Mapmaking

SciNet Parallel Scientific Computing Course

Aug 31 - Sept 4, 2009

First, Why?

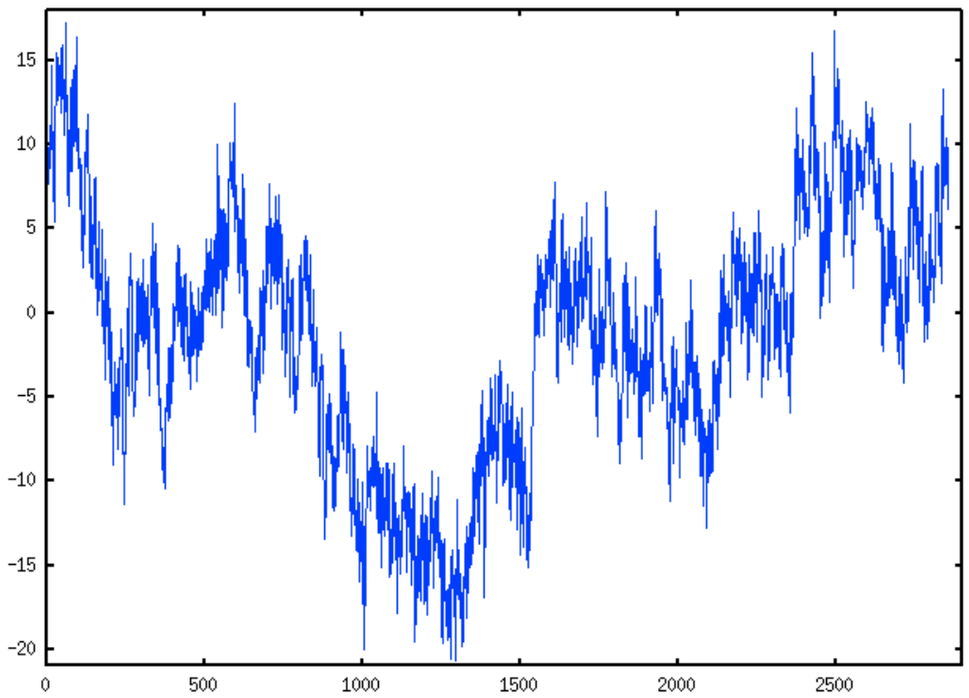
- To look at CMB/mm-wave sky, typically use bolometers. Getting more & more detectors.
- Cameras scan back & forth across sky, so need to convert scans to maps.
- Noise properties of most any total-power detector complicated. 1/f noise big. So, best map made from all data.

Touches on Lots of Topics

- Optimal map is a linear least-squares solution. Can't solve directly, so we will meet iterative matrix solutions.
- Noise is defined in Fourier space, so will meet FFTW.
- Will meet thread safety using other people's routines. Something you will have to be aware of.
- Will see new OpenMP (un)synchronization commands.

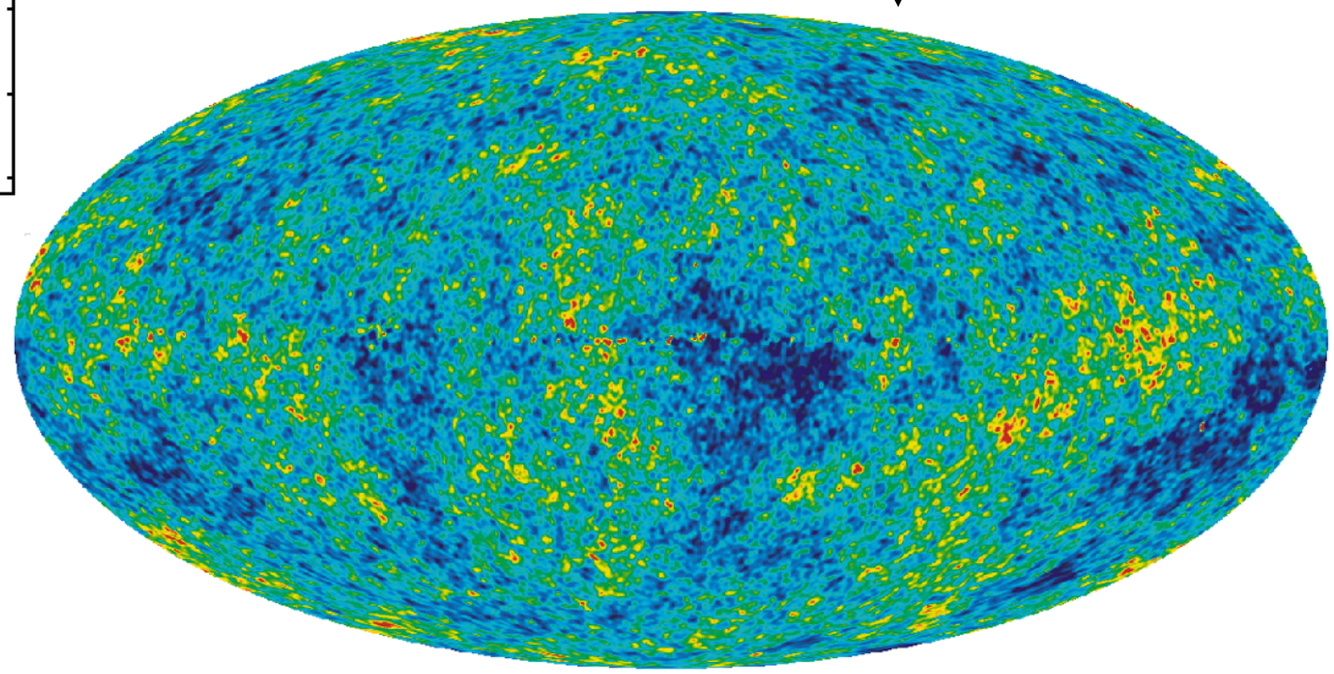
Goal:

Simulated 1/f timestream



Turn this.

Into this.

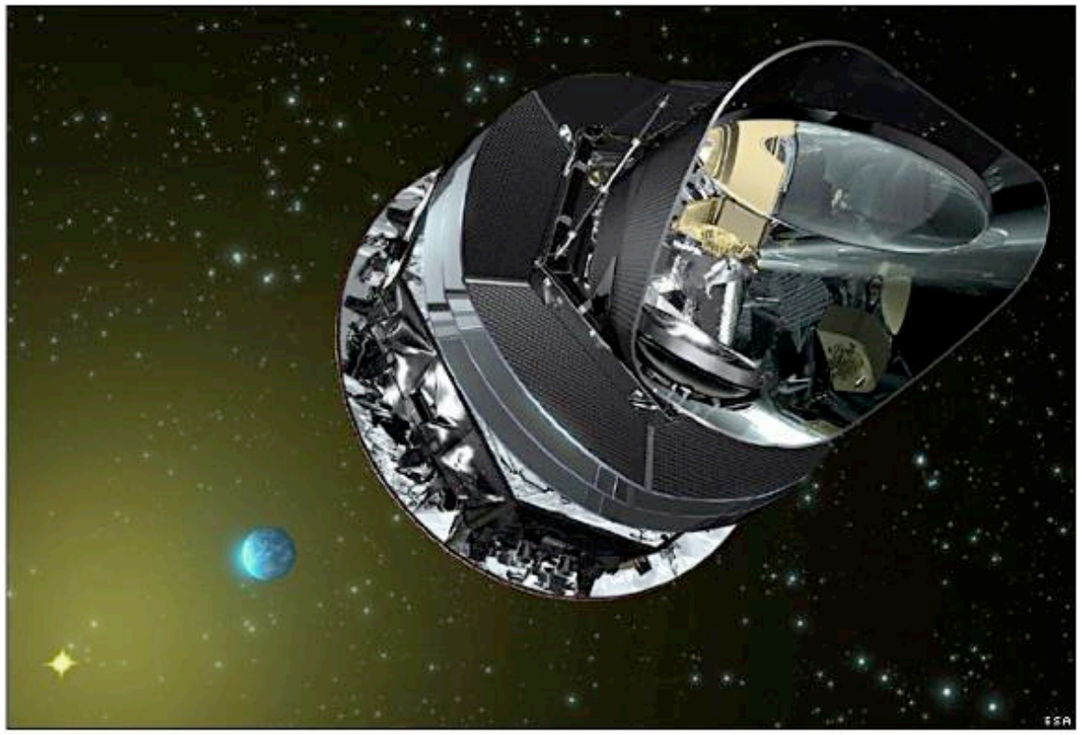


WMAP 5-year.

And do it Quickly

- WMAP has .15 TB, takes ~week on 50 cores.
- Modern experiments will have to deal with 10-100 TB of data.
- One drive, reading 50 MB/s takes *23 days* to read 100 TB.
- If nodes have 8 GB, holding 100 TB in memory takes 12,500 nodes.

Lest You Think I'm Joking...



Planck - 10^{10} pts/detector/year
70 detectors, ~3 years,
single-precision = 8 TB
Launched already.

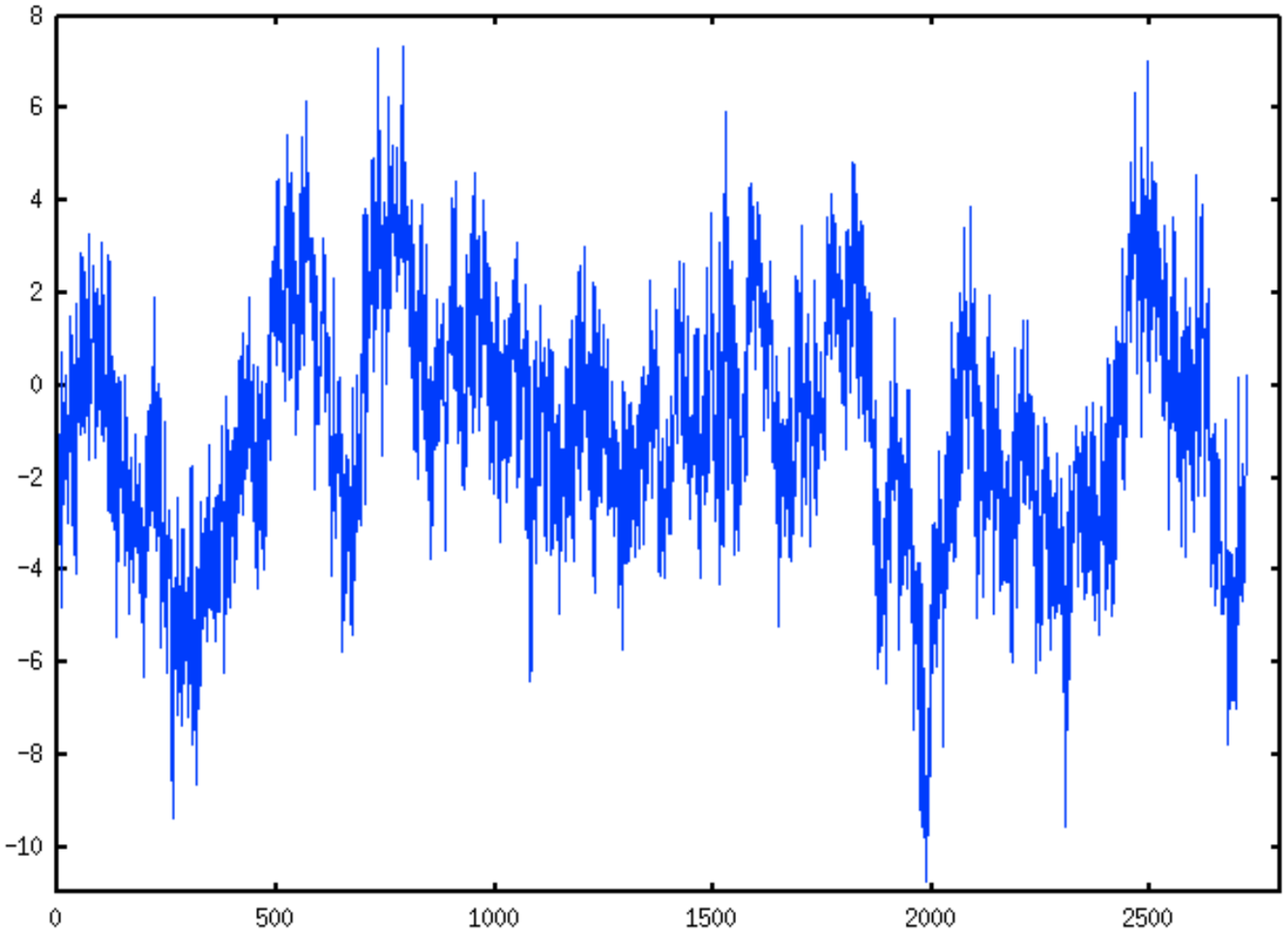
ACT - 3000 detectors, 400Hz readout, 12 hrs/night, up to 200 GB/day. 2 years = 150 TB.
Have > 1 year already
SPT, SCUBA2, others...

1/f? Why do I hate it?

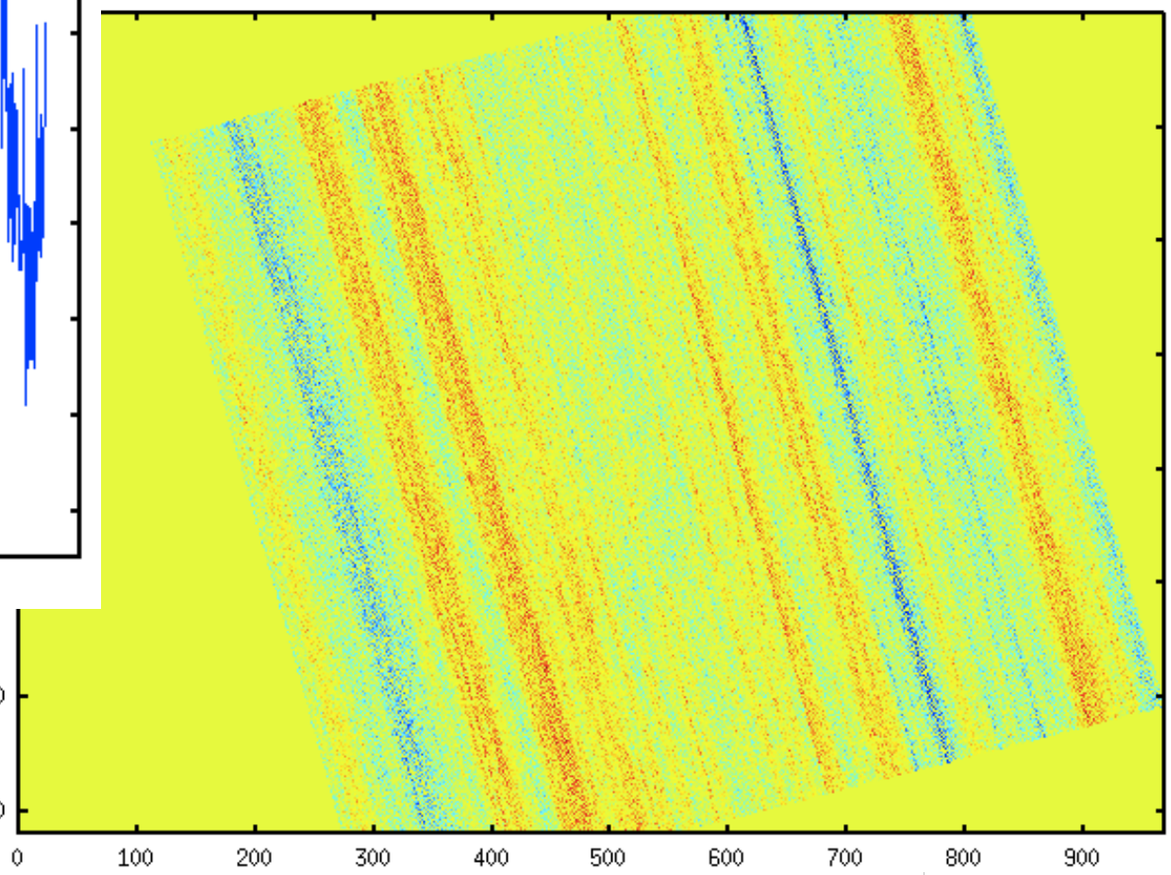
- For most total-power detectors, detector noise is larger on long timescales.
- I could make a map where ith pixel was the average of all data that fell in it.
- Lots of noise in common in data taken at similar times.
- Usually scan back & forth on sky - low-frequency noise makes stripes in the maps. Not good for science...

Stripey Map

Typical timestream

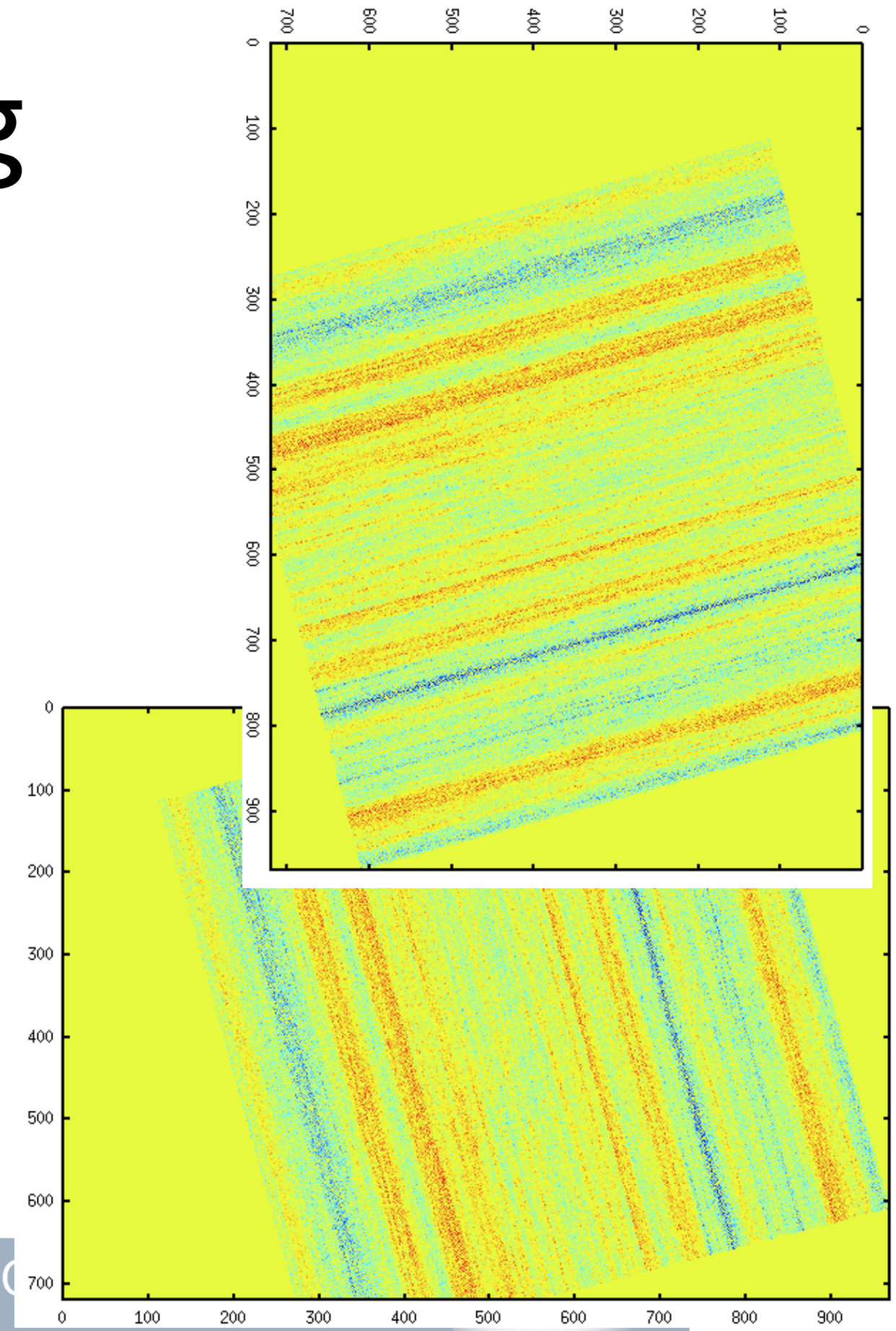


Resulting map



Cross-linking

But say I had data taken in a different direction. Ought to be able use information in one direction from one, the other from the other, right? Yep. But how?



And now for some math...

- Optimal map minimizes χ^2 of the data.
- Because the data is correlated, χ^2 requires work.

- Want to use data from different directions to beat down striping.
- How do we do this? Linear least-squares.

Fundamental Equation

(We really, really hope)

$d=Pm+n$

d=data.

m=map.

n=noise

P=Pointing matrix. It turns sky into expected data.

For a given map, I think noise=d-Pm. If N is my expected noise matrix, then $\chi^2 = n^T N^{-1} n$ or $(d-Pm)^T N^{-1} (d-Pm)$

Mapmaking Equation

- At χ^2 minimum, $\partial\chi^2/\partial m=0$
- Matrix algebra gives $P^T N^{-1}(Pm-d)=0$
- Or, $P^T N^{-1} Pm = P^T N^{-1} d$. Need to solve this.
- You have seen this before: simply linear least squares for parameters *m* given data *d*.

How Big is This?

- For Planck, maps are ~10^8 pixels.
- N^{-1} is size of data. 10^{12}-10^{13} on a side.
- P tells you which data lives in which pixel. It is n_{data} x n_{pixel}. Ideally, one non-zero element per column (since each data point lives in one pixel).

How Long?

- We often can pull tricks to get fast multiplies by P, N^{-1}. Hopefully, at least.
- We have to invert $P^TN^{-1}P$, because $m=(PTN^{-1}P)^{-1}\ P^TN^{-1}d$ (now, this assumes all other matrix operations are free!).
- So, down to inverting 10^8x10^8 matrix. 1 sec. for 10^3x10^3. So, 10^{15} core-seconds. That's 10^4 cores for 3000 years. If we'd started during Trojan war, we'd be done!

So, What Next?

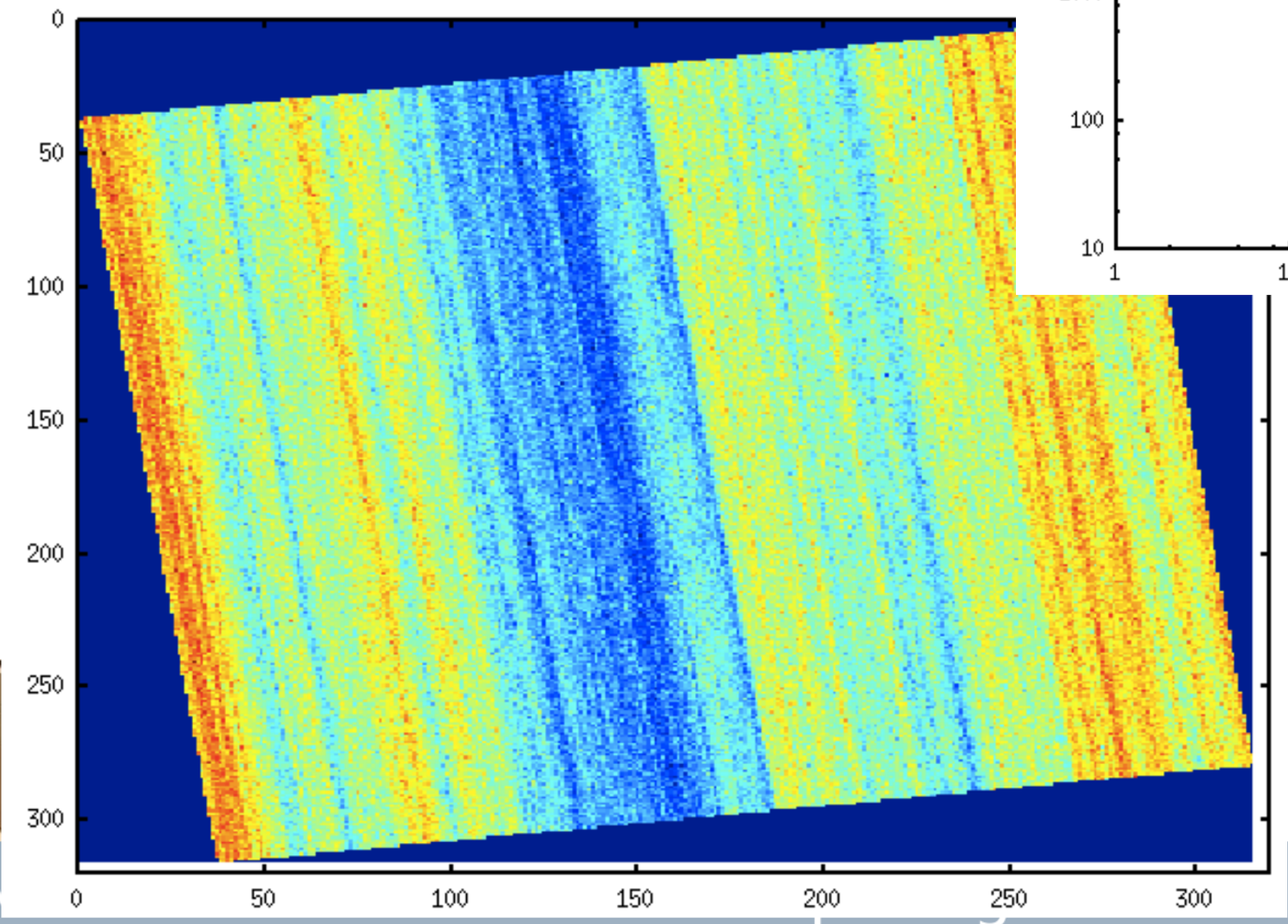
- If someone gave us the right answer, would we know? Let's look at terms.
- RHS: $N^{-1}d$ means I've divided the data by the noise (squared). If I knew how to divide data by its noise, I might be in business.
- $P^T(N^{-1}d)$ sums (noise-filtered) data into a map (this is the operation I used to make the striped map). Takes order(#data) FLOPS. Doable.

So, What Next? Contd.

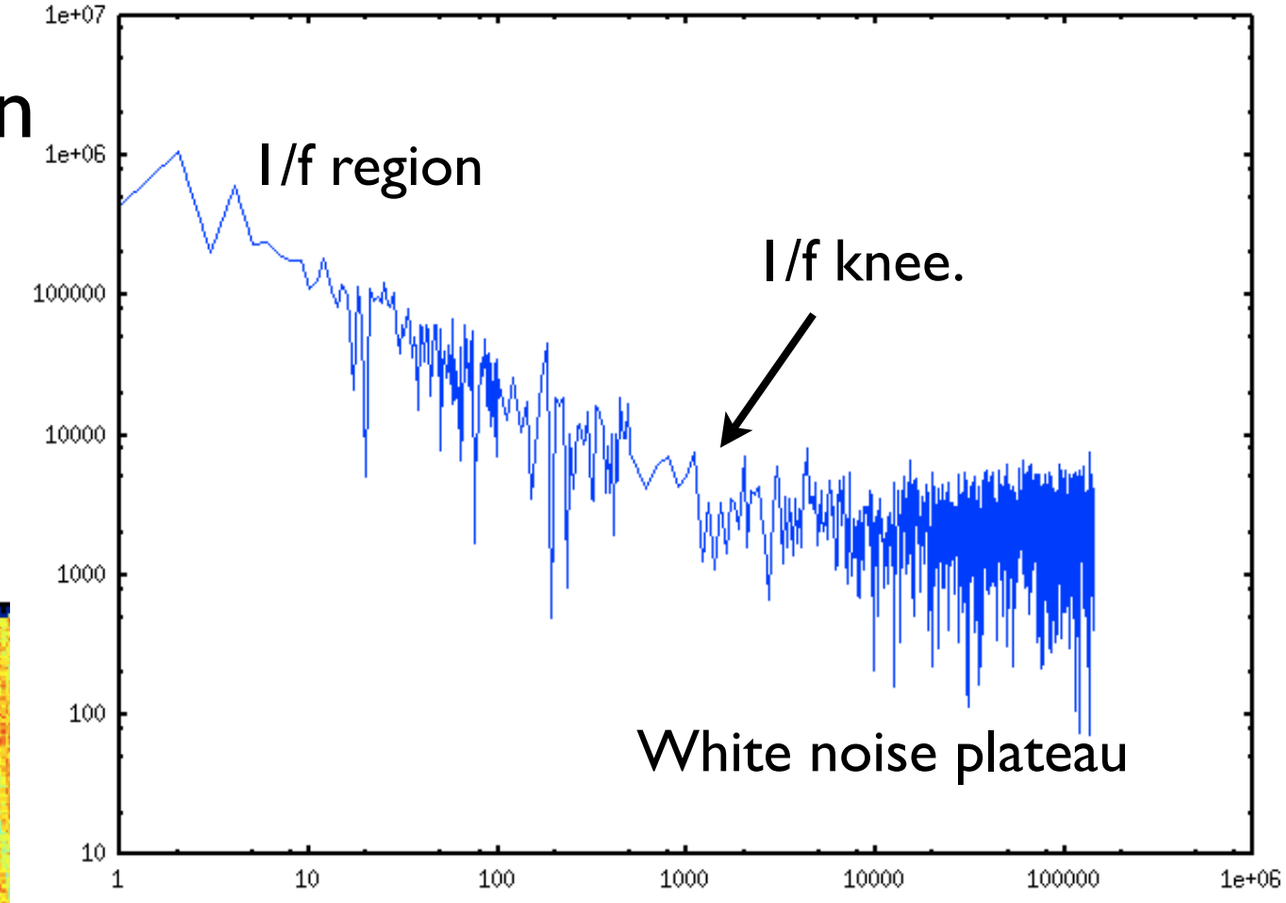
- LHS: Pm says make the data I should have seen if this were the true map. Again, about as many FLOPS as I have data.
- $P^T N^{-1}(Pm)$ is same work as $P^T N^{-1} d$, so if I can do this N^{-1} thing, I stand a chance.
- So, how do I divide the data by the noise? Find a space in which the noise is simple.

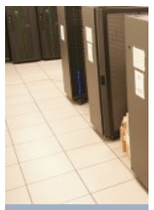
1/f giveth, 1/f taketh away.

Fourier modes are uncorrelated in 1/f. Not bad in real life. So, if I divide by noise (squared) in Fourier space, I can make $N^{-1}d$.



FT of data.





Calculating $N^{-1}d$

- Fourier transform data.
- Fit 1/f+constant (or more complicated, if need be) to data FT.
- Now whenever I want to have N^{-1} on a vector, I just use that Fourier model for N^{-1}.

- How much work? FFT goes like *n*log(*n*). If data is broken into hour chunks, have ~10^6 elements, log(*n*)~20. For 10^{13} elements, need $2*10^{14}$ FLOP. Good core can do 10^{10} FLOPS (4 FLOP/clock@2.5GHz). So, 100 cores can *test* guessed answer in ~3 min.

But is My First Guess Right?

- No. But it doesn't have to be.
- There's a whole literature on solving Ax=b. People have been doing this at least since Gauss.
- Given current guess, we can make a guess as to a better map. Keep doing this until we're happy.
- Cookbook scheme: a method called conjugate gradient tells us how to take our next step.

Conjugate Gradient

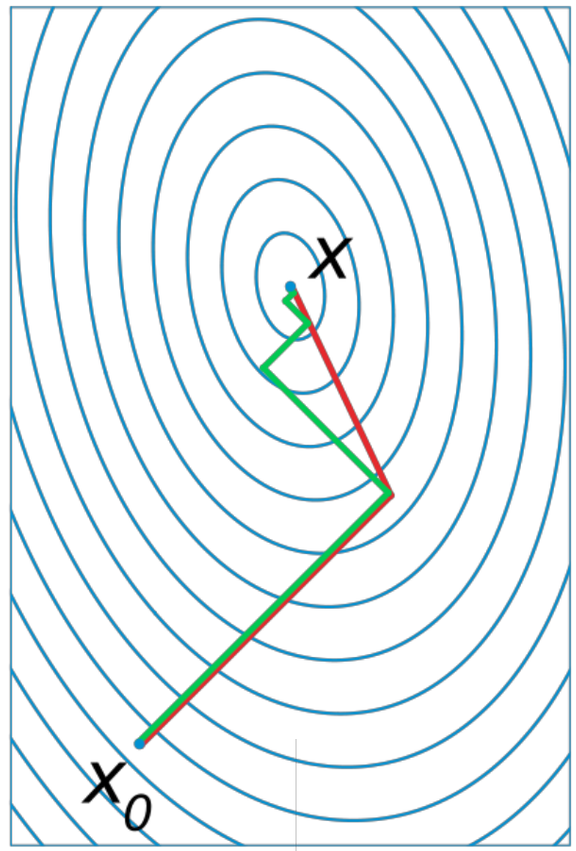
From Wikipedia:

$r_0 := b - Ax_0$
$p_0 := r_0$
$k := 0$
repeat

$$\alpha_k := \frac{r_k^\top r_k}{p_k^\top A p_k}$$
$$x_{k+1} := x_k + \alpha_k p_k$$
$$r_{k+1} := r_k - \alpha_k A p_k$$
if r_{k+1} is "sufficiently small" **then** exit loop **end if**
$$\beta_k := \frac{r_{k+1}^\top r_{k+1}}{r_k^\top r_k}$$
$$p_{k+1} := r_{k+1} + \beta_k p_k$$
$$k := k + 1$$

end repeat
The result is x_{k+1}

Here's the cookbook recipe. At each step, I have to multiply A(=$P^T N^{-1} P$) by p_k, where p_k looks like a map. Then I do a bunch of map times map operations, which are fast.



Conjugate gradient steps somewhat intelligently downhill to get to a minimum error.

Finally, Memory.

- "But wait - you can't fit 150 TB in memory, can you?" Well, no. But... First, only need $P^T N^{-1} d$, which is the size of a map, don't need actual data.
- Different chunks of data don't have noise in common. So, in matrixese, N^{-1} is block-diagonal
- P has one 1 per column, so it doesn't mix things.
- This means that I can make the dirty map for a chunk of model data (Pm), then sum those maps, which are small. BTW, this should scream "parallelize me".
- Of course, I still need P, where I was pointed...

Timing, etc. concerns

- Conjugate-gradient can take few hundred steps to converge.
- Noise is often correlated between detectors. Must take this into account as well.
- Applying P takes time (not as much as FFT's, but close)
- Still have to read data - tens of TB non-trivial to read.
- My noise estimate depends on the map (n=d-Pm). So, will have to update the noise during mapmaking.
- Can't store pointing for all the data, so need fast way to reconstruct it.

Disk Usage

- Disks read ~50 MB/s (a bit more perhaps).
- If I have 1000 disks reading, then get 50 GB/s, 3 TB/min, or 50 min to read entire dataset. Hard, but not impossible.
- Alas, need to re-read data every time I update noise. May need to do this several times during mapmaking.
- The lower your S/N, the fewer times you have to do this. Yay, crappy data!

Bird's Eye View of Steps

$$P^T N^{-1} P m = P^T N^{-1} d$$

Read the data.

- Decide on a map, and how to assign data to map.
- Figure out noise and its inverse.
- Figure out how to apply N^{-1}, P, and P^T to objects.
- Use iterative techniques to solve LLS problem.

Step 1: Read the data.

```
int readOneTOD(TOD *tod,char *froot, int mytod)
{
  char c[512];
  sprintf(c,"%s%d.dat",froot,mytod);
  fprintf(stderr,"Reading %s\n",c);
  FILE *infile=fopen(c,"r");
  if (!infile) {
    fprintf(stderr,"Error opening file %s for reading in readOneTOD.\n",c);
    return 1;
  }

  fread(&tod->n,1,sizeof(int),infile);
  allocateOneTOD(tod,tod->n);

  fread(tod->d,sizeof(NType),tod->n,infile);
  fread(tod->x,sizeof(NType),tod->n,infile);
  fread(tod->y,sizeof(NType),tod->n,infile);
  fread(tod->filt,sizeof(NType),tod->n,infile);

  fclose(infile);
  return 0;
}
```

```
int readAllTOD(TOD *tod, char *froot, int ntod)
{
  for (int i=0;i<ntod;i++)
    if (readOneTOD(&tod[i],froot,i))
      return i;
  return 0;

}
```

Have code to read one timestream (TOD, Time-Ordered Data). We have a bunch of timestreams that need to be read. How would you do this with OpenMP? With MPI?

Reading, cont'd

- In the data, we have the data, x and y positions for each data point (could be, say, RA and Dec). Finally, have a representation of the noise.
- Just a big loop over data. Don't want all processes reading all data. Trivial in OpenMP. Requires thinking in MPI.
- You may notice if you read the data twice, second time is much faster. Hello, disk caching.

Step 2: Set the Pointing

- Have both x and y coordinates in high precision. Really, just want to know in which map pixel each data point lives.
- So, make a 1-D vector corresponding to a 2-D map. Then we need to store 1 integer per data point instead of 2 float/doubles.
- Need to make sure map is big enough. Need outer limits for all data. MPI alert.

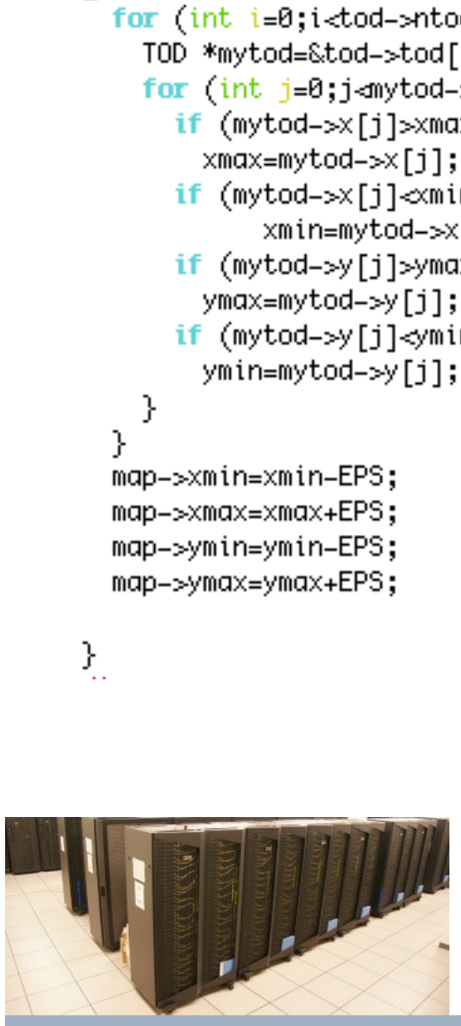
Pointing, cont'd.

```
void findMapLimits(MAP *map, TODvec *tod)
{
  float xmin,xmax,ymin,ymax;
  xmin=tod->tod[0].x[0];
  xmax=xmin;
  ymin=tod->tod[0].y[0];
  ymax=ymin;

  for (int i=0;i<tod->ntod;i++) {
    TOD *mytod=&tod->tod[i];
    for (int j=0;j<mytod->n;j++) {
      if (mytod->x[j]>xmax)
        xmax=mytod->x[j];
      if (mytod->x[j]<xmin)
          xmin=mytod->x[j];
      if (mytod->y[j]>ymax)
        ymax=mytod->y[j];
      if (mytod->y[j]<ymin)
        ymin=mytod->y[j];
    }
  }
  map->xmin=xmin-EPS;
  map->xmax=xmax+EPS;
  map->ymin=ymin-EPS;
  map->ymax=ymax+EPS;

}
```

Find my min/max x and y. Have to look at all of the data. Have a map structure that stores the map limits.



If I have coords ix,iy, then set my pixel to ix+nx*iy. Now I can throw away x&y and save memory.

```
void assignTODIndices(MAP *map, TODvec *tod)
{
  for (int i=0;i<tod->ntod;i++) {
    int ix,iy;
    TOD *mytod=&tod->tod[i];
    for (int j=0;j<mytod->n;j++) {
      ix=(mytod->x[j]-map->xmin)/map->oversamp;
      iy=(mytod->y[j]-map->ymin)/map->oversamp;
      mytod->ind[j]=ix+iy*map->nx;
      assert(mytod->ind[j]>=0);
      assert(mytod->ind[j]<map->npix);
    }
  }
}
```

Step 3: Applying the Pointing

- So, now we have an index for each data point. This is a compressed representation of P.
- To multiply P*m, we have: for i=1 to n, data(i)=map(index(i))
- To multiply P^T*d, we have: for i=1 to n, map(index(i))+=data(i).
- Again, these loops scream parallelize me!

Applying the Pointing II

```
void map2tod(MAP *map, TODvec *tod)
{
  for (int i=0;i<tod->ntod;i++) {
    TOD *mytod=&tod->tod[i];
    for (int j=0;j<mytod->n;j++)
      mytod->d[j]=map->map[mytod->ind[j]];
  }

}
```

map2tod carries out d=Pm. I loop over all of my TOD's, and over all the data in each TOD. All data is separate, so embarassingly parallel.

```
void tod2map_simple(MAP *map, TODvec *tod)
{

  clearMap(map);
  for (int i=0;i<tod->ntod;i++) {
    TOD *mytod=&tod->tod[i];
    filterTOD(mytod);
    for (int j=0;j<mytod->n;j++)
      map->map[mytod->ind[j]]+=mytod->data_filt[j];
  }
}
```

tod2map does $m=P^T d$. Note that many pixels can write to a single map pixel, so will need to watch for race conditions in parallel.

(filterTOD command applies noise inverse to data. Will get there soon.)

Applying the Pointing III

- To parallelize the loop, where we sum data into map, each thread should have its own copy of the map to avoid race condition.
- Happens naturally in MPI, requires making private map copies in OpenMP.
- At end, must combine maps. MPI use MPI_Allreduce. OpenMP, well, this may be one of few times where MPI is simpler.

Reducing in OpenMP

- Simplest map reduction in OpenMP is to use #pragma omp critical. This will work - how does it scale?
- Say data parts scale perfectly, so time $\propto 1/n_p$. Reduction time $\propto n_p$.
- How bad? Say data is 100x work of map sum. 2 CPUs: 50 time units on data, 2 for reduction, so 4% in reduction. 10 CPUs: 10 time units on data, 10 on reduction. So have *already* lost factor of 2. 16 CPU's: ~75% of time reducing!

Makes Amdahl look optimistic!

Reducing in OpenMP II

- First thing we can do: end of *#pragma omp for* has an implied barrier. Everybody waits. Well, if one thread is done, start it on the reduction while others still work. Add *nowait* clause to *#pragma omp for*.
- *#pragma omp critical* locks a whole block of code. If I'm reducing, if I've finished a piece of map, next guy can start.
- One note: Could put critical around each row. Would this help? Not offhand. Critical is a global structure: only one thread allowed *anywhere* in code in a critical.
- Can give names to critical regions, but who wants to name every row? Overhead on criticals also potentially high.

Reduction: Locks.

- Solution: locks.
- What is a lock? Low-level routine with a variable. Can create locks (*omp_lock_t* type in C, pointer size in Fortran).
- If a thread asks for lock, waits until it gets it. Only one thread allowed to have a lock at a time.
- Make a lock for each row. Each thread sets the lock when starts, releases when done.
- Might be lower overhead than critical: only care about lock I ask for. (Of course compiler might do critical with a lock)

Locks in Action

```
void assignMapLocks(MAP *map)
{
  //allocate space for a bunch of locks.
  map->locks=(omp_lock_t *)malloc(sizeof(omp_lock_t)*map->nx);
  for (int i=0;i<map->nx;i++)
    omp_init_lock(&(map->locks[i]));  //we have to set up each lock we want.

}
```

```
    for (int i=0;i<map->nx;i++) {
      omp_set_lock(&(map->locks[i]));
      for (int j=0;j<map->ny;j++)
        map->map[i*map->ny+j]+=mymap->map[i*map->ny+j];
      omp_unset_lock(&(map->locks[i]));
    }
```

```
Reduction took      0.04499 seconds.
Reduction took      0.04601 seconds.
Reduction took      0.04814 seconds.
Reduction took      0.05106 seconds.
Reduction took       0.0526 seconds.
Reduction took      0.04634 seconds.
Reduction took      0.04972 seconds.
Reduction took      0.04265 seconds.
Reduction took      0.05058 seconds.
Reduction took      0.05094 seconds.
Reduction took      0.05037 seconds.
Reduction took       0.0465 seconds.
Reduction took      0.04559 seconds.
Reduction took      0.04672 seconds.
```

Have to set up locks, but now I can have processes reducing simultaneously. @8 threads, locks win by 2 (and maybe more if I tweak #of locks). Also, w/ locks 2 threads takes 0.02, so reduction time sub-linear.

```
Reduction took       0.0836 seconds.
Reduction took      0.08277 seconds.
Reduction took      0.08134 seconds.
Reduction took       0.0828 seconds.
Reduction took      0.08185 seconds.
Reduction took       0.0818 seconds.
Reduction took      0.08145 seconds.
Reduction took      0.08307 seconds.
Reduction took      0.08314 seconds.
Reduction took       0.0815 seconds.
Reduction took      0.08057 seconds.
```

```
    for (int i=0;i<map->nx;i++) {
      omp_set_lock(&(map->locks[i]));
      for (int j=0;j<map->ny;j++)
        map->map[i*map->ny+j]+=mymap->map[i*map->ny+j];
      omp_unset_lock(&(map->locks[i]));
    }
```

Step 3: Noise

- I have kindly given you noise in the form you'll need. Doesn't happen in real life, so enjoy!
- You have noise in Fourier Space, where it is diagonal. To apply, FFT data, divide by noise, and FFT back.
- FFT's so common, going to discuss them.

FFTW

- Almost everybody uses FFTW for FFT's these days.
- FFTW=Fastest Fourier Transform in the West. Ask them...
- Unlike Numerical Recipes, FFTW works on all sizes - not just powers of 2.
- Be careful of powerful tool. Can blow off own foot.

Go to www.fftw.org for info, source code etc.

FFTW in Action

Here's how to actually calculate an FFT using FFTW. You will have to do this at some point in your career, so here's an example.

```c
//fftw_example1
#include <stdio.h>
#include <stdlib.h>
#include <fftw3.h>
#include <assert.h>
#include "../util/pca_utils.h"

int main(int argc, char *argv[])
{
  int n_in=1000;  //default
  if (argc>1)
    n_in=atoi(argv[1]);
  assert(n_in>0);

  for (int n=n_in;n<n_in+20;n++) {
    double *x=(double *)malloc(sizeof(double)*n);
    int nn=n/2+1;  // we win a factor of 2 by doing real FFT's
    //fftw has its own types and its own memory-aligned malloc
    fftw_complex *vec=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*nn);
    //FFTW works by creating plans for doing fft's, then executing them.
    fftw_plan p_forward=fftw_plan_dft_r2c_1d(n,x,vec,FFTW_ESTIMATE);
    pca_time tt;
    tick(&tt);
    //here's where we actually do the FFT
    fftw_execute(p_forward);
    printf("Toock %12.4g seconds to do FFT of length %d\n",tocksilent(&tt),n);
    fftw_destroy_plan(p_forward);
    fftw_free(vec);
    free(x);
  }
}
```

Program Output

```
[fedora@localhost mapmaker]$ ./fftw_example1 1048666
Toock       0.1827 seconds to do FFT of length 1048666
Toock       0.3006 seconds to do FFT of length 1048667
Toock        0.258 seconds to do FFT of length 1048668
Toock       0.6576 seconds to do FFT of length 1048669
Toock       0.1758 seconds to do FFT of length 1048670
Toock       0.2233 seconds to do FFT of length 1048671
Toock       0.3094 seconds to do FFT of length 1048672
Toock       0.3143 seconds to do FFT of length 1048673
Toock       0.1858 seconds to do FFT of length 1048674
Toock       0.6913 seconds to do FFT of length 1048675
Toock       0.2031 seconds to do FFT of length 1048676
Toock       0.5409 seconds to do FFT of length 1048677
Toock       0.2073 seconds to do FFT of length 1048678
Toock       0.7554 seconds to do FFT of length 1048679
Toock       0.1949 seconds to do FFT of length 1048680
Toock        1.434 seconds to do FFT of length 1048681
Toock       0.3835 seconds to do FFT of length 1048682
Toock       0.2919 seconds to do FFT of length 1048683
Toock       0.1852 seconds to do FFT of length 1048684
Toock       0.2723 seconds to do FFT of length 1048685
[fedora@localhost mapmaker]$
```

```
[fedora@localhost mapmaker]$ ./fftw_example1 1048666
Toock       0.1271 seconds to do FFT of length 1048666
Toock       0.2807 seconds to do FFT of length 1048667
Toock       0.2017 seconds to do FFT of length 1048668
Toock       0.8031 seconds to do FFT of length 1048669
Toock       0.2312 seconds to do FFT of length 1048670
Toock       0.2034 seconds to do FFT of length 1048671
Toock       0.2424 seconds to do FFT of length 1048672
Toock        0.449 seconds to do FFT of length 1048673
Toock        0.192 seconds to do FFT of length 1048674
Toock       0.6341 seconds to do FFT of length 1048675
Toock       0.1891 seconds to do FFT of length 1048676
Toock       0.8338 seconds to do FFT of length 1048677
Toock        0.251 seconds to do FFT of length 1048678
Toock       0.7319 seconds to do FFT of length 1048679
Toock       0.2291 seconds to do FFT of length 1048680
Toock        1.469 seconds to do FFT of length 1048681
Toock       0.3299 seconds to do FFT of length 1048682
Toock       0.2752 seconds to do FFT of length 1048683
Toock       0.2328 seconds to do FFT of length 1048684
Toock       0.2914 seconds to do FFT of length 1048685
[fedora@localhost mapmaker]$
```

Two runs of FFTW for different sizes. Note badness of 1048681. I could win a factor of 7 in CPU time by tossing 1 data point out of 1,000,000. Keep this in mind when FFTing.

More FFTW

- Say I really needed to calculate all those FFT's. Well, looks perfect for OpenMP!
- Slap a parallel for with a dynamic schedule, should leak to happiness, right?
- Let's see.

Multi-threaded FFTW

```
[sievers@tpb4 mapmaker]$ setenv OMP_NUM_THREADS 1
[sievers@tpb4 mapmaker]$ ./fftw_example1
Toock      2.5e-05 seconds to do FFT of length 1000
Toock     0.000113 seconds to do FFT of length 1001
Toock     0.000508 seconds to do FFT of length 1002
Toock     0.000159 seconds to do FFT of length 1003
Toock     0.000183 seconds to do FFT of length 1004
Toock     0.000146 seconds to do FFT of length 1005
Toock     0.000201 seconds to do FFT of length 1006
Toock     0.000316 seconds to do FFT of length 1007
Toock      2.3e-05 seconds to do FFT of length 1008
Toock     0.000146 seconds to do FFT of length 1009
Toock      8.2e-05 seconds to do FFT of length 1010
Toock     0.000144 seconds to do FFT of length 1011
Toock      3.2e-05 seconds to do FFT of length 1012
Toock       0.0002 seconds to do FFT of length 1013
Toock      2.3e-05 seconds to do FFT of length 1014
Toock      3.3e-05 seconds to do FFT of length 1015
Toock      8.6e-05 seconds to do FFT of length 1016
Toock     0.000142 seconds to do FFT of length 1017
Toock     0.000199 seconds to do FFT of length 1018
Toock     0.000189 seconds to do FFT of length 1019
[sievers@tpb4 mapmaker]$ setenv OMP_NUM_THREADS 8
[sievers@tpb4 mapmaker]$ ./fftw_example1
Segmentation fault
[sievers@tpb4 mapmaker]$
```

Works fine on 1 thread. Dies miserably on 8. Did we screw up?

Did We Screw Up?

```
#pragma omp parallel for shared(n_in) default(none) schedule(dynamic,1)
  for (int n=n_in;n<n_in+20;n++) {
    double *x=(double *)malloc(sizeof(double)*n);
    int nn=n/2+1;  // we win a factor of 2 by doing real FFT's
    //fftw has its own types and its own memory-aligned malloc
    fftw_complex *vec=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*nn);
    //FFTW works by creating plans for doing fft's, then executing them.
    fftw_plan p_forward=fftw_plan_dft_r2c_1d(n,x,vec,FFTW_ESTIMATE);
    pca_time tt;
    tick(&tt);
    //here's where we actually do the FFT
    fftw_execute(p_forward);
    printf("Toock %12.4g seconds to do FFT of length %d\n",tocksilent(&tt),n);
    fftw_destroy_plan(p_forward);
    fftw_free(vec);
    free(x);
  }
```

No! All variables used in loop are declared in there.
All we did was use stuff and call FFTW. Oh wait...

Thread Safety

"Users writing multi-threaded programs must concern themselves with the thread safety of the libraries they use—that is, whether it is safe to call routines in parallel from multiple threads. FFTW can be used in such an environment, but some care must be taken because the planner routines share data (e.g. wisdom and trigonometric tables) between calls and plans.

The upshot is that the only thread-safe (re-entrant) routine in FFTW is fftw_execute"

-FFTW Documentation

In English

- Sometimes you have to use other peoples code.
- Sometimes that code only likes to run one copy at a time.
- If you try to run more copies, you will be punished.
- If you want to call library routines in parallel sections, check on their thread safety. May save you much pain.
- What's going on to make this a problem?

In a Word, Static

- C supports static variables. They are variables in routines that stick around and keep their values between calls.
- If one thread expects a static not to have changed between calls, but another thread changed it, well...
- The only static I ever use is to tell me if a routine is being called for the first time. Even then I feel dirty. Better to put inside a structure.
- So am I trashing FFTW? No. Come talk to me about statics when your FFT is as fast as theirs.

FFTW Loop Working

```
#pragma omp parallel for shared(n_in) default(none) schedule(dynamic,1)
  for (int n=n_in;n<n_in+20;n++) {
    double *x=(double *)malloc(sizeof(double)*n);
    int nn=n/2+1;  // we win a factor of 2 by doing real FFT's
    //fftw has its own types and its own memory-aligned malloc
    fftw_complex *vec=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*nn);
    //FFTW works by creating plans for doing fft's, then executing them.
    fftw_plan p_forward;
#pragma omp critical
    p_forward=fftw_plan_dft_r2c_1d(n,x,vec,FFTW_ESTIMATE);

    pca_time tt;
    tick(&tt);
    //here's where we actually do the FFT
    fftw_execute(p_forward);
    printf("Toock %12.4g seconds to do FFT of length %d\n",tocksilent(&tt),n);
#pragma omp critical
    fftw_destroy_plan(p_forward);
    fftw_free(vec);
    free(x);
  }
```

```
[sievers@tpb4 mapmaker]$ setenv OMP_NUM_THREADS 8
[sievers@tpb4 mapmaker]$ ./fftw_example1 1048666
Toock        0.189 seconds to do FFT of length 1048666
Toock       0.1925 seconds to do FFT of length 1048670
Toock       0.4061 seconds to do FFT of length 1048667
Toock       0.2046 seconds to do FFT of length 1048668
Toock       0.3024 seconds to do FFT of length 1048671
Toock       0.2004 seconds to do FFT of length 1048672
Toock       0.3796 seconds to do FFT of length 1048669
Toock       0.2086 seconds to do FFT of length 1048676
Toock       0.4724 seconds to do FFT of length 1048673
Toock        0.172 seconds to do FFT of length 1048674
Toock        0.334 seconds to do FFT of length 1048675
```

Since plans aren't thread-safe, lock 'em up in critical regions. And we're in business. Another solution: make plans once & save them.

} } } (Closing Many Clauses)

Now that we have met FFTW and how to use it in parallel, here's noise filtering.

```
void filterTOD(TOD *tod)
{
  if (tod->dofilt) {
    fftw_complex *vec=(fftw_complex *)fftw_malloc(sizeof(fftw_complex)*(tod->n));
    fftw_execute_dft_r2c(tod->p1,tod->d,vec);

    for (int i=0;i<tod->n/2+1;i++) {
      vec[i][0]=vec[i][0]/tod->filt[i]/tod->n;
      vec[i][1]=vec[i][1]/tod->filt[i]/tod->n;
    }
    fftw_execute_dft_c2r(tod->p2,vec,tod->data_filt);
    fftw_free(vec);
  }
}
```

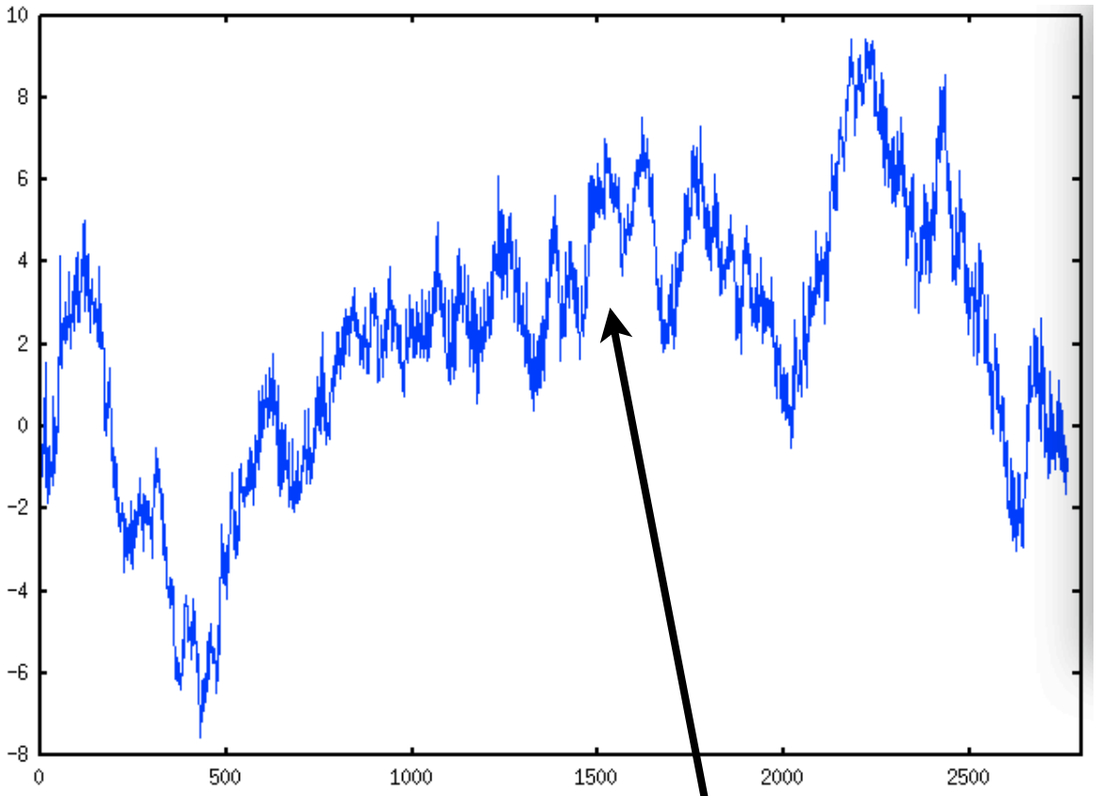
Need a temp space to FFT into.

Run the FFT. This call lets you use same plan for different data vectors.

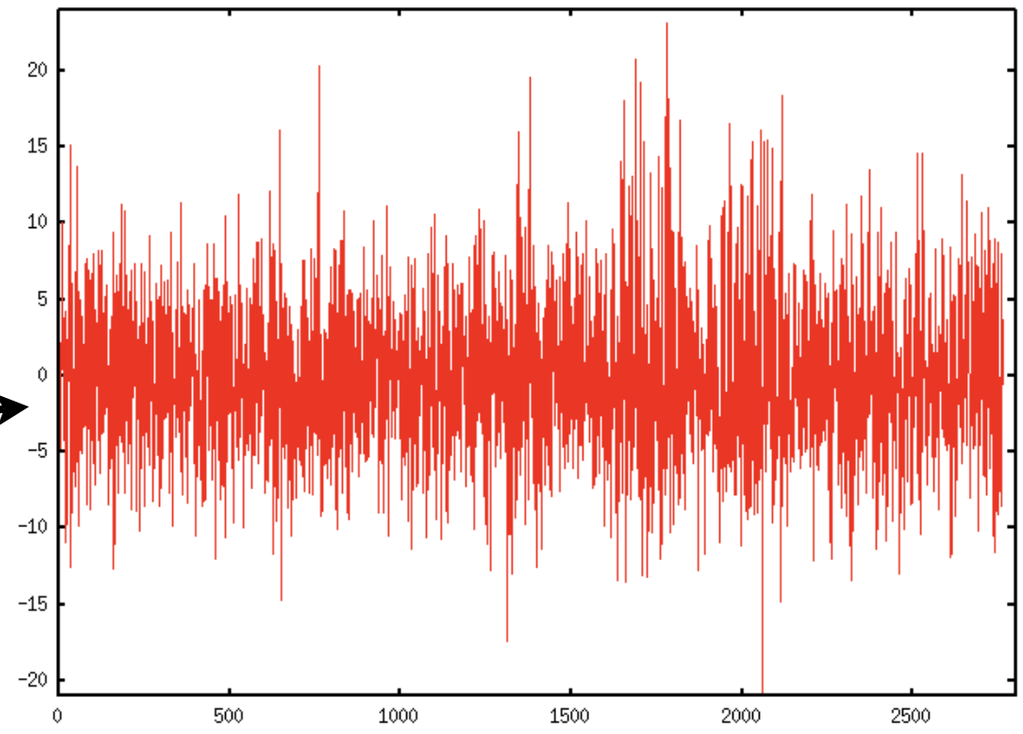
Apply the noise to the FT

Get filtered data back to time domain.

What Does This Look Like?



Turn this into this



Back to Mapmaker

$$P^T N^{-1} P m = P^T N^{-1} d$$

- Now we know how to do each operation.
- Next problem: given a guess for m, come up with better guess for m. Repeat until happy.
- Conjugate-gradient always works. Tries to solve Ax=b for positive-definite A.
- Will spare you details, but works by taking optimal steps in (sort-of) orthogonal directions.

Pre-conditioning

- Conjugate gradient solves Ax=b.
- For (non-singular) A^*, $A^*Ax=A^*b$ if Ax=b
- If we pick a good A*, in the sense that it is close to A-1, CG converges faster. Solving Ix=b is fast!
- A* is called a pre-conditioner. Picking them is an art. No more here, but if you use CG, investigate.

Giant Matrix Solving in Action

```
void runPCG(MAP *map, TODvec *tod)
{
  createFFTWplans(tod);
  tod2map(map,tod);
  MAP *r=makeMapCopy(map);
  MAP *p=makeMapCopy(map);
  MAP *x=makeMapCopy(map);
  clearMap(x);
```

Setup FFT Plans

Make $P^T N^{-1} d$

Setup CG quantities.

If you recall (which you don't), I put noise filtering inside the P^T part. Since N^{-1} never appears without P^T in mapmaking equation, it makes sense.

```
for (int i=0;i<50;i++)
  {
    tick(&tt);
    fprintf(stderr,"residual is %14.5e\n",PCGstep(r,p,x,tod));
    tock(&tt);
    displayMap(x);
  }
```

Run it!

Actual Solving

```
NType PCGstep(MAP *r, MAP *p, MAP *x, TODvec *tod)
{
  MAP *ap=makeMapCopy(p);
  map2map(ap,tod);
  NType rsqr=mapTimesMap(r,r);
  NType alpha_k=rsqr/mapTimesMap(p,ap);
  MAP *rk=makeMapCopy(r);
  MAPaxpy(rk,ap,-alpha_k);
  NType beta_k=mapTimesMap(rk,rk)/rsqr;
  MAP *pk=makeMapCopy(rk);
  MAPaxpy(pk,p,beta_k);
  MAPaxpy(x,p,alpha_k);

  copyMap2Map(r,rk);
  copyMap2Map(p,pk);
  destroyMap(rk);
  destroyMap(pk);
  destroyMap(ap);

  return rsqr;
}
```

This step is the biggie. It calculates $P^T N^{-1} P m$. Consists of map2tod followed by tod2map.

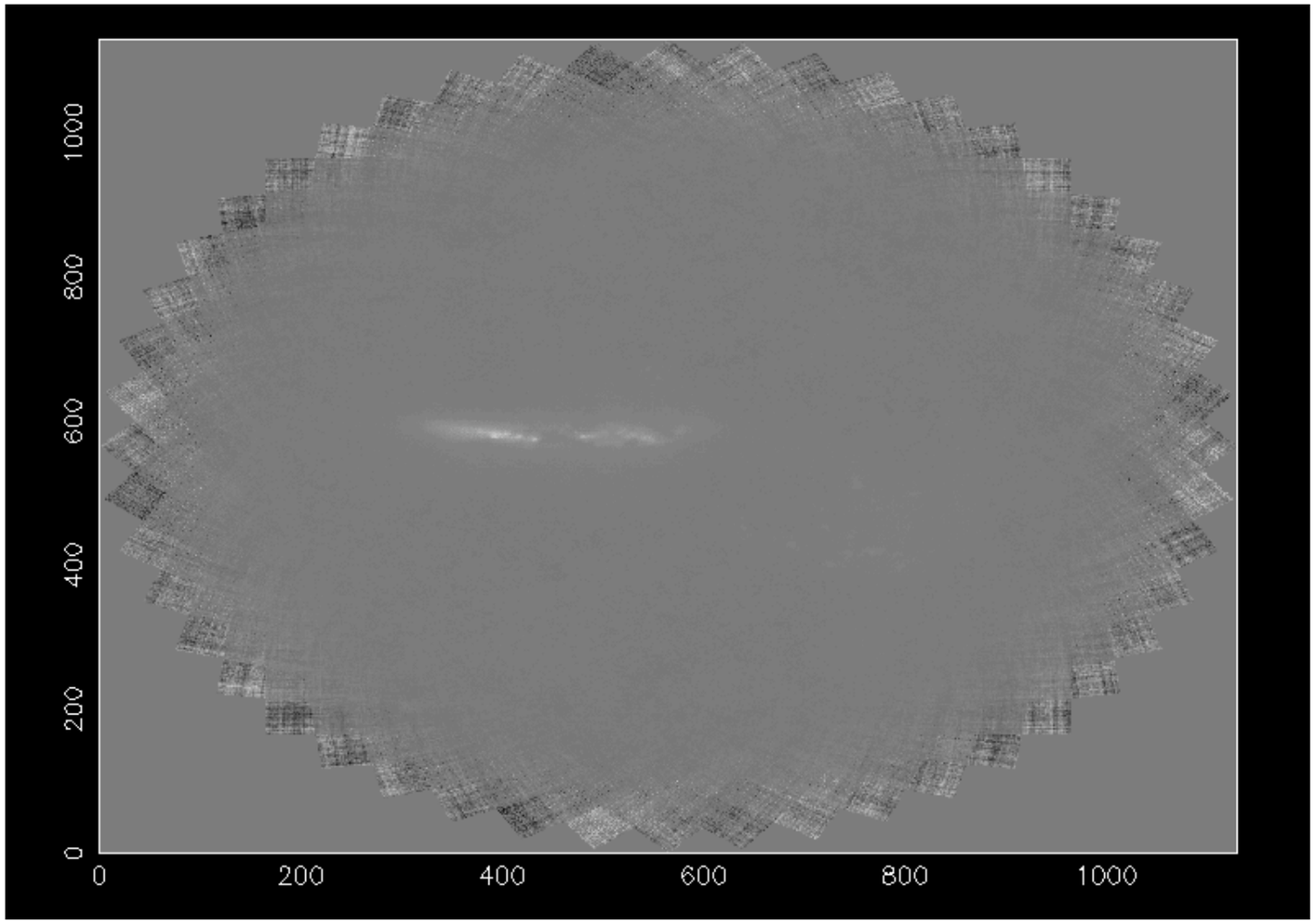
Take steps. The objects are all size of maps (small), not data (big)

Update quantities & clean up.

And this is my current error

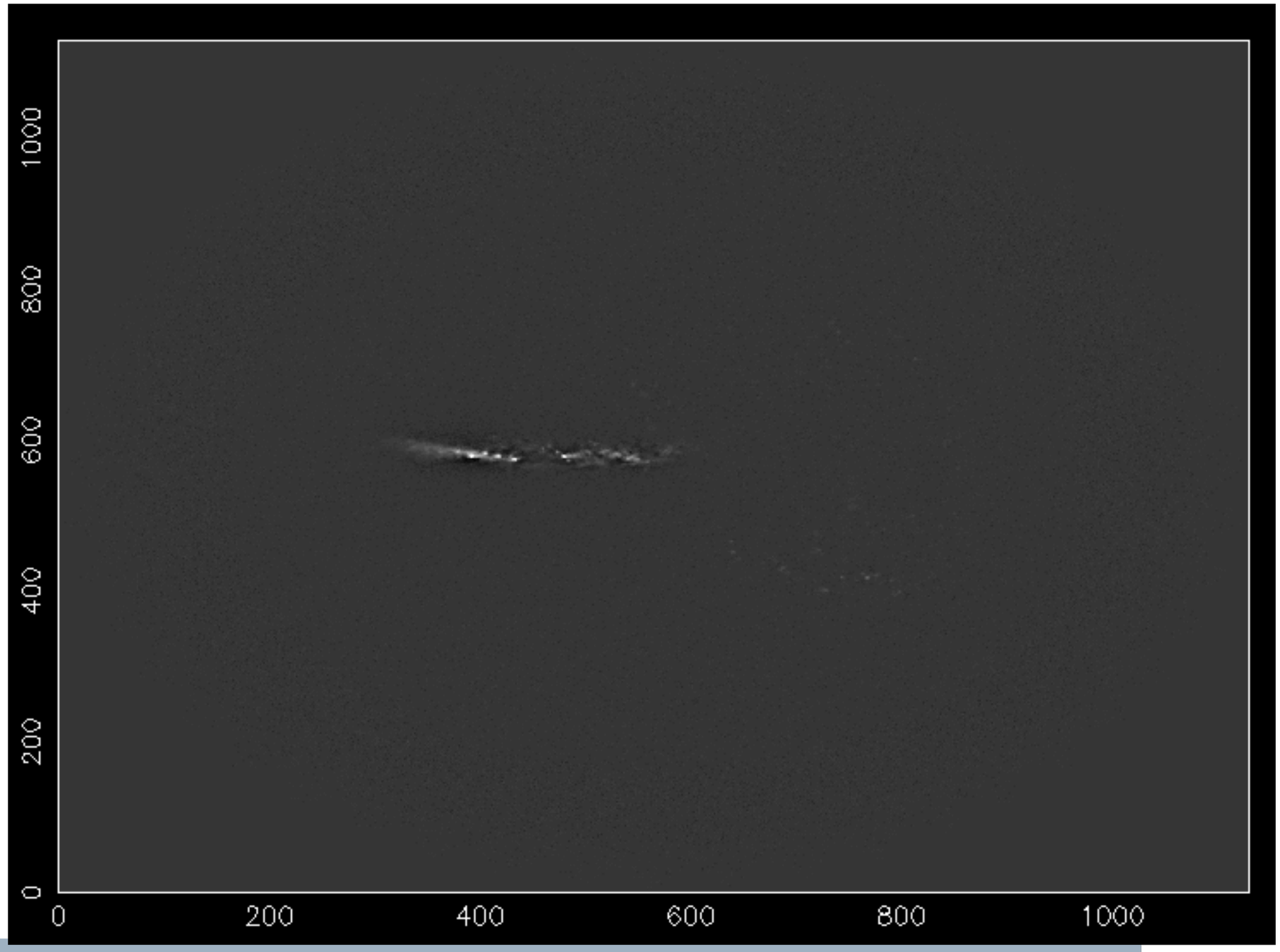
So, Let's See it in Action!

Merging galaxy pair Arp148

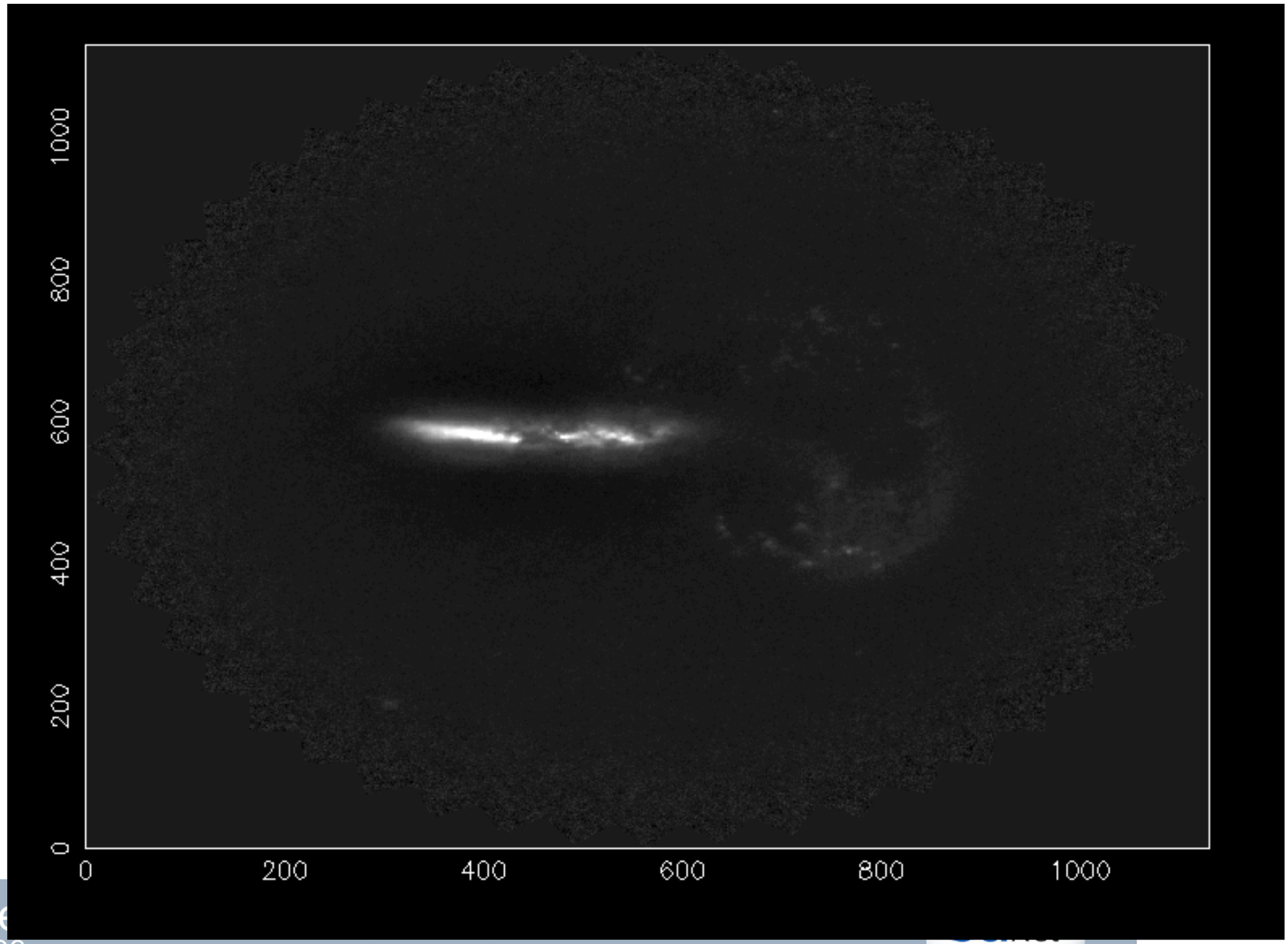


This is what we get from just adding up the data.

Initial Guess:



After Several Iterations



And now, homework...

Step 0: go to ~/pca/src/mapmaker. Compile mapmaker.c. Run it. Did it work? If it can't find data, make sure froot in main is set correctly. Copy mapmaker.c to mapmaker_omp.c Now we will OpenMP mapmaker_omp.c

Step 1: OpenMP map2map

- First, OpenMP map2tod. There is no conflict in this routine: a simple *omp parallel for* should work.
- Second, OpenMP tod2map. You will have to make a private copy of the map for each thread. You may find the call: MAP *mymap=makeMapCopy(map); useful. At the end, call destroyMap(mymap); Feel free to use criticals in summation. Use the nowait clause on the for loop. Why is this OK?
- Did it run? Did you get same answer (look at residuals). What was the speedup?

Step 2: OpenMP PCG

- We have left CG serial. Is this bad? The CG stepper uses the routines mapTimesMap and MAPaxpy.
- MAPaxpy is a simple parallel for. OpenMP it.
- mapTimesMap is just a dot product. OpenMP it with a reduce.
- Now there are no serial bits left. Re-run. Did it speed up?

Step 3: MPI the data

- There are only a few places where MPI kicks in.
- Start with a fresh copy called mapmaker_mpi.c
- First, need to split up data. Each process only gets a piece of data. Throughout code, tod.ntod is total number of tod's on a node. Simplest to keep it that way, and add field tod.total_ntod to the TODvec structure.
- Have each thread decide how many tod's it gets (and put in tod.ntod). Next, have readAllTOD read the right subset of data. Run - did every tod get read once?

Step 4: MPI the Limits

- Maps need to have the same x/y limits. Need to know the global max/min of both x and y.
- MPI findMapLimits. You can use MPI_Allreduce to globally share the x and y limits. Can you share all the limits with a single MPI_Allreduce? Might think about flipping sign of x_{min} and y_{min}, then flipping back...
- Do you get same x/y limits as single processor job?

Step 5: MPI the mapping

- tod's don't depend on each other, so if map is globally agreed on, map2tod is already set!
- tod2map requires the same reduction that OpenMP tod2map needed. If you use an MPI_Allreduce, everyone agrees on the map at the end.
- PCG: I think (but don't know for sure) that communications costs in PCG are steeper than running serially. So, skip it. You should be done now!
- Does your code run? Do you get the right answers?