On-policy** or **off-policy**.
  - Improve current policy or use another for exploration?
- **Monte Carlo** methods.
  - Average over **episodes** of experience.
  - Update after every episode.
- **Temporal-difference** methods.
  - Update after every time step.

Sarsa and Q-learning

Sarsa

```
foreach episode do
    foreach step of episode do
        For current state $s$, take action $a$ according to
          some $\epsilon$-greedy policy wrt $Q$.
        Observe immediate reward $r$ and next state $s'$.
        Choose next action $a'$ according to policy.
        $Q(s,a) := Q(s,a) + \alpha\Big[r + \gamma Q(s',a') - Q(s,a)\Big]$.
        $s = s'$.
```

Q-learning very similar but we instead update using:

$$Q(s,a) := Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big].$$

Approximate dynamic programming

Very large state spaces are problematic—this is the **curse of dimensionality**.

Need to **generalize** and approximate the value (or Q) function, so we can estimate its value for unseen states and actions,

$$V(s) \approx \hat{V}(s, \theta),$$

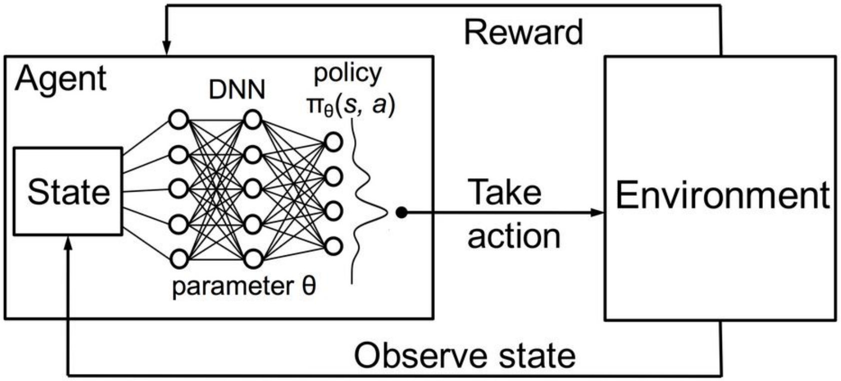
$$\text{or } Q(s, a) \approx \hat{Q}(s, a, \theta),$$

where θ is some (finite) vector of parameters.

This is **approximate dynamic programming**.

Deep reinforcement learning

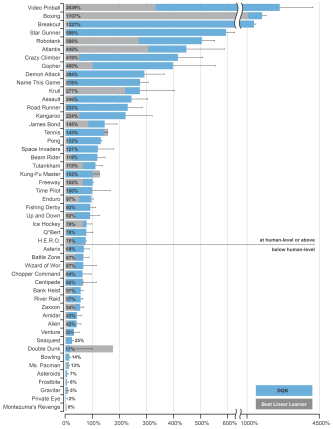
The most popular approach is to use **deep neural networks**.
This is called **deep reinforcement learning**.



Why reinforcement learning?

Applying RL follows naturally from the idea of planning along the critical path of the DAG. But is it practical?

Recent successes suggest traditional issues can be overcome.



Atari games.

- Transfer of learning.
- See [1, Mnih et al., 2015].

Board game Go.

- $\approx 10^{172}$ possible states.
- See [2, Silver et al., 2016] and [3, Silver et al., 2017].

Outline

Long-term goal

What we ultimately want:

A scheduler that can find a near-optimal schedule for any given application on any given HPC architecture in a reasonable time.

If using RL in particular we need to be practical by:

- Minimizing the cost of gathering training data,
- Making the best possible use of the data we have.

Generic RL issues

- **Exploration** vs. **exploitation**.

  How do we avoid getting stuck in local optima?
  - Standard ϵ-greedy policies.
  - UCB, Thompson sampling?

- The **credit assignment problem**.

  How do we identify the features that are truly useful for making decisions?

- Crafting the problem.

  How do we define states, actions and rewards?

Training data

RL needs lots of data but in HPC hours of runtime can be hundreds of dollars in energy costs—*everything* must be **fast**!

Solution: why not just simulate?

- Much cheaper/faster/easier.
- Can consider arbitrary architectures.
- Mature software available—reliable results.

Key is to identify what data we really require.

Transfer of learning

How do we generalize from the data we have?

- Can we exploit DAG structure to cluster them?
- What about parameterized task graphs (PTGs)?
  - Never see the entire DAG.
  - Used in StarPU, PaRSEC, ...
- Similar issues for new environments.

Rather than learning how to schedule a given DAG on a given system, need to learn rules that can be applied to many different DAGs on many different systems.

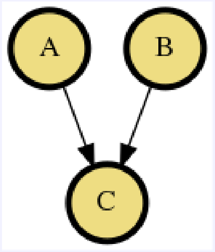
Can we improve HEFT?

There are some obvious issues with HEFT.

- Using average values across all processors is simple but not necessarily optimal.
- Greedy—can't avoid large future costs.

A simple example

Environment: Node with two Kepler K20 GPUs, one 12-core Sandy Bridge CPU.



Optimal policy is to schedule everything on one of the GPUs.

But HEFT schedules Tasks A and B on different GPUs, so can't avoid large later communication cost.

Questions

Thank you for your attention.

Any questions?

References I

V. Mnih, et al.
Human-level control through deep reinforcement learning.
Nature 518.7540 (2015): 529.

D. Silver, et al.
Mastering the game of Go with deep neural networks and tree search.
Nature 529.7587 (2016): 484.

D. Silver, et al.
Mastering the game of Go without human knowledge.
Nature 550.7676 (2017): 354.

References II

S. Tomov, J. Dongarra, and M. Baboulin.
Towards dense linear algebra for hybrid GPU accelerated manycore systems.
Parallel Computing 36.5-6 (2010): 232-240.

A. Zafari, E. Larsson, and M. Tillenius.
DuctTeip: A task-based parallel programming framework for distributed memory architectures.
Technical Report 2016-010, Uppsala University, Division of Scientific Computing, 2016.