OpenMP experiments

I wrote my own C programs and used the output to show how the following clauses work. I benchmarked the C OpenMP programs, compiled in Visual Studio 2022, on my older desktop with 16GB RAM, Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz, 3501 Mhz, 4 Core(s), 8 Logical Processor(s).

Contents

[private and shared difference 1](#_Toc121859023)

[Work sharing – reduction 2](#_Toc121859024)

[reduction (op:list) 2](#_Toc121859025)

[Synchronization -- critical, atomic, barrier, and ordered 2](#_Toc121859026)

[critical -- 1b steps to compute pi with various (1-32) number of threads 2](#_Toc121859027)

[atomic -- 1b steps to compute pi with various (1-32) number of threads 3](#_Toc121859028)

[barrier – inexplicably encounter a rare *race condition* 3](#_Toc121859029)

[ordered -- 1b steps to compute pi with various (1-32) number of threads 5](#_Toc121859030)

[Data sharing -- private, firstprivate and lastprivate 6](#_Toc121859031)

[private 6](#_Toc121859032)

[firstprivate 6](#_Toc121859033)

[lastprivate 6](#_Toc121859034)

# private and shared difference

In OpenMP *private* is the clause that contains the variables that each thread in the OpenMP parallel region will have a copy of. These copies are not initialized, specifically, they do not have the value that the original variable had at the time it was passed in the private clause. In *#pragma omp parallel* ***private*** is the default, i.e. it does not have to be specified. The ***shared*** clause declares the variables in the list to be shared among all the threads. All threads have access to the same storage area for *shared* variables. However, unlike the theory, access is not the same, i.e. it depends on the location of the cache w.r.t. the specific threads.

**begin index = -1**

**begin tid = -1**

thread = 0

thread = 3| index = 3

| index = 0

thread = 2| index = 2

thread = 1| index = 1

**serial index = 3 (private)**

**serial tid = -1 (public)**

# Work sharing – reduction

## reduction (op:list)

The reduction clause creates a local copy of the list variable, here variable *sum* and op-initializes it, here *reduction (+:sum)* means all threads local variable copy *sum* is initialized to 0.

|  |  |  |
| --- | --- | --- |
| threads | execution time | pi |
| 1 | 6.16500s: 1 | 3.1415926495899708648096293473 |
| 2 | 3.07950s: 2 | 3.1415926495899011428036828875 |
| **4** | **1.94702s: 4** | **3.1415926495898212067459098762** |
| 8 | 1.95751s: 8 | 3.1415926495897692483083574189 |
| 16 | 2.65602s:16 | 3.1415926495898318648869462777 |
| 32 | 2.32203s:32 | 3.1415926495897559256320619170 |

The *reduction (+:sum)* appears to be the fastest on 4 threads.

# Synchronization – critical, atomic, barrier, and ordered

## critical – 1b steps to compute pi with various (1-32) number of threads

|  |  |  |
| --- | --- | --- |
| threads | execution time | pi |
| 1 | 5.98193s: 1 | 3.1415926535899707516819034936 |
| 2 | 3.15777s: 2 | 3.1415926535900071669971111987 |
| 4 | 1.96814s: 4 | 3.1415926535897682470022118650 |
| **8** | **1.59247s: 8** | **3.1415926535898273108671219234** |
| 16 | 1.61137s:16 | 3.1415926535898446303463060758 |
| 32 | 1.61256s:32 | 3.1415926535898455185247257759 |

Using *critical section* (mutex) guaranteed the variable **sum** from each of the threads was safely added (as critical implied the thread is finished before passing the *#pragma critical* line for each thread) to the main thread’s variable **pi**. As anticipated the physical threshold defined **the best** possible execution time on my old Win10 desktop using **8 threads on an 8 hyperthreaded CPUs**. *Critical section* is a preferred way to beat the *false sharing* phenomenon whereby each update will cause the cache lines to “slosh back and forth” between threads. *False sharing* is one of the major reasons for poor scaling up. *Critical section* should never be placed inside a parallel loop because it essentially would serialize it.

## atomic – 1b steps to compute pi with various (1-32) number of threads

|  |  |  |
| --- | --- | --- |
| threads | execution time | pi |
| **1** | **5.96108s: 1** | 3.1415926535899707516819034936 |
| 2 | 3.29295s: 2 | 3.1415926535900071669971111987 |
| 4 | 2.02469s: 4 | 3.141592653589768691091421715**1** |
| **8** | **1.62016s: 8** | **3.1415926535898277549563317734** |
| 16 | 1.62391s:16 | 3.141592653589845518524725775**9** |
| 32 | 1.69354s:32 | 3.1415926535898450744355159259 |

Very similar to the *critical* is the *atomic section* (also mutex, but for operator overload memory location, e.g. a++, ++b, --c, d--, or +=, -=, \*=, /=). In the *critical section* table it is obvious that the results are almost identical to those from the *atomic section*. The *atomic section* performed slightly faster only on 1 thread, and all other experiments a little bit slower than the *critical section*.

## barrier – inexplicably encounter a rare *race condition*

It worked as anticipated for the most part – waiting for all threads to finish at the barrier clause – however on rare occasions I encountered race conditions – see in red.

requested cores = 4

requested steps = 1000000000

thread # 1 sum = 0.7853981636473857941282972206

thread # 3 sum = 0.7853981626474767496759454843

thread # 2 sum = 0.7853981631474401536863183537

thread # 0 sum = 0.7853981641474657715562557314

**1.91105s: 4 pi = 3.1415926535897686910914217151**

requested cores = 4

requested steps = 1000000000

thread # 1 sum = 0.7853981636473857941282972206

thread # 2 sum = 0.7853981631474401536863183537

thread # 3 sum = 0.7853981626474767496759454843

thread # 0 sum = 0.7853981641474657715562557314

**2.08747s: 4 pi = 3.1415926535897682470022118650**

requested cores = 4

requested steps = 1000000000

thread # 2 sum = 0.7853981631474401536863183537

thread # 0 sum = 0.7853981641474657715562557314

thread # 1 sum = 0.7853981636473857941282972206

thread # 3 sum = 0.7853981626474767496759454843

**1.82107s: 4 pi = 2.3561944904423284263828008989**

In the above example 1 of 4 thread didn’t finish, which explains the wrong pi.

requested cores = 8

requested steps = 1000000000

thread # 1 sum = 0.3926990823236878869195720654

thread # 7 sum = 0.3926990808238750574332698307

thread # 6 sum = 0.3926990810737580051004158577

thread # 5 sum = 0.3926990813237561939175179759

thread # 2 sum = 0.3926990820737114584737526002

thread # 3 sum = 0.3926990818237206526397642392

thread # 4 sum = 0.3926990815736759454779303269

thread # 0 sum = 0.3926990825736425549941088775

**1.60893s: 8 pi = 3.1415926535898277549563317734**

requested cores = 8

requested steps = 1000000000

thread # 7 sum = 0.3926990808238750574332698307

thread # 5 sum = 0.3926990813237561939175179759

thread # 6 sum = 0.3926990810737580051004158577

thread # 2 sum = 0.3926990820737114584737526002

thread # 4 sum = 0.3926990815736759454779303269

thread # 3 sum = 0.3926990818237206526397642392

thread # 0 sum = 0.3926990825736425549941088775

thread # 1 sum = 0.3926990823236878869195720654

**1.60192s: 8 pi = 2.7488935720161520315230063716**

In the above example 1 of 8 thread didn’t finish, which explains the wrong pi.

requested cores = 8

requested steps = 1000000000

thread # 7 sum = 0.3926990808238750574332698307

thread # 6 sum = 0.3926990810737580051004158577

thread # 3 sum = 0.3926990818237206526397642392

thread # 5 sum = 0.3926990813237561939175179759

thread # 0 sum = 0.3926990825736425549941088775

thread # 1 sum = 0.3926990823236878869195720654

thread # 2 sum = 0.3926990820737114584737526002

thread # 4 sum = 0.3926990815736759454779303269

**1.64762s: 8 pi = 2.3561944909422321003944489348**

In the above example 2 of 8 thread didn’t finish, which explains the wrong pi.

Overall, I am unable to figure out why I encounter a rare race condition in *barrier condition* without further debug/investigation which doesn’t seem to be trivial in OpenMP.

## ordered – 1b steps to compute pi with various (1-32) number of threads

|  |  |  |
| --- | --- | --- |
| threads | execution time | pi |
| 1 | 6.03670s: 1 | 3.1415926535899707516819034936 |
| 2 | 3.20380s: 2 | 3.1415926535900071669971111987 |
| 4 | 2.10890s: 4 | 3.1415926535897682470022118650 |
| 8 | 1.68219s: 8 | 3.1415926535898277549563317734 |
| **16** | **1.64702s:16** | **3.1415926535898446303463060758** |
| 32 | 1.67611s:32 | 3.1415926535898450744355159259 |

The *ordered block* (also a mutex) executes in sequential order, however it is not clear what is the sequence? The thread id, the thread finishing place, or any other sequence. Similarly, to the other two *mutex blocks* – *critical and atomic* – the ordered block produced the same results for pi, but the execution times were slightly higher, perhaps for the overhead of ordering the threads? The fastest in my case was the 16 thread execution.

# Data sharing – private, firstprivate and lastprivate

## private

is the clause that contains the variables that each thread in the OpenMP parallel region will have a copy of. These copies are not initialized, specifically, they do not have the value that the original variable had at the time it was passed in the private clause.

## firstprivate

The *firstprivate* clause provides a superset of the functionality provided by the *private* clause. The *firstprivate* variable is initialized by the original value of the private variable when the parallel construct is encountered.

## lastprivate

The *lastprivate* clause provides a superset of the functionality provided by the private clause. The *lastprivate* variable is updated after the end of the parallel construct. The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with the *lastprivate* clause.

**begin a = -1**

**begin b = -1**

**begin c = -1**

a = 1 | b = -2 | c = 2

a = 3 | b = -2 | c = 4

a = 2 | b = -2 | c = 3

a = 0 | b = -2 | c = 1

**serial a = -1 (private)**

**serial b = -1 (firstprivate)**

**serial c = 4 (lastprivate)**