Dynamic load-balancing on multi-FPGA systems

a case study

Volodymyr Kindratenko
Innovative Systems Lab (ISL)
National Center for Supercomputing Applications (NCSA)

Robert Brunner and Adam Myers
Department of Astronomy
University of Illinois at Urbana-Champaign (UIUC)
Angular Correlation Function

• TPACF, denoted as $\omega(\theta)$, is the frequency distribution of angular separations $\theta$ between celestial objects in the interval $(\theta, \theta + \delta\theta)$
  – $\theta$ is the angular distance between two points

• Blue points (random data) are, on average, randomly distributed, red points (observed data) are clustered
  – Blue points: $\omega(\theta)=0$
  – Red points: $\omega(\theta)>0$

• Can vary as a function of angular distance, $\theta$ (yellow circles)
  – Blue: $\omega(\theta)=0$ on all scales
  – Red: $\omega(\theta)$ is larger on smaller scales

Image source: http://astro.berkeley.edu/~mwhite/
The Method

• The angular correlation function is calculated using the estimator derived by Landy & Szalay (1993):

\[ \omega(\theta) = \frac{1}{n_D^2} \cdot DD(\theta) - \frac{2}{n_D n_R} \sum DR_i(\theta) + 1 \]

\[ \frac{1}{n_R^2} \sum RR_i(\theta) \]

• where \( DD(\theta) \) and \( RR(\theta) \) are the autocorrelation function of the data and random points, respectively, and \( DR(\theta) \) is the cross-correlation between the data and random points.
// pre-compute bin boundaries, binb

// compute DD
doCompute{CPU|MAP}(data, npd, data, npd, 1, DD, binb, nbins);

// loop through random data files
for (i = 0; i < random_count; i++)
{
    // compute RR
    doCompute{CPU|MAP}(random[i], npr[i], random[i], npr[i], 1, RRS, binb, nbins);

    // compute DR
    doCompute{CPU|MAP}(data, npd, random[i], npr[i], 0, DRS, binb, nbins);
}

// compute w
for (k = 0; k < nbins; k++)
{
    w[k] = (random_count * 2*DD[k] - DRS[k]) / RRS[k] + 1.0;
}
for (i = 0; i < ((autoCorrelation) ? n1-1 : n1); i++)
{
    double xi = data1[i].x, yi = data1[i].y, zi = data1[i].z;
    for (j = ((autoCorrelation) ? i+1 : 0); j < n2; j++)
    {
        double dot = xi * data2[j].x + yi * data2[j].y + * data2[j].z;
        register int k, min = 0, max = nbins;
        if (dot >= binb[min]) data_bins[min] += 1;
        else if (dot < binb[max]) data_bins[max+1] += 1;
        // run binary search
        else {
            while (max > min+1)
            {
                k = (min + max) / 2;
                if (dot >= binb[k]) max = k;
                else min = k;
            };
            data_bins[max] += 1;
        }
    }
}
for (i = 0; i < random_count; i++) {
    #pragma omp parallel sections
    #pragma omp section
doComputeMAP1(⋯, mapC);

    #pragma omp section
doComputeMAP2(⋯, mapE);
}
Simplified Performance Model

- Analysis of a data/random file with 100 data points each
  - Autocorrelation between the points in the random data file requires 100*(100-1)/2=4,950 steps
  - Cross-correlation between the observed data and random data requires 100*100=10,000 steps
  - MAP Series C processor is idle about 50% of the time!
Consider Data Partitioning…

Dataset A: 100 points

Dataset B: 100 points

Autocorrelation Jobs
- A1-A1 (ac)
- A1-A2 (cc)
- A1-A3 (cc)
- A2-A2 (ac)
- A2-A3 (cc)
- A3-A3 (ac)

Cross-correlation Jobs
- A1-B1 (cc)
- A1-B2 (cc)
- A1-B3 (cc)
- A2-B1 (cc)
- A2-B2 (cc)
- A2-B3 (cc)
- A3-B1 (cc)
- A3-B2 (cc)
- A3-B3 (cc)

MAP C

MAP E
Consider Data Partitioning…

- Analysis of a data/random file with 100 data points each
  - Each data file is divided into 3 equally sized segments
  - Autocorrelation is computed first, followed by the cross-correlation
  - Each MAP processor is invoked with the first available unprocessed pair of segments
  - MAP Series C processor is idle about 7% of the time!
for each pair of d1/d2 segments, $p_{ij}$
for each MAP processor, $m$
if $m$ is free
   assign $p_{ij}$ to $m$
break
endif
endfor
endfor
Job Scheduler Implementation

```c
do {
    for (k = 0; k < K; k++) {
        if (job[k].status == running) continue;  // let it run
        if (job[k].status == done) continue;    // nothing to do anymore
        if (job[k].status == finished) {        // need to get results back
            pthread_join(job[k].thread, (void **)&mytd);  // join the thread
            for (i = 0; i < nbins+2; i++) res[i] += mytd->res[i];  // copy results
            job[k].status = done;  // set status to done
            TOTAL++;  // count number of fully executed jobs
            continue;
        }
    }
    for (t = NPROCS-1; t >= 0; t--) {  // is there a free MAP to run this job?
        if (thread_stat[t] == busy) continue;  // thread is busy
        if (self && i == j && t == 1) continue;  // not suitable thread for 'self'
        struct my_thread_data *mytd = (struct my_thread_data *)malloc(sizeof(struct my_thread_data));  // lock it
        pthread_create(&(job[k].thread), NULL, my_map_proc, (void *)mytd);
        thread_stat[t] = busy;  // set status to running
        job[k].status = running;  // no need to check the rest of the MAPs
        break;
    }
} while (TOTAL != K);
```
Load-balanced Implementation

for (i = 0; i < random_count; i++)
{
    JobScheduler(data, random);
    JobScheduler(random, random);
}
Conclusions

• Pros
  – A 9% performance improvement due to a better utilization of the idle resources
  – Near-identical load on each of the MAPs
  – Scalable solution that allows to mix compute subroutines with different performance characteristics

• Cons
  – Performance hit for the smaller datasets due to the overhead in calling the MAP processors
  – More complex execution flow and data management
Acknowledgements

• This work is funded by NASA Applied Information Systems Research (AISR) award number NNG06GH15G
  – Prof. Robert Brunner and Dr. Adam Myers from UIUC Department of Astronomy

• NCSA Collaborators
  – Dr. Rob Pennington, Dr. Craig Steffen, David Raila, Michael Showerman, Jeremy Enos, John Larson, David Meixner, Ken Sartain

• SRC Computers, Inc.
  – David Caliga, Dr. Jeff Hammes, Dan Poznanovic, David Pointer, Jon Huppenthal