Royal Institute of Technology

Introduction to High-Performance Computing, DN2258

Final project, parallel search Draft

Sindri Magnússon, sindrim@kth.se, 871209-7156
Brynjar Smári Bjarnasson, bsbj@kth.se, 840824-4690

September 28, 2012

Contents

1 Introduction

In today's scientific environment the amount of data that is generated is increasing like never before. One consequence of these revolutionary time is the ability to process and work with the data in acceptable time. We plan to look further into one of the most important problem in the data universe which is searching.

embarrasly parallel

1.1 The data

To isolate the main problem and to be able to analyse the results in as derect way as possible we decided to use as simple data as we could think of. Anohter benefit of a simple data set is the ability of creating a data with a large number of rows. The data set is a file containing in every line an id and a sequence of six uniform random integers on the interval $[0, 9]$. All of the id's and all the sequences of random numbers have the same number of characters so the number of characters in each line is fixed.

Here below we se an example of the data with 1000 rows.

```
0000000 124523
0000001 561313
0000002 921836
    ...
9999999 196142
```

example_data.txt

The data we used for testing contains 10^9 lines and each line has 17 characters. Since a character is one byte the file is 17×10^9 bytes (17 Gbyte).

2 Theoretical performance estimation

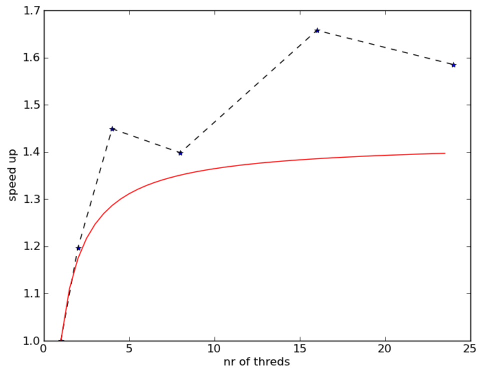
Since each comparison is independent we see that the problem of searching is actually embarrassingly parallel. However we need to read the data file from disk which is a sequential operation. So if we only look at the searching part of the algorithm then we would expect a linear speedup. But if we look at the whole thing and let P be the time percent of the search in the parallel part then from Amdahl's law we expect the speedup on N threads to be:

$$S_N = \frac{1}{(1-P) + \frac{P}{N}}$$

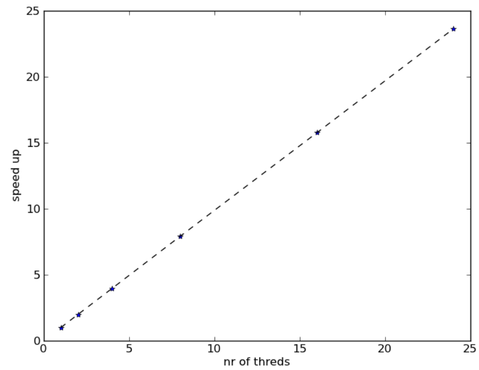
3 Results

3.1 OpemMP

Some graphics of the result we got. We have still not been able to run it on Lindgren but these results are from our personal computers.



(a) The red-line is the theoretical speed-up from Ahmads-law, the stars are show the acctual speed-up.



(b) Speed up of the search part

3.2 MPI

A Code

A.1 OpenMP

```c
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <omp.h>
#include <stdlib.h>
long long ae_load_file_to_memory(const char *filename, char **result)
{
  //  filename:   path to the file to read
  //  result:   pointer to character array that contains the content of the file.
  long long size = 0;
  FILE *f = fopen(filename, "rb");
  if (f == NULL)
  {
    *result = NULL;
    return -1; // -1 means file opening fail
  }
  fseek(f, 0, SEEK_END);
  size = ftell(f);
  fseek(f, 0, SEEK_SET);
  *result = (char *)malloc(size+1);
  if (size != fread(*result, sizeof(char), size, f))
  {
    free(*result);
    return -2; // -2 means file reading fail
  }
  fclose(f);
  (*result)[size] = 0;
  return size;
}

int read_file(char* input_file, char* key, int result_size_block, long long nr_lines, int line_size){
  //  input_file:        is direction to a data file.
  //  key:               the search string.
  //  result_size_block: is the block size of the array that
  //                     contains the resutls which contains the id
  //                     of the mathing data.
  //  nr_lines:          nr of lines in file.
  //  line_sie:          size of line in char
  //
  // It is assumed that all lines in input_file have the same length and are on the form: "0000001  123123\n" where first is an index and then the search values

  char line[line_size]; // line in file
  long long i;
  int result_size = result_size_block; // initualized result size
  int *result;
  int result_counter;
  int key_len = strlen(key);

  result = malloc(sizeof(*result)*result_size);
```

```
  #pragma omp parallel for private(i,line)
      shared(result,result_counter,result_size,key,result_size_block)
  for ( i = 0; i < nr_lines; i++ ) {
    strncpy(line,&input_file[i*line_size],line_size);

    line[line_size - 1] = '\0';
    // line contains a single line from the file. We compare the
        last part of the line to the search key.
    if ( strcmp(&line[line_size-(key_len+1)], key) == 0){
      // if we have used all spaces in result_counter we need to
          reallocate and increase the size.
      if (result_size == result_counter ){
        result_size = result_size + result_size_block;
        result = realloc(result , result_size*sizeof(*result) );
        if (result == NULL){
          printf("Error reallocating memory\n");
          exit(1);
        }
      }
      // we keep the line number where we found the key
      result[result_counter] = atoi(strtok(line,"\t\n"));
      result_counter++;
    }
  }
  return result_counter;
}


int main( int argc, const char* argv[] )
{
  double start,end;
  double dif;

  char* file_name = "../data/file.txt";
  char* result;
  char* search_key = "123123";

  long long nr_bytes;
  long long i;
  long long nr_lines;

  int string_size = strlen(search_key);
  int block_size;
  int read_count;
  int line_size;

  start = omp_get_wtime();
  nr_bytes = ae_load_file_to_memory(file_name,&result);
  end = omp_get_wtime();
  dif = end-start;
  printf("LoadFile: %f\n", dif);

  // assume each line in file is equally long. here we get the
      line size.
  for ( i=0 ; i<nr_bytes ; i++ ){
    if ( result[i]=='\n' ){
      line_size = i+1;
      break;
    }
  }

  //
```

```
  nr_lines = nr_bytes / line_size;

  block_size = 1.5*(nr_lines/pow(20,string_size));
  read_count;
  start = omp_get_wtime();
  read_count =
      read_file(result,search_key,block_size,nr_lines,line_size);
  end = omp_get_wtime();
  dif = end - start;
  printf("Search: %f\n",dif);
  printf("result found: %i\n", read_count);

}
```

../src/search_openmp.c