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*[*].

Exercise

[*] 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?

Solution

For n indices, there are n choices for the first index, $n-1$ choices for the second, ..., so the total number of choices is $n! \equiv n \cdot n-1 \cdot \ldots \cdot 1$. For $n=3$, that yields 6 permutations of $\{1,2,3\}$ of length 3.

Note that $I(i,j) = N_2 I(i) + j = N_2 i + j$; similarly, $I(i,j,k) = N_3 I(i,j) + k = N_3 N_2 i + N_3 j + k$. Adding a new dimension d to the index is thus as simple as scaling the first $d-1$ dimensions by the d-th size, and adding the new index. This is really simple for *C*-languages. For column-major languages like Fortran and Julia, it's the reverse: each new dimension gets shifted by the product of the sizes of the prior dimensions: $I(i,j,k) = i + N_1 j + N_1 N_2 k$.

This is a rather simple program, but annoyingly long without metaprogramming; a partial listing of the timing function is below:

timing-solution.c

```c
42 double time_ijk(int N){
43
44   int i, j, k;
45
46   double A[N][N][N];
47   double B[N][N][N];
48   double C[N][N][N];
49
50   clock_t t0 = clock();
51   for(i=0; i<N; i++){
52     for (j=0; j<N; j++){
53       for (k=0; k<N; k++){
54         C[i][j][k] = A[i][j][k] + B[i][j][k];
55       }
56     }
57   }
58   double t1 = ((double)clock() - t0) / CLOCKS_PER_SEC;
59   return t1;
60 }
```

With the corresponding portion of `main(...)`:

timing-solution.c

```c
163 int main(int argc, char **argv){
164
165   int N = atoi(argv[1]);
166   int M = atoi(argv[2]);
```

```
167   double t;
168
169   printf("3D version:\n");
170
171   t=0.0;
172   for (int m=0; m < M; m++){
173     t += time_ijk(N);
174   }
175   printf("\tijk: %1.16e s\n",t/M);
```

Here are results on my MacBook Pro (Apple M1 Pro), for outer → inner order[†]:

c_timing.txt

```
3D version:
  ijk: 8.3599609374999798e-05 s
  jik: 1.1076171874999944e-04 s
  kij: 6.4437011718749854e-04 s
  kji: 1.3183330078125023e-03 s
  ikj: 2.1346386718749721e-04 s
  jki: 1.9664707031250084e-03 s
2D version:
   ij: 1.0107421875000122e-06 s
   ji: 9.6386718750001443e-07 s
```

f_timing.txt

```
3D version:
  ijk:    1.3992850326758344E-003  s
  jik:    2.0666131749749184E-003  s
  kij:    3.0821701511740685E-004  s
  kji:    1.5980890020728111E-004  s
  ikj:    7.0454692468047142E-004  s
  jki:    1.7376337200403214E-004  s
2D version:
  ij:    2.6319175958633423E-006  s
  ji:    9.9278986454010010E-007  s
```

So, for C, the `ijk` ordering is the fastest. The complete rankings are (`a > b` means "a slower than b"):

`jki > kji > kij > ikj > jik > ijk`.

This is rather surprising to me, since in C it is the first axis which is memory-contiguous (fastest), but it appears that the `ijk`-ordering is fastest, on average, with it's reverse `kji` being (close to) slowest.

I also tried compiling the C code in `clang` – due to this machine having an Arm-based and non-standard Apple-designed CPU, one might expect the Apple-forked `clang` (clang-1500.3.9.4) compiler to produce a *much* faster executable than `gcc`. Likewise, I did the same experiment in Julia because it has column-major ordering (like Fortran, the opposite of C's row-major ordering), but substantial metaprogramming facility which can be used to beat `gcc` and `gfortran` with comparative ease[‡] – though still substantially slower than `clang`.[§]

j_timing.txt

```
3D version:
  BC:  6.425e-5 s
  CI:  6.4083e-5 s
  ijk:   0.001294416 s
  jik:   0.0019575 s
  kij:   0.000234875 s
```

c1_timing.txt

```
3D version:
  ijk: 2.4999999999999962e-07 s
  jik: 2.9101562500000019e-07 s
  kij: 2.4218749999999953e-07 s
  kji: 7.0703125000000819e-07 s
  ikj: 8.8183593750001191e-07 s
```

[†] (`N=64`, `M=1024`), with versions 14.1.0 of `gcc` and `gfortran`, `-O3 -march=native`

[‡] The julia code is run with `julia -O3 --threads=1` for fairness.

[§] In truth, `clang` is managing this exceptionally low-latency executable by cheating... the computation is being skipped due to a lack of C-dependent output from these functions.

```
    kji:   6.425e-5 s
    ikj:   0.000635875 s
    jki:   9.9167e-5 s
  2D version:
    BC:  6.245116279069768e-7 s
    CI:  5.233403141361257e-7 s
    ij:  2.2916666666666666e-6 s
    ji:  6.2330232558139 54e-7 s
```

```
    jki: 6.9042968750000874e-07 s
  2D version:
    ij: 6.7871093750000803e-07 s
    ji: 6.7773437500000800e-07 s
```

Now for Julia (and Fortran), `kji` is the fastest, and `ijk` is (nearly) the slowest. I suspect that in both cases, the compiler is doing things to optimize for the 2D case so the rankings are pairwise perturbed based on the index of the inner loop. Note that the shortest time is about $2.5 \times 10^{-7} s$, so we're more than a factor of $206\times$ slower in the best case for Julia, $334\times$ slower in the best case for *C* with `gcc`, and $640\times$ slower for the best case for Fortran with `gfortran`.

On a modern CPU, single-core memory bandwidth ($\mathrm{BW}_{\mathrm{SC}}$) is in the $15 - 55$ GB/s range, typically. For $N = 64$ and $D = 3$, we should then expect typical times no faster than

$$t \sim 4 \times 10^{-5} s \gtrsim 8 \mathrm{BW}_{\mathrm{SC}}^{-1} N^D$$

or roughly 40 µs, which works out to $\ll 1$ ns per element. It's remarkable that we can get a good order-of-magnitude estimate of the throughput for the computation based solely on the bandwidth of a single CPU core – this is an indication that the computation in bandwidth-limited.

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*?¶

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.

¶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.

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?

Solution

In OpenMP, we should use the `#pragma omp parallel for` pragma for these nested for loops. From this baseline, we only need to understand two additional options: `collapse(n)` and `OMP_NUM_THREADS`.

Obviously, how one sets the environment variable `OMP_NUM_THREADS` determines the maximum number of threads being used (up to some subtleties[‖]). However, you should expect to use N threads effectively with a single `#pragma omp parallel for` because it applies to the outermost loop *only* with each of those N threads performing N^2 additions.

Adding a `collapse(3)` clause to the pragma tells the compiler to treat the three nested loops as one large loop, and you might expect to use up to N^3 threads (i.e., every thread accesses its own element of `A`, `B`, and `C` and completes the work with a single addition). If you try this, however, you will find that it's absurdly inefficient.

You may also notice that this only has an impact for very large N – much larger than the values I have used here. The truth is that creating a thread pool is not free – while you can't add arbitrarily more threads to a computation to speed it up even theoretically (as in Amdahl's Law), the situation is somewhat worse in that creating threads is costly. Typically, you will find that optimizing the single-threaded version of your program improves the multi-threaded version substantially, while the reverse is rarely true.

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 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

[‖] You can also set the number of threads within your program, by calling an OpenMP library function `void omp_set_num_threads(int num_threads);`.

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 lower the minimal working size from $\frac{N}{[]}$ 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