Recitation 4

Announcements

- PS2 grades are ready soon.
- PS4 is out, deadline 17.10, pass/fail
- We'll use a lab for PS5 and PS6 (with GPUs), access card form will come soon.

Cheating

Problem set 4

- Shared memory multithreading with OpenMP/Pthreads.
- Some theory
- Parallelize matrix multiplication
- Parallelize k-means

Theory

- One of the questions asks you to correct a error in a program.
- You should include the lines you change (which will be few) with your theory answers, not as a separate code file.

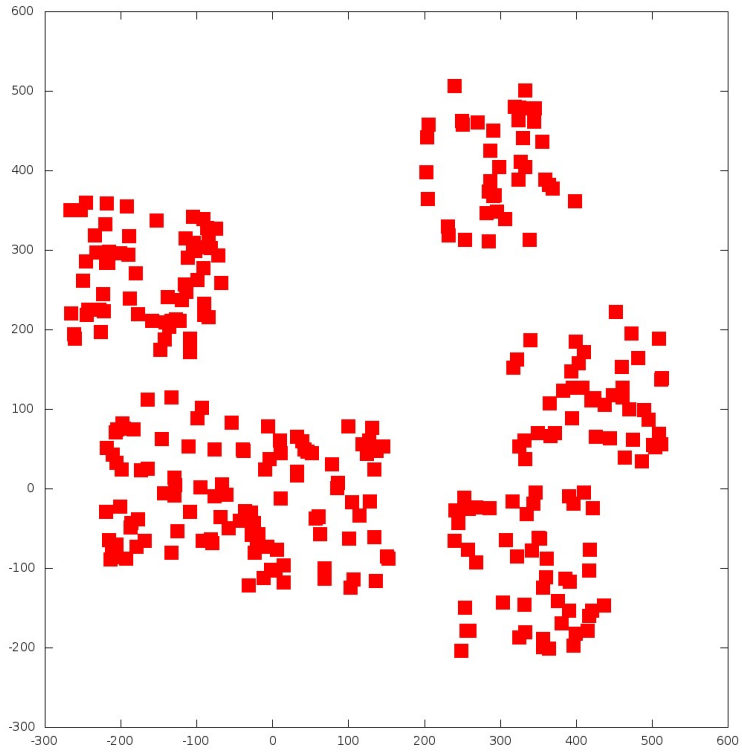
Parallelizing Matrix multiplication

- Easy to parallelize, intended as warm up.
- You'll get a serial version.
- You should do one version with OpenMP, and one version with PThreads.

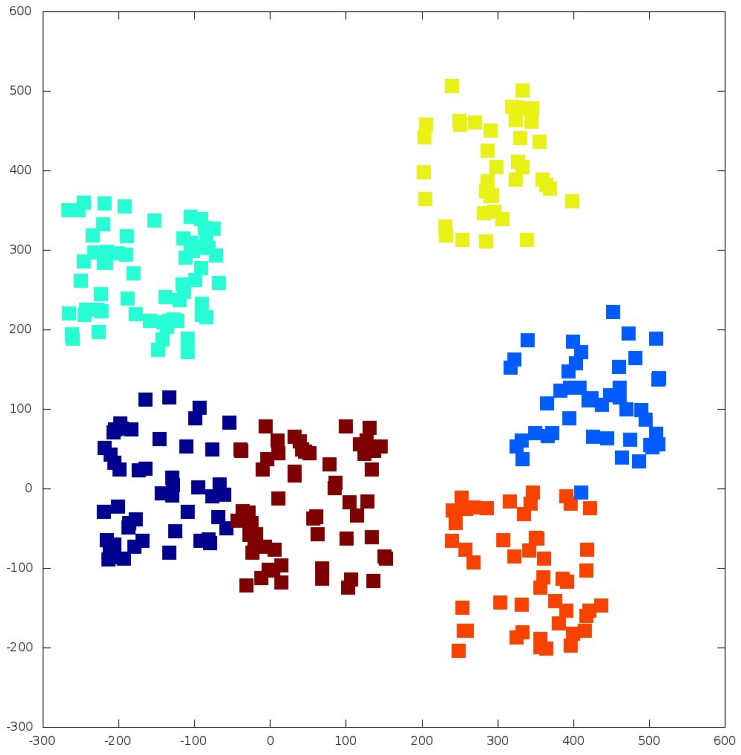
Cluster analysis

- Clustering is the task of grouping a set of objects in such a way that objects in the same group/cluster are more similar to each other than to those in other groups/clusters.

Example



(a) Input



(b) Output

k-means

Algorithm 1 k-means

Input: points
Output: clusters

```
assign clusters randomly
while any point changed cluster do
    for each cluster c do
        compute new centroid;
    end for
    for each point p do
        reassign to cluster with closest centroid
    end for
end while
```

Examples

Computing the centroid

- The centroid is the average position of the points in the cluster.
- One can find it by computing the sum of all the points, and divide by the number of points.
- E.g. the centroid of $(1, 3), (-1, 4), (2, 0)$ is $(\frac{1+(-1)+2}{3}, \frac{3+4+0}{3}) = (0.67, 2.33)$

Reassigning points, termination

- After the centroids have been updated, the distance between each point, and each centroid must be computed, and points should be assigned to the cluster with the closest centroid.
- The algorithm runs until no points are moved to a new cluster.

Assignment 4

- You're given a serial k-means implementation (*kmeans.c*)
- You're supposed to parallelize it, using both OpenMP, and pThreads.
- That is, you're supposed to hand in two versions, one with OpenMP, the other with pThreads.
- You should parallelize as much of the application as possible, that is, get as close to linear speedup as possible.

Practicalities

- The input is randomly generated.
- However, if you use the same seed, both the serial and your parallel versions will generate the same sequence of random numbers, so the input will be the same, allowing you to test for correctness.
- The output is dumped to stdout, printing to the screen can be quite slow, so you should redirect the output to a file when timing.

Compilation

- The included makefile will build all the three versions, for both k-means and matrix multiply.
- For k-means, `make plot` will build the serial program, run it, and make a plot of the output (like mine). You can modify it to plot the output of the paralle programs to check for correctness.
- If you want to do it on your own, `-pthread` is needed for Pthreads, `-fopenmp` for OpenMP

Useage

- The programs are used like this:
  ```
  kmeans nThreads nClusters nPoints
  gemm nThreads k n m
  ```
- Your versions should work like that too.
- While the serial program just ignores the number of threads specified, the parallel versions should use that number of threads for the computation.

Timing

- To time program we can use the `time` command, e.g. `time ./test`
- It doesn't have millisecond resolution, but once we let the program run for a few seconds, that's not an issue.
- Reports three values:
  - real: wall clock time
  - user: CPU time spent in user mode
  - system: CPU time spent in system mode (e.g. system calls)

Minimal pthreads

```c
#include <stdio.h>
#include <pthread.h>

void* run_thread(void* arg){
  printf("Thread␣%ld\n", (long)arg);
  pthread_exit(NULL);
}

int main(){
  pthread_t threads[4];
  for(long i = 0; i < 4; i++){
    pthread_create(&threads[i], NULL,
      run_thread, (void*)i);
  }
  pthread_exit(NULL);
}
```

Passing arguments to threads

```
typedef struct{
  int arg1;
  int arg2;
} args;
...
void* run_thread(void* a){
  args* b = (args*)a;
  printf("%d\n", b->arg1)
  ...

main(){
  ...
  args* a = malloc(...
  a->arg1 = 4;
  a->arg2 = 5;
  pthread_create(..., (void*)a);
```

Minimal OpenMP

```
#include <stdio.h>
#include <omp.h>

int main(){

#pragma omp parallel for
  for(int c = 0; c < 8; c++){
    printf("Thread:␣%d␣of␣%d\n",
      omp_get_thread_num(),
      omp_get_num_threads());
  }
}
```

Barriers in pThreads

- A *barrier* is a usefull synchronization method.
- When a thread reaches the barrier, it has to wait until all other threads reach the barrier before it can continue.
- In OpenMP it is simply `#pragma omp barrier`
- Barriers is an optional part of the Pthreads standard, the version on clustis does not implement them.
- So I've included an implementation (from the textbook, section 4.8.3) on the next slide.

Barriers with pThreads

```
//Shared variables
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
...
//Barrier
pthread_mutex_lock(&mutex);
counter++;
if(counter == thread_count){
    counter = 0;
    pthread_cond_broadcast(&cond_var);
} else{
    while(pthread_cond_wait(&cond_var, &mutex)!=0);
}
pthread_mutex_unlock(&mutex);
```