Distributed Computation with Mutual Exclusion and Precedence Constraints

Davey Proctor

May 9, 2019

Abstract

I propose a model of distributed parallel computation that observes both precedence and mutual-exclusion constraints. I discuss the apparent need to extend prior models that have been presented to handle precedence or mutual-exclusion but not both; I show that doing so would unduly burden the maxweight-optimizing scheduler. Instead, I suggest a simple "minimum computational unit" programmer-level technique that fits within existing frameworks to deal with both constraints. Given appopriate use of this technique, the fullest capacity of the network can still achieved for seemingly the lion's share of real-world scenarios.

1 Introduction

The problem of parallel computation in a distributed setting has gotten recent attention due to the increased demand in big data processing jobs that can be allocated across many servers housed in data centers. These servers naturally have different capacities for different types of jobs; for instance, GPUs are particularly good at the floating point arithmetic and matrix math that arise in machine learning contexts.

The authors in [1] presented the "Generalized Constrained Queueing System" (GCQS) model, in which servers with random response times probabilistically advance arrivals to subsequent queues. They used the popular Max-Weight algorithm (originally in [2]) with more sophisticated batched recomputation (only recompute the policy every so often). The authors of [3] additionally sped up the max-weight computation by not computing the exact max-weight value (which is NP-Hard in several examples) but rather converging over time to higher values of the max-weight using random distributed choices that maintain the model's constraints while still maximizing throughput.

Although the GCQS model supports parallel computation, and they further show the model captures so-called "feedback" in which an n-degree fork is joined and probabilistically advanced among other m-degree forks, they do not support general precedence constraints among different degrees of parallelism

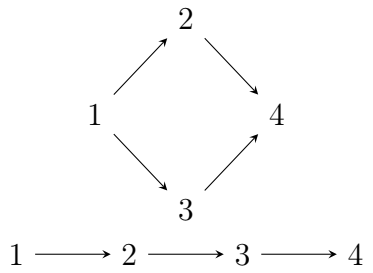

Figure 1: Top: A four-task job with directed edges representing precedence constraints. Bottom: additional constraints added to decrease virtual queue complexity without sacrificing throughput.

(often modeled by directed acyclic graphs (DAG)). The model is easily extensible to enforce mutual exclusion, if for instance a constraint is introduced to limit more than one server working at the same time on a certain node representing a task with that mutual exclusion constraint. Not discussed in the paper, one can introduce different job types to model a system where known rather than random degrees of parallelism occur from task to task within a job; still, these tasks must be started and completed synchronously, far short of the freedom of the general asynchronous DAG precedence model.

Towards this end of a general DAG model for precedence-constraints among a job's tasks, the authors in [4] maximize throughput by using a virtual queueing network and a maxweight-style policy. Because there are no server-side constraints besides not over-allocating a server to more than one task at any given time (there are not, for instance, inter-server constraints as there were in GCQS) - their max-weight optimization is linear in the number of virtual queues, and this computation can be done asynchronously and in a distributed manner among the servers, with each server scheduling itself myopically based on greatest local demand. Although the number of virtual queues is in general exponential in a DAG's tasks, they find that throughput does not suffer if additional precedence constraints are introduced (even to the point of making all jobs into a chain of tasks, see figure 1, which results in a linear number of corresponding virtual queues).

[5] extends this model to networks in which communication latency is a significant concern. Yet neither of these models observe mutual exclusion constraints among tasks. My contribution is to determine the extent to which this limits the applicability of the model; I present a simple technique working within these models that seems to capture much of what is desired in real-world cases.

1.1 Motivating Example

Consider the problem of writing and reading values to and from a database. If we want any given computation to be persistent, we better be able to store

values for later access. To consider a concrete example, imagine a social media site that must handle a new post from a user. This job might involve a series of partially-parallelizable tasks some of which must precede others (i.e. if one's output is the other's input), and some of which can't occur at the same time:

1. Decide whether the post is toxic.
2. Decide whether the post is fake news.
3. Decide who should see it.
4. Store one copy of the post centrally.
5. Store a pointer to the centrally-stored post on each news feed for those who should see it.

Now suppose the DAG representing precedence constraints among the tasks of this computation has no mutual-exclusion constraints among the tasks (and a single task can't be self-synchronized). A sequence of interleaved database reads and writes could leave the wrong post stored on the wrong news feed - opportunities for error are endless.

2 Model for Locks and Precedence

To model this and similar examples, we start with [4] for precedence constraints (carrying over definitions and notation where possible). We introduce lock-protected groups of tasks $\mathcal{L}_l \subseteq \{1, \ldots, K\} : 1 \leq l \leq L$ where K is the number of tasks and L is the number of groups of tasks that must be mutually-exclusive in their operation (within a group at most one task is served per unit time). See Figure 2 for precedence-constrained tasks with mutual-exclusion groups. Note that for simplicity we use chain dependencies as in [5] and as identified as equivalent to the general DAG precedence model for throughput-optimality in [4].

Formally, suppose there are J servers where $\mathcal{A}_j$ are the activities server j can do. $k(a)$ is the task associated with activity a. Suppose $q \in \mathcal{A}^J$ is the server-allocation at a given time instant. The mutual-exclusion constraint is that, at all times:

$$\sum_{j=1}^{J} \mathbf{1}\{k(q_j) \in \mathcal{L}_l\} \leq 1.$$

3 NP-Hard Maxweight-Style Policy

Suppose we wanted to use the maxweight-style policy from [4] on this extended model. They take:

$$\underset{q \text{ is feasible}}{\arg\max} \; -(Q^n)^T D q,$$

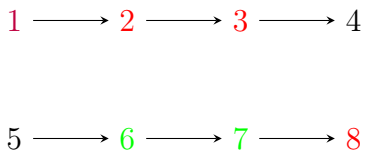

Figure 2: Two four-task jobs in the new model. As in [4], arrows represent precedence-constraints. Similar to [1], inter-server constraints, in this case mutual exclusion, exist. No two tasks of the same group (colored the same way, save black, in the figure) should be serviced at the same time. Because the same job type arrives multiple times and receives asynchronous service from task to task, mutex constraints just among tasks in the same job type (e.g. {1}; {6, 7}) must still be treated with care.

Where Q is the vector of queue lengths, D is the completion-rate-normalized transition matrix to subsequent queues given choices of service, and q is the service allocation vector introduced previously. Intuitively, the optimization asks for service to long queues that will add work to short queues (if anywhere) and, all else equal, not take too long. "q is feasible" deserves attention. The authors go to great lengths in their queueing network construction to free q from many constraints; the only set of constraints is that no server can be allocated to more than one activity at any given time. This means the optimization is distributed, in that any one server need only worry about itself, seeking put itself to best use; it is in linear time.

The optimization in the new model is the same (we choose not to formally show its throughput optimality). The issue is that the complexity of the optimization is much larger due to these inter-server constraints. I feel it is NP-Hard (TODO).

4 Minimum Computational Units: Reduction of Model with Locks to Previous Model

Instead of adding in inter-server constraints, which vastly complicates the max-weight optimization, we see what we can get far enough with just the original model of [4]. We introduced the freedom of lock groups to help us model the mutual exclusion constraints; now we show how we can do away with them at the level of implementation. We hope to maintain the mutual-exclusion constraints and have the same network capacity as before.

The basic idea of the reduction is simple: for each lock group, have exactly one server that is able to serve the tasks in that group (the server might additionally be able to serve tasks outside the group). That is, we require that any synchronized group of tasks $\mathcal{L}_l$ has a single server $j(l)$ that is capable of servicing the task. $j(l)$ might additionally be able to serve other tasks.

If this invariant holds, we can argue (formally, in Theorem 4.1), that mutual-

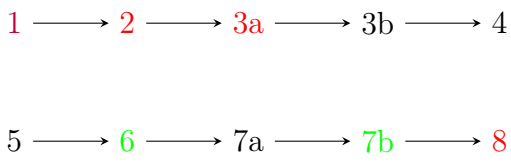

Figure 3: Without loss of generality or capacity region, we simplify the precedence-constrained model with lock groups to one where only one server is capable of serving any of the tasks in the same lock group. The scheduler requires cooperation from the programmer in that any task in a lock group should be as trivial as possible, not hiding latent opportunities for distributed computation. Splitting tasks may be required.

exclusion holds as originally specified, and the "q is feasible" max-weight optimization can function in linear time just as it did in [4].

4.1 Capacity Region Maintained

Seemingly, there's no telling how much throughput would be lost in this reduction, because previously multiple servers could have capabilities that intersect with the same lock group. In response, I question how many real-world situations really require that dimension of generality. A lock-protected database read is on a database typically on a single server; if it's a distributed database, it probably doesn't need to be synchronous. Now a task might have a small section that requires mutual exclusion (imagine computing the result to be stored in a database, and then indeed storing it). Maybe any one of multiple servers could accomplish the first part of this computation, and then the database write is implemented as a synchronized communication with a central server. The programmer-level (rather than scheduler-level) strategy is to make this latent server actual: split up the task so that the piece that can be done by multiple servers is one task, and a dependent task (e.g. communicate the result synchronously to the database server) is its own task (with only one server serving that one). This programmer-level strategy is illustrated by example in 3.

4.2 Mutual Exclusion Constraints Maintained

Because a server is only allocated to one task at a time, the total number of tasks in any mutual-exclusion group being served at the same time is less than or equal to 1. This implies the mutual-exclusion constraints are observed. This is shown formally below.

Theorem 4.1. *Mutual exclusion is maintained in the reduced model. At all times, for all lock groups* $1 \leq l \leq L$,

$$\sum_{j=1}^{J} \mathbf{1}\{k(q_j) \in \mathcal{L}_l\} \leq 1.$$

Proof. We know that any lock $1 \leq l \leq L$ has exactly one server $j(l)$ that is capable of serving any task in $\mathcal{L}_l$. This server can serve at most one task anywhere at any given time; it might or might not be a task in $\mathcal{L}_l$. We have, for any $1 \leq l \leq L$,

$$\begin{aligned}
&\sum_{j=1}^{J} \mathbf{1}\{k(q_j) \in \mathcal{L}_l\} \\
&= \mathbf{1}\{k(q_{j(l)}) \in \mathcal{L}_l\} \\
&\leq 1.
\end{aligned}$$

□

5 Conclusion

I presented a model for distributed computating of tasks that can't be done in the wrong order (precedence constraints) or, sometimes, at the same time (mutual-exclusion constraints). None of the authors for contending models mentioned both of these cases together, even as they presented promising parallel computation models for one or the other. I extended the model from [4] to precisely state the problem before noticing that naively using max-weight optimization to achieve maximum throughput could be prohibitively slow. Instead, I introduced a programmer-level technique to speed up the computation without sacrificing throughput or model generality.

Future work would be to take the model in [5] and deal with the issue of communication latency in this context. Locks are an important synchronization primitive; extending the model to condition variables and the like might present additional difficulties and opportunities.

References

[1] Leandros Tassiulas and Partha P. Bhattacharya. Allocation of interdependent resources for maximal throughput. *Communications in Statistics. Stochastic Models*, 16(1):27–48, 2000.

[2] L. Tassiulas and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control*, 37(12):1936–1948, Dec 1992.

[3] Devavrat Shah and Jinwoo Shin. Randomized scheduling algorithm for queueing networks. *CoRR*, abs/0908.3670, 2009.

[4] R. Pedarsani, J. Walrand, and Y. Zhong. Scheduling tasks with precedence constraints on multiple servers. In *2014 52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1196–1203, Sep. 2014.

[5] Chien-Sheng Yang, Ramtin Pedarsani, and Amir Salman Avestimehr. Communication-aware scheduling of serial tasks for dispersed computing. *CoRR*, abs/1804.06468, 2018.