Concurrent Programming

Gavin Lowe and Bernard Sufrin

Hilary Term 2016-17

5: Example: Numerical Integration by Process-Farming

Numerical integration

We will look at the calculation in parallel of an estimate of a definite integral:

$$\int_a^b f(x)\,dx$$

This is not typical of the kind of problem for which we advocate the use of concurrent programming with channels. But it is very simple, so we can focus more on the structure and organization of the computation than on the problem itself.

Trapezium rule

The trapezium rule estimates the integral by splitting the interval $[a, b]$ into n intervals $[a + i.\delta, a + (i + 1).\delta]$, for $i = 0, \ldots, n - 1$, where $\delta = (b - a)/n$.

The integral over the range $[a + i.\delta, a + (i + 1).\delta]$ can then be estimated as

$$\frac{f(a + i.\delta) + f(a + (i + 1).\delta)}{2} * \delta.$$

Summing over all intervals gives us the estimate

$$\left(\frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(a + i.\delta) \right) * \delta.$$

Sequential code

Here's a straightforward sequential procedure to implement the trapezium rule:

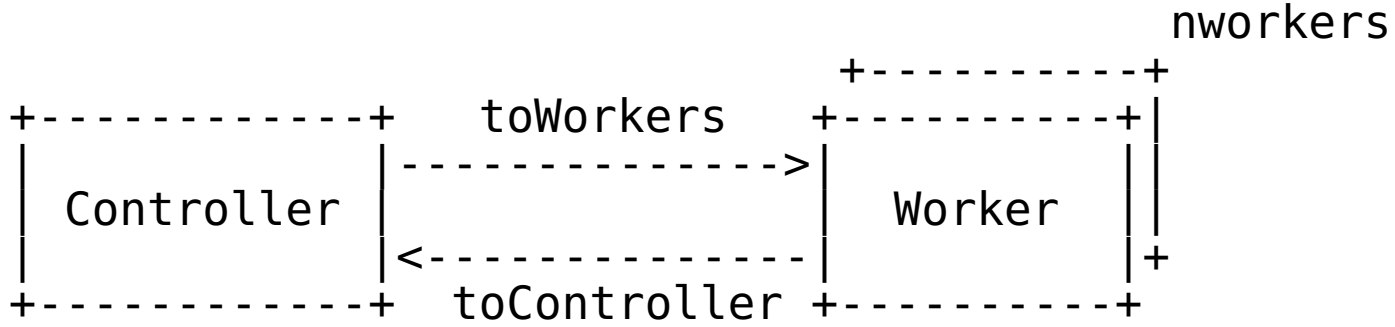
```
def integral(f: Double⇒Double, a: Double, b: Double, n: Int, delta: Double)=
  // assert(n*delta == b-a) // would fail because of rounding errors!
  assert(Math·abs(n*delta - (b-a)) < 0·000000001)

  var sum: Double=(f(a)+f(b))/2·0
  for (i ← 1 until n) sum += f(a+i*delta)
  sum*delta
}
```

Towards parallel code

Idea:

- ⋆ Split the interval $[a, b]$ into `nworkers` equal size ranges;
- ⋆ Have `nworkers Worker` processes (one per range);
- ⋆ A controller process tells each worker which range to work on;
- ⋆ Each worker runs the trapezium rule over its range, and returns the result to the controller;
- ⋆ The controller sums the sub-results to give the overall result.



- ⋆ This is an architectural pattern for *data parallelism* – sometimes known as a *farm*, with the controller as *farmer*.

Communication channels

The controller needs to tell the workers their tasks, which comprise:

- ⋆ the function `f`,
- ⋆ the range `[l,r]` to work on,
- ⋆ the number `taskSize` of intervals in its range,
- ⋆ and the size `delta` of each interval (so `taskSize*delta = r-l`).

We pass this information as a 5-tuple `(f, l, r, taskSize, delta)`, so define:

```
type Task = (Double⇒Double, Double, Double, Int, Double)
```

We will eventually use an `N2N` channel because there is one controller and several workers:

```
// val toWorkers = N2N[Task](1, nworkers, "toWorkers")
```

Each of the several workers eventually sends a `Double` to the controller so we will eventually define:

```
// val toController = N2N[Double](nworkers, 1, "toController")
```

A worker

```
def Worker(fromController: ?[Task], toController: ![Double]) = proc {
  val (f, l, r, taskSize, delta) = fromController?()
  val result = integral(f, l, r, taskSize, delta)
  toController!result
}

··· Integral
```

The controller

For simplicity, we will assume that `n` (the number of intervals) is divisible by `nworkers` (the number of workers).

So each worker receives a range of size `taskRange = (b-a)/nworkers`, containing `taskSize = n/nworkers` intervals, each of size `delta = (b-a)/n`.

```
def Controller
  (f: Double ⇒ Double, a: Double, b: Double, n: Int, nworkers: Int,
   toWorkers:   ![Task],
   fromWorkers: ?[Double]
  ) = proc {
      assert(n%nworkers==0)

      val delta = (b-a)/n
      val taskSize = n/nworkers
      val taskRange = (b-a)/nworkers

      var left = a // left hand boundary of current range
      for(i ← 0 until nworkers) {
        val right = left+taskRange
        toWorkers!(f, left, right, taskSize, delta)
        left = right
      }

      ...
```

```
        ...
        // Receive and sum results
        var result:Double = 0.0
        for(i ← 0 until nworkers) result += fromWorkers?()
        println(result)
    }
```

Putting the system together

```
def System(f:Double⇒Double, a:Double, b:Double,  n:Int, nworkers:Int): PROC =
{ val toWorkers    = N2N[Task](1, nworkers, "toWorkers")
  val toController = N2N[Double](nworkers, 1, "toController")

  val Workers =
      || (for (i ← 0 until nworkers) yield  Worker(toWorkers,toController))

  ( Workers || Controller(f, a, b, n, nworkers, toWorkers, toController) )
}
```

Exercise: what changes could be made to turn `System` into a procedure that returns the value of the integral to its caller.

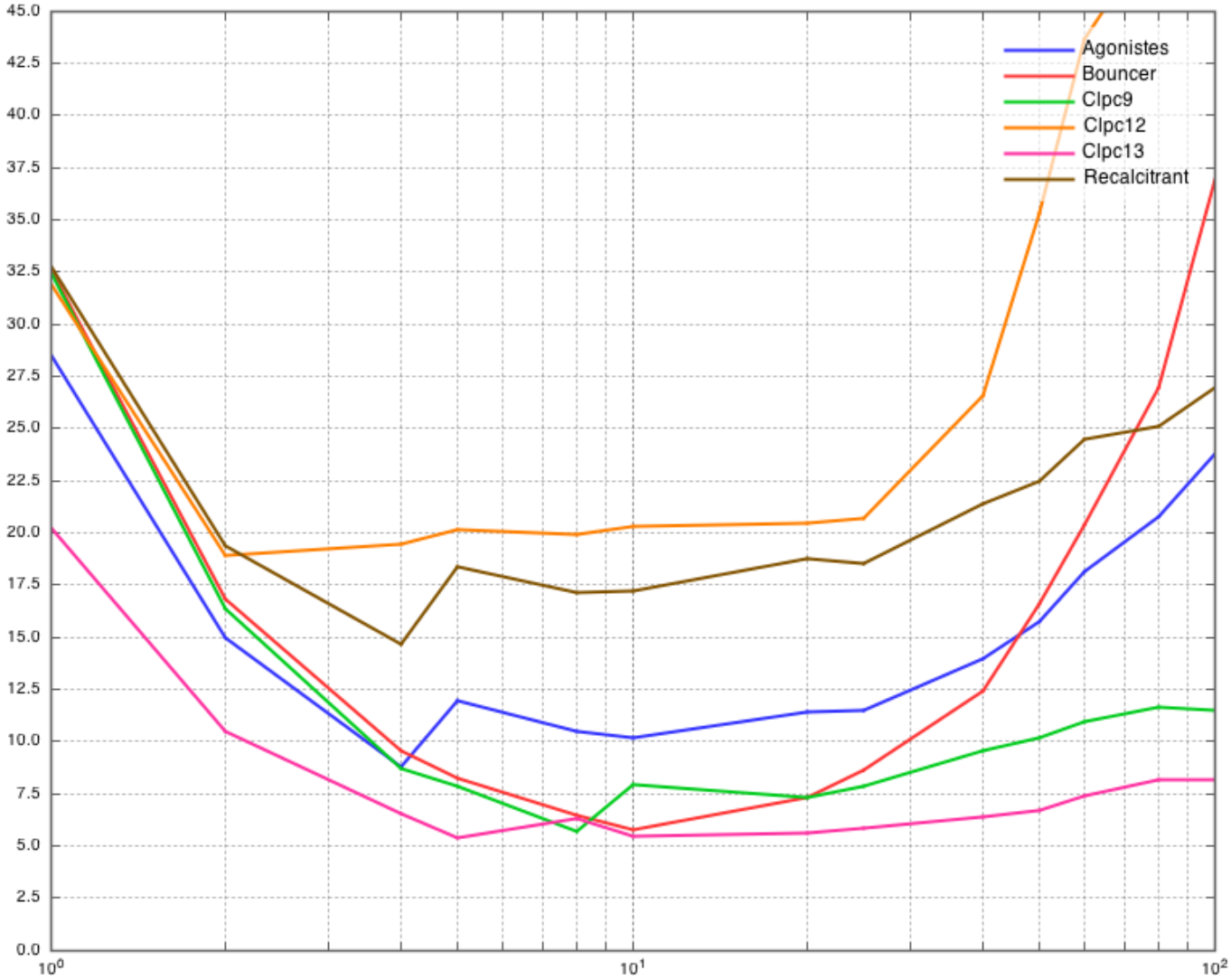
Empirical Performance Measurements

Integral

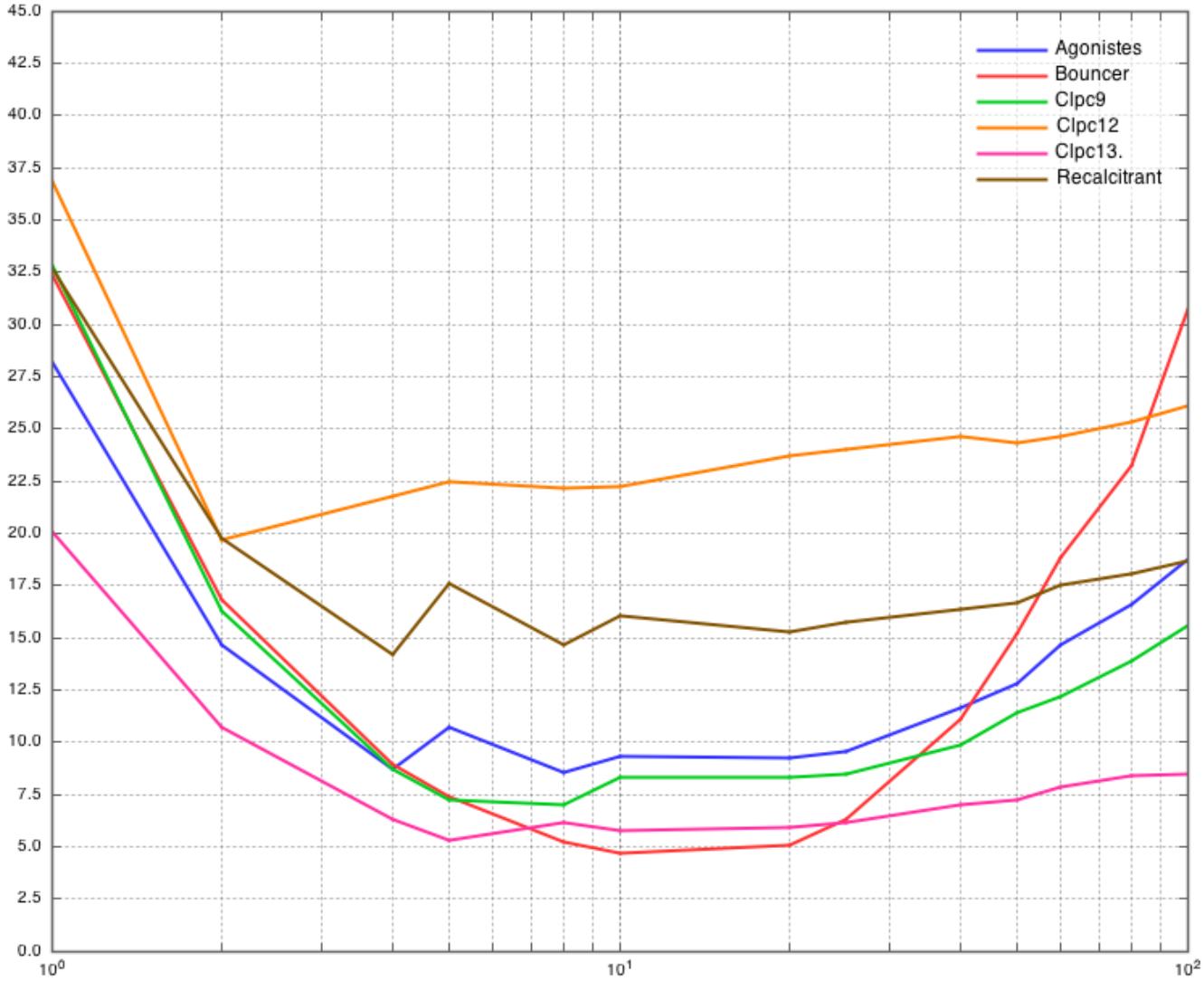
$$\int_{1.0}^{3.0} x^2 \, dx, \text{ with } n = 28.8 \cdot 10^5$$

Computer Details (for reference)

	$Sockets$	$\left(\frac{Cores}{Socket}\right)$	$\left(\frac{HWThreads}{Core}\right)$	$VCPUs$	Ghz
Agonistes	2	2	1	4	2.66
Bouncer	4	10	2	80	2.39
Clpc12	1	2	1	2	1.0
Clpc13	1	4	2	8	1.6
Clpc9	2	4	1	8	2.39
Mimi	1	2	2	4	2.9
Recalcitrant	1	2	2	4	1.66



T (ms/integration) vs. log $nworkers$. Size of a worker's task (number of strips) is $\frac{28.8 \cdot 10^5}{nworkers}$



Same integration: pooled threads with 4-second timeout: performance marginally better.

Diminishing the relative overhead of process creation

- ⋆ Each worker was created, performed a single task, and then terminated.
- ⋆ The worker-creation-cost to task-cost ratio was large.
- ⋆ An alternative is to make each worker perform more than one task, by splitting the interval into `ntasks` subintervals where `ntasks > nworkers`.
- ⋆ The tasks are distributed between the workers, each receiving a new task when it has finished its previous one.
- ⋆ This pattern is known as *bag of tasks*, or *process farm*: the controller (farmer) holds a bag of tasks, which it distributes to workers.

A worker

The worker process has to be willing to repeatedly receive tasks (at least until one of the channels is closed).

```
def Worker(fromController: ?[Task], toController: ![Double]) = proc {
  repeat {
    val (f,l,r,taskSize,delta) = fromController?;
    val result = integral(f,l,r,taskSize,delta);
    toController!result
  }
}
```

Each worker will terminate when its `fromController` channel is closed.

Towards a concurrent controller

We could design the controller to:

1. Distribute `nworkers` tasks;
2. Repeat `ntasks - nworkers` times: receive a result and distribute the next task;
3. Receive the final `nworkers` results;
4. Close the `toWorkers` channel so they can terminate.

We can simplify this by exploiting the potential for concurrency.

Concurrent Controller

The two tasks:

1. Distributing the tasks (and closing the `toWorkers` channel)
2. Receiving and summing the results

are independent, so we split them between two concurrent processes.

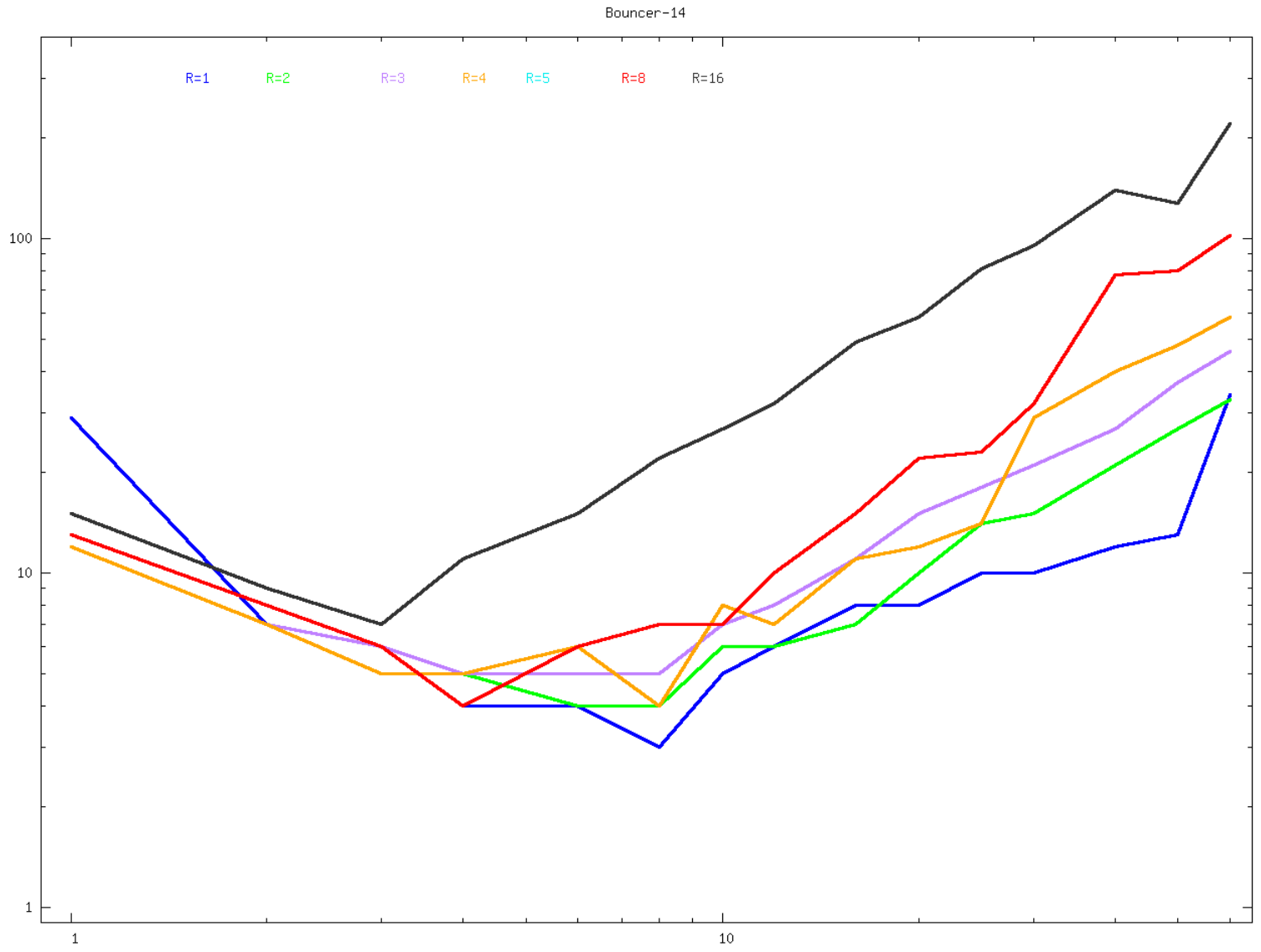
```
def Controller
  (f: Double⇒Double, a:Double, b:Double, n: Int, nworkers: Int, ntasks: Int,
   toWorkers:    ![Task],
   toController: ?[Double]) : PROC =
{
   val Distributor = ···
   val Collector   = ···

  (Distributor || Collector)
}
```

```
val Distributor = proc {
  val delta     = (b-a)/n            // size of each interval
  val taskSize  = n/ntasks           // # intervals in each task
  val taskRange = (b-a)/ntasks       // size of range of task

  var left = a;                     // left hand boundary of current range
  for (i ← 0 until ntasks) {
    val right = left+taskRange
    toWorkers!(f, left, right, taskSize, delta)
    left=right
  }
  toWorkers·close
}
```

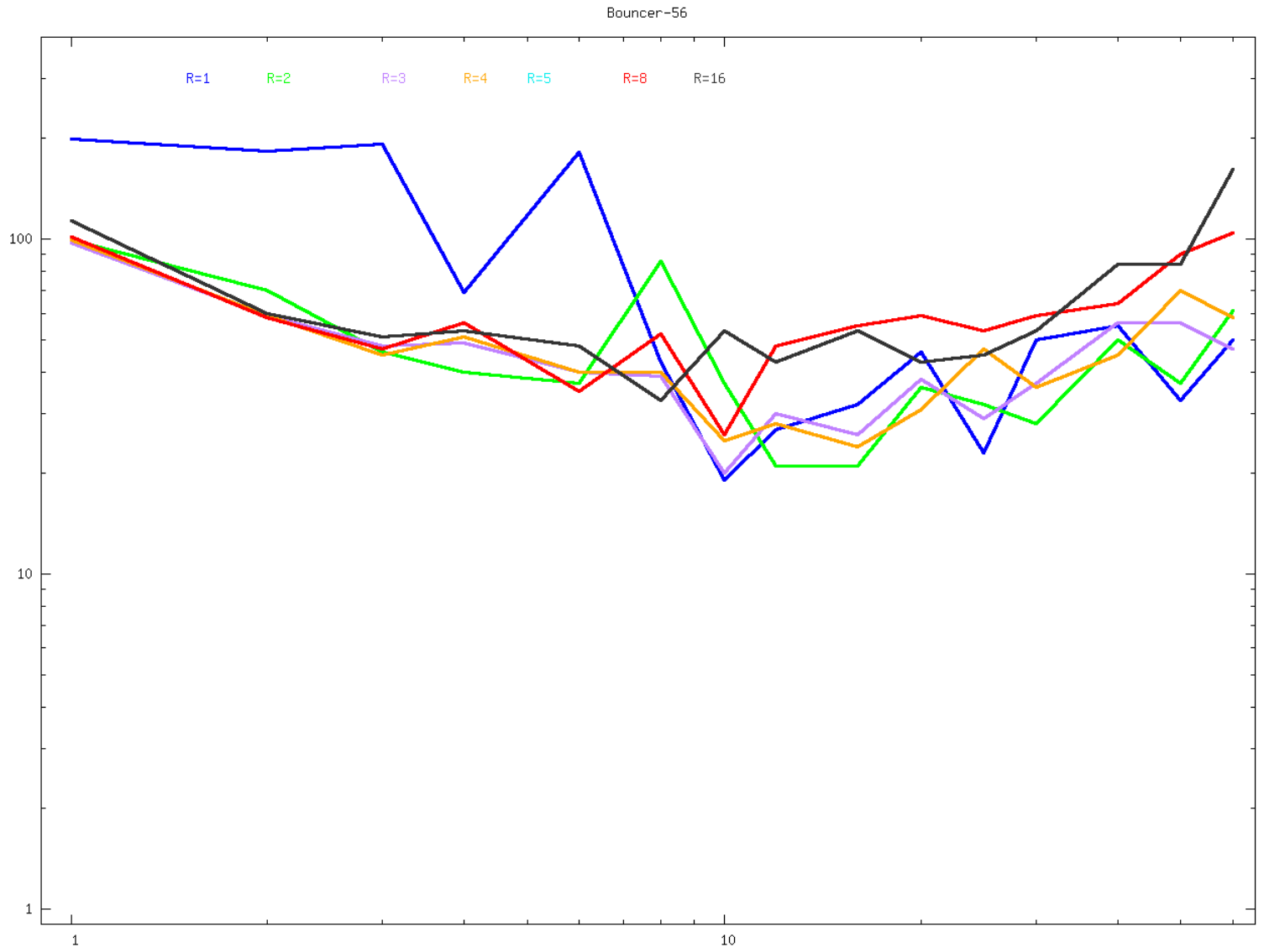
```
val Collector = proc {
  var result: Double = 0·0;
  for (i ← 0 until ntasks) result += (toController?)
  println(result);
}
```



$log\ T$

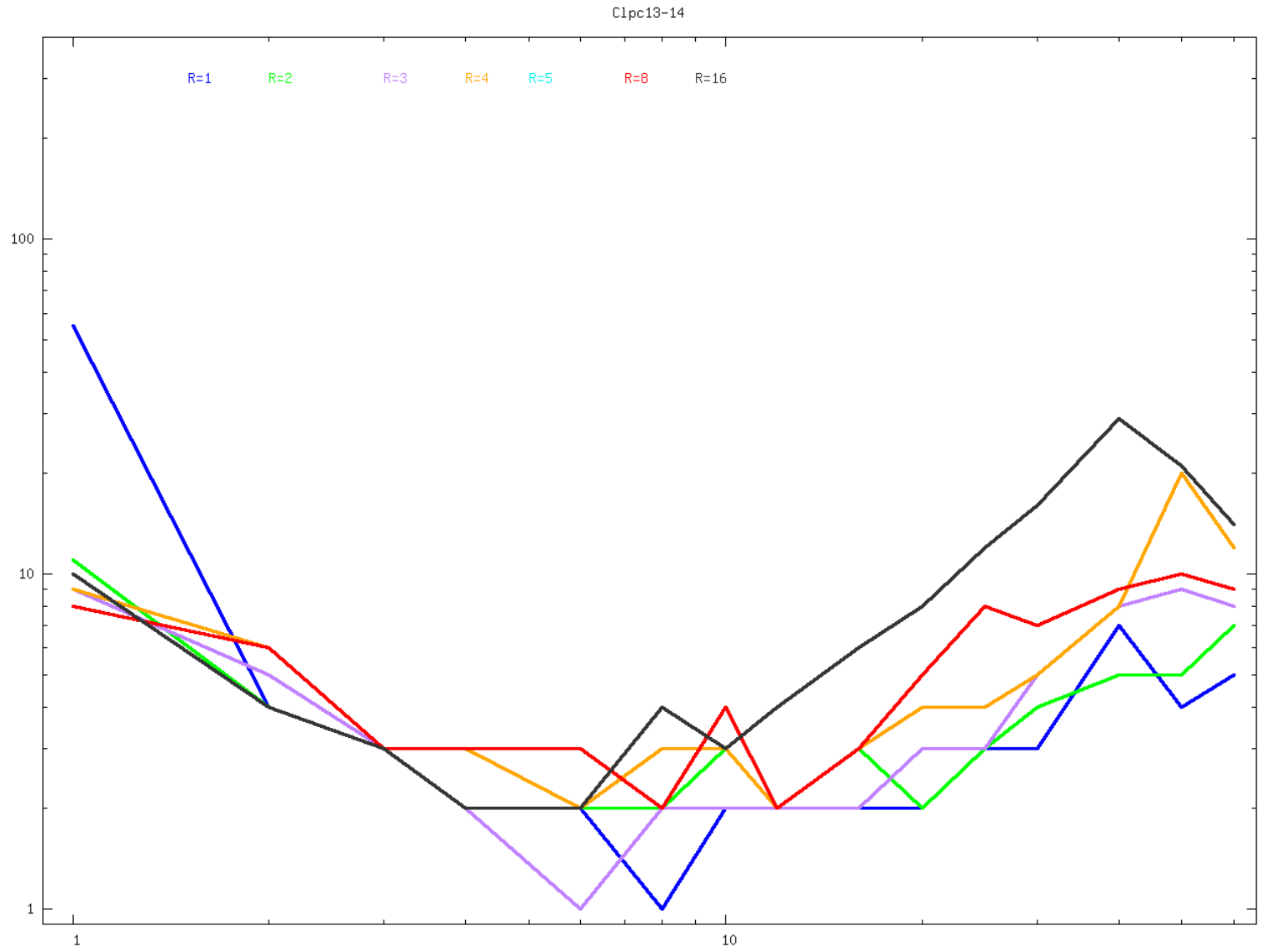
Bouncer

Bouncer: $log\ nWorkers$ [colour-coded R($ntasks/nWorkers$)] ($n = 14.4 \cdot 10^5$)



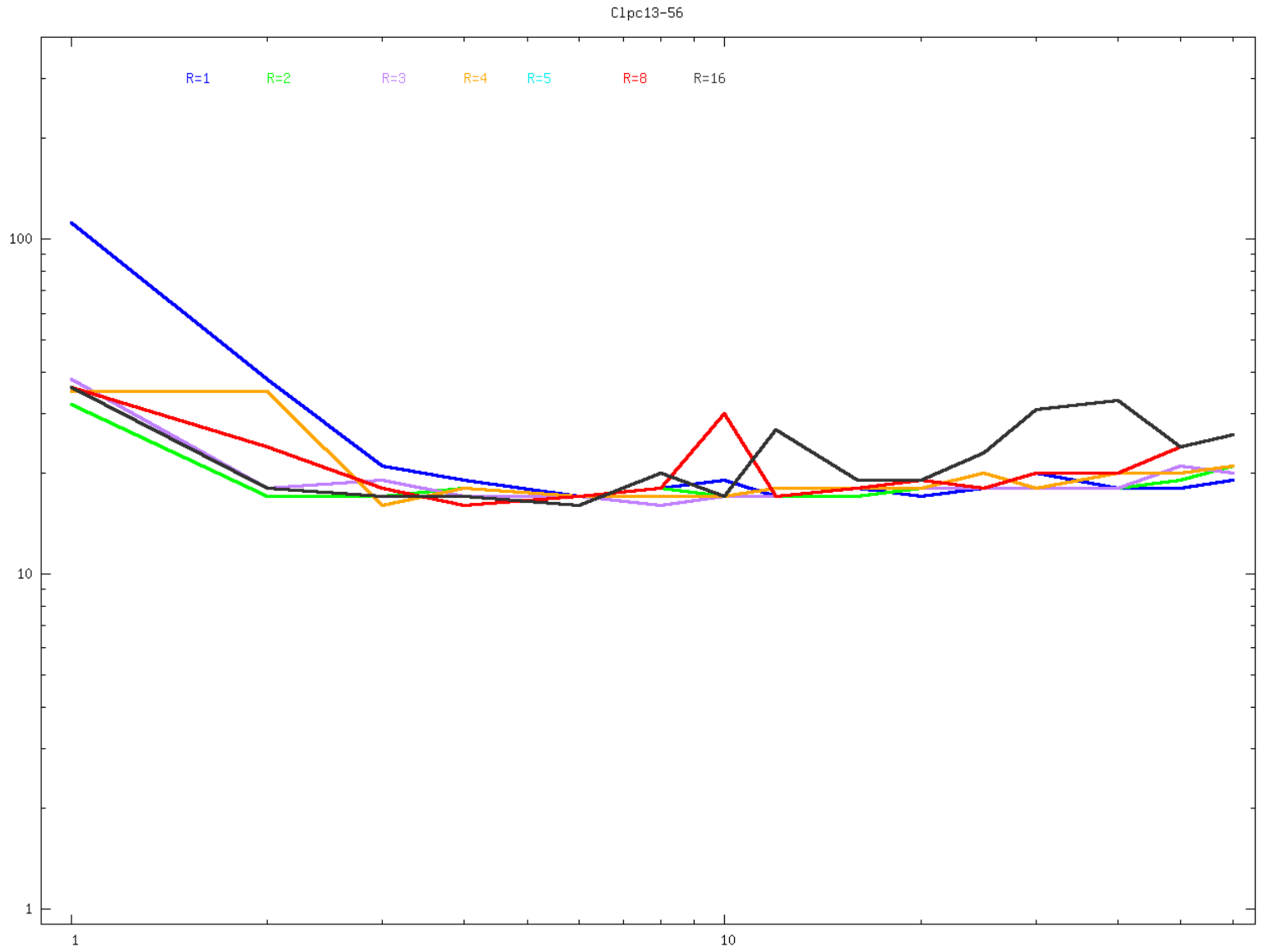
$log\ T$

Bouncer: $log\ nWorkers$ [colour-coded R($ntasks/nWorkers$)] ($n = 4 \cdot 14.4 \cdot 10^5$)



$log\ T$

CLPC13: $log\ nWorkers$ [colour-coded R($ntasks/nWorkers$)] $(n = 14.4 \cdot 10^5)$



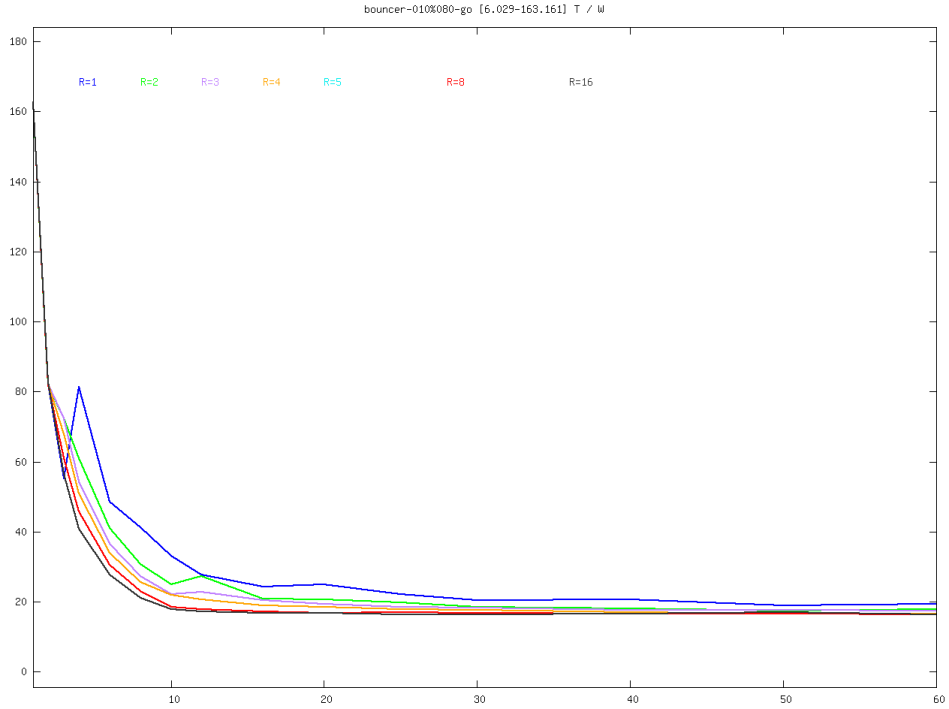
$log\ T$

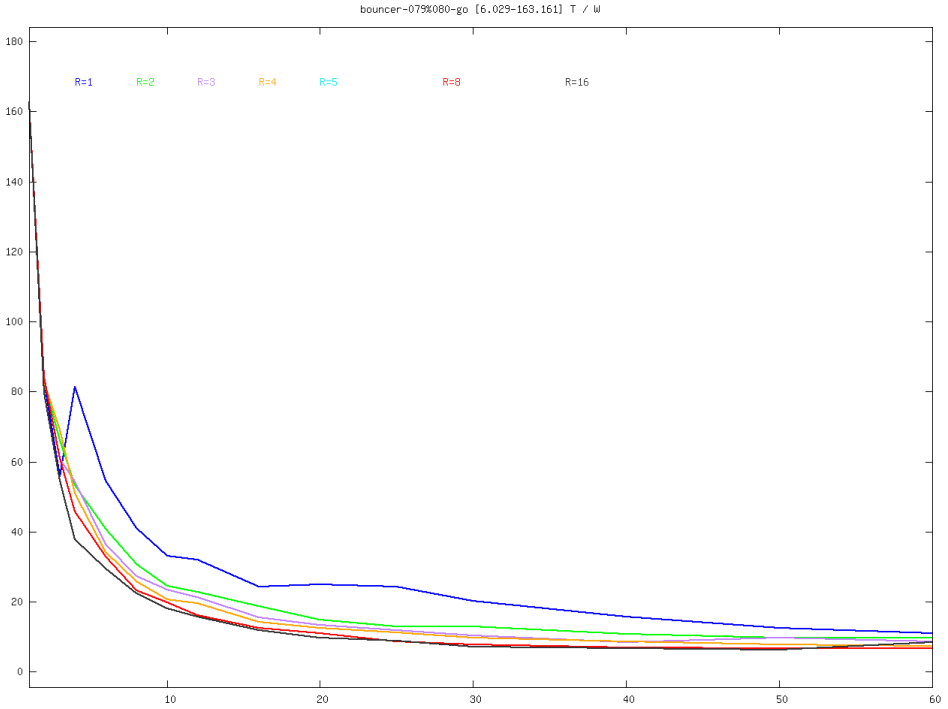
CLPC13: $log\ nWorkers$ [colour-coded R($ntasks/nWorkers$)] $(n = 4 \cdot 14.4 \cdot 10^5)$

Observations

- ⋆ Overall performance worsens on both machines as the task size decreases (n increases).
- ⋆ Overall performance is better on the (8-cpu 1.6Ghz) machine with the simpler memory system than on the (80 cpu, 2.39Ghz) machine.
- ⋆ On the better-performing machine the number of tasks allocated per worker doesn't significantly affect performance after the number of workers reaches about 3.
- ⋆ Other experiments seem to suggest that once the sweet-spot for the number of workers has been reached, increasing the number of tasks allocated per worker adversely affects performance.

Two experiments in Go on Bouncer. Linear axes. Many more tasks.





Bouncer 10 and 79 CPUs Max; n=$288 \cdot 10^5$; $nWorkers \in [1..60]$, $T \in [6..163]$

Tentative conclusions

- ⋆ Communication overhead for the JVM is more significant when task size is smaller because the individual tasks take less time.
- ⋆ The machines with the simpler memory subsystems perform better for the JVM than the other machines, and their performance scales better with increasing $nworkers$. The details of memory system architecture are probably highly significant.
- ⋆ This suggests that a lot of work needs doing on JVM implementations before JVM can be used this way for *computationally intensive tasks.*

- ⋆ Additional experiments demonstrate that Go program performance scales very well as the number of allocated CPUs increases.
- ⋆ Asymptotic shape of the Go graphs seems to demonstrate that Go interprocess communication imposes a *considerably* lower overhead than JVM communication.

Contents

Note 1: 4 ☞
We will explore process-farming in more detail in a later chapter, under the name Bag of Tasks.

Note 2: 8 ☞
We could remove the assumption `n%nworkers==0` by setting

```
val right = Math·min(left+taskRange, b)
```

Note 3: 10 ☞
Remind yourself of the notation for indexed parallel composition.

Note 4: Why this system architecture? 10 ☞
We could have constructed the workers with the appropriate values for `f, l, r, taskSize, delta`, rather than having the controller distribute them; but we wanted to illustrate the pattern in which a single controller distributes tasks to the workers.

Note 5: ASIDE (on CSO notation) 10 ☞
It is important to understand that the body of `System` is not itself constructed as a **`proc`**. Instead its result is the parallel composition of the `PROC`-valued expressions: `Workers` and `Controller(···)`.

`Workers` is itself the parallel composition of individual worker processes, whereas `Controller(···)` is a single process, constructed as a **`proc`**.

Note 6: Performance Evaluation context and observations 13 ☞
Threads are said to be pooled with a timeout of t if when a process terminates its thread is reused to run processes that are started no more than t seconds later.

Context:

- ⋆ Machines were running almost nothing except the benchmark when timings were done.
- ⋆ Times were averaged over 100 trials.
- ⋆ JVM was version 1.6 for all but Recalcitrant and Bouncer.
- ⋆ The standard JVM implementation is poor ("notoriously poor" says Michael Goldsmith) at keeping a thread running on the same processor as it goes through the context switches and rescheduling caused by interprocess communication and contention for shared resources.

  When a thread "migrates" across processors the unavoidable processor cache-flushes and invalidations have an even more pronounced effect on the overall memory subsystem performance than if JVM were better at organising "thread-processor affinity".

- ⋆ Bouncer has a more complex memory subsystem than the other machines, with multiple layers of caching.

Observations and discussion:

- ⋆ Sequential and $nworkers = 1$ performance are comparable for all machines.
- ⋆ Best-performance is not directly correlated to the number of available virtual processors (hardware threads) or to their speed.
- ⋆ As $nworkers$ increases, machines climb to their performance "sweet region", after which their performance declines.

  The decline is very pronounced in Bouncer's case.

  The decline in performance is less marked when threads are pooled.
- ⋆ As $nworkers$ increases beyond the sweet region, it is likely that the costs of the trapezium calculations become dominated by the increasing amount of communication overhead; including contention among workers for the shared ends of the channels `toWorkers` and `toController`. Contention costs are likely to increase faster in Bouncer than in the other machines because of its more complex memory subsystem.

Note 7: 14 ☞
Here we will show the most primitive and unrefined variant of bag-of-tasks. Later we will show a more refined bag-of-tasks architecture in which workers return sub-tasks to the controller.

Note 8: 25 ☞
The differences seem likely to be a consequence of the fact that CSO/JVM associates each logical process with a single kernel thread once it has started whereas the Go runtime system has an internal, user-level, scheduler that multiplexes processes among available kernel threads (which are scheduled across available CPUs by the kernel). A single communication across a CSO channel always causes several context switches between sender-thread, kernel, and receiver-thread whereas a single communication across a GO channel may not cause anywhere near as many.

Note 9: CAVEAT 25 ☞
The experiments reported here were done hastily some years ago. I have reservations about the suitability of this kind of experiment for exploring JVM performance. Conclusions about performance of the Go implementation are likely to remain sound.

A good starting place for reading about the difficulties of "microbenchmarking" on the JVM is https://www.ibm.com/developerworks/library/j-jtp12214/