Introduction to Supercomputing

TMA4280 · Problem set 2

Exercise 1. The previous supercomputer at NTNU, *Njord*, was based on the POWER5 dual-core chip, which had cache sizes:

- L1: 32 kB,
- L2: 1.875 MB,
- L3: 36 MB.

Assuming double precision, how many floating point numbers can fit in each cache? What is the dimension of the largest square matrix that can fit in each cache?

Solution. One double precision floating point number needs eight bytes of storage. Therefore the number N_i of doubles in cache Li is

$$\begin{aligned} N_1 &= \frac{32 \cdot 1024}{8} = 4096, \\ N_2 &= \frac{1.875 \cdot 1024^2}{8} = 245760, \\ N_3 &= \frac{36 \cdot 1024^2}{8} = 4718592. \end{aligned}$$

Since a square matrix of size n must store n^2 numbers, the maximal n_i is the square root of the above numbers,

$$n_1 = 64, \qquad n_2 = 495, \qquad n_3 = 2172.$$

Exercise 2. What limits are there to the speed of electronic circuits?

What is the maximum distance a memory unit could be from an arithmetic unit (in a processor), and still allow a memory access time of 100 ps? (1 ps = 10^{-12} s.)

Solution. The speed of electronic circuits is limited by the speed of light,

$$c = 299\,792\,458\,\mathrm{m\,s^{-1}}.$$

The memory access time t over a distance d is then bounded as

$$t \geq \frac{d}{c} \implies d \leq ct = 299\,792\,458\,\mathrm{m\,s^{-1}} \cdot 10^{-12}\,\mathrm{s} \approx 300\,\mu\mathrm{m}.$$

So such a memory unit cannot be more than 0.3 mm from the arithmetic unit.

Exercise 3. Assume that we have a scalar c and two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of length n. We consider three types of linear algebra operations:

- Add c to all elements in $\boldsymbol{a}$.
- Add $\boldsymbol{a}$ and $\boldsymbol{b}$ and store the result in $\boldsymbol{a}$.
- Multiply $\boldsymbol{b}$ with c and add $\boldsymbol{a}$ to the resulting vector. Store the result in $\boldsymbol{a}$.

Below we show three Fortran subroutines implementing these operations, however, the particular choice of programming language doesn't matter.

```fortran
subroutine op1(a,c,n)
  real a(n),c
  do i=1,n
    a(i) = a(i) + c
  end do
  return
end

subroutine op2(a,b,n)
  real a(n),b(n)
  do i=1,n
    a(i) = a(i) + b(i)
  end do
  return
end

subroutine op3(a,b,n)
  real a(n),b(n),c
  do i=1,n
    a(i) = a(i) + c*b(i)
  end do
  return
end
```

These three routines were tested on an older supercomputer at NTNU around the year 2000. In order to check the single-processor performance, a test program was run which called each of these routines many times. The mean number of floating-point operations for different vector lengths n was then computed. The results are summarized in the following figure.

Explain these results. In particular,

- Why does the speed increase with n for small n, followed by being largely independent of n?
- Why is there a sudden drop at a particular vector length?

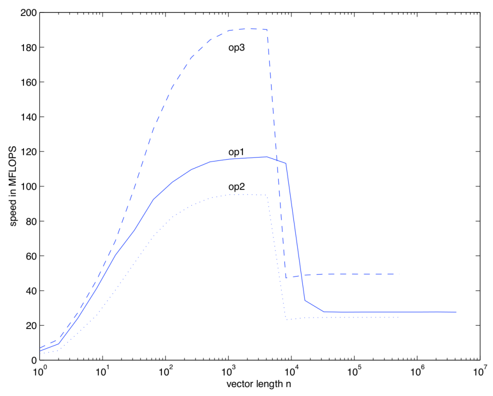

Figure 0.1: Performance on a single alpha chip on a Cray T3E supercomputer.

- Why does this drop happen sooner for operations 2 and 3 than for operation 1? (At $n = 4096$ and $n = 8192$ respectively. This is difficult to see in the figure.)
- Why does operation 3 run faster than operation 1, and why does operation 2 run slower still?

Solution. There are a number of operations involved that produces *overhead*. In particular, we might expect the following sources of overhead:

- Filling the processor pipeline. After a number of cycles (determined by how many it takes to compute one element of the resulting vector) the pipeline is "full" and the processor is operating at maximum speed.
- The initial fetching of the vectors from memory to cache.
- The loading of the actual program into memory (this depends on how the results were timed, information that is unfortunately lost).
- Subroutine calls.

For very short vectors, we expect that most of the time spent by the program is involved with these operations, and comparatively less time spent doing actual work. For long vectors, the maximal processor performance will dominate. This is why the program exhibits low performance on short vectors.

The sudden drop for a particular vector length is due to the fact that the vector can no longer fit entirely in the processor (L1) cache, and that vector elements must be fetched one by one from a memory unit lower in the hierarchy.

This happens later for operation 1 because this operations does not need to access the vector $\boldsymbol{b}$. Since the cache can be totally dedicated to storing $\boldsymbol{a}$, it can effectively handle twice the vector length before running into the cache limitation problem.

Operation 3 runs faster than the other operations, because it does both a multiplication and an addition at each iteration. We can conclude that the processor in question is capable of doing these simultaneously. Even though operation 3 outputs as many *vector elements* per second as operation 2, it still performs more floating point operations per element.

Operation 1 runs faster than operation 2 because it only needs to fetch one vector element per iteration. The constant c can be stored in a register on the CPU, instead. In comparison, operation 2 must fetch two numbers from cache every iteration.

Exercise 4. Consider the vector operation $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$, where all vectors are of length n. One way to implement this operation is:

```
for i=1,n
  c(i) = a(i) + b(i)
end
```

To maximize perfomance, we want to ensure that the vector elements are stored interleaved in memory:

$$\ldots, a_i, b_i, c_i, a_{i+1}, b_{i+1}, c_{i+1}, \ldots$$

- Why could this storage scheme be advantageous?
- How would you realize this in C and/or in Fortran?
- Do you think this scheme would pay off compared to the extra implementation effort?

Solution. Storing data in this way follows the rule of storing data together that is used together. In particular, it can avoid cache trashing.

One implementation might be as follows. In C, allocate memory for a matrix with dimension $n \times 3$. Then initialize it, and compute.

```
double M[n][3];
for (size_t i = 0; i < n; i++) {
  M[i][0] = a[i];
  M[i][1] = b[i];
  M[i][2] = c[i];  // Not strictly necessary in this case
}
for (size_t i = 0; i < n; i++) {
  M[i][2] = M[i][0] + M[i][1];
}
```

C uses a *row-major* storage scheme: elements in the same row in the matrix are stored together (i.e. the *last* index is the one that varies the quickest). Thus, this will access entries in memory order.

Fortran uses the opposite storage scheme: it is *column=major*. In Fortran, elements in the same column are stored together, and it is the *first* index that varies the quickest. Thus, we effectively need to transpose the matrix to get the same effect.

```
real M(3,n)
for i = 1,n
  M(1,i) = a(i)
  M(2,i) = b(i)
  M(3,i) = c(i)  ! Not strictly necessary in this case
end
for i = 1,n
  M(3,i) = M(1,i) + M(2,i)
end
```

It is unlikely that a rearrangement like this would be worth the trouble, at least in this case. The chance of cache trashing is not very high in the first place, and with more advanced cache strategies (associative caches), it will probably be a non-issue. Moreover, planning and rewriting a program to use memory strategies like this is not trivial.

Exercise 5. Implement a program in C or Fortran which performs the following three different operations:

$$\begin{aligned} \boldsymbol{x} &= \boldsymbol{a} + \gamma \boldsymbol{b} \\ \boldsymbol{y} &= \boldsymbol{a} + \boldsymbol{A}\boldsymbol{b} \\ \alpha &= \boldsymbol{x}^\mathsf{T} \boldsymbol{y} \end{aligned}$$

You can use any compatible vectors and matrices you see fit (e.g. random ones). The constant γ should be read from the command line.

Solution. In C:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double random_double()
{
    return (double)rand() / RAND_MAX;
}

int main(int argc, char **argv)
{
    if (argc < 3) {
        printf("Usage: ex2.5 n g\n");
        printf("  n = vector/matrix size\n");
```

```c
        printf("  g = constant gamma\n");
        return 1;
    }

    // Collect input arguments
    int n = atoi(argv[1]);
    double gamma = atof(argv[2]);

    // Allocate vectors and matrices (this is valid C99)
    double a[n], b[n], A[n][n], x[n], y[n];

    // Fill a, b, A with random data in [0,1]
    srand(time(NULL));
    for (size_t i = 0; i < n; i++) {
        a[i] = random_double();
        b[i] = random_double();
        for (size_t j = 0; j < n; j++)
            A[i][j] = random_double();
    }

    // Compute x
    for (size_t i = 0; i < n; i++)
        x[i] = a[i] + gamma * b[i];

    // Compute y
    for (size_t i = 0; i < n; i++) {
        y[i] = a[i];
        for (size_t j = 0; j < n; j++)
            y[i] += A[i][j] * b[j];
    }

    // Compute alpha
    double alpha = 0.0;
    for (size_t i = 0; i < n; i++)
        alpha += x[i] * y[i];

    // Finalize
    printf("alpha = %f\n", alpha);
    return 0;
}
```