Exercise for Course

Parallel High-Performance Computing

Dr. S. Lang

Return: 5. November 2015 at the beginning of the exercise or earlier

---

Task 3 C++ Introduction: Debugging (5 points)

The following program shall add all natural numbers from given $a \in \mathbb{N}$ to $b \in \mathbb{N}$:

```
1 #include <iostream>
2
3 // sums all natural numbers in [a, b]
4 int sum(int a , int b)
5 {
6   int result;
7   for (int i=a; i<=b ; i++)
8   {
9     int result = result + i;
10  }
11
12  return 0 ;
13 }
14
15 int main()
16 {
17  std::cout << sum(1, 10) << std::endl;
18  return 0 ;
19 }
```

Although the progrmm is syntactically correct, it calculates the wrong result. Find the errors and correct them without modifying the program purpose. What do you have to change, if further natural numbers shall be added, but for the number domain applies: $a, b \in \mathbb{R}$.
Tip: If you just want to try compile the program on the computer in a file `debug.cc`, the C++ compiler initiated with the option `-Wall` gives you hints to erroneous code. In the command line use for compilation: `g++ -Wall debug.cc`.

Task 4 Measurement von MFLOPS (15 points)

In this task we want to measure for two numerical applications, how many arithmetic operations per second are achievable on our pool machines. Herefore we implement the following mathematical operations:

1. *Matrix Multiplication.*
   Given two matrices $A, B \in \mathbb{R}^{n\times n}$. Then the matrix product $C = AB$ is again a matrix $C \in \mathbb{R}^{n\times n}$ with the entries:
   $$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

2. *Gauß-Seidel 2d.*
   Given a domain in d dimensions defined by
   $$\Omega_n^d = \left\{ (i_0, \ldots, i_{d-1}) \in \mathbb{Z}^d \mid \forall\ 0 \le k < d, 0 \le i_k < n \right\}.$$
   In $2D$ this would be for example a mesh with n^2 points. We choose a mesh with equidistant points, therefore $\Omega = [0, n-1]^2$. On this mesh a mesh function $u^m : \Omega_n^2 \to \mathbb{R}$ is defined. For this the iteration procedure
   $$u^{m+1}(i,j) = \frac{1}{4}\left\{ u^{m+1}(i-1,j) + u^{m+1}(i,j-1) + u^m(i,j+1) + u^m(i+1,j) \right\} \quad (i,j) \in [1, n-1]^2$$
   defines the so-called Gauss-Seidel iteration.

Subtask (a) (5 points)

Implement the matrix multiplication in the programming language C/C++ and use for storing the matrices an arbitrary data structure of your choice (e.g. one-dimensional or two-dimensional arrays or `std::vector`). Determine the number of floating point operations and calculate herefrom and of the measured runtime the speed of the program in „Million FLoating point OPerations per Second "(MFLOPS).

For time measurement you can choose the functions provided in `timer.h`. You can find the header file on the lecture homepage, hints for usage at the end of the exercise sheet. Be careful to choose the problem size n in a size, that the time measurement is not influenced by the measurement error, in the pool about $n \geq 1000$. Initialise the arrays with meaningful data (not 0.0), e. g. $u(i, j) = i + j$.

Compile the program with maximal optimization level. For the `GNU C/C++` compiler is e.g. `-O3 -funroll-loops` recommendable.

Visualize all results in graphical form, MFLOPS over problem size n. Discuss the curvature of the graph, especially why and when the MFLOPs rate decreases. For graphics generation the program `gnuplot`, also installed in the pool, is recommended.

Subtask (b) (5 points)

Repeat the investigations of subtask (a) for the Gauss-Seidel scheme.

Subtask (c) (5 points)

Introduce for the matrix multiplication a better cache usage by tiling as proposed in the lecture, and determine the acceleration for different blocking sizes.

Remarks for Time Measurement

The different timings

During time measurement on the computer the problem arises that the time a program needs depends on the load of the whole system. Are there many processes active a single process has only few time and runs accordingly long in wall clock time. The processor time instead measures how many seconds the processor has been active executing the program. The clock tics as long as the program runs and when the process is idle it waits.

timer.h

In the header file `timer.h` there are several auxiliary functions implemented, that can read the used processor time. There are three functions available:

- `void reset_timer(struct timeval* timer)`: reset/initialise counter.
- `double get_timer(struct timeval timer)`: read used seconds.
- `void print_timer(struct timeval timer)`: print used seconds.

Example

```
1 #include "timer.h"       // Header file for time measurement
2
3 int main()
4 {
5   struct timeval timer; // variable for time measurement
6   reset_timer(&timer);  // reset and initialize counter
7   ...                   // Do something that needs time
8   print_timer(timer);   // print counter
9 }
```

More about the internal time measurement can be read in the manpage for `getrusage (2)`.