



Optimal Number of Workers for Parallel Julia

CHRISTOPHER RACKAUCKAS

READ REVIEWS

WRITE A REVIEW

CORRESPONDENCE:
me@chrisrackaukas.com

DATE RECEIVED:
August 18, 2018

DOI:
[10.15200/winn.153459.99003](https://doi.org/10.15200/winn.153459.99003)

ARCHIVED:
August 18, 2018

CITATION:
Christopher Rackaukas,
Optimal Number of Workers for
Parallel Julia, *The Winnower*
5:e153459.99003, 2018, DOI:
[10.15200/winn.153459.99003](https://doi.org/10.15200/winn.153459.99003)

© Rackaukas This article is
distributed under the terms of
the [Creative Commons](#)
[Attribution 4.0 International](#)
[License](#), which permits
unrestricted use, distribution,
and redistribution in any
medium, provided that the
original author and source are
credited.



How many workers do you choose when running a parallel job in Julia? The answer is easy right? The number of physical cores. We always default to that number. For my Core i7 4770K, that means it's 4, not 8 since that would include the hyperthreads. On my FX8350, there are 8 cores, but only 4 floating-point units (FPUs) which do the math, so in mathematical projects, I should use 4, right? I want to demonstrate that it's not that simple.

WHERE THE INTUITION COMES FROM

Most of the time when doing scientific computing you are doing parallel programming without even knowing it. This is because a lot of vectorized operations are "implicitly paralleled", meaning that they are multi-threaded behind the scenes to make everything faster. In other languages like Python, MATLAB, and R, this is also the case. Fire up MATLAB and run

```
A = randn(10000,10000)
```

```
B = randn(10000,10000)
```

```
A.*B
```

and you will see that all of your cores are used. Threads are a recent introduction to Julia, and so in version 0.5 this will also be the case.

Another large place where implicit parallelization comes up is in linear algebra. When one uses a matrix multiplication, it is almost surely calling an underlying program which is an implementation of BLAS. BLAS (Basic Linear Algebra Subroutines) is aptly named just a set of functions for solving linear algebra problems. These are written in either C or Fortran and are heavily optimized. They are well-studied and many smart people have meticulously crafted "perfect code" which minimizes cache misses and all of that other low level stuff, all to make this very common operation run smoothly.

Because BLAS (and LINPACK, Linear Algebra Package, for other linear algebra routines) is so optimized, people say you should always make sure that it knows exactly how many "real" processors it has to work with. So in my case, with a Core i7 with 4 physical cores and 4 from hyperthreading, forget the hyperthreading and thus there are 4. With the FX8350, there are only 4 processors for doing math, so 4 threads. Check to make sure this is best.

WHAT ABOUT FOR YOUR CODE?

Most likely this does not apply to your code. You didn't carefully manage all of your allocations and tell the compiler what needs to be cached etc. You just wrote some beautiful Julia code. So how many workers do you choose?

Let's take my case. I have 4 real cores, do I choose 4? Or do I make 3 workers to allow for 1 to "command" the others freely? Or do I make 7/8 due to hyperthreading?

I decided to test this out on a non-trivial example. I am not going to share all of the code (I am submitting it as part of a manuscript soon), but the basic idea is that it is a high-order adaptive solver for stochastic differential equations. The code sets up the problem and then calls pmap to do a Monte Carlo simulation and solve the equation 1000 times in parallel. The code is mostly math, but there is a slight twist where some values are stored on stacks (very lightweight datastructure). To make sure I could trust the times, I ran the code 1000 times and took the average, min, and max times.

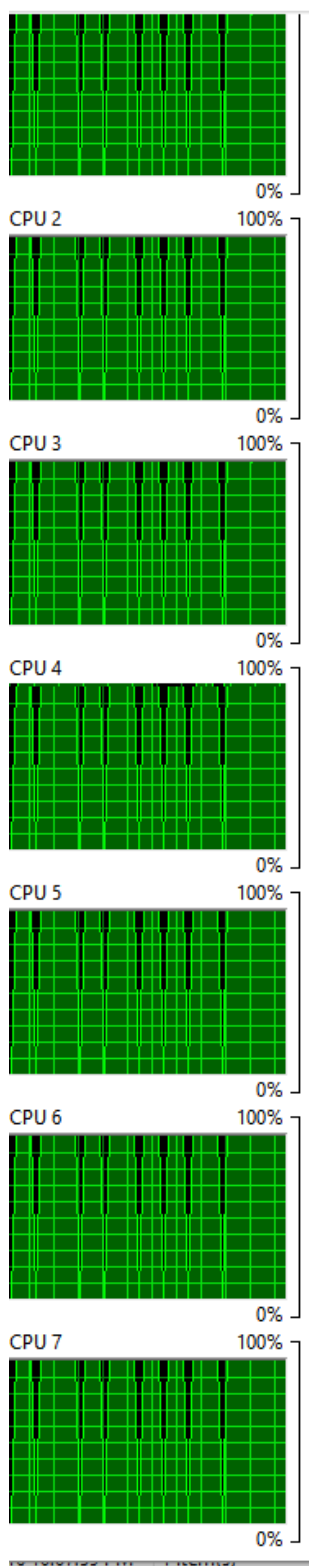
So in this case, what was best? The results speak for themselves.

Average wall times vs Number of Worker Processes, 1000 iterations

Number of Workers	Average Wall Time	Max Wall Time	Min Wall Time
1	62.8732	64.3445	61.4971
3	25.749	26.6989	25.1143
4	22.4782	23.2046	21.8322
7	19.7411	20.2904	19.1305
8	19.0709	20.1682	18.5846
9	18.3677	18.9592	17.6
10	18.1857	18.9801	17.6823
11	18.1267	18.7089	17.5099
12	17.9848	18.5083	17.5529
13	17.8873	18.4358	17.3664
14	17.4543	17.9513	16.9258
15	16.5952	17.2566	16.1435
16	17.5426	18.4232	16.2633
17	16.927	17.5298	16.4492

Note there are two "1000"s here. I ran the Monte Carlo simulation (each solving the SDE 1000 times itself) 1000 times. I plotted the mean, max, and min times it took to solve the simulation. From the plot it's very clear that the minimum exists somewhere around 15. 15!

What's going on? My guess is that this is because of the time that's not spent on the actual mathematics. Sometimes there are things performing logic, checking if statements, allocating new memory as the stacks grow bigger, etc. Although it is a math problem, there is more than just the math in this problem! Thus it seems the scheduler is able to effectively let the processes compete and more fully utilize the CPU by pushing the jobs around. This can only go so far: if you have too many workers, then you start to get cache misses and then the computational time starts to increase. Indeed, at 10 workers I could already see signs of problems in the resource manager.



Overload at 10 Workers as
seen in Window's Resource
Manager

However, allowing one process to start re-allocating memory but causing a cache miss (or whatever it's doing) seems to be a good tradeoff at low levels. Thus for this code the optimal number of workers is far above the number of physical cores.

MORAL OF THE STORY

The moral is, test your code. If your code is REALLY efficient, then sure, making sure you don't mess with your perfect code is optimal. If your code isn't optimal (i.e. it's just some Julia code that is pretty good and you want to parallelize it), try some higher numbers of workers. You may be shocked what happens. In this case, the compute time dropped more than 30% by overloading the number of workers.