Efficient Parallelization Techniques for Exact Algorithms in the Planted Motif Problem

Samuel Van Pelt

NSF Research Fellow
University of Connecticut
Bio-Grid 2013

samuelvanpelt@gmail.com

July 31, 2013
Overview

1. The Importance of Parallelization
   - Planted Motif Search

2. Algorithmic Changes
   - Loop-Level Parallelization
   - Block-Level Parallelization

3. Results
   - Synthesized Data
   - Real Data

4. Discussion

5. Conclusions
First generate all $l$-mers from each of the $t$ input sequences. This can be done in $O(tn)$ time. Then, from each of these sequences, given a particular input sequence, generate all possible $l$-mers that would satisfy the hamming distance requirement. This results in:

$$O(tn{l\choose d} \sum |d|)$$  (1)

Given $p$ cores, this is reduced to:

$$O\left(\frac{tn{l\choose d} \sum |d|}{p}\right)$$  (2)

time.
Loop-Level Parallelization

- Philosophy
  - Interdependence of Data
  - Use of Standard Sequential Structures
- High Memory Usage
- Easy To Implement
- OpenMP
#pragma omp parallel for
for (int i = 0; i < The Number of Sequences; i++)
{
    #pragma omp parallel for
    for (int j = 0; j < The Number of L_mers; j++)
    {
        genMotifCandidates(sequences[i], L_mer);
    }
    radix_sort(Clmer[i]);
    elimDupes(Clmer[i]);
}
Block-Level Parallelization

- Unwinding loops
  - Moving interdependent data out of loops
  - Separating tasks across processors
- Taking into consideration the number of cores
- Doing data interaction as efficiently as possible
#pragma omp parallel for
for (int i = 0; i < NUM_THREADS; i++)
{
    i_neighborhood[i] = c_lmer[i*sequences_per_thread];
    #pragma omp parallel for
    for (All 1/NUM_THREADS sequences)
        intersect(c_lmer[j], i_neighborhood[i]);
}
for (int i = 1; i < NUM_THREADS; i++)
    intersect(i_neighborhood[i], i_neighborhood[0]);
if (remainder != 0)
    for (int i = fN-1; i >= (fN - remainder); i--)
        intersect(c_lmer[i], i_neighborhood[0]);
An Alternative Method

It would be advantageous to access 2 elements at a time on each core and merge them, and then apply this method to the merged lists until all lists have been merged into one. This is depicted below:
This graphs the ratio of sequential run-time to parallel run-time on a quad-core machine. Given a theoretical peak efficiency of a parallel implementation of sequential algorithm of $p$, where $p$ is the number of cores, of 4, this implementation shows an efficiency of approximately 82%.
Synthesized Data

Run Times of PMS1 when $d = 4$

Sequential

Parallel

Sequential

Parallel
Optimized Run-Time

Run Times of PMS1 when $d = 4$

- **Sequential**
- **Parallel**
Real Data

Run Times of PMS1 when $d = 4$

Sequential

Parallel
3-D for Parallel Run Times on Real Data

Parallel
Efficiency in an Optimized Environment

Ratio of Sequential to Parallel Run-Time

- $d = 1$
- $d = 2$
- $d = 3$
- $d = 4$

Samuel Van Pelt (CMU)
Conclusions

- OpenMP
- Run-time Efficiency
- Future Work
  - Planted Motif Search 8
- Parallel Algorithms
- Significance