COMP4300 Assignment 2

Jackson Kilrain-Mottram (u6940136)

May 2023

1 Question 1

1.1 Performance

In order to properly quantify the performance impacts that the various configurations of OpenMP directives used in each case have, we will utilise a linux tool called `perf`. This captures event statistics for the compute stack, from the OS level, to kernel and hardware. We are particularly interested in hardware, at the L1 data cache misses and LLC load misses. To give some context to these quantites we will examine, let's first mention that the program has roughly $5.4 \times 10^{1}0$ instruction total to be executed for 100 iterations of a 1000×1000 grid.

```
void omp1dUpdateBoundary(...) {
    // ... snip ...
    #pragma omp for private(j)
    for (j = 1; j < N + 1; j++) { ... }
    #pragma omp for private(i)
    for (i = 0; i < M + 2; i++) { ... }
}

void omp1dUpdateAdvectField(...) {
    // ... snip ...
    #pragma omp for private(i)
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            V(u, i, j) = ...;
}

void omp1dCopyField(...) {
    // ... snip ...
    #pragma omp for private(i)
    for (i = 0; i < N; i++)
        for (j = 0; j < M; j++)
            V(u, i, j) = V(v, i, j);

}

void omp1dAvect(...) {
    // ... snip ...
    for (size_t r = 0; r < reps; r++) {
        #pragma omp parallel shared(u,ldu,v,ldv)
        {
            omp1dUpdateBoundary(u, ldu);
            omp1dUpdateAdvectField(&V(u, 1, 1), ldu, &V(v, 1, 1), ldv);
            omp1dCopyField(&V(v, 1, 1), ldv, &V(u, 1, 1), ldu);
        }
    }
}
```

Utilising the performance counters from `perf stat -d`, we see that on average `L1-dcache-load-misses` is approximately 0.72% and `LLC-load-misses` fluctuate between $55 - 62.08\%$. Quantitatively, these equate to 5.6×10^7 in L1 dcache misses and 1.4×10^6 in LLC misses. This implies that there are few misses in the cache for loads, as the locality and alignment of data shared between threads is well situated. The overhead seen here can be attributed to the edge cases between the outside (starting and ending point) of the contiguous part of the advection field accessed for a thread. These are less ideal in their alignment and block saturation, leading to the minimum overhead seen in the L1 data cache load misses.

1.2 Parallel Region Entry/Exits

```
void omp1dUpdateBoundary(...) {
    // ... snip ...
    #pragma omp for private(j)
    for (j = 1; j < N + 1; j++) { ... }
    #pragma omp for private(i)
    for (i = 0; i < M + 2; i++) { ... }
```

```
}

void omp1dUpdateAdvectField(...) {
    // ... snip ...
    for (i = 0; i < N; i++)
        #pragma omp for private(j)
        for (j = 0; j < M; j++)
            V(u, i, j) = ...;
}

void omp1dCopyField(...) {
    // ... snip ...
    for (i = 0; i < N; i++)
        #pragma omp for private(j)
        for (j = 0; j < M; j++)
            V(u, i, j) = V(v, i, j);

}

void omp1dAvect(...) {
    // ... snip ...
    for (size_t r = 0; r < reps; r++) {
        #pragma omp parallel shared(u,ldu,v,ldv)
        {
            omp1dUpdateBoundary(u, ldu);
            omp1dUpdateAdvectField(&V(u, 1, 1), ldu, &V(v, 1, 1), ldv);
            omp1dCopyField(&V(v, 1, 1), ldv, &V(u, 1, 1), ldu);
        }
    }
}
```

1.3 Cache Misses Coherent Reads

TODO

1.4 Cache Misses Coherent Writes

```
void omp1dUpdateBoundary(...) {
    // ... snip ...
    #pragma omp for private(j)
    for (j = 1; j < N + 1; j++) { ... }
    #pragma omp for private(i)
    for (i = 0; i < M + 2; i++) { ... }
}

void omp1dUpdateAdvectField(...) {
    // ... snip ...
    #pragma omp for private(i) schedule(dynamic)
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            V(u, i, j) = ...;
}

void omp1dCopyField(...) {
    // ... snip ...
    #pragma omp for private(i) schedule(dynamic)
    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            V(u, i, j) = V(v, i, j);

}

void omp1dAvect(...) {
    // ... snip ...
    for (size_t r = 0; r < reps; r++) {
        #pragma omp parallel shared(u,ldu,v,ldv)
        {
            omp1dUpdateBoundary(u, ldu);
            omp1dUpdateAdvectField(&V(u, 1, 1), ldu, &V(v, 1, 1), ldv);
            omp1dCopyField(&V(v, 1, 1), ldv, &V(u, 1, 1), ldu);
        }
    }
}
```

Using `OPENMP_NUM_THREADS=48 perf stat -d ./testAdvect 1000 1000 100` we can keep track of performance counters of hardware behaviour while the advection solution is executing. More specifically, paying attention to the `L1-dcache-load-misses` and `LLC-load-misses`. Form the execution profile, we note that `L1-dcache-load-misses` sits at $6.80 - 14.60\%$ miss rate consistently across multiple runs. For `LLC-load-misses`, this tends to fluctuate heavily between 75.73% overall. From this we can see that cache misses occur frequently for loads that are localised to blocks that overlap between processors. This causes frequent invalidation due to coherence reads between the processors to ensure up to date cache block data. Qualitatively, there is approximately 5.9×10^8 total L1 dcache misses, compared to the 5.6×10^7 for the optimise variation in case 1. Similar,y the LLC misses are an order of magnitude greater in quantity, at 1.35×10^7 on average compared to 1.4×10^6.

2 Question 2

In order to formulate a model, we need to establish a few points on the hardware properties and problem decomposition. Firstly, using `cat /proc/cpuinfo | grep cache_alignment` we can determine the cache block size for the Xeon 8274 CPUs being utilised. Using this on the Gadi nodes, we see that there is 64 bytes in a cache block. To set the scene, an $M \times N$ grid, requires $\lceil \frac{M \cdot N}{64} \rceil$ cache blocks to fully contain it, these are distributed across the contiguous grid (figure 1).

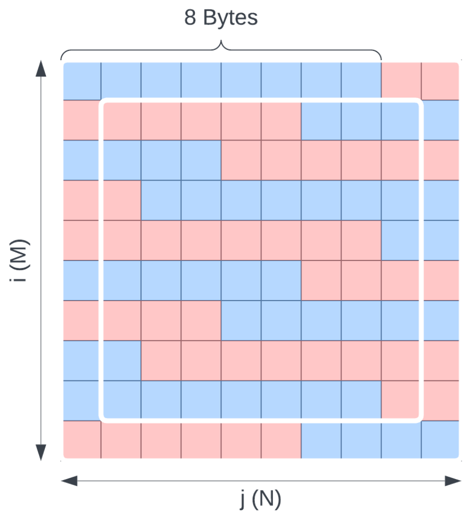

Figure 1: 8 byte cache block distribution over an example 2D advection grid.

In order to determine cache coherence behaviour on read and write operations for the solution, we need to first define a function $f : \mathbb{Z}^2 \to \mathbb{Z}$ that maps a 2D grid index (i, j) to the associated cache block it resides in.

$$f(i,j) \stackrel{\text{def}}{=} \left\lfloor \frac{j + (i \cdot N)}{64} \right\rfloor$$

2.1 Field Update

For the `omp1dUpdateAdvectField` method, there are two sets of operations to consider for coherent-write cache misses and coherent-read cache misses.

2.1.1 Write Misses

Each row starts at a given cache block and ends at the same or another cache block. Using this we can determine how many cache lines (and which they are) are involved. For each write, we need to invalidate these cache lines on each of the other processors. Given that each of the i indices are in parallel, we need to find the maximal cost line update across all processors. Therefore the total can be expressed as follows

$$M_L = \left\lfloor \frac{M}{P} \right\rfloor$$

$$U_w = t_{w,W} \cdot \underset{\forall p \in [0,P]}{\operatorname{argmax}} \left(\max \left\{ 1, \sum_{i=1}^{M_L} [f((p \cdot M_L) + i, N) - f((p \cdot M_L) + i, 1)] \right\} \right)$$

2.1.2 Read Misses

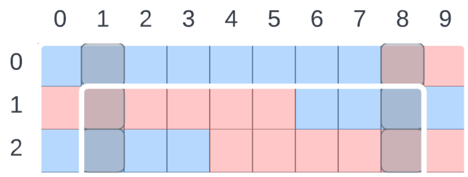

Figure 2: First and last cache block encounters over a subset of the example advection grid.

Each row element has a convolution applied over the previous, current and next row elements. The rows start at a given cache block and end at the same or different cache block. For each transition between cache blocks (as highlighted in figure 2), we will see a coherent read miss. Given we load these rows in chunks for each thread, we consider misses for each row, summed together.

$$U_r = t_{w,R} \cdot \underset{\forall p \in [0,P]}{\text{argmax}} \left(\max \left\{ 1, \sum_{i=1}^{M_L} \left(\sum_{l=-1}^{1} \left[f((p \cdot M_L) + i + l, N) - f((p \cdot M_L) + i + l, 1) \right] \right) \right\} \right)$$

2.2 Copy Field

When copying the field from v to u to allow for the next iteration or finalisation to occur, we have both read and write operations occurring simultaneously. Note that these are orthogonal to each other, as the read operations are performed over v and the write operations over u. We can abstract the common argmax into a single formulation applied to both the read and write coefficients as they equate equivalently.

$$C_{\max} = \underset{\forall p \in [0,P]}{\text{argmax}} \left(\max \left\{ 1, \sum_{i=1}^{M_L} \left[f((p \cdot M_L) + i, N) - f((p \cdot M_L) + i, 1) \right] \right\} \right)$$

$$C_w = t_{w,W} \cdot C_{\max}$$

$$C_r = t_{w,R} \cdot C_{\max}$$

2.3 Full Model

Combining the above operational breakdowns, we can achieve the full model over r iterations, ignoring the boundary update (as stipulated by the question):

$$t = r \cdot (U_r + U_w + C_r + C_w)$$

2.4 Coefficient Evaluation

TODO

3 Question 3

4 Question 4

5 Question 5

6 Question 6

7 Question 7

Between the three models of MPI, OpenMP and CUDA, design wise CUDA provides the greatest diversity for design of highly parallel tasks. More specifically, with regards to problems that have thread independent discretisation models, such as matrix evolutions (Heat equation, advection, etc). However, when it comes to approaches that require interaction between threads (not necessarily synchronisation), it introduces complexity that is otherwise more manageable in the MPI model. As for OpenMP, it yields similar benefits that CUDA does, however in much more significant context given the restrictions on compiler driven threading models as opposed to the more explicit nature of CUDA.

Implementation wise, OpenMP has the most approachable interfaces for both new and adaptive implementation. This extends to understanding the behaviour of how execution, compaction and grouping function relative to the behaviour of the pragma directives. That being said, it has a high-end implementation overhead when it comes to optimising it for fine-grained control of execution behaviour due to the high-level of abstraction. MPI has provides a well rounded SPI from simplistic interaction models for control over exchange behaviour and optimisations to a per-thread level to the point of fine-grained memory behaviour.

Over CUDA, it is substantially easier to control complex overlapping behaviour where the implementation does not follow SIMD/SIMT structuring well (diversity in condition evaluation for instance). This allows for reduced algorithmic complexity when handling these kinds of diverse behaviour on a thread-to-thread basis. Lastly, for CUDA, implementation has a high-entry barrier in terms of understanding how to appropriately structure your implementation and understand the resource decomposition required for appropriate allocation and usage. This requires more work to structure the implementation differently, especially when considering how conditional overheads (warp divergence/convergence) are easy to introduce with easy-to-overlook implementation aspects. Substantially more effort and understanding is required for the CUDA model, however, even with this overhead the tool set it provides for programmability is far and above the others allowing full control over the behaviour and program structure across threads when compared to OpenMP.

During development, in any of the three paradigms/models, the debugging of the core algorithm with regards to structure, is essentially the same. This is specific to the arithmetic operations and memory operations. However, considering the halo exchanges and the different approaches for managing these, it differs quite a bit ween the communication model versus the shared memory model. For MPI, exchange debugging relies on logging tagged entries communicated between threads for message issues. With contention issues much of the debugging hinges on manual checking and verification of parameters for communication handlers.

On the other hand, in the land of shared memory models, OpenMP is straightforward to debug for most cases. In the most difficult of cases (that I had encountered), it comes down to data ownership between threads and the global program space. Missing a marker for shared or private variables can lead to undefined behaviour that is hard to track down, however, due to the simplicity of compiler directives for simple cases (such as advection solution), these tend to be easy to find. Compare that to CUDA, which has improvement with regards to scoping as mentioned previously, as runtime errors occur for segmentation violations and improper host/device pointer usage. These become much clearer, however, sourcing the problem behind them can be arduous for complex logic. With proper alteration of kernel execution and isolation, it does however provide an edge over OpenMP with the clarity of debugging techniques.

References