Cache is King: Presenting a Suite of Fast, Correlation Function Codes

Manodeep Sinha[a]

[a] *6902 Stevenson Center, Department of Physics & Astronomy, Vanderbilt University, Nashville, TN 37235*

Abstract

We have entered the era of 'Big Data' where we are gathering a tremendous amount of data that needs to be processed as well as modeled accurately. In particular, large galaxy surveys like the Sloan Digitial Sky Survey requires computing correlation functions. While measuring the correlation function in the data is not a bottle-neck, modeling the observed galaxy distribution correctly requires a MCMC chain and repeated measurements of $\xi(r)$ and/or $\xi(r_p, \pi)$. However, the architecture of modern CPU's means that best performance is only obtained when cache misses are kept to a minimum. Here, I present a suite of 3 correlation function codes that exploit current CPU architecture and hand-written AVX

The accompanying codes are publicly available at `https://bitbucket.org/manodeep/Corrfunc/`.

Keywords: methods: numerical, (cosmology:) large-scale structure of universe,

1. INTRODUCTION

The large-scale structure of the Universe can be now measured using $\mathcal{O}$(million) galaxies using data from current surveys like SDSS/BOSS surveys. Upcoming surveys like LSST will probe even deeper and wider and aim to target on $\mathcal{O}$(10's of million) galaxies. With such a large galaxy data, we can measure the galaxy density field fairly accurately in the data. While the number of galaxies observed in these current and upcoming surveys is

Email address: `manodeep.sinha@vanderbilt.edu` (Manodeep Sinha)

large, we have to compute the galaxy density fields *only once*. Thus, even a slow, correlation function code will be fine when measuring the data correlation function. However, predicting this galaxy density field data will require an even larger number of model galaxies – increasing the computational load of determining the density field even further. Typically, the modeling process also involves an MCMC – requiring $\sim$ millions of evaluations of the correlation function.[1]

Modern cpu's have a hierarchy of memory locations; the smallest and fastest are closest to the cores while the largest and slowest are the farthest. All cpu instructions need to be carried out from cpu registers - there are $\sim 10-30$ registers typically available and the access times can be thought of as instantaneous. Next up is the $L1$ cache divided into $L1D$ for data and $L1I$ for instructions cache. Since the cpu always necessarily executes instructions that are close together, we will ignore the instruction cache from now on. Typical $L1$ cache sizes range from 64KB to 128 KB. Next level up is the $L2$ cache, typically $\sim$ 256KB to 1 MB. The last level cache or the $L3$ cache is usually shared across all cores on the socket and can be 10 MB to 40 MB.

2. Methods

We need to compute pairwise distances to get the correlation function. A naive implementation of a correlation function would compute *all possible* pairwise separations with a complexity $\mathcal{O}(N^2)$. However, for almost all correlation functions, we are only interested in separations less than a certain r_{max}, where r_{max} is much smaller than the domain of the point distribution itself. We can then immediately see a way to prune pairs that can not *possibly* be within r_{max}. If we impose a 3-d grid, with cell-size r_{max}, then two points separated by more than one cell size (r_{max}) in any one dimension can not be within r_{max} of each other. Thus, given one point which is the target galaxy and a grid with cell-size r_{max}, immediately allows us to prune *all* of the points that are not within 1 cell offset in each dimension. However, even with this pruning, the actual implementation of the algorithm matters. For instance, a linked-list in each cell performs $\sim 60\times$ worse than the algorithm

[1] I will assume that creating the models themselves is much faster compared to computing the correlation functions.

described here.[2]

2.1. Partitioning the Particles based on r_{max}

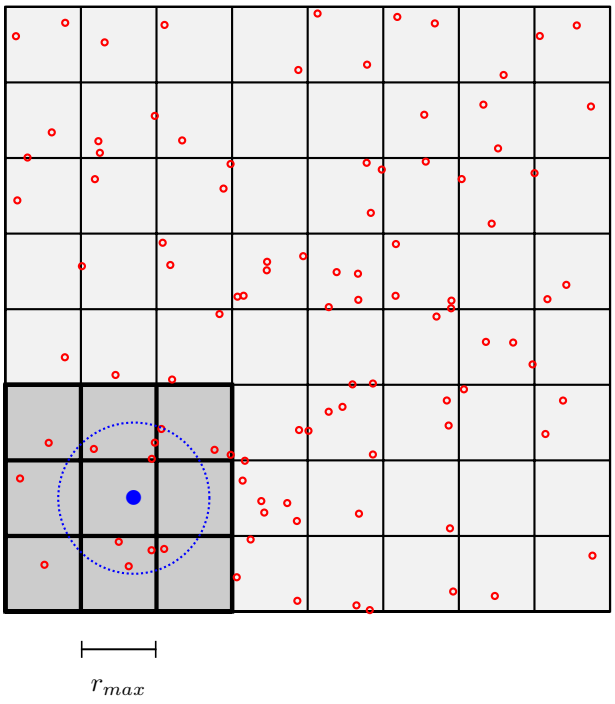

Figure 1: A 2-D grid showing the bin-lattice partitioning scheme. The bigger square show the entire domain, the red circles show a random distribution of 100 particles. Let's say we want to compute all pairs for the target blue point, then we would only have to consider red points that are within one cell (the dark shaded region). A circle with radius r_{max} is also drawn to shown the actual pairs that will eventually count in the correlation function.

2.2. How to Maintain Cache Locality within the Grid

For all pairs around a given target galaxy, we need to compute distances to all points within all neighbouring 3-d cells. We ensure that the particle locations are contiguous by moving them into the following `C struct` in the order in which they arrive.

[2]The usual linked-list is very cache-unfriendly. Each dereference requires a read from a new region of memory and an almost guaranteed cache miss.

```
1 typedef struct{
2   DOUBLE *x;
3   DOUBLE *y;
4   DOUBLE *z;
5   int64_t nelements;
6 } cellarray;
```

Since the typical particle data is small ($\sim$ 20-30 MB), duplicating the entire particle distribution does not impose a strong constraint on the memory requirements. However, this duplication allows us to store the particles contiguously and produces fewer cache misses while looping over particles in a cell. The entire 3-D particle distribution is then deposited on the uniform grid. Now, for each particle we need to visit 27 total cells to compute all possible pairs within r_{max}.

3. The pair-counting algorithm

3.1. $\xi(r)$

3.2. $\xi(r_p, \pi)$

3.3. $w_p(r_p)$

3.4. Hand-written Vectorization Support

Advanced Vector Extensions (AVX) has been available in cpu's more recent than 2011. AVX allows the processing of 8 floats or 4 double simultaneously – thus, potentially increasing the throughput by a factor of 8x/4x. However, automatic vectorization is not always possible by the compiler and in those cases we can write AVX vector intrinsics to directly manipulate 8 floats/4 doubles.

4. Benchmarks & Scaling

In this section we present the runtimes and OpenMP scaling for the different codes.

4.1. $\xi(r)$

4.2. $\xi(r_p, \pi)$

4.3. $w_p(r_p)$

5. Conclusions

Foreman-Mackey et al. (2013)

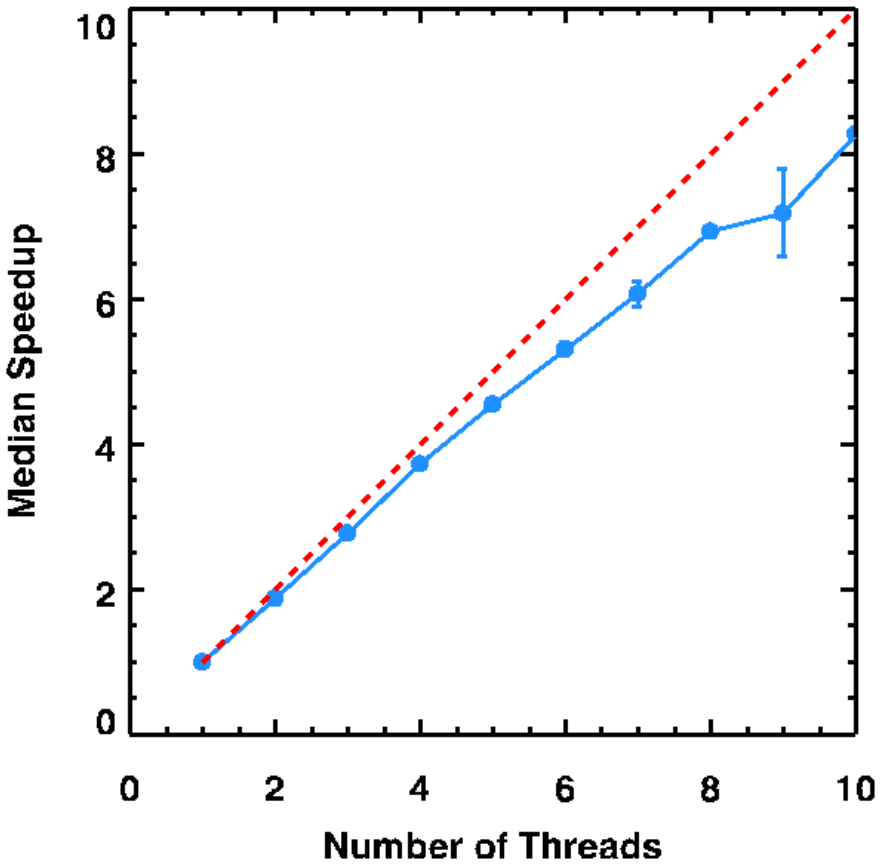

Figure 2: OpenMP scaling for the $\xi(r)$

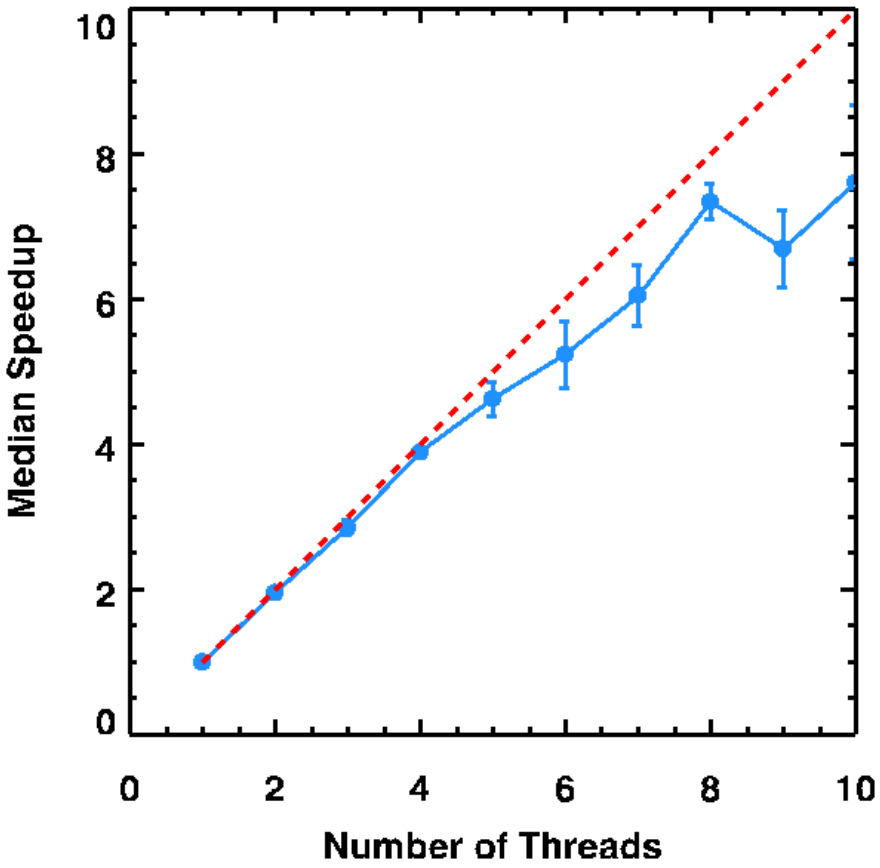

Figure 3: OpenMP scaling for the $\xi(r_p, \pi)$

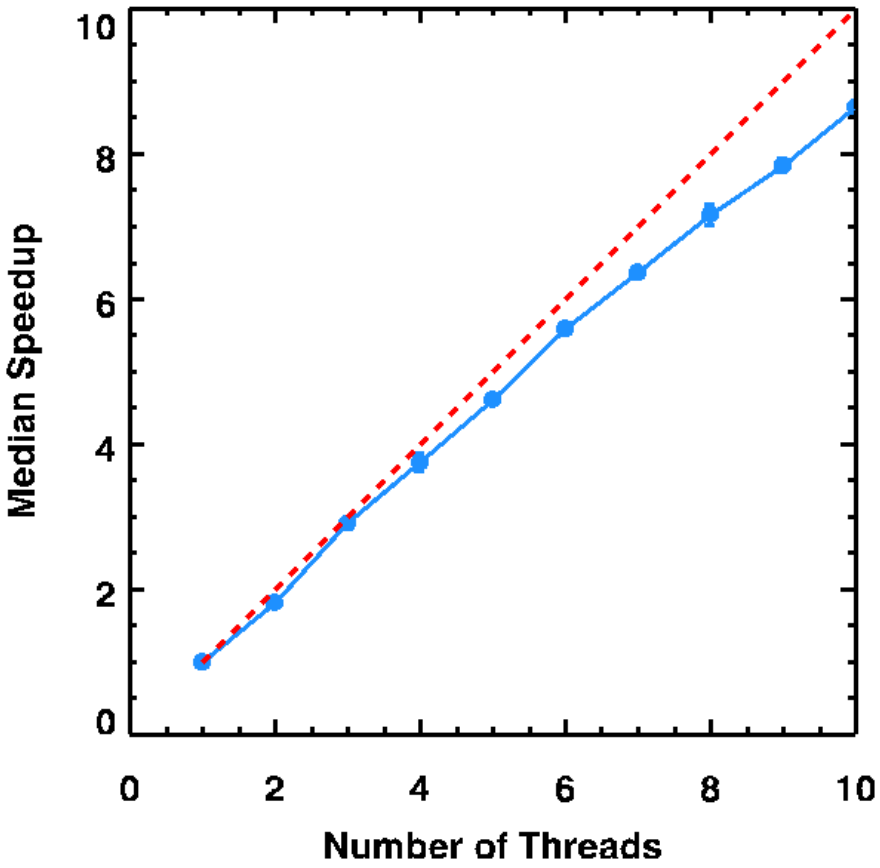

Figure 4: OpenMP scaling for the $w_p(r_p)$

Acknowledgements

Foreman-Mackey, D., Hogg, D. W., Lang, D., Goodman, J., Mar. 2013. emcee: The MCMC Hammer. PASP125, 306–312.