Influence of atomic FAA on ParallelFor and a cost model for improvements

Ran Shuai
Microsoft, Silicon Valley Campus, Mountain View, CA, United States
rashuai@microsoft.com

November 29, 2021

Abstract - this paper focuses on one of the most frequently visited multithreading library interfaces - ParallelFor. In this study, it is inferred that ParallelFor’s end-to-end latency performance is noticeably affected by the frequency with which fetch-and-add (FAA) is called during program execution. This can be explained by ParallelFor’s uniform semantics and the utilization of atomic FAA. To prove this assumption, a battery of tests was designed and conducted on diverse platforms. From the collected performance statistics and overall trends, several conclusions were drawn and a cost model is proposed to enhance performance by mitigating the influence of FAA.

Introduction

For decades, increasingly high-performance workstations have been adopted in production environments to boost productivity. These machines are commonly equipped with multi-core CPU(s) that allow for parallelism in applications and services. Each CPU core hosts one or two hardware threads (e.g., hyper-threading) that undertake assigned tasks independently. With a suitable task-breakdown policy and wait-and-join semantics, an application or service could allow several of its tasks to run simultaneously, instead of sequentially, which reduces latency and promotes responsiveness.[1][2][3]

To fully utilize the hardware's computing power, multi-threading software libraries serve to bridge upstream software applications and hardware threads. Usually, such libraries create, host, and maintain a set of software threads known as a thread pool. Moreover, the libraries provide API(s), such as ParallelFor(...) or ForEach(...), that take a task/function and an integer number as the iterations to run. Upon being called, the libraries will distribute the task to all threads in the pool. When the maximum iterations are reached and all threads have completed the assigned iterations, ParallelFor(...) or ForEach(...) will return to the caller. One such scenario is OpenMP[4] - a built-in library for C++ compilers, such as GNU gcc[5] and LLVM[6]. IntelTBB and a recently published library—TaskFlow[7] serve similar purposes.

According to Amdahl’s law[8], a parallelized application could provide end-to-end performance benefits as:

$$SpeedUp(T) = 1/((1 - P) + P/T)$$

where T is the number of threads and P is the parallelizable fraction of the application. For ParallelFor, P is always 1; thus, we could expect the gain to be:

$$SpeedUp(T) = T$$

Problem statement

The measurement of performance gains should neither be limited to Amdahl's law, nor the assumption that n threads in total “should” bring down the overall cost to total_cost/number_of_threads. The parallelism provided by multi-threading libraries leads to some nontrivial costs that could challenge the estimation of...
end-to-end latency. Considering ParallelFor as an example, its most representative implementations follow a simple semantic:

```c++
void ParallelFor(
    function<void(int)>& task,
    int N) {
    atomic<int> counter{0};
    int block_size{...};
    function<void(void)> thread_task =
        [&]() {
            int begin{0};
            while ((begin =
                    counter.FetchAndAdd(block_size))<N){
                for (int iter = begin;
                     iter < min(N,begin+block_size);
                     iter++) {
                    task(iter);
                }
            }
        };
    for(Thread& thread: threadpool) {
        thread.Enqueue(thread_task);
    }
    thread_task();
    for (Thread& thread: threadpool) {
        thread.wait(...)
    }
}
```

As seen in this code snippet, the ParallelFor function first wraps up the input task with a thread task, which utilizes atomic fetch-and-add (FAA) to acquire a range of iterations for execution. The atomic component serves to synchronize between threads to avoid a data race. Then, ParallelFor assigns the thread task to all threads in the pool and waits for all the returns. Based on the listed logic, the ParallelFor's caller could be assured that the input task will be called exactly \( N \) times, with an input iterating from 0 to \( N-1 \).

Among all the listed details, “block size” is a key variable that may significantly affect the overall end-to-end latency. It decides for how many iterations the thread should run each input task in the “for” loop. Equivalently, “block size” is a variant that determines the size by which “\( N \)” could be divided.

The issue here is that the calling of atomic FAA results in nontrivial cost because cache invalidation and resynchronization are required to ensure that all threads visiting the atomic variable see the same value. [9] presented a keen estimation on overheads as follows:

\[
L(A, S) = R(S) + E(A) + O
\]

where \( A \) is the type of atomic operation, such as FAA, compare-and-swap (CAS), or swap (SWP); \( S \) is the cache state among Modified, Owned, Exclusive, Shared, Invalid [10]. \( R(S) \) is the time-cost of acquiring ownership of the cache in \( S \) state, \( E(A) \) is the cost of performing \( A \) on the cache, and \( O \) represents other miscellaneous costs. In our case, \( A \) is always FAA and \( S \) is assumed to be Shared; hence, a simplified version of our case is:

\[
L = R + E + O
\]

According to [9], \( R \) takes the most significant part of \( L \), implying that threads spent most of the time during the interval on acquiring ownership of the cache. Furthermore, the number of threads and their locality also influence the cost of \( R \), and therefore, the overall \( L \). For example, two threads in the cores sharing the same \( L3 \) cache tend to spend less time on \( R \) than when the threads are in cores across sockets.

Assuming that the total number of iterations \( N \) is to be divided into several blocks of block size \( B \), and the number of threads is \( T \). Consequently, the overall end-to-end latency of ParallelFor may be formulated as

\[
Cost(T, N, L) = N/B \times L + O(N)/T
\]

Based on this formula, we could reasonably infer that \( Cost(T, N, L) \) is lower when \( B \) is larger, i.e., a larger block size means less calling of FAA, resulting in a lower total atomic overhead.

But what is the upper bound of \( B \)? We cannot assert that \( Cost(T, N, L) \) will always decrease as the block sizes increase. For instance, considering that we have \( T \) threads fully at our disposal, by setting \( B \) to exceed \( N/T \), only fewer than \( T \) threads are allowed the chance to run the input task—the potential of parallelism is thereby jeopardized. The test that follows shows that the end-to-end latency would begin falling before \( B \) reaches \( N/T \) because the threads are scheduled to run on a physical CPU core at different times, and as the load sharing among cores is occasionally imbalanced, the threads running time may vary for a period, which implies that smaller sized blocks are more likely to match the threads' running quota perfectly. [11]

Hence, knowing that the block size is expected to be larger to avoid excessive atomic FAA overhead, it should not exceed a certain limit. The problem lies in how to determine the proper \( B \) value.

**Test and statistics**

To address this problem, a series of tests was conducted to resolve the problem of determining how FAA with varied block sizes would affect the end-to-end latency of ParallelFor. Additionally, different “sized” tasks were included, as a task that reads and writes 32 bytes of memory with simple computations would not have the same performance as a more complex task. Furthermore, a task that works mostly on IO should behave differently from one that is more CPU intensive. To address these variants, we implemented a configurable unit task function:
unit_task =
[unit_read, unit_write,
unit_computation, ...] (...) {

    uint8_t* read_at = ...;
    uint8_t* write_at = ...;
    uint64_t per_read_computation =
        unit_computation / unit_read;
    uint64_t write_count = 0;
    uint8_t integer = 0;

    for (uint64_t i = 0; i < unit_read; ++i) {
        integer = read_at[i];
        for (uint64_t j = 0; j <
            per_read_computation; ++j) {
            integer += 1;
        }
        if (write_count < unit_write) {
            write_at[write_count++] = integer;
        }
    }
    while (write_count < unit_write) {
        write_at[write_count++] = integer;
    }
};

Note that unit_task will be sent to ParallelFor as
an input argument. As observed from the implementa-
tion, unit_task references three external variables
unit_read, unit_write, and unit_computation. Here,
unit_read denotes the number of bytes of memory to
read; unit_write the number of bytes to write; and
unit_computation the number of computations that
should occur along with execution. By using these vari-
ables, we could run tests with tas’s of varied “sizes” to
obtain unbiased conclusions.

Furthermore, we implemented a thread pool to pro-
vide the ParallelFor function, following the semantic
listed in the previous section. The thread pool allows
a number of threads with fixed affinity settings to be
configured to restrict the threads on certain cores. This
helps to reduce the noise from thread rescheduling
between cores and to maintain load balancing.

Finally, to ensure that the test results are more gen-
eralized, we prepared a set of platforms with diverse
hardware specifications and computing capabilities.

- Dell M4 workstation with Intel Xeon® W-3225R
  @ 3.70 GHz, Windows 10 Pro 18362.1171
- AMD Ryzen Threadripper 3970X 32-Core Proces-
sor @ 3.69 GHz, Windows 10 Pro 19042.1165
- Dell M4 workstation with two Intel® Xeon® Gold
  5225R @ 2.20 GHz, Windows 10 Pro 19042.1165

The W-3225R has one CPU of eight cores, each has
its propriety L1 and L2 but share the same L3; the
AMD 3970X has one CPU of 32 cores, every 4 cores
share an L3; the Gold 5225R has 2 CPUs sitting on
separate sockets, each has 24 cores sharing the same
L3, whereas L1 and L2 are core private. By using hwloc

[12], graphs of the internal hardware topology were
created as follows:
Influence of atomic FAA on ParallelFor and a cost model for improvements

With the task, thread pool, and platforms all implemented and ready, the end-to-end ParallelFor performance was first tested with different block sizes, varying unit computation, and thread pool sizes on the W-3225R. The latency was measured using the CPU clocks:

|         | Unit Read 1024 | Unit Write 1024 | Unit Comp 1024 |
|---------|----------------|----------------|---------------|
| Block Sizes | 2 Threads | 4 Threads | 8 Threads |
| 1        | 1394900 | 957700 | 569100 |
| 2        | 1291900 | 762200 | 445600 |
| 4        | 1240400 | 706200 | 430200 |
| 8        | 1112800 | 644700 | 462000 |
| 16       | 1078100 | 646900 | 437200 |
| 32       | 1060400 | 643700 | 447000 |
| 64       | 1082100 | 643000 | 494000 |
| 128      | 1081400 | 691700 | 558000 |
| 256      | 1192000 | 798200 | 632000 |
| 512      | 1179800 | 796900 | 800500 |
| 1024     | 1585000 | 799100 | 1309500 |

W-3225R: Unit Comp 1024, e2e latency in clocks

For the listed cases, the best performance varies by block size. However, noting the highlighted cells, it is apparent that mounting the unit computation decreases the most preferred block size, and increasing the number of threads that participate in computation produces a similar outcome. To confirm the observation, we conducted the following tests on the G-5225R:
Influence of atomic FAA on ParallelFor and a cost model for improvements

| block sizes | 4 threads | 8 threads | 16 threads |
|-------------|-----------|-----------|------------|
| 1           | 948300    | 555200   | 311700     |
| 2           | 900800    | 532700   | 272200     |
| 4           | 872500    | 516900   | 267300     |
| 8           | 868900    | 522600   | 269100     |
| 16          | 865200    | 505900   | 283100     |
| 32          | 864000    | 512300   | 302600     |
| 64          | 889200    | 502500   | 366500     |
| 128         | 874500    | 512200   | 341600     |
| 256         | 1023300   | 665100   | 511700     |
| 512         | 1041200   | 809700   | 1003300    |
| 1024        | 1332100   | 1334300  | 1584000    |

Gold 5225R: 1024^3 unit comp

| block sizes | 4 threads | 8 threads | 16 threads |
|-------------|-----------|-----------|------------|
| 1           | 889300    | 545300   | 314700     |
| 2           | 847100    | 494100   | 269500     |
| 4           | 826500    | 495900   | 263700     |
| 8           | 822900    | 484400   | 288100     |
| 16          | 827500    | 489800   | 273500     |
| 32          | 839300    | 498900   | 296000     |
| 64          | 817400    | 513000   | 305600     |
| 128         | 898800    | 502500   | 336600     |
| 256         | 1025700   | 515600   | 389600     |
| 512         | 1009900   | 614000   | 674600     |
| 1024        | 1015400   | 1335000  | 1037500    |

Gold 5225R: 1024^4 unit comp

| block sizes | 4 threads | 8 threads | 16 threads |
|-------------|-----------|-----------|------------|
| 1           | 913000    | 545000   | 313100     |
| 2           | 860600    | 521200   | 281000     |
| 4           | 830100    | 496800   | 263300     |
| 8           | 832000    | 503200   | 264500     |
| 16          | 827500    | 498200   | 273400     |
| 32          | 814300    | 512300   | 303400     |
| 64          | 827200    | 502000   | 294700     |
| 128         | 899600    | 517700   | 337200     |
| 256         | 1013100   | 671700   | 412000     |
| 512         | 1036000   | 660700   | 673700     |
| 1024        | 1013900   | 1337500  | 1022000    |

Gold 5225R: 1024^5 unit comp

| block sizes | 4 threads | 8 threads | 16 threads |
|-------------|-----------|-----------|------------|
| 1           | 931700    | 550100   | 322000     |
| 2           | 718300    | 626700   | 388600     |
| 4           | 639200    | 487700   | 315700     |
| 8           | 601600    | 401500   | 236200     |
| 16          | 560300    | 351500   | 197200     |
| 32          | 535800    | 323600   | 174000     |
| 64          | 524200    | 322700   | 165800     |
| 128         | 546100    | 321100   | 166000     |
| 256         | 632600    | 316400   | 162900     |
| 512         | 623300    | 323200   | 320800     |
| 1024        | 640600    | 621600   | 625800     |

Gold 5225R: 1024^6 unit comp

| block sizes | 8 threads | 16 threads | 32 threads |
|-------------|-----------|------------|------------|
| 1           | 931700    | 550100     | 322000     |
| 2           | 718300    | 626700     | 388600     |
| 4           | 639200    | 487700     | 315700     |
| 8           | 601600    | 401500     | 236200     |
| 16          | 560300    | 351500     | 197200     |
| 32          | 535800    | 323600     | 174000     |
| 64          | 524200    | 322700     | 165800     |
| 128         | 546100    | 321100     | 166000     |
| 256         | 632600    | 316400     | 162900     |
| 512         | 623300    | 323200     | 320800     |
| 1024        | 640600    | 621600     | 625800     |

AMD 3970X: 1024^4 unit comp

For the Gold 5225R, 24 threads were run on a single core group, whereas 36 or 48 threads required two core groups; for the AMD 3970X, every four cores form a core group, therefore, the tests listed above covered two, four, and eight core groups.

It is evident that the preferred block size increases by adding core groups, indicating that the task of N iterations would be split into fewer pieces, hence fewer FAA would be triggered. The explanation could be that the cache resynchronizations among the core groups are via media that is markedly less performant than by defining cores that share the same L3 as a core group, the opposite trend was observed when adding more core groups to the tests:

| block sizes | 24 threads | 36 threads | 48 threads |
|-------------|------------|------------|------------|
| 1           | 309600     | 325100     | 490600     |
| 2           | 269400     | 234400     | 498600     |
| 4           | 302000     | 228600     | 381200     |
| 8           | 264700     | 274800     | 376700     |
| 16          | 273700     | 235500     | 236100     |
| 32          | 296400     | 257900     | 212000     |
| 64          | 301900     | 257800     | 193600     |
| 128         | 337000     | 360400     | 420400     |
| 256         | 389100     | 406000     | 553600     |
| 512         | 701000     | 840800     | 1161600    |
| 1024        | 1066900    | 1744600    | 2402800    |

The Gold 5225R cases confirm that a larger unit computation reduces the preferred block size. That there is a constant range of how much computation a thread can handle in its average CPU quota is a possible explanation. This means that the larger the unit computation, the smaller the block size should be to keep the overall block computation in that range. Note:

\[
\text{computation of block} = \text{block size} \times \text{unit comp}
\]

In addition, FAA overheads are low because threads in the tests are running on cores sharing the same L3, thus by adding more threads, block size should be kept smaller to allow for better parallelism. Contrastingly,

in the tests are running on cores sharing the same L3, thus by adding more threads, block size should be kept smaller to allow for better parallelism. Contrastingly,
shared L3, such as the hyper-transport link, therefore, FAA overheads are considerably higher than previous cases, accordingly, block sizes should be kept larger to reduce the number of FAA calls.

It has been established that the best block size is proportional to the number of core groups and inversely proportional to the number of threads and size of the unit computation. Further investigation commences on the unit read and write test results:

| unit read 64, unit write 1024, unit comp 1024<sup>th</sup> | block sizes | 4 threads | 16 threads | 24 threads |
|---|---|---|---|---|
| 1 | 1053600 | 3358800 | 497400 |
| 2 | 967600 | 335400 | 347500 |
| 4 | 937800 | 252100 | 280300 |
| 8 | 923400 | 245400 | 247400 |
| 16 | 912400 | 244100 | 240400 |
| 32 | 873600 | 284700 | 153100 |
| 64 | 511900 | 270300 | 197600 |
| 128 | 549800 | 436300 | 202500 |
| 256 | 612900 | 373100 | 276500 |
| 512 | 626900 | 717400 | 550800 |
| 1024 | 655100 | 687700 | 1124600 |

**Gold 5225R: 64 unit read**

| unit read 256, unit write 1024, unit comp 1024<sup>th</sup> | block sizes | 4 threads | 16 threads | 24 threads |
|---|---|---|---|---|
| 1 | 736900 | 352700 | 299200 |
| 2 | 695500 | 196200 | 202100 |
| 4 | 631600 | 209800 | 181300 |
| 8 | 636700 | 182000 | 178400 |
| 16 | 634900 | 200400 | 179100 |
| 32 | 621700 | 187000 | 192100 |
| 64 | 632200 | 238400 | 261800 |
| 128 | 683600 | 227500 | 253500 |
| 256 | 769800 | 426200 | 367200 |
| 512 | 790800 | 480800 | 781600 |
| 1024 | 787100 | 654200 | 1521700 |

**Gold 5225R: 256 unit read**

| unit read 4096, unit write 1024, unit comp 1024<sup>th</sup> | block sizes | 4 threads | 16 threads | 24 threads |
|---|---|---|---|---|
| 1 | 3511000 | 974500 | 797600 |
| 2 | 3436000 | 948600 | 766800 |
| 4 | 3433400 | 940300 | 767100 |
| 8 | 3581400 | 942900 | 742000 |
| 16 | 3841100 | 975600 | 794000 |
| 32 | 3845200 | 942100 | 808800 |
| 64 | 4071100 | 1069800 | 1065700 |
| 128 | 4644200 | 1110300 | 1069800 |
| 256 | 4661300 | 1088300 | 1760700 |
| 512 | 4638300 | 2215800 | 3464700 |
| 1024 | 4212500 | 3271700 | 7041200 |

**Gold 5225R: 4096 unit read**

From the numbers listed above, the best block size is clearly also inversely proportional to the unit read. Notably, with more engaged threads, the block size decreases accordingly, as was observed in the tests on unit computation. Next, varied unit write statistics are considered:

| block read 1024, unit write 2<sup>12</sup>, unit comp 1024<sup>th</sup> | block sizes | 8 threads | 16 threads | 32 threads |
|---|---|---|---|---|
| 1 | 1375600 | 457100 | 324900 |
| 2 | 1456100 | 639800 | 526100 |
| 4 | 1302800 | 487800 | 396400 |
| 8 | 1193800 | 422500 | 341700 |
| 16 | 1158400 | 383800 | 309400 |
| 32 | 1140500 | 378000 | 304800 |
| 64 | 1127600 | 362900 | 353200 |
| 128 | 1125500 | 362800 | 383700 |
| 256 | 1389300 | 365300 | 373100 |
| 512 | 1379700 | 690000 | 688800 |
| 1024 | 1384100 | 1383200 | 1379600 |

**AMD 3970X: 2<sup>12</sup> unit write**

| unit read 1024, unit write 2<sup>14</sup>, unit comp 1024<sup>th</sup> | block sizes | 8 threads | 16 threads | 32 threads |
|---|---|---|---|---|
| 1 | 3818400 | 1259400 | 954400 |
| 2 | 3932900 | 2068000 | 1273700 |
| 4 | 3758100 | 2000000 | 1180800 |
| 8 | 3721500 | 1755200 | 1127300 |
| 16 | 3692200 | 1931600 | 1118500 |
| 32 | 3653800 | 1641100 | 1146200 |
| 64 | 3673800 | 1577100 | 1207800 |
| 128 | 3919200 | 1530900 | 1316300 |
| 256 | 4445700 | 1513100 | 2117700 |
| 512 | 4446400 | 2774100 | 2844900 |
| 1024 | 4503600 | 4421800 | 4490600 |

**AMD 3970X: 2<sup>14</sup> unit write**

| unit read 1024, unit write 2<sup>16</sup>, unit comp 1024<sup>th</sup> | block sizes | 8 threads | 16 threads | 32 threads |
|---|---|---|---|---|
| 1 | 13781800 | 8311600 | 7729600 |
| 2 | 13626800 | 10221600 | 11584900 |
| 4 | 13509000 | 10178700 | 11673600 |
| 8 | 13542500 | 10186700 | 11597300 |
| 16 | 13703600 | 10284900 | 11546900 |
| 32 | 13749300 | 10498000 | 11579000 |
| 64 | 13714900 | 10178800 | 11541200 |
| 128 | 14650200 | 10367800 | 10999900 |
| 256 | 16670700 | 9950500 | 10291700 |
| 512 | 16645700 | 12600000 | 12558400 |
| 1024 | 16739400 | 16789600 | 16820400 |

**AMD 3970X: 2<sup>16</sup> unit write**

Predictably, the unit-write tests suggest similar trends to those of unit read; as unit write increases, the preferred block size for better performance decreases. Combining the observations from unit computation as well as read and write, we can assume that, when the task size is “larger,” the preferred block size is smaller. Note that the total number of operations in one block should be calculated as follows:

\[
\text{complexity_of_block} = \text{block_size} \times \text{task_size}
\]

\[
\text{task_size} = \text{unit_read} + \text{unit_write} + \text{unit_comp}
\]
Cost model and improvements

In summary, we have the following observations:

- The best block size is proportional to the number of core groups (cores share the same L3 in a core group);
- The best block size is inversely proportional to the number of threads, unit read, unit write, and unit computation.

This confirms that the distribution of the preferred block size varies following fixed rules. Intuitively, the proposed cost model is formulated as follows:

\[
B = \frac{\alpha \times G + \delta_0}{\beta_0 \times T + \beta_1 \times R + \beta_2 \times W + \beta_3 \times C + \delta_1}
\]

where \(B\) is the expected block size; \(G\) is the number of core groups; and \(T, R, W,\) and \(C\) respectively represent the number of threads, unit read, unit write, and unit computation. \(\alpha, \beta,\) and \(\delta\) are unknown parameters that require tuning. To determine the parameters and evaluate the model’s efficacy, we implemented a linear regression model using Pytorch [13].

```python
class CostModel(nn.Module):
    def __init__(self):
        super(CostModel, self).__init__()
        self.power = nn.Linear(1,1)
        self.cost = nn.Linear(4,1)

    def forward(self, x):
        power = self.power(x[:,:1])
        cost = self.cost(x[:,1:])
        return torch.div(power, cost)
```

where \(x\) is the batched input vectors of the core groups, threads, unit read, unit write, and unit computation. The raw training input is the list of previously collected numbers. Each training vector has six columns.

| G | T | R | W | C | B |
|---|---|---|---|---|---|
| 1 | 2 | 1024 | 1024 | 1024 | 128 |
| 1 | 2 | 1024 | 1024 | 1024 | 128 |
| 1 | 2 | 1024 | 1024 | 1024 | 128 |
| 1 | 2 | 1024 | 1024 | 1024 | 128 |
| 1 | 2 | 1024 | 1024 | 1024 | 128 |

Note that the last column is the preferred block size, which is excluded from \(x\). Next, owing to the overly sparsified inputs compared with the expected output, we performed data normalization in case the training converges slowly, albeit with acceptable losses [14]. Hence, we did the following:

- Multiple core group with 100

Finally, the cost function is:

\[
loss = (y - y')^2
\]

The training was then conducted on an NVIDIA Quadro M4000 with the Cuda Toolkit 11.4 [15]. After 30 h and approximately \(10^7\) epochs, the training data loss on over 200 cases was reduced to 2001.48. Hence, for each input data, on average, the loss was less than 10. Several examples are listed in the following table:

| G | T | R | W | C | B | Inferred B |
|---|---|---|---|---|---|-----------|
| 100 | 2 | 10 | 10 | 1 | 128 | 125 |
| 100 | 2 | 10 | 10 | 3 | 64 | 51 |
| 100 | 2 | 10 | 10 | 4 | 32 | 39 |
| 100 | 2 | 10 | 10 | 6 | 16 | 27 |
| 100 | 8 | 10 | 10 | 2 | 32 | 36 |
| 100 | 8 | 10 | 10 | 3 | 32 | 30 |
| 100 | 8 | 10 | 10 | 5 | 16 | 22 |
| 100 | 8 | 10 | 10 | 6 | 16 | 22 |
| 100 | 4 | 6 | 10 | 6 | 64 | 80 |
| 100 | 4 | 8 | 10 | 6 | 32 | 37 |
| 100 | 4 | 12 | 10 | 6 | 16 | 17 |
| 100 | 4 | 16 | 10 | 6 | 16 | 11 |
| 100 | 8 | 8 | 10 | 6 | 16 | 27 |
| 100 | 8 | 10 | 10 | 6 | 16 | 19 |
| 100 | 8 | 16 | 10 | 6 | 4 | 10 |
| 200 | 8 | 10 | 10 | 1 | 128 | 108 |
| 200 | 8 | 10 | 10 | 2 | 64 | 85 |
| 200 | 8 | 10 | 6 | 6 | 64 | 112 |
| 200 | 8 | 10 | 8 | 6 | 64 | 65 |
| 200 | 8 | 10 | 10 | 6 | 64 | 46 |
| 200 | 8 | 10 | 14 | 6 | 32 | 29 |
| 200 | 8 | 10 | 16 | 6 | 16 | 24 |
| 400 | 16 | 6 | 10 | 6 | 128 | 126 |
| 400 | 16 | 8 | 10 | 6 | 128 | 92 |
| 800 | 32 | 6 | 10 | 6 | 128 | 136 |
| 800 | 32 | 10 | 10 | 6 | 64 | 98 |
| 800 | 32 | 16 | 10 | 6 | 64 | 69 |

The formula with trained weights thereby becomes:

\[
B = \frac{1558.31 - 61.84 \times G}{693.13 - 10.48 \times T - 33.71 \times R - 34.50 \times W - 26.84 \times C}
\]
Related work and comparison

A recently published multi-threading library with public access on Github—Taskflow[7]—has implemented the ParallelFor semantic. According to the corresponding paper, Taskflow provides a powerful interface that assembles tasks in topological order and executes with maximum parallelism. Regarding ParallelFor (under the name of for_each, atomic faa synchronized threads compete for the assigned iterations. Each time a thread attempts to acquire a range of [begin, end], it intends to multiply a decimal constant as q (= 0.5/ < number_of_threads >) with the unfinished part of N, defined as r. Thereafter, the end is equated to “begin + q * r,” implying that the block size is “q * r.” Subsequently, when r is smaller than 4 * <number of threads>, the block size will reduce to 1 until the execution ends. This approach, though somewhat dynamic, is generally less performant than the one we have implemented herein with the cost model. The performance data for comparison are as follows:

| unit_read | Taskflow | CostModel | block sizes |
|-----------|----------|-----------|-------------|
| 2^6       | 3205000  | 257100    | 46          |
| 2^8       | 420400   | 259500    | 27          |
| 2^10      | 462600   | 390400    | 19          |
| 2^12      | 1364700  | 1242900   | 15          |
| 2^14      | 5822300  | 4470400   | 12          |
| 2^16      | 1920330  | 16524300  | 10          |

W-3225R: 8 T, 1024 unit_read, 2^60 unit_comp, in clocks

| unit_write | Taskflow | CostModel | block sizes |
|------------|----------|-----------|-------------|
| 2^6        | 580100   | 403400    | 48          |
| 2^8        | 673500   | 377000    | 28          |
| 2^10       | 1176700  | 430700    | 19          |
| 2^12       | 1077100  | 847600    | 15          |
| 2^14       | 3707600  | 3746300   | 12          |
| 2^16       | 15411800 | 15498900  | 10          |

W-3225R: 8 T, 1024 unit_write, 2^60 unit_comp, in clocks

| unit_comp | Taskflow | CostModel | block sizes |
|-----------|----------|-----------|-------------|
| 1024      | 1334200  | 750500    | 46          |
| 1024^2     | 790100   | 744800    | 36          |
| 1024^3     | 496600   | 456000    | 30          |
| 1024^4     | 508500   | 412700    | 25          |
| 1024^5     | 527300   | 429400    | 22          |
| 1024^6     | 479000   | 435300    | 19          |

W-3225R: 8 T, 1024 unit_comp, in clocks

| unit_read | Taskflow | CostModel | block sizes |
|-----------|----------|-----------|-------------|
| 2^6       | 420900   | 172200    | 17          |
| 2^8       | 459800   | 157100    | 13          |
| 2^10      | 764700   | 228400    | 11          |
| 2^12      | 797400   | 633500    | 9           |
| 2^14      | 3514300  | 3542800   | 8           |
| 2^16      | 16775400 | 14511900  | 7           |

G-5225R: 24 T, 1024 unit_read, 2^60 unit_comp, in clocks

| unit_comp | Taskflow | CostModel | block sizes |
|-----------|----------|-----------|-------------|
| 1024      | 604700   | 287000    | 17          |
| 1024^2     | 549700   | 272300    | 15          |
| 1024^3     | 439200   | 195100    | 14          |
| 1024^4     | 211390   | 192000    | 13          |
| 1024^5     | 367100   | 190100    | 12          |
| 1024^6     | 402900   | 186100    | 11          |

G-5225R: 24 T, 1024 unit_read/write, in clocks

| unit_comp | Taskflow | CostModel | block sizes |
|-----------|----------|-----------|-------------|
| 1024      | 312700   | 269600    | 13          |
| 1024^2     | 348200   | 182100    | 11          |
| 1024^3     | 367600   | 320000    | 9           |
| 1024^4     | 819500   | 337500    | 8           |
| 1024^5     | 1913400  | 1382500   | 7           |
| 1024^6     | 7120100  | 4541800   | 6           |

AMD 3970X: 64 T, 1024 unit_read/write, 2^60 unit_comp

| unit_comp | Taskflow | CostModel | block sizes |
|-----------|----------|-----------|-------------|
| 1024      | 354300   | 199000    | 13          |
| 1024^2     | 339200   | 183500    | 11          |
| 1024^3     | 374600   | 274100    | 9           |
| 1024^4     | 514400   | 320300    | 8           |
| 1024^5     | 1488600  | 1569900   | 7           |
| 1024^6     | 8166500  | 8368800   | 6           |

AMD 3970X: 64 T, 1024 unit_read, 2^60 unit_comp

| unit_comp | Taskflow | CostModel | block sizes |
|-----------|----------|-----------|-------------|
| 1024      | 413800   | 34500     | 13          |
| 1024^2     | 439100   | 315100    | 12          |
| 1024^3     | 413300   | 358900    | 11          |
| 1024^4     | 444500   | 340500    | 10          |
| 1024^5     | 5496700  | 352200    | 10          |
| 1024^6     | 398700   | 336800    | 9           |

AMD 3970X: 64 T, 1024 unit_read/write

As the listed results indicate, with the fine-tuned cost model included in ParallelFor, a general performance boost of over 20% was observed. There are also multiple cases in which ParallelFor with the cost model required less than one-fifth of the time taken by Taskflow. Admittedly, there are also several cases in which ParallelFor underperforms compared to Taskflow, however, the lag in each case is relatively negligible. Moreover, we believe that these issues could be solved by using fine-tuned training data that are more representative; for example, the training data could be enriched with more fine-grained case information on task size.
Conclusion and future work

Based on the analysis and results presented in this paper, we demonstrated that atomic FAA has a significant effect on the end-to-end latency of ParallelFor on various platforms. Furthermore, by rigorously examining the collected performance statistics on neutralizing the FAA overheads, the best block size for splitting tasks within ParallelFor was found to be virtually dependent on a few parameters, such as core groups, the number of threads, and task size. Based on this information, a cost model is proposed. The training and inferencing of the model confirmed that it was effective when applied to the collected statistics and offers a significant advantage over existing counterparts that employ different approaches.

In future work, the CPU frequency and cache latency parameters, which may significantly promote cost model precision on general platforms, must be further investigated.

References

[1] D. Marr and F. Binns. “Hyper-Threading Technology Architecture and Microarchitecture 1 Hyper-Threading Technology Architecture and Microarchitecture”. In: 2002.

[2] Akhtar Jason Roberts. “Multi-Core Programming Increasing Performance through Software Multi-threading Shameem”. In: 2006.

[3] N. Tuck and D.M. Tullsen. “Initial observations of the simultaneous multithreading Pentium 4 processor”. In: 2003 12th International Conference on Parallel Architectures and Compilation Techniques. 2003, pp. 26–34. doi: 10.1109/PACT.2003.1237999

[4] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: vol. 5. 1. 1998, pp. 46–55. doi: 10.1109/99.660313

[5] Richard M S Brian J. Gough. In: An Introduction to GCC. 2004.

[6] Chris Lattner. In: LLVM and Clang: Next Generation Compiler Technology. 2008.

[7] T. Huang et al. “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System”. In: 1. Los Alamitos, CA, USA: IEEE Computer Society, 5555, pp. 1–1. doi: 10.1109/TPDS.2021.3104295

[8] Mark D. Hill and Michael R. Marty. “Amdahl’s Law in the Multicore Era”. In: vol. 41. 7. 2008, pp. 33–38. doi: 10.1109/MC.2008.209

[9] Hermann Schweizer, Maciej Besta, and Torsten Hoefer. “Evaluating the Cost of Atomic Operations on Modern Architectures”. In: 2020. arXiv: 2010.09852 [cs.DC]

[10] MOESI Protocol. 2021. url: en.wikipedia.org/wiki/MOESI_protocol

[11] Mohan Rajagopalan, Brian T Lewis, and Todd A Anderson. “Thread Scheduling for Multi-Core Platforms.” In: HotOS. 2007.

[12] Portable Hardware Locality. 2021. url: openmpi.org/projects/hwloc

[13] PyTorch. 2021. url: pytorch.org

[14] Rob van der Goot, Barbara Plank, and Malvina Nissim. “To Normalize, or Not to Normalize: The Impact of Normalization on Part-of-Speech Tagging”. In: 2017. arXiv: 1707.05116 [cs.CL]

[15] NVidia Cuda Toolkit. 2021. url: developer.nvidia.com