TDT4200 Parallel Computing
Problem Set 3

Stian Hvatum (hvatum)
MTDT

September 23, 2012

Contents

Task 1 Theory

a) Reduce branches

1. If we are using if's with elsif's, we can put the most common case first, so that we usually only do one branch. Code Listing a) 1.
2. Some conditional statements can be sustituted with equals that are not conditional. Code listing a) 2.
3. We can do loop-unrolling. If we know number of repetitions is known to be a multiple of N>1, this can easily be done. Code listing a) 3.

a) 1.

```
for (int i = 0; i < N; i++)
{
  if (i == N/2 + 1)
  {
    do_seldome_stuff();
  }
  else if (i % N/4 == 1)
  {
    do_stuff();
  }
  else if (i % 2 == 0)
  {
    do_common_stuff();
  }
}
```

Will be faster if we put the common stuff first (less checking). This implies that the conditions are exclusive.

```
for (int i = 0; i < N; i++)
{
  if(i % 2 == 0)
  {
    do_common_stuff();
  }
  else if(i == N/2 + 1)
  {
    do_seldome_stuff();
  }
// This condition will be more common,
// but have overlaping conditions with previous statement
  else if(i % N/4 == 1)
  {
    do_stuff();
  }
}
```

a) 2.

We can substitute this static conditional assignment with some clever logic statement.

```
  if (n % 2 == 0)
    a[n] = 2;
  else
    a[n] = 0;
```

```
    a[n] = (n & 0b) << 1;
```

a) 3.

In this example, N/2 is allways a multiple of 2 (since we are dividing in half, which gives an even number).

```
for (int i = 0; i < N/2; i++)
{
    do_stuff(i);
}
```

Since i is known to be a multiple of 2, we can unroll the loop at least twice, like shown here.

```
for (int i = 0; i < N/2; i++)
{
    do_stuff(i);
    do_stuff(++i);
}
```

b) Improve switches

We can improve the performance of a long switch by placing the most common case first. We can also splitt it into nested switch statements or nest inside the else-block of an if-else-statement where the true-block of the if-statement contains the most frequent cases.

Example: Let us assume *answer()*'s most common return value is 42, but can return a lot of other values, then this statement will have improved performance over a regular switch statement with sequentially ordered cases. Of cource, if this switch is converted into a jump-table, or 42 was the first case AND the switch was converted into if-else-statements by the compiler, this optimization won't matter that much, as it would be implicit.

```
int a = answer();
if (a == 42)
    hurray();
else
    switch (a)
    {
    case 1:
        ...
        break;
    case 2:
        ...
        break;
    ...
    ...
    default:
        blah();
        break;
    }
```

c) Reduce cache misses

1. One can change loops to operate on blocks instead of rows of data. This may help to reduce directe-mapped cache conflicts.
2. Two arrays that are concurrently accessed can be merged with an array of structs, where the struct contains one element from each original array. This helps if both elements are needed, and the original arrays conflicts in cache.
3. If we are looping though a 2-dim array, and we have the first index looped over at the inner loop, we can switch the order of the loops, such that the inner loop controlls the second index. This helps since the first index effects a stride of the size of the inner array (or random if malloc'ed in C), while the second index gives a stride of 1 element.

d) Implementing SIMD

1. As loop unrolling

In its simplest form, vectorization can be used as a form of loop unrolling, as we do two and two double value arithmetics at the same time. This will speed up the loop, and is fairly simple. Disadvantages is that vectorization in general may increase overhead in flow control of the program[1].

2. For complex artithmetics

Intel has a lot of docs on how to do complex artithmetics with SSSE3. Eg. can you do complex multiplication in just a few steps, while the C99 Complex library implements a rather large routine for this purpose. SSSE3 also has an *addsub* instruction which is ideal for the last step of complex multiplication, where we add and subtract in the same operation. This is theoretically much faster, but code readability are greatly reduced.

3. General math

When we are facing math where multiple steps involve the same values, we can use SIMD to speed up the process. This might be tricky and hard to read, but might also speed up the program.

4. Let the compiler do it

When specifyng -O3 -march=native, and your CPU supports SSSE3, GCC will use it where it finds it usefull.

Task 2 Code optimization

Task 2.1 Optimizing Taylor series for Sinus approximation

1. Less branches

One obvious optimization we can do[2], is that the *factorial* could be done iterative with a loop, instead of recursive calls. This saves a lot of function calls, which contains a lot

[1] `http://en.wikipedia.org/wiki/Vectorization_(parallel_computing)#Reducing_vectorization_overhead_in_the_presence_of_control_flow`

[2] At first look, I thought the function *factorial*did not need to return a double, but since its values are too large, integers would overflow, thus we need the double value.

of stack- and register-maintenance. The *slow_sin* is doing 100 branches with different outcome at each repetition, this is bad for branch prediction. We can modify the code to instead of checking and doing two different operations, we can do the same thing each time. On my computer, these two optimizations gives the function some more than double speed[3].

```
/* calculate n! */
double factorial(int n)
{
    double answer = n;
    for (int i = n-1; i > 1; i--)
        answer *= i;
    return answer;
}
/* calculate sin(x) using Taylor series expansion */
double slow_sin(double x)
{
    double r=0; int i;
    for(i=0;i<100;i++)
    {
    int sign = (i & 1) * -2 + 1;
    r += pow(x, i*2+1) / factorial(i*2+1) * sign;
    }
return r;
}
```

2. Loop unrolling

Instead of doing the sign test for + or -, we could simply unroll the loop, being shure each even step is + and each odd step is -. We could unroll the loop as many times we want, effect is determined by cache size, since we want to keep the inner instructions in cache, instead of fetching them from a long distance for each entry. With this optimization, along with the optimized *factorial*, speed is about 3x of the original[4].

```

/* calculate sin(x) using Taylor series expansion */
double slow_sin(double x)
{
    double r=0; int i;
    for(i=0;i<100;i++)
    {
    r += pow(x, i*2+1) / factorial(i*2+1);
    i++;
    r -= pow(x, i*2+1) / factorial(i*2+1);
    }
return r;
}
```

3. Using threads

Since the result is not dependent of the order of the partial results, we can use pthreads, MPI or similar to split the problem into smaller problems. In practice, this slowed things down at my computer.

[3]tested with *time* and setting the sine-loop to 10000 instead of 100. Results was new function using 0.4 sec, old function using 0.9 sec. (Intel Core i7 930@ 2.8GHz, GCC -O0)

[4]0.32 sec. on my machine, same configuration as previous test

```
...
/* calculate sin(x) using Taylor series expansion */
double slow_sin_new(double x, int start, int stride)
{
    double r=0; int i;
    for(i=start;i<10000;i+stride)
    {
    r += pow(x, i*2+1) / factorial_new(i*2+1);
    i++;
    r -= pow(x, i*2+1) / factorial_new(i*2+1);
    }
return r;
}
...
void* start_thread(void* rank)
{
    double* ans = malloc(sizeof(double));
    *ans = slow_sin_new(input, (int)rank * 2, 15);
    pthread_exit((void*)ans);
}
...
int main()
{
    pthread_t threads[8];
    for (int i = 0; i < 8; i++)
        pthread_create(&threads[i], NULL, start_thread, (void*)i);
    for (int i = 0; i < 8; i++)
    {
        double val;
        pthread_join(threads[i], (void*)&val);
        r += val;
   }
   printf(”%f\n”, r);
}
```

4. Using SSSE3

So instead of threads, we can use vectorization via SSSE3 or equal instruction set. This function uses pow and factorial, but we can also implement them inline with SSSE3 instructions. Speedup was disapointingly only double from original[5], but this is most likely since I am calling pow and factorial and stores the result into special registers inbetween the other SSSE3 instructions. This can most likely be avoided by inlining these two functions with SSSE3 instructions[6].

```
/* calculate sin(x) using Taylor series expansion */
double slow_sin_new(double x)
{
    int i; double r = 0;
    __m128d factor, res, part, cx, nx;
    double xarr[2];
    res = _mm_set_pd(0,0);

    for(i=0;i<10000;i+=2)
    {
    factor = _mm_set_pd(factorial(i*2+1), factorial(i+2+3));
    part = _mm_set_pd(pow(x, i*2+1), pow(x, i+2+3));
    res = _mm_addsub_pd(res, part);
    }
    _mm_storeu_pd(xarr, res);
    return xarr[0] + xarr[1];
return r;
}
```

[5]0.45 seconds

[6]Pow can be sustituted with a multiplication with x for each step, and factorial can be a multiplication with i at each step.

Task 2.2 Loops and indexed access optimizations

As explained in Task 1 c) 3., this code access the a-array with the first index in the inner loop, which gives a large stride in the accessed data. This gives the cache a hard time keeping up, since accessed data is not aligned in memory the same way it is accessed. Set loop over j as outer loop, and over i as inner loop in order to speed up function. This loop could also benefit from being unrolled and/or vectorized.