Royal Institute of Technology

Introduction to High-Performance Computing, DN2258

Final project, parallel search

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

September 23, 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.

```
000 367534
001 356216
002 915278
003 093367
    ...
999 849392
```

../data/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 sequential code 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

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';
```

```
56      // line contains a single line from the file. We compare the
            last part of the line to the search key.
57      if ( strcmp(&line[line_size-(key_len+1)], key) == 0){
58        // if we have used all spaces in result_counter we need to
              reallocate and increase the size.
59        if (result_size == result_counter ){
60          result_size = result_size + result_size_block;
61          result = realloc(result, result_size*sizeof(*result) );
62          if (result == NULL){
63            printf("Error reallocating memory\n");
64            exit(1);
65          }
66        }
67        // we keep the line number where we found the key
68        result[result_counter] = atoi(strtok(line,"\t\n"));
69        result_counter++;
70      }
71    }
72    return result_counter;
73  }
74
75
76  int main( int argc, const char* argv[] )
77  {
78    double start,end;
79    double dif;
80
81    char* file_name = "../data/file.txt";
82    char* result;
83    char* search_key = "123123";
84
85    long long nr_bytes;
86    long long i;
87    long long nr_lines;
88
89    int string_size = strlen(search_key);
90    int block_size;
91    int read_count;
92    int line_size;
93
94    start = omp_get_wtime();
95    nr_bytes = ae_load_file_to_memory(file_name,&result);
96    end = omp_get_wtime();
97    dif = end-start;
98    printf("LoadFile: %f\n", dif);
99
100   // assume each line in file is equally long. here we get the
          line size.
101   for ( i=0 ; i<nr_bytes ; i++ ){
102     if ( result[i]=='\n' ){
103       line_size = i+1;
104       break;
105     }
106   }
107
108   //
109   nr_lines = nr_bytes / line_size;
110
111   block_size = 1.5*(nr_lines/pow(20,string_size));
112   read_count;
113   start = omp_get_wtime();
```

```
114     read_count =
            read_file(result,search_key,block_size,nr_lines,line_size);
115     end = omp_get_wtime();
116     dif = end - start;
117     printf("Search: %f\n",dif);
118     printf("result found: %i\n", read_count);
119
120 }
```

../src/search_openmp.c