www.locuz.com

OpenMP Performance Tuning using Intel VTune Profiler

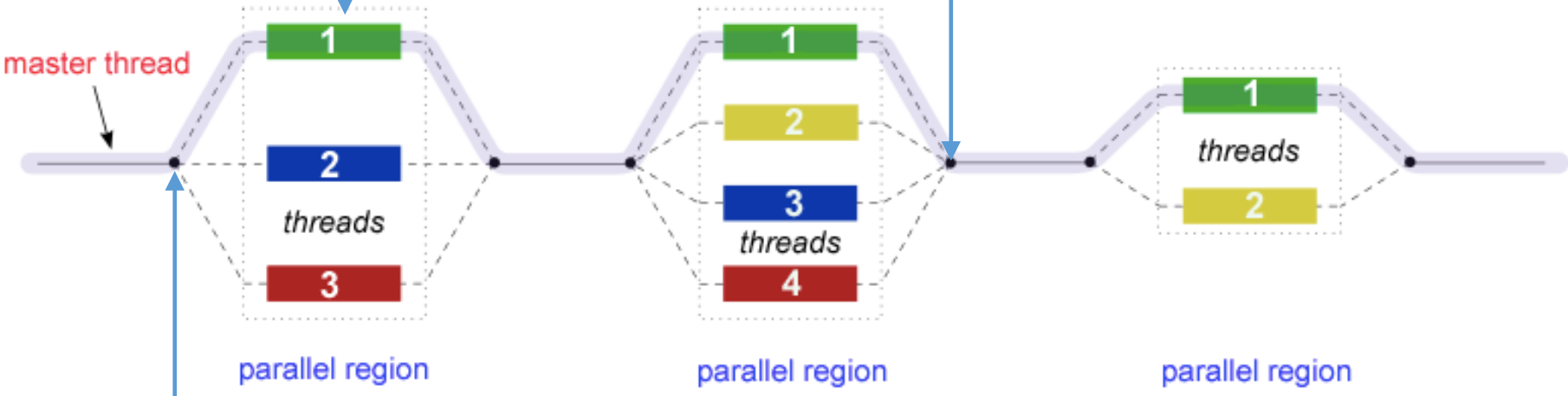
Presented By: Mandeep Kumar

Converge to the Cloud

There are several reasons for OpenMP overhead

When work-sharing constructs are used, the OpenMP runtime must divide the work between the threads. This is **scheduling overhead.**

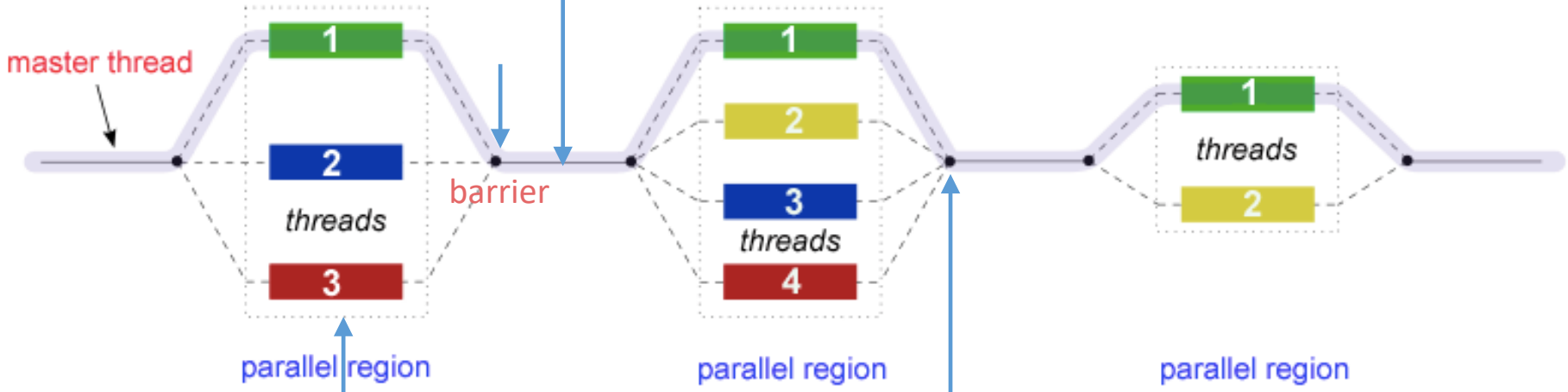
Whenever the OpenMP regions are reduction variables, the OpenMP runtime needs to spend more time performing reductions once all threads have finished executing. This is **reduction overhead.**



When the master thread is encounters the first parallel region, it needs to create a team of OpenMP worker threads. This is **thread creation overhead.**

Why do worker threads wait

When the master thread is executed a serial region, the worker threads are in the OpenMP runtime waiting for the next parallel region



When synchronization objects are used inside a parallel region, **threads can wait on a lock release**, contending with other threads for a shared resource **(Synchronization on locks)**

- When a thread finishes a parallel region, it waits at a barrier for the other threads to finish. (**Load imbalance**)
- The number of loop iterations < the number of working threads so several threads from the team are waiting at the barrier not doing useful work at all **(Not enough parallel work)**

Running OpenMP Analysis

Compiler flags:

- Build the app with -g to provide debug information and with: **-qopenmp**
- Add **-parallel-source-info=2** compiler option to embed source file name to a region name (optional)

Intel Compiler Command line Example:

- Linux: **icc -qopenmp -O2 -g -parallel-source-info=2 omptest.c -o omptestoutput**

Intel VTune Profiler Analysis:

- Choose Hotspots or Advanced Hotspots analysis type (with or w/o stacks)

OpenMP Imbalance and Scheduling Overhead

- This recipe shows how to detect and fix frequent parallel bottlenecks of OpenMP programs such as imbalance on barriers and scheduling overhead.
- **Application:** an application calculating prime numbers in a particular range. The main loop is parallelized with the OpenMP parallel for construct.
- **Compiler:** Intel Compiler
- **Performance Analysis Tools:** Intel VTune Profiler

Create a Baseline

- The initial version of the sample code uses the OpenMP parallel for pragma for the loop by numbers with the default scheduling that implies static

```
#include <stdio.h>
#include <omp.h>

#define NUM 100000000

int isprime( int x )
{
    for( int y = 2; y * y <= x; y++ )
    {
        if( x % y == 0 )
            return 0;
    }

    return 1;
}
```

```
int main( )
{
    int sum = 0;

#pragma omp parallel for reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }

    printf( "Number of primes numbers: %d", sum );

    return 0;
}
```

Compilation and Analysis

- **Compilation:**

  icc -qopenmp -O2 -g -parallel-source-info=2 ompprimev0.c -o ompprimev0

- I**ntel VTune Amplifier Analysis:**

  - vtune -collect hotspots -result-dir vtune_datav0 ./ompprimev0
  - vtune -report summary -result-dir vtune_datav0 -format html -report-output summaryv0.html

- Analysis collects and show important metrics that help understand such performance bottlenecks as CPU utilization (parallelism), memory access efficiency and vectorization.

- For application run with the Intel OpenMP runtime, you can benefit from special OpenMP efficiency metrics that help identify issues with threading parallelism.

- Start your analysis with the Summary view that displays application-level statistics. Flagged **Effective Physical Core Utilisation** metric (on some systems just **CPU Utilization**) signals a performance problem that should be explored.

Identify OpenMP Imbalance

Intel® oneAPI VTune™ Profiler 2021.1.1 Gold

Elapsed Time:	10.857s
CPU Time:	168.785s
Effective Time:	160.192s
Idle:	0s
Poor:	43.870s
Ok:	71.305s
Ideal:	45.016s
Over:	0s
Spin Time:	8.552s
Imbalance or Serial Spinning:	8.090s
Lock Contention:	0s
Other:	0.462s
Overhead Time:	0.041s
Creation:	0s
Scheduling:	0s
Reduction:	0s
Atomics:	0s
Other:	0.041s
Total Thread Count:	24
Paused Time:	0s

Top Hotspots:

Function	Module	CPU Time
isprime	ompprime	159.872s
__kmp_barrier	libiomp5.so	5.822s
__kmp_fork_barrier	libiomp5.so	2.730s
mainompparallel@21	ompprime	0.320s
_INTERNALba1907af::__kmp_itt_thread_name	libiomp5.so	0.021s
__kmp_get_global_thread_id_reg	libiomp5.so	0.020s

Effective Physical Core Utilization: 78.6% (9.436 out of 12)

```
The metric value is low, which may signal a poor physical CPU
cores utilization caused by:
    - load imbalance
    - threading runtime overhead
    - contended synchronization
    - thread/process underutilization
    - incorrect affinity that utilizes logical cores instead of
physical cores
Explore sub-metrics to estimate the efficiency of MPI and OpenMP
parallelism or run the Locks and Waits analysis to identify
parallel bottlenecks for other parallel runtimes.
```

Effective Logical Core Utilization: 65.3% (15.665 out of 24)

```
The metric value is low, which may signal a poor logical CPU
cores utilization. Consider improving physical core utilization
as the first step and then look at opportunities to utilize
logical cores, which in some cases can improve processor
throughput and overall performance of multi-threaded
applications.
```

Collection and Platform Info:

Application Command Line:	./ompprime
Operating System:	4.15.18 DISTRIB_ID=Ubuntu DISTRIB_RELEASE=18.04
Computer Name:	s001-n017

Apply Dynamic Scheduling

- The imbalance could be caused by static work distribution and assigning large numbers to particular threads while some of the threads processed their chunks with small numbers quickly and wasted the time on the barrier.

- To eliminate this imbalance, apply dynamic scheduling with the default parameters:

```
#pragma omp parallel for schedule(dynamic) reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }
```

- Re-compile the application and compare the execution time versus the original performance baseline to verify your optimization.

Identify OpenMP Scheduling Overhead

Intel® oneAPI VTune™ Profiler 2021.1.1 Gold

Elapsed Time:	8.632s
CPU Time:	183.134s
Effective Time:	153.352s
Idle:	0s
Poor:	36.158s
Ok:	95.684s
Ideal:	21.510s
Over:	0s
Spin Time:	2.546s
Imbalance or Serial Spinning:	2.514s
Lock Contention:	0s
Other:	0.032s
Overhead Time:	27.237s

A significant portion of CPU time is spent in synchronization or threading overhead. Consider increasing task granularity or the scope of data synchronization.

Creation:	0s
Scheduling:	27.205s

The thread scheduling threading runtime function consumed a significant amount of CPU time. This occurs when the threads are frequently returning to the scheduler for more work, which can indicate an inefficient work chunk size. To reduce scheduling overhead, increase the size of tasks or iteration chunks being performed by working threads.

Reduction:	0s
Atomics:	0s
Other:	0.032s
Total Thread Count:	24
Paused Time:	0s

Top Hotspots:

Function	Module	CPU Time
isprime	ompprimev1	152.534s
[OpenMP dispatcher]	libiomp5.so	25.897s
__kmp_fork_barrier	libiomp5.so	2.546s
mainompparallel@21	ompprimev1	0.778s
_INTERNAL5d437bb1::[OpenMP dispatcher]	libiomp5.so	0.636s
[Others]	N/A	0.744s

Apply Dynamic Scheduling with a Chunk Parameter

- Use the chunk parameter 20 for the schedule clause as follows:

```
#pragma omp parallel for schedule(dynamic,20) reduction (+:sum)
    for( int i = 2; i <= NUM ; i++ )
    {
        sum += isprime ( i );
    }
```

- Re-compile the application and compare the execution time versus the original performance baseline to verify your optimization.

Final Performance

Intel® oneAPI VTune™ Profiler 2021.1.1 Gold

Elapsed Time:	7.714s
CPU Time:	159.155s
Effective Time:	155.221s
Idle:	0.010s
Poor:	0.345s
Ok:	15.132s
Ideal:	139.734s
Over:	0s
Spin Time:	3.202s
Imbalance or Serial Spinning:	3.086s
Lock Contention:	0s
Other:	0.116s
Overhead Time:	0.732s
Creation:	0s
Scheduling:	0.712s
Reduction:	0s
Atomics:	0s
Other:	0.020s
Total Thread Count:	24
Paused Time:	0s

Top Hotspots:

Function	Module	CPU Time
isprime	ompprimev2	154.915s
__kmp_fork_barrier	libiomp5.so	3.202s
[OpenMP dispatcher]	libiomp5.so	0.520s
mainompparallel@21	ompprimev2	0.296s
_INTERNAL5d437bb1::[OpenMP dispatcher]	libiomp5.so	0.156s
[Others]	N/A	0.066s

Effective Physical Core Utilization: 86.8% (10.413 out of 12)
Effective Logical Core Utilization: 86.4% (20.739 out of 24)

Thanks!