DLisp: Automatically distributed computation

Maxwell Bernstein
Tufts University

Matthew Yaspan
Tufts University

Abstract

Many of today's programs are written sequentially, and do not take advantage of the computer's available resources. This is in a large part due to the difficulty of using any given available threading or parallelization API. Moreover, these programs often fail to take advantage of the network to distribute work not just among processes to take advantage of the scheduler, but use available computing power over the network.

We solve this problem by automatically parallelizing or distributing computation across cores or even across a datacenter, then analyze the performance of our distribution algorithms across several modes. We created a toy programming language based on lisp and built the underlying parser, basis, and evaluator in Erlang, which also handles our network protocols and distribution algorithms.

1 Design

1.1 Language

We began with a Lisp-like language with support for most forms: `fixnum`, `boolean`, `symbol`, `lambda`, `funcall`, `define`, `val`, `quote`, `if`, `let`, `let*`, `eval`, `apply`, built-in functions, and closures. Then we decided that users would be less comfortable with a Lisp than a programming language with a syntax that mirrors existing programming languages like SML and OCaml, and changed the syntax. Mergesort, for example, begin as:

```
(define mergesort (xs)
  (if (or (null? xs) (null? (cdr xs)))
    xs
    (let* ((size    (length xs))
           (half    (/ size 2))
           (fsthalf (take half xs))
           (sndhalf (drop half xs)))
      (merge (mergesort fsthalf)
             (mergesort sndhalf)))))
```

and, after the syntax transformation, ended as:

```
fun mergesort(xs) =
  if null?(xs) or null?(cdr(xs))
  then xs
  else let* val size = length(xs),
            val half = size/2,
            val fsthalf = take(half, xs),
            val sndhalf = drop(half, xs)
       in merge(mergesort(fsthalf),
                mergesort(sndhalf))
       end;;
```

Both programs map to the same abstract syntax tree, which means that we even allow mixing and matching of styles in the same program, as in:

```
fun mergesort(xs) =
  if [or [null? xs] [null? [cdr xs]]]
  then xs
  else let* val size = length(xs),
            val half = size/2,
            val fsthalf = take(half, xs),
            val sndhalf = drop(half, xs)
       in [merge mergesort(fsthalf)
                 mergesort(sndhalf)]
       end;;
```

One language property that *did not* change in the transition was mutation; DLisp forces variable immutability. In forms that introduce new environments, such as `let` and `lambda`, shadowing is allowed – but never mutation. This, as it turns out, is key when attempting to parallelize computation.

For this reason, `map`, `pmap` (parallel map), and `dmap` (distributed map) are all built-in special forms.

1.2 Network

We use several terms across the span of this writeup:

- *Master* — Main controller computer from which the program is run and distributed. Communicates with several Machines and Workers.
- *Machine* — Logical or physical computer, which contains many Agents and Workers.
- *WorkPacket* — A serializable tuple of the form `{Exp` *x* `Env}` that is sent to Workers.
- *Worker* — Process whose sole purpose is to receive WorkPackets, evaluate them, and send the results back to the Master.
- *SlowWorker* — A Worker that has been artificially slowed down by a constant factor.
- *Agent* — Process whose sole purpose is to manage a work queue.
- *StealingAgent* — Agent that occasionally steals from other Agents when its worker is moving quickly.
- *RoundRobinMode* — Distribution mode that uses a circular queue to hand out work to Workers in order.
- *ByMachineMode* — Distribution mode that uses per-Machine statistics to determine which Worker should receive a given WorkPacket. Currently, this has two sub-modes:
  - *LowLatency* — Hand out work to whoever can respond the fastest to a HealthCheck.
  - *ByMemory* — Hand out work to whoever has the most computational power (currently measured by memory pressure) currently available.

We use normal (non-stealing) Agents to begin with, then proceed to demonstrate the utility and speed gains by using StealingAgents. Additionally, we introduce some SlowWorkers into the Worker pool to demonstrate that work stealing is an effective means of combating heterogeneous computational power.

Additionally, we demonstrate the results of different distribution modes (enumerated above) and their affects on end-to-end computation speed.

1.3 Startup Procedure

A Master node is started, and runs on node `M`. Independently, anywhere from 0 to N¿0 Machines are started up on the same network, with knowledge of the Master. In this case, we use Erlang nodenames, such as the atom `master@some.ip.address.here`, to identify the Master.

Each of those Machines will spawn some number of Agents (see `calculate1` and `calculate2` functions for details) based on the capacity of the machine. Currently, this is based primarily on the number of physical CPU cores.

Each Machine will then register with the Master, sending over its list of Agents and hardware stats. Machines can register at any time, but `dmap` will fail if no machine has registered with the Master.

2 Distribution Methods

The goal of this project was to allow for a client with basic coding abilities to vastly improve the performance of their program which may take up a significant amount of memory or processing power by distributing it in a more effective way, either by using Erlang's ability to spawn threads and collate responses to take advantage of the local scheduler, or by distributing the work across multiple machines over network.

2.1 Local Parallelization

The function `pmap` in DLisp takes in a function, which can be anonymous, and a list, as arguments. The parser decomposes these inputs into a list of WorkPackets consisting an expression and the environment in which the expression is to be carried out. The expression consists of an operation and a member of the list on which it is to be evaluated. For each member of the list, an Erlang process is spawned in which our `eval` module is called on the WorkPacket and then the evaluated result is sent back to the Master Erlang process. An `assemble` function collates all of the results and returns the mapped list.

The advantage of this is that it allows for more optimal scheduling of processes that are not dependent on one another. Because this is not a reduce operation, there's no data dependency to resolve, and evaluating sequentially wastes scheduling time for no discernible benefit.

2.2 Distributed Parallelization

The function `dmap` in DLisp works semantically just like `pmap`. However, under the hood, a significant amount is different. There are three possible methods of distributed map: RoundRobin, LowLatency, and ByMemory. Each

of these is a different load balancing technique that aims to distribute work in the most efficient way possible.

2.2.1 RoundRobin

In the round robin scheme, all machines that are to contain Worker processes are initialized with a number of processes calculate based on the amount of memory and/or number cores on the respective machines. Each machine sends their list of Worker processes identifiers to master, and the Master concatenates the lists into a queue and randomizes the order of the workers. When dmap is invoked, The `dmap` call is decomposed in the same way as `pmap` into WorkPackets for each item in the list. In the case of `dmap`, however, the WorkPacket is sent to processes selected from the aforementioned queue. A process is popped off the queue, sent a WorkPacket, and requeued at the back, in a RoundRobin scheme. After all of the work is sent, a list of IDs for each individual job is returned so that when Workers send results back, they can be collated and assembled into a mapped list by Master who sits in receiving mode until all the packets have been recovered.

2.2.2 LowLatency

In the LowLatency scheme, initialization occurs by Machine. When Machines register with Master, the Machine is appended to a list of Machine representations, each element containing the machine Pid, a queue of Processes local to the Machine. When `dmap` is called, for each iteration through the list given as an argument, a message is sent to all of the worker Machines, and the first one to respond is given the WorkPacket. The process is then repeated until work is entirely allocated.

The goal of this algorithm is to account for disparities in latency between machines. Although this was not an issue we experienced, it is plausible that a large organization with multiple data centers could see this issue if the pool of machines contained nodes with a significant enough distance from each other or nodes that are simply slower. Collation works the same as in RoundRobin, where the Master waits for each packet to return before in order returning the collated results.

2.2.3 ByMemory

The By Memory scheme is similar to low latency, except in this case available memory is prioritized. Much like in the low latency scheme, initialization occurs by machine, such that each machine, when it connects, sends its worker queue and statistics and is represented in a list of Machines by Master having its own distinct WorkerQueue. For each element in the list, the corresponding WorkPacket is sent to the machine that presents as having the most available memory. Machines are polled through each iteration of this process. ByMemory was the scheme we tested the least, because it was difficult to contrive scenarios of heterogeneity between machines in terms of memory usage in the scope of the time we were given.

2.3 Work Stealing

A major component in how we distributed work is through work stealing. The principle of work stealing is that when certain processes are busy, others can take on more work. This is a way to augment the load balancing above in having a failsafe in case a process or processes stall on work. In order to accomplish this, when we create worker processes on a node, these are actually pairs of Workers and Agents, which were briefly defined above.

The motivation is that the worker simply waits for work, performs the work when the work order is received and returns it, then repeats ad infinitum. The Agent, however, is the point-of-contact for the worker and anyone who wishes to send it work. It is called an agent because it behaves like an agent for an actress, sports player, or celebrity: the worker does the work, and the agent manages the worker and gives it work from outside sources. The list of processes inside of Master is a list of agents, in reality. The Master sends the Agent work, who passes it along to the worker or builds up a queue of jobs for the worker to do.

The Agent also negotiates, sending a message to a random agent asking to steal work from their queue when the worker is idle. This setup allows for any inefficiencies in the system to be mitigated. If in the low latency scheme there is one clear favorite machine, its scheduler may be bogged down with potentially high-powered jobs. While it is processing work, an Agent on another machine can steal from the queues being built up by the agents on the lowest latency machine. To the Master, this is irrelevant, but it allows for work to get done as fast as possible with little central micromanagement.

3 Results

Our findings (located in `results.csv`) indicates several things about the performance of `map(fac, range(0, 1000));;` across the different types of map, and across the different configurations of `dmap`. An integral part of our testing was artificially slow Workers.

3.1 SlowWorkers

To accomplish this, we added calls to `timer:sleep/1`. This doubled the length of time needed to accomplish any given task. Of course, slowing down every machine does not help demonstrate anything — we instead made a probabalistic model at startup time to determine which fraction of Workers would be "slow".

3.2 Test Code

In order to test our distribution method, we wrote some DLisp code.

```
fun foldl(f, acc, xs) =
  if null?(xs)
  then acc
  else foldl(f, f(car(xs), acc), cdr(xs));

fun vartimes(...) =
    foldl(bintimes, 1, ...);
fun fac(n) =
    apply(vartimes, range(1, n+1));

fun range(start, fin) =
  if (start > fin) or (start == fin)
  then nil
  else cons(start, range(start+1, fin));
```

We figured that `dmap(fac, range(0, 1000));;` would be a good test because:

- Not all units of work are the same difficulty.
- Most of those work items are not of insignificant difficulty, especially given that they are not running at "native" Erlang speed.
- There are many units of work, more than fit on any given Machine in our setup.

3.3 Machine Setup

We used one Master and three Machines, each with 8GB RAM and 4 cores. They were all on the same network, even in the same network closet.

Unfortunately, this is not a good test for the LowLatency mode, which is designed to handle systems with more heterogeneous network setups.

3.4 Findings

1. `map` is slow.
2. `pmap` is faster than `map` if the machine has more than one core.
3. `dmap` is faster than `map` and `pmap` if you have more than one machine helping out, and that machine is at least as powerful as the Master.
4. `dmap` with Timed/LowLatency mode is the fastest.
5. `dmap` with WorkStealing enabled is the fastest.

4 Conclusion

The regression results (see `regression_results.txt`) give a summary of the partial effects of each of the variables we incorporated based on the table shown previously. The intercept value (12476 ms) refers to the value of the reference observation, in which the type of parallel map used was Low Latency, there were no slow workers, and no work stealing. Outside of the intercept, none of the variables were statistically significant, which is almost certain to the lack of observations and high number of variables and interactions by comparison. With more time, a more robust dataset could be easily compiled and results much more definitive.

Generally speaking, RoundRobin clearly had an advantage over Low Latency. All else equal, it performed over 4 seconds faster, nearly halving the amount of time. These improvements were amplified when work stealing was enabled by about one more second. The changes were even robust to slower worker processes, with somewhat of a speedup indicated. Its possible that some sleeping processes allowed for the schedulers on machines to more easily allocate work, but these results are not strong enough for any conclusions to be made in this regard. Curiously, work-stealing made the Low Latency algorithm dramatically slower when slow workers were used. This makes sense, but makes a peculiar juxtaposition to the miniscule slowdown of Round Robin algorithms, which arguably went faster, and work-stealing without any slow workers in the low latency algorithm, which was about the same speed.