COMP60611 Laboratory Exercise 4: Adaptive Quadrature Solutions

Team:

Gold- Group 5

Team Members:

Christos Theofilou

Zijiang Yang

Yini Zhang

Kai Shen

Question 1

Table 1 Timing Results of aquad.c

Function	func1	func2	Func3
Timing Result(seconds)	0.0450	0.4478	4.4748
Standard Deviation	2.318E-05	8.589E-05	0.0002

The results for the functions are 0.333333. The numbers are based on an average of 3 runs.

Question 2

```
double stackops(int stackpointer, Interval *stack, double (*func)(double)) {
//#include <math.h>

  double result, abserror, l, r, m, est1, est2, eps;
  int stacksize = 1000;
  int condition = 0;
  int gotwork;
  result = 0.0;
#pragma omp parallel default(none)\
                private(l,r,m,est1,est2,abserror,eps,gotwork)\
                shared(stackpointer,stack,func,result,condition,stacksize)

  while (stackpointer >= 1 || condition >= 1) {
#pragma omp critical
{
    // pop next interval off stack
    gotwork = 0;
    if (stackpointer >= 1) {
    gotwork = 1;
    condition++;
    l = stack[stackpointer].left;
    r = stack[stackpointer].right;
    eps = stack[stackpointer].epsilon;
    stackpointer--;
    }
}

    // compute estimates
    if (gotwork == 1) {

    m = 0.5 * (l + r);
    est1 = 0.5 * (r - l) * (func(l) + func(r));
    est2 = 0.5 * ((m - l) * (func(l) + func(m)) + (r - m) * (func(m) + func(r)));
    abserror = fabs(est2 - est1) / 3.0;

    // check for desired accuracy: push both halves onto
    // the stack if not accurate enough
#pragma omp critical
{
    if (abserror <= eps) {
      result += est2;

    } else {
      if (stackpointer+2 > stacksize) {
        printf("Stack too small, track doubling stacksize");
        exit(0);
      }
```

```
        stackpointer++;
        stack[stackpointer].left = l;
        stack[stackpointer].right = m;
        stack[stackpointer].epsilon = eps * 0.5;

        stackpointer++;
        stack[stackpointer].left = m;
        stack[stackpointer].right = r;
        stack[stackpointer].epsilon = eps * 0.5;
      }
      condition--;
  }
  }
  }
   return result;
  }
```

Question 3

Table 2 Timing results of different functions on different cores

Cores	1	2	3	4	6	8	12	24	36
Func1	0.0451	0.0232	0.0157	0.0118	0.0081	0.0063	0.0049	0.0206	0.0366
Func2	0.4478	0.2247	0.1505	0.1127	0.0761	0.0568	0.0394	0.0219	0.0177
Func3	4.4859	2.2428	1.5057	1.1241	0.7591	0.5656	0.3877	0.2016	0.1427

The results for the functions are 0.333333. The numbers are based on an average of 3 runs.

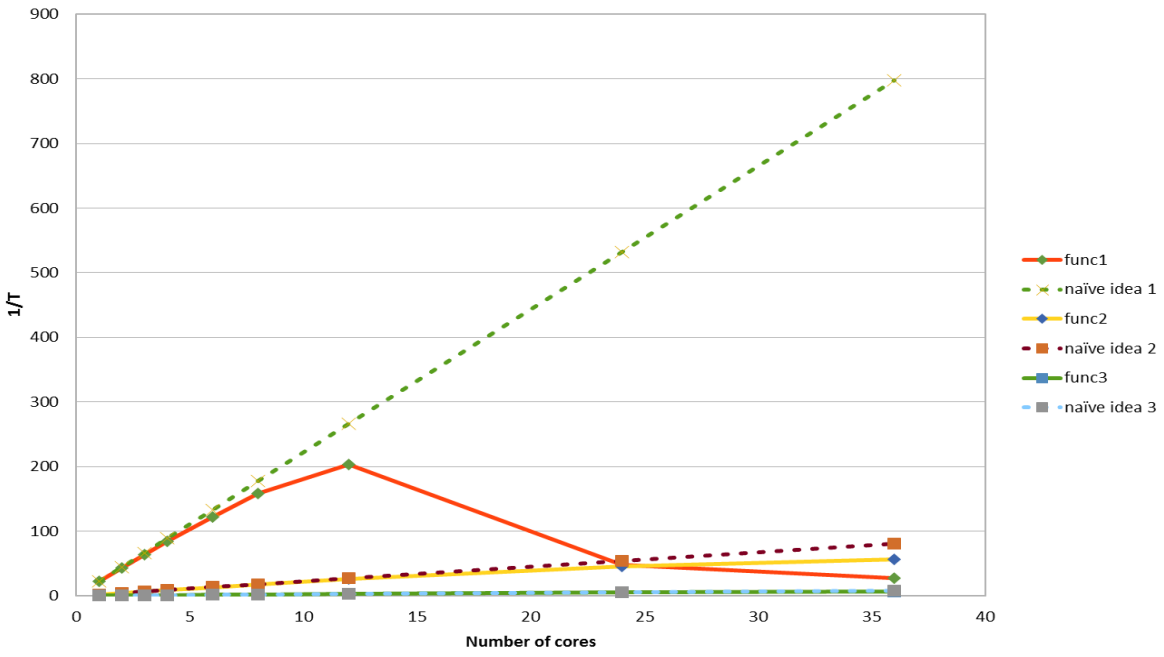

Figure 1 Temporal performance of different functions on different cores

We can observe that the temporal performance of func1 keeps increasing but when running on 24 and 36 cores its efficiency drops significantly, which is quite different from its naive ideal. The performance of func2 and func3 are slightly lower than their naive ideal respectively. What cannot be ignored is that the temporal performance of func1 is even lower than that of func2 when running on 36 cores.

Efficiency equals T_s/pT_p and we can compute the efficiency of different functions on different cores as well.

Table 3 Efficiency of different functions on different cores

Cores	1	2	3	4	6	8	12	24	36
Func1	1	0.9694	0.9571	0.9543	0.9197	0.8911	0.7622	0.0910	0.0343
Func2	1	0.9962	0.9913	0.9930	0.9799	0.9853	0.9472	0.8518	0.7003
Func3	1	1	0.9931	0.9977	0.9849	0.9913	0.9640	0.9270	0.9731

The efficiency graph is as follows:

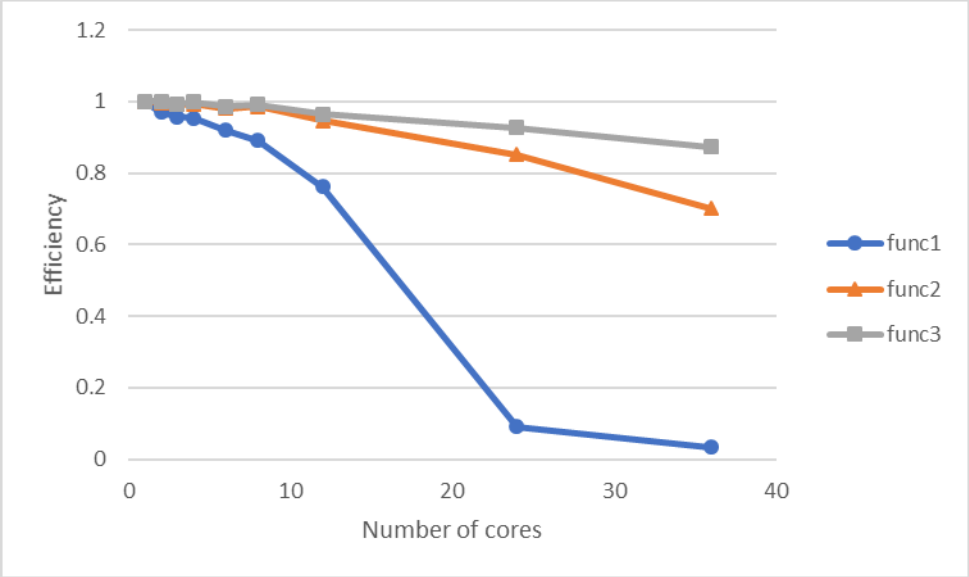

Figure 2 Efficiency of different functions on different cores

From the graph we can observe that the efficiency of all these three functions keep dropping when running on increasing number of cores and apparently the efficiency of func1 drops fastest, ending at 0.0343, which is a quite small number.

We thought the principal source of overheads in func1 is scheduling overhead. The more cores are, the more significant scheduling overhead is. It increases to a large proportion since the runtime of func1 is shorter and as a result, the temporal performance of func1 drops significantly when func1 runs on 24 and 36 cores(the performance is even lower than func2 when running on 36 cores).

The main source of overheads in func2 and func3 is synchronisation. When the number of cores increases, more threads will be created and called, making synchronisation an important part to ensure correct results (since there are some shared variables like stackpointer and result) and this costs some time.