CompSci 590.03: Introduction to Parallel Computing

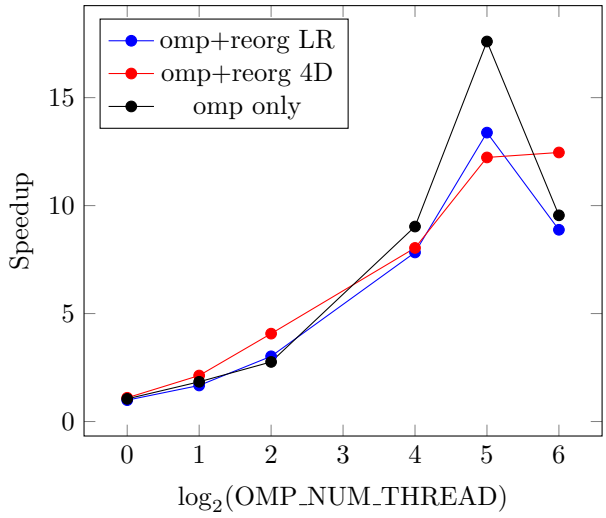
Homework 2 - Data Reorganization (09/15/15)

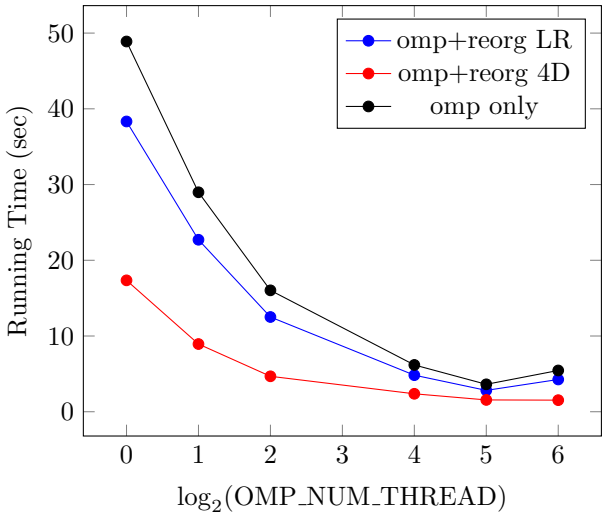
Lecturer: Alvin R. Lebeck *Scribes: Mengke Lian, Kai Fan, Chaoren Liu*

Contribution distribution:

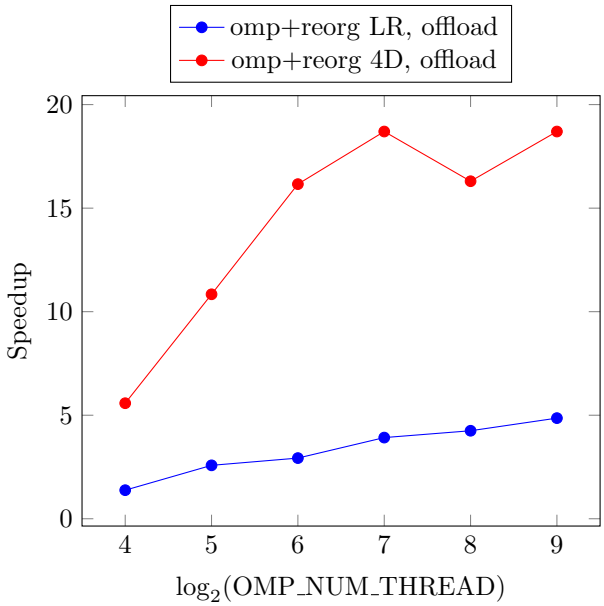
- Mengke Lian: finished data reorganization with loop reorder (LR) and 4D-array partition approaches and their vector version, finished Morton order approach but it is too slow since computing index is expensive. Did not try Morton-hybrid approach since it shares same idea with 4D-array approach except the arrangement of blocks are different.
- Kai Fan: The OpenMp version is to add one line of parallel for to serial version. The reorganization method is implemented by 4D-array. The OpenMp reorganization is to add one line of parallel for to previous version. The offload version is similar. A block multiplication function is defined and called in the matrix multiplication function, leading to 2x acceleration than 6-loop in my programs, but I do not understand this counter-intuitive phenomenon.
- Chaoren Liu: Reorganized the matrix from A[row][column] to A [column][row] by simply transforming the matrix. The computational time is shortened but not significant. Implemented the openmp, and openmp+data reorganization method. The computational time is shortened to 2-3 seconds. Implemented offload + openmp + dataReorg, but the computational time is 17s, which may be because the data reorganization is not optimal. 4D-array partition method further improves the performance to 2s, as being introduced below.

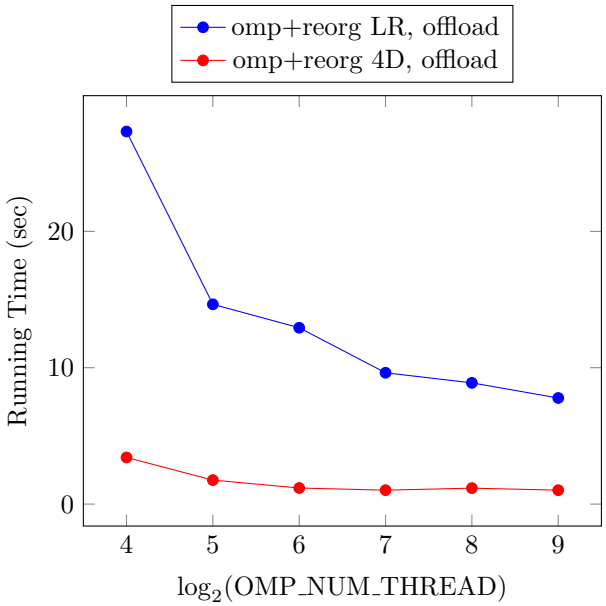
Note: for 4D-array partition approach, the block size is chosen to be 64 for native and 256 for offload. In the following running time curves, we use `-O2` without `-fast` in the Makefile: The running time for serial code is 48.41 seconds, and reorganization with reorder is 37.82 seconds, reorganization with 4D-array is 19.07 seconds. All speedup is computed according to corresponding reorganization running times.





For curves of offload programs, note that when x tick is 9, it means the `OMP_NUM_THREADS` is default, not 2^9.





Discussion

From the speedup curves and running time curves we can see that for data reorganization only case, the 4D-array partition approach is faster than just reorder the loops. When combining both data reorganization and OpenMP, data reorganization always outperforms OpenMP only version, and 4D-array approach is faster than reorder loops. This is because of the avoidance of cache miss, refer to next section for details.

For offload curves, unfortunately performance boost does not exist for loop reorder approach, but in contrast we succeed to reduce the running time to 1s when `OMP_NUM_THREADS` is 128 or default for 4D-array partition approach.

Also, we notice for both data reorganization approaches, high dimensional data (4D array in our case) structure can be linearized into a vector. By properly manipulating the index, extra computation burden for index can be effectively alleviated and the program can benefit from 1D array structure.

In summary, data reorganization is a technique to avoid cache miss. However, while trying to use more complicated cache friendly data structures, extra computation of index is also introduced. Thus, avoid recomputing same value for index is also very important. In one sentence, data reorganization is a trade-off problem among data size, thread numbers, cache miss, structure parameters (e.g. block size of 4D-array partition approach) and index computation.

Reorder Loops

The original matrix multiplication C of two large matrix A with size $N \times P$ and B with size $P \times N$ follows the pseudo code below

```
for i = 1 : N do
    for j = 1 : M do
        for k = 1 : P do
            C[i][j] += A[i][k] * B[k][j]
        end
    end
end
```

The drawback of this approach is that inside the inner loop for k column element are read in matrix B and write in matrix C. Since in C language 2D array is stored in row-first order, when the column number is large, reading element in one column causes cache miss frequently and this slows down the program. The cache miss happens when j changes, i.e. $N \cdot M \cdot P$ times.

To avoid cache miss, reorder loops as:

```
for i = 1 : N do
    for k = 1 : P do
        for j = 1 : M do
            C[i][j] += A[i][k] * B[k][j]
        end
    end
end
```

Note that for the inner loop for j, matrix B, C are accessed within a row and cache miss happens when i, k change, i.e $N \cdot P$ times

4D-Array Partition

Another approach is first partition the large matrix into $D \times D$ smaller blocks and perform matrix multiplication for small blocks:

```
for i = 1 : D do
    for j = 1 : D do
        for k = 1 : D do
            for ii = 1 : N/D do
                for kk = 1 : P/D do
                    for jj = 1 : M/D do
                        C[i][j][ii][jj] += A[i][k][ii][kk] * B[k][j][kk][jj]
                    end
                end
            end
        end
    end
end
```

Notice inside one block reordering loops is also applied. Choose each block is not large enough to cause cache miss, then cache miss only happens when block changes, i.e. when i, j, k changes, which is D^3 times.