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

In the era of 'Big Data', we need special computing techniques to both process the raw data as well generate and evaluate models to test our theories. In particular, large galaxy surveys like the Sloan Digitial Sky Survey requires computing a variety of n-point statistics, e.g., the 2-pt projected correlation function, $w_p(r_p)$. 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 $w_p(r_p)$. To efficiently measure the correlation functions, we need to write code that accounts for cache hierarchies while simultaneously exploits the wide vector registers present in the relatively modern CPUs. Here, I present a suite of OpenMP parallelized, clustering codes that exploit current CPU micro-architecture with hand-written AVX intrinsics. Given a 3-D distribution of points in a cartesian system, these codes compute spatial clustering statistics 3-D correlation functions $DD(r)$, $\xi(r)$, $\xi(r_p, \pi)$; 2-point projected correlation function $w_p(r_p)$; counts-in-spheres $pN(r)$. For a 3-D particle distribution on the sky, the codes can compute projected correlation function $DD(r_p, \pi)$, angular correlation function $DD(\theta)$ as well as $pN(r)$. These codes are designed to be blazing fast and can compute $w_p(r_p)$ for $\mathcal{O}$(1 million) galaxies in ~ 6 seconds on a post-2011 CPU, which is a factor of few faster than the existing public correlation function routines. The accompanying codes are publicly available at `https://github.com/manodeep/Corrfunc/`.

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

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

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 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 physically located close to the computing cores while the largest (and slowest) are the farthest. All cpu instructions need to be carried out from cpu registers - there are $\sim$ 100 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 (shared between instruction and data). 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

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

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 described here.[2]

2.1. Partitioning the Particles based on r_{max}

Fig. 1 shows a schematic 2-D grid that corresponds to our partitioning scheme.

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.

```
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}.

[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.

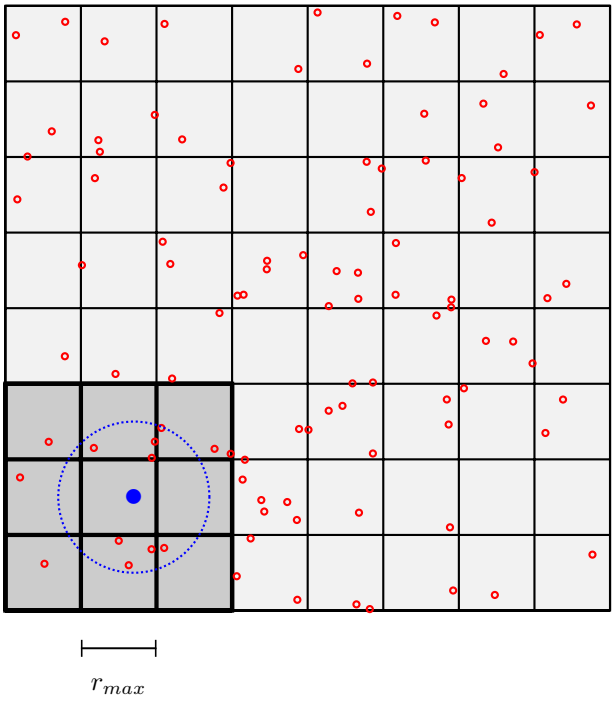

Figure 1: A 2-D grid showing the bin-lattice partitioning scheme. The bigger square show the entire domain, the red circles show a random distribution of 100 particles. Let's say we want to compute all pairs for the target blue point, then we would only have to consider red points that are within one 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.

3. The pair-counting algorithm

3.1. $DD(r)$

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

3.3. $w_p(r_p)$

3.4.

3.5. 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[3].

4. Benchmarks & Scaling

In this section we present the runtimes and scalings for different number of particles, r_{max} and OpenMP threads for the codes. For all of the scaling tests, only an auto-correlation calculation was used and the fiducial catalog contains ~ 1.2 million galaxies on a periodic cube of side 420 $h^{-1}Mpc$.

4.1. Scaling with Number of Particles

In Fig. 2, we show the scaling for the three codes with the number of particles. For this scaling, we subsampled the fiducial mock to attain 10 logarithmic steps in particle number ranging from 1.2×10^4 to 1.2×10^6.

4.2. Scaling with r_{max}

4.3. Scaling with OpenMP threads

5. Conclusions

I have presented a suite of three fast correlation function codes that take advantage of the underlying hardware. The reasons why the codes are much faster for typical workloads are:

[3]Another option would be to use the vectorclass written by Agner Fog here: `http://agner.org/optimize/vectorclass/`

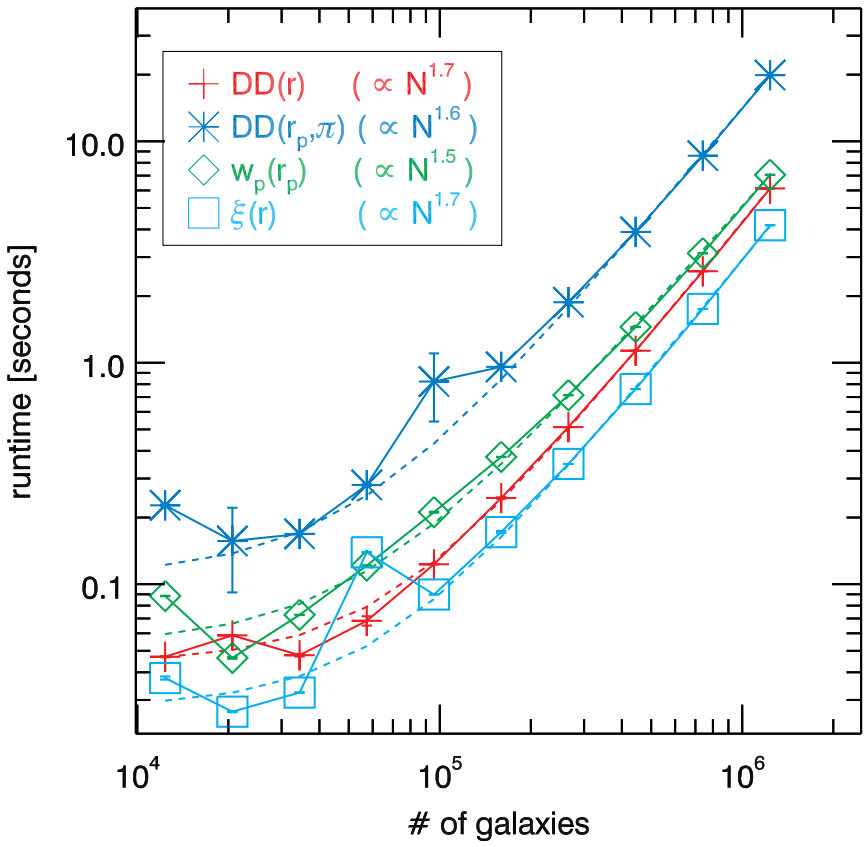

Figure 2: Scaling with particle number for $DD(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$ and $\xi(r)$

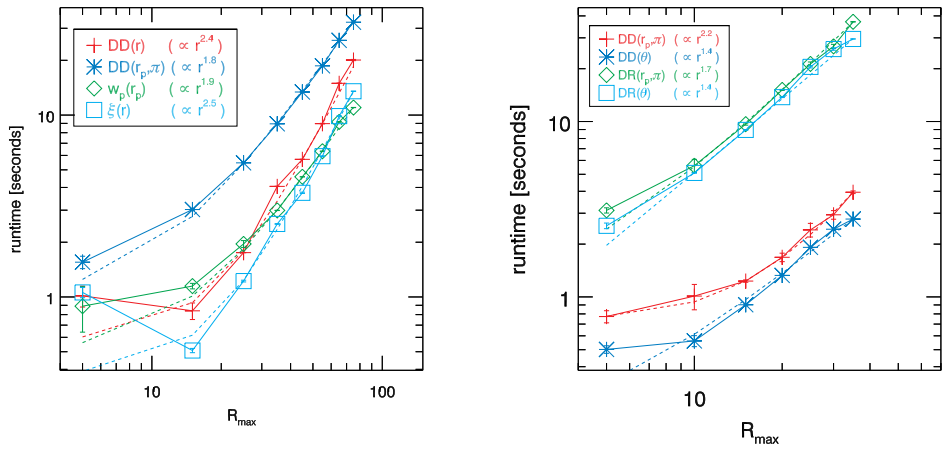

Figure 3: Scaling with r_{max} for $DD(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$ and $\xi(r)$

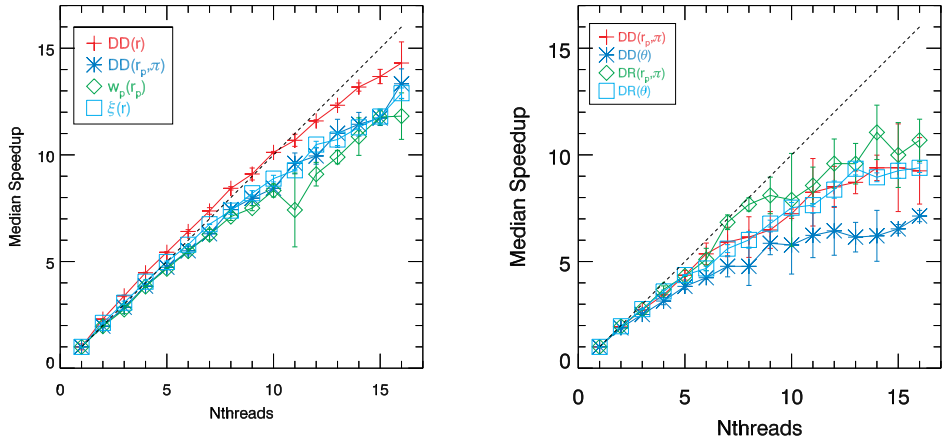

Figure 4: OpenMP scaling for $DD(r)$, $\xi(r_p, \pi)$, $w_p(r_p)$ and $\xi(r)$

Table 1: OpenMP scaling for the three codes. Efficiencies are defined as `Speedup/Nthreads`. Note, that $DD(r)$ has super-linear scaling with `nthreads` up to ~ 10 threads.

Nthreads	**Efficiency**[%]							
	Periodic Boxes				**Spherical Geometry**			
	$\boldsymbol{DD(r)}$	$\boldsymbol{\xi(r_p,\pi)}$	$\boldsymbol{w_p(r_p)}$	$\boldsymbol{\xi(r)}$	$\boldsymbol{DD(r_p,\pi)}$	$\boldsymbol{DD(\theta)}$	$\boldsymbol{DR(r_p,\pi)}$	$\boldsymbol{DR(\theta)}$
1	100	100	100	100	100	100	100	100
2	115	98	98	106	96	94	97	98
3	113	95	91	102	91	83	91	92
4	111	96	95	101	85	78	92	89
5	108	94	93	99	87	76	86	87
6	106	91	91	96	89	70	84	77
7	105	90	89	96	84	68	97	80
8	105	92	88	92	76	59	95	75
9	101	88	83	91	72	65	89	75
10	101	84	83	88	72	57	78	75
11	97	87	67	84	74	56	77	69
12	96	82	75	87	70	53	79	69
13	94	84	76	82	67	47	73	71
14	94	81	77	80	67	44	78	63
15	91	78	78	78	62	43	66	61
16	89	83	73	80	57	44	66	58

- A 'bin-lattice' scheme is used to first partition the computational domain into 3-D cells that can then be used to prune majority of the volume.
- To obtain better cache-locality, the particle list is duplicated into a contiguous array for each dimension. Thus, all particles that fall into the same 3-D cell are stored in a contiguous array.
- With the AVX instruction set, modern CPU's can process 8 floats/4 doubles simultaneously. The codes contain hand-written AVX intrinsics that offer a factor of few speedup compared to the compiler generated scalar code.

Foreman-Mackey et al. (2013)

Acknowledgements

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