Process Cooperativity as a Feedback Metric

in Concurrent Message-Passing Languages

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Matthew Fluet, Supervisor

Dr. James Heliotis, Reader

Dr. Rajendra K. Raj, Observer

Process Cooperativity as a Feedback Metric in Concurrent Message-Passing Languages

by

Alexander Dean, B.S.

THESIS

Presented to the Faculty of the Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

Rochester Institute of Technology

August 2014

Abstract

Process Cooperativity as a Feedback Metric

in Concurrent Message-Passing Languages

Alexander Dean, M.S.

Rochester Institute of Technology, 2014

Supervisor: Dr. Matthew Fluet

Runtime systems for concurrent languages have begun to utilize feedback mechanisms to influence their scheduling behavior as the application proceeds. These feedback mechanisms rely on metrics by which to grade any alterations made to the schedule of the multi-threaded application. As the application's phase shifts, the feedback mechanism is tasked with modifying the scheduler to reduce its overhead and increase the application's efficiency.

Cooperativity is a novel possible metric by which to grade a system. In biochemistry the term cooperativity is defined as the increase or decrease in the rate of interaction between a reactant and a protein as the reactant concentration increases. This definition translates well as an information theoretic definition as: the increase or decrease in the rate of interaction between a process and a communication method as the number of processes increase.

This work proposes several unique takes on feedback mechanisms and scheduling algorithms which take advantage of cooperative behavior. It further compares these algorithms to other common mechanisms via a custom extensible runtime system developed to support swappable scheduling mechanisms. A minimalistic language with interesting characteristics, which lend themselves to easier statistical metric accumulation and simulated application implementation, is also introduced.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Runtime systems can be broken up into multiple distinct parts: the garbage collector, dynamic type-checker, resource allocator, and much more. One sub-system of a language's run-time is the task-scheduler. The scheduler is responsible for order of task evaluation and the distribution of these tasks across the available processing units.

Tasks are typically spawned when there is a chance for parallelism, either explicitly through `spawn`, or `fork` commands or implicitly through calls to parallel built-in functions like `pmap`. In either case it is assumed that the job of a task is to perform some action concurrent to the parent task because it would be quicker if given the chance to be parallel.

It is up to the scheduler of these tasks to try and optimize for where there is opportunity for parallelism. However, it's not as simple as evenly distributing the tasks over the set of processing units. Sometimes, these tasks need particular resources which other tasks are currently using, or maybe some tasks are waiting for user input and don't have anything to do. Still worse, some tasks may be trying to work together to complete an objective, like in the `pmap` example above.

Tasks however, in functional language verbiage, are typically called *processes* due to the inherent isolation this term brings and the language paradigm calls for. So how do these processes share information or return a value back to their parent? Message passing is a common abstraction to shared memory. Message passing is akin to emailing a colleague a question. You operate independently, and your colleague can check her mailbox when they want to and respond when they want to. Meanwhile you are free to operate on an assumption until proven wrong, wait until she gets back to you, or even ask someone else.

While message passing is a good method for inter-process communication, it is also a nice mechanism for catching when two processes are working together. Let's, for example, consider a purely functional `pmap`, where all workers are copied subsections of the list. Each worker thread will have no need to cooperate and thus no messages will need to be passed amongst them. However, suppose the function being mapped uses several processes

accessing a piece of shared memory to update it's particular cell in the list. These processes would do little good if treated like the course-grained parallelism the `pmap` workers create.

Bad example, should be more real world, and doesn't flow well into the next paragraph.

There are a large number of mechanics that scheduling systems can use in an attempt to improve work-load across all processing units. Some of these mechanics use what's called a feedback system. Namely, they observe the running behaviour of the application as a whole, (i.e. collect *metrics*), and modify themselves to operate better.

Process cooperativity is an interesting metric by which to grade a system. In biochemistry the term cooperativity can be defined as an increase or decrease in the rate of interaction between a reactant and a protein as the reactant concentration increases. We can translate this into an information theoretic definition:

Definition 1. *The* **degree of cooperativity** *of a system is the increase or decrease in the rate of interaction between processes and an inter-process communication method as the concentration of processes fluctuate.*

Thus, when a process attempts to pass a message to another we know it's trying to cooperate on some level. When this frequency of interaction is high, it may indicate a tight coupling of processes or fine-grained parallelism. If it is low, this could indicate course-grained parallelism. In either event, a scheduler able to recognize these clusters of cooperative and non-cooperative processes should have an edge over those that don't.

Chapter 2 will look first at the background of classical scheduling systems as well as the recent feedback-enabled approaches. Then we will also examine the types of message passing implementations and how these effect scheduling decisions, now that we are looking at process cooperativity. Chapter 3 introduces our work on a language and compiler, built to easily simulate system cooperativity and visualize the effects of scheduling mechanisms on these systems. We also discuss a few example mechanics which take advantage of cooperativity. Some example applications which demonstrate different degrees of cooperativity and phase changes are also explained. In Chapter 4 we run our cooperativity-enabled schedulers along with a few common non-feedback-enabled schedulers on the example applications and discuss the results. Finally, in Chapter 5 we give some concluding remarks and avenues of future research we believe would be fruitful.

Chapter 2

Background

Since the formalization of feedback driven systems and the advent of Cybernetics, multiple fields have attempted to mold these principles to their own models; and run-time schedulers are no exception. This is due, in part, from process scheduling in parallel systems being fundamentally an NP-Complete problem [**?**]. Instead, focus has been more fruitful when pursuing the optimization of various metrics using some particular objective function [**?**] to tune for particular edge cases. As such, scheduling based on feedback metrics is not new [**?**].

There is a big distinction though, which can be made between the effects of control theory in classical cybernetic applications versus that of run-time systems. This is primarily in the adaptation of the controller in the generic feedback loop (figure 2.1). In typical mechanical feedback loops there are two scenarios which need to be avoided: resonance and rapid compensation. It can be seen that most controller models will attempt to damp the adjustments to reduce oscillation which could cause resonance or sharp spikes in behavior based on its output. This is due to the limitations of the physical space in which they are having to deal with.

However, in run-time scheduling systems we would very much like to do the opposite. We would prefer tight oscillations or consistent behavior of our runtime so as to achieve minimal overhead from our modifications. We can also compensate, to reach our reference signal, as quickly as we need to as there are no physical restrictions for our modifications.

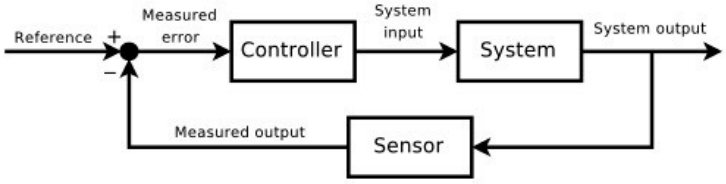

Figure 2.1: A classical feedback loop representation.

Also, note that run-time systems tend to extend to more than just the process-scheduling systems. They also encompass garbage collection, dynamic type checking, statistical collection and other debugging mechanisms, and general resource allocation (*e.g.* I/O). For the purposes of this paper we will be exclusively talking about the process scheduling system and intentionally ignore the effects of these other, albeit important, sub-systems. This distinction must be made however, as some scheduling systems also take into account the effects of other sub-systems on the placement and order of process evaluation. For example, White *et al.* [**?**] discuss heap size as a metric for process selection as a means of curtailing garbage collection effects.

Not sure why the image is blurry

2.1 Classical Run-Time Scheduling

Another distinction must be made as far as the level of foresight the scheduling systems have, at least, within this paper. There is a spectrum of clairvoyance in classical job-scheduling, in that on one end, job-schedulers have full foresight over the jobs which will enter the queue and their order. These schedulers have the opportunity to optimize for future events, which is a luxury the scheduling systems that this paper discusses do not have.

Citation needed, taxonomy paper?

However, as it is a spectrum, there is a single point of knowledge this subrange of schedulers can assume. Namely, that the first job will always be the last, and all other jobs will spawn from it. Thus there will always be a single process in the queue at the beginning. This is true as the runtime will always require an initial primary process (*e.g.* the 'main' function), and once that function is completed, the system is terminated (despite the cases of unjoined children). Apart from this, all other insights will need to be gleaned from the evaluation of this initial process.

2.2 Message Passing

In concurrent systems, there are a number of methods for inter-process communication. Arguably though, one of the more popular abstractions is the idea of message passing. Message passing in general can be broken down into two types, asynchronous or synchronous, and then further by how they are implemented. However, when discussing process scheduling the method of their implementation is often of some consequence .

Needs citation?

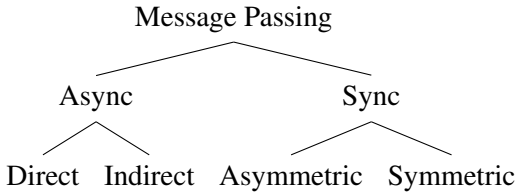

Figure 2.2: A High-Level Message-Passing Taxonomy

In asynchronous message passing a process can either be provided a rendezvous point or an identifier for another process. To send a message in either case requires pushing/copying the message into a shared or global memory space for another process to access (possibly) at a later time. This push/copy can be done in a lock free manner with some lower level atomic data structures such as a double-ended queue. But in either a locked or lock-free manner, the process performing the send still blocks (if only for a moment) on the operation.

In terms of scheduling, a language with asynchronous message passing can ignore the effects of blocking operations, but will need to look closer at process placement to take advantage of possible gains in cache affinity [**?**]. For example, the effects of the cache on direct message passing (*e.g.* a process mailbox) can be substantial as two processes on different cores will need to copy stack frames to memory and back rather than just a pointer to the message in the originator's stack frame. In indirect message passing the task is even worse as multiple processes may need access to the same data.

In synchronous message passing, a process must meet another at a provided rendezvous point but can either be symmetrical or asymmetrical. Note that the rendezvous point is not a requirement in the sense that direct synchronous messaging isn't possible. Instead we think of a rendezvous point in synchronous communication to be time bound rather than location bound (*i.e.* two processes are blocked until communication occurs, the implementation of this passing is irrelevant to this classification).

Asymmetrical message passing is synonymous with Milner's Calculus of Communicating Systems [**?**] or standard Π-Calculus [**?**], in that you have a sender and a receiver which will block on their respective functions around an anonymous channel until the pass has been completed. This differs from a symmetrical message passing in that the only operation on the channel is a blocking function which swaps values with the process at the other end.

Note, it is possible to simulate symmetrical message passing on asymmetrical message channels, but in terms of scheduling of synchronizing processes, order is now a factor that needs to be considered. On top of this, directionality can also be a factor which complicates the channel implementation. Namely, the internal queuing of senders or receivers may not percolate hints up to the scheduler regarding their queue position. For the alternative, symmetrical message passing or swap channels, the order is directly handled by the scheduling of the system (*i.e.* the order at which the channels evaluate the $swap$ command can be directly governed).

2.3 Example Scheduling Feedback Systems

A basic feedback scheduler can be explained using a metric like 'waiting time vs running time'. A standard round-robin scheduler may utilize a queue of processes and attempts to be fair by giving every process an equal chance to run. However, in a contrived example such as a fork-bomb, the scheduler may never reach the end of the queue and the rest of the system could starve. In our example feedback enabled system each process could be ranked according to its ratio of waiting time over running time. Thus as time went on, even in the event of a fork-bombing effects the beginning of the queue would still get some time to run.

There is quite a literature behind the types of metrics used though. Yet statistical collection is only a third of the feedback loop. A classification of the system must be made by a control mechanism based on the recorded state, and then a modification must be made to the scheduler which should positively effect the state of the system's future. The key is to make sure this effect is transparent and results in minimal overhead, while still achieving the objective function.

This objective function, for the purposes of simplicity is typically execution time. However, it can also take into account resource utilization, process/channel fairness, fulfillment of real-time guarantees, *etc.* to rate the schedule as a whole. Note that this function is an external measure (evaluation) of the schedulers effectiveness, which will be brought up further in section **??**. For now however, note that the objective function and control metrics are separate and evaluate two different things.

Before discussing Cooperativity as a feedback metric though, it is important to introduce previous metrics and their respective control schemes. Interactivity is one metric which evaluates the system's current scheduler. It looks at the ratio of communication to

computation currently taking place in the system as a whole. One scheduling system which utilizes this metric indirectly is in CML [**?**].

It's method by which it grades the system is the recognition of *computation*-bound processes versus *communication*-bound processes. A *communication*-bound process is one which has previous blocked itself without using it's entire scheduled quantum, the rational for this is only a synchronizing thread would be willing to block; thus a *computation*-bound process is the opposite (*i.e.* one that always uses its full quantum). After labeling the processes the scheduler pushes *computation*-bound processes to a secondary queue so they don't eat up all of the time. This improves the time the scheduler can spend attending the processes which need to synchronize, which in turn improves the Interactivity of the application (*i.e.* GUI widgets will finish their message passing while the back end computes).

2.4 Cooperativity as a Metric

This section feels off, I wish I had referenced a "generic" definition by which to base all assumptions and initial observations off of.

As mentioned, note that the Interactivity metric looks only at the ratio of *communication*-bound processes to *computation*-bound processes and misses the quantity of communication on a per-process level which could indicate needed changes in scheduling quantum or process locality with those which are frequently in touch. This hints at a need to look for some level of cooperation between processes in the phases of an application's life-time.

Rather than a ratio of *communication* to *computation*, instead we need to look at a frequency of communication and between which processes (*i.e.* rate at which a process communicates with a particular channel). One method this can be achieved by is, in addition to recording which processes are *communication* and *computation*-bound, the scheduler can also tag the process with it's frequency of communication on each channel it touches over time. By doing so, the scheduler can calculate a process's cooperativity with one or more processes by comparing frequency counts on particular channels, perhaps by a sum of ratios.

Cooperativity, in the general sense, has been a probabilistic interpretation of how two actors can be affected by the actions of others. This has been a sociological term as well as a biological one. Abeliovich [**?**], thusly defines a notion of positive cooperativity as:

> A process involving multiple identical incremental steps, in which intermediate

> states are statistically underrepresented relative to a hypothetical standard system [...] where the steps occur independently of each other. Likewise, a definition of negative cooperativity would be a process involving multiple identical incremental steps, in which the intermediate states are overrepresented relative to a hypothetical standard state in which individual steps occur independently.

In other words, while a process can be cooperative, a system can be shown to possess *positive* or *negative* cooperativity as the set of cooperative processes increases or decreases (respectively). This lends itself quite readily as a metric for an evolving system of phases. Assuming the cooperativity can be directly probed we can check it against an evolving standard state the scheduler proposes. The feedback mechanism would be to attempt to raise or lower the cooperativity of the system to get it back to this standard state. It can do so by increasing or decreasing the synchronization allowance each scheduler gets per a given time frame for example. This would have the effect of bounding the frequencies of communication.

Chapter 3

Methodology

3.1 Overview

3.2 ErLam

3.2.1 The ErLam Language

3.2.2 Channel Implementation

3.2.3 The Scheduler API

3.2.4 Example: The CML Scheduler

3.3 Simulation & Visualization

3.3.1 Runtime Log Reports

3.3.2 Cooperativity Testing

As part of the thought experiment, we needed to implement a set of test cases which would give a decent coverage of applications which

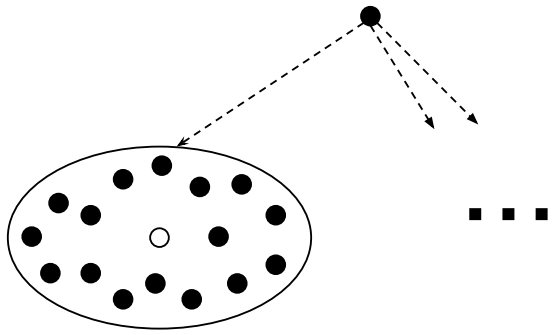

Figure 3.1: Graphical representation of $PTree$, N Parallel work groups.

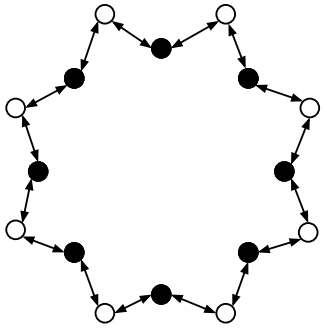

Figure 3.2: Graphical representation of $PRing$, full system predictable cooperation.

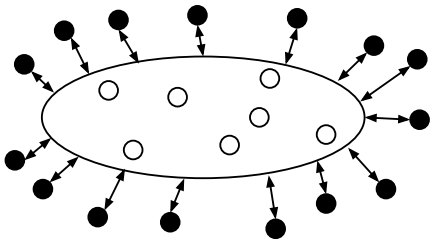

Figure 3.3: Graphical representation of $ClusterComm$, N processes to M channels for unpredictable full system cooperation.

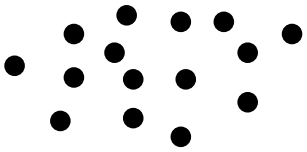

Figure 3.4: Graphical representation of $ChugMachine$, N worker processes without cooperation.

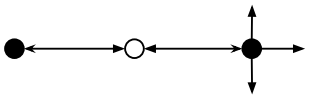

Figure 3.5: Graphical representation of $UserInput$, single randomly hanging process.

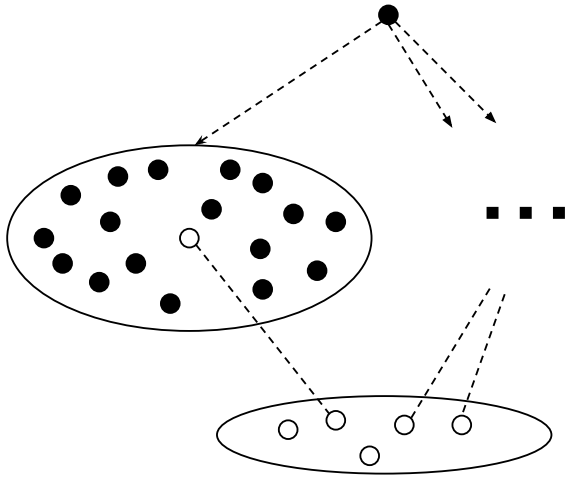

Figure 3.6: Graphical representation of $JumpShip$, N Parallel phase shifting work groups.

3.4 Cooperativity Mechanics

3.4.1 Overview

3.4.2 Longevity-Based Batching

3.4.3 Channel Pinning

3.4.4 Bipartite Graph Aided Shuffling

Chapter 4

Results and Discussion

Chapter 5

Conclusion and Future Work

Bibliography

[1] Hagai Abeliovich. An empirical extremum principle for the hill coefficient in ligand-protein interactions showing negative cooperativity. *Biophysical journal*, 89(1):76–79, 2005.

[2] Kunal Agrawal, Charles E Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS)*, 26(3):7, 2008.

[3] John L Bruno, Edward Grady Coffman, RL Graham, WH Kohler, R Sechi, K Steiglitz, and JD Ullman. *Computer and job-shop scheduling theory*. Wiley, 1976.

[4] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2):141–154, 1988.

[5] Kurt Debattista, Kevin Vella, and Joseph Cordina. Cache-affinity scheduling for fine grain multithreading. *Communicating Process Architectures*, 2002:135–146, 2002.

[6] Richard D Dietz, Thomas L Casavant, Mark S Andersland, Terry A Braun, and Todd E Scheetz. The use of feedback in scheduling parallel computations. In *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings., Second Aizu International Symposium*, pages 124–132. IEEE, 1997.

[7] Michael R Garey, Ronald L Graham, and DS Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, 1978.

[8] Yuxiong He, Wen-Jing Hsu, and Charles E Leiserson. Provably efficient online non-clairvoyant adaptive scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 19(9):1263–1279, 2008.

[9] Robin Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

[10] Catuscia Palamidessi. Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–265. ACM, 1997.

[11] John H Reppy. Concurrent ml: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, 1993.

[12] Carl G Ritson, Adam T Sampson, and Frederick RM Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727–740, 2012.

[13] David R White, Jeremy Singer, Jonathan M Aitken, and David Matthews. Automated heap sizing in the poly/ml runtime. *Trends in Functional Programming*, 2012.