OpenMP Array Timing

Dr. Christopher Marcotte — *Durham University*

Since this might be the first time some of you have seen *C* and thus been compelled to care about details like column ordering... Let's do a simple timing exercise.

Given three arrays, `double A[N][N][N], B[N][N][N], C[N][N][N]`, we add each element in `A` to the corresponding element in `B` and store the result in the corresponding element of `C`: i.e. $C_{ijk} = A_{ijk} + B_{ijk}$.

Timing

We time how long it takes to iterate over all N^3 elements, depending on how the arrays are accessed.

Warning

If you take `N` much larger than `64`, you will likely hit stack limit size, leading to a segmentation fault. Put `ulimit -s unlimited` at the command line prompt to remove this limitation.

Hint

You may wonder how to write a function which performs this timing experiment. Here is a two-index example function you can build on:

timing.c

```c
double time_ij(int N){

  int i, j;

  double A[N][N];
  double B[N][N];
  double C[N][N];

  clock_t t0 = clock();
  for(i=0; i<N; i++){
    for (j=0; j<N; j++){
      C[i][j] = A[i][j] + B[i][j];
    }
  }
  double t1 = ((double)clock() - t0) / CLOCKS_PER_SEC;
  return t1;
}
```

Serial

First, we should think about this (serial) problem – for each element, we load two values from distinct regions of memory, do a trivial computation with them, and store them in a different region of memory. Where, precisely, we load and store from and to is itself a computation. The linear index I from double index $(i, j) \in [0, N_1) \times [0, N_2)$ is $I(i, j) = iN_2 + j$, for column-major ordering, like *C*[1].

Exercise

[1] For row-major ordering $I(i, j) = i + jN_1$.

For three indices, how many loop orderings are there? What is $I(i, j, k)$ for $(i, j, k) \in [0, N_1) \times [0, N_2) \times [0, N_3)$?

Write a small function which performs the addition $C_{ijk} = A_{ijk} + B_{ijk}$, and returns a `double` time for the execution (do not time the allocations).

Wrap each timing call in a loop so you can get an average of the timings across `M` calls. Make `N` and `M` command line arguments.

Which loop ordering is fastest? What is the ranking for the different loop orderings? Is the ranking different on Hamilton than on your machine? What other factors influence the timing?

Challenge

If you've be using *C* for some time, you might consider the multi-dimensional array pattern being used here somewhat aberrant – "why wouldn't you just allocate `double A[N*N]` and avoid all this complexity?" I can hear faintly in the distance. For small `N` this is certainly viable, one only pays the price in manually computing the index offset (i.e. managing the mapping $I(i, j, k)$). Try the timing exercise with a single integer index `int i` – with the one-dimensional array analogues `double A[NNN], B[NNN], C[NNN]`. Is it faster? Is it convenient? Is it *necessary*?[2]

Parallel

Note that for $C_{ijk} = A_{ijk} + B_{ijk}$ there are no dependencies between the indices – every (i, j, k) computation can, in principle, be done independently.

The relevant question is whether and how to parallelize this task, and to quantify the effect(s) of doing so. As a baseline, you should expect to parallelize the task using a shared-memory approach, i.e. OpenMP, because the problem is too small to benefit from multiple memory spaces, and these present some significant conceptual and programmatic hurdles. In the following, use your understanding to implement a parallel update of the three-index array `C`, similarly to the serial version we've considered thus far.

Exercise

How would we parallelise these (i, j, k) loops using OpenMP – what form of pragma should you use?

How many threads *could* we utilize effectively with $N = 64$? How much work would each thread be responsible for?

Try to parallelise the code using `#pragma omp parallel for`. How many threads are utilized? How many *could* be utilized? How do you *know*?

Try adding a `collapse(3)` clause to the `parallel for`. How many threads are utilized *now*? How many *could* be utilized? How do you *know*?

Should you continue to use `clock()` for timing?

Challenge

Given that we just computed *M* timing results to get a single value (the mean time per iteration), it might be worth computing the variance (or standard deviation) of that result simultaneously. The idea is, for each

[2] Note that this is trivial in Julia: all arrays are linear-indexable. Similarly, in *C* we work with pointers and thus we need only know the (linear) offset to an index to access that memory.

iteration $1 \leq k \leq M$, compute (t_k, S_k) from (t_{k-1}, S_{k-1}). Due to potentially catastrophic cancellation errors, we do so with the following algorithm due to Welford:

$$\mu_{k+1} = \frac{k}{k+1}\mu_k + \frac{1}{k+1}t_k, S_{k+1} = S_k + \frac{k}{k+1}(t_{k+1} - \mu_k)^2,$$

where the variance of $t_{1\ldots k}$ is just $k^{-1}S_k$ and $\mu_k = k^{-1}\sum_{i=1}^{k} t_k$ is the mean. This might be a fun exercise in numerical computing.

Summary

In this exercise we've considered the cost of traversing multi-dimensional data in a multi-index fashion, i.e. the cost of computing $C_{ijk} \leftarrow A_{ijk} + B_{ijk}$ by iterating over the three-index tuple $(i, j, k) \in [0, N)^3$.

We saw that performance is remarkably different depending on the loop index ordering for bandwidth-bound problems. We saw that the performance improves somewhat when using multiple threads by parallelizing the loop work using `#pragma omp parallel for`, and that by further appending `collapse(n)` we improve the assignment of work as the number of threads increases, permitting larger N.

Of course, the dimensionality of the *data* need not be reflected in our *data structure*, and indeed there are often very good reasons for choosing to make these different.

Aims

- Introduction to multidimensional array access performance
- Introduction to parallel loops with OpenMP
- Introduction to the `collapse(n)` OpenMP clause
- Introduction to some basic numerical computing techniques