Software

# CASE STUDY OF SHARED-MEMORY PARALLELIZATION

Jackson Marusarz - Intel

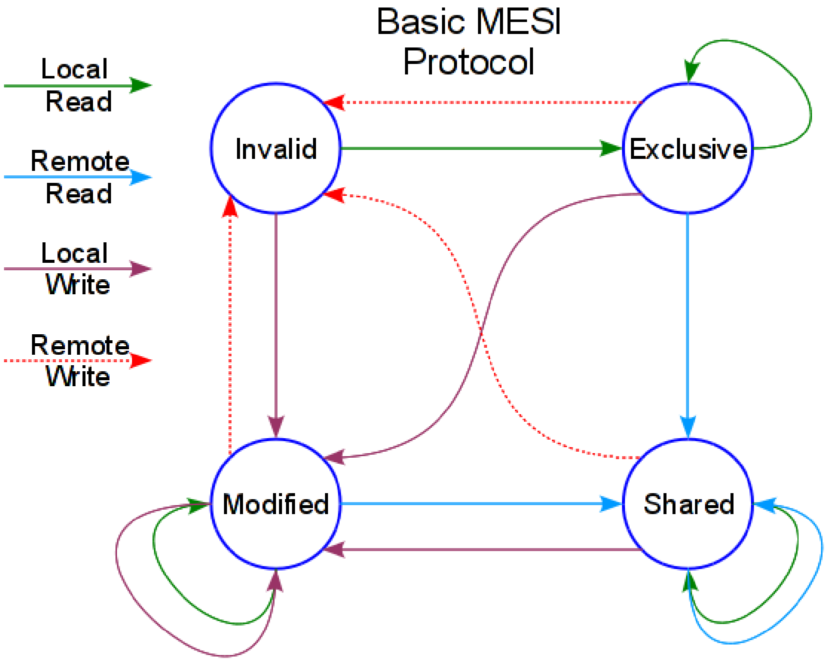
# Agenda

- Shared Memory Review
- Performance implications of shared memory hardware
  - Data sharing
  - Contested accesses
- Performance implications of shared memory software
  - Data races
  - Deadlocks
  - Poor synchronization
  - Static thread scheduling
  - Scalability
  - Load imbalance
  - Oversubscription
  - Lock contention

# Shared Memory Review

# Performance Implications of Shared-Memory Hardware

- Modern CPUs have a shared address space for all the cores
- Need to maintain correctness, as different cores work on the same data
- Hardware protocols maintain coherency, but can have performance impacts




https://people.cs.pitt.edu/~xianeizhang/notes/cache.html

# Performance Implications of Shared-Memory Hardware

- Metrics available in Intel® VTune™ Amplifier General Exploration

**General Exploration** General Exploration viewpoint (change)

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Grouping: Function / Call Stack

| Function / Call Stack | Back-End Bound | | | | |
|---|---|---|---|---|---|
| | Memory Bound | | | | |
| | L2 Bound | L3 Bound | | | |
| | | Contested Accesses | Data Sharing | L3 Latency | SQ Full |
| grid_intersect | 4.9% | 0.6% | 0.4% | 97.8% | 0.0% |
| sphere_intersect | 0.0% | 0.0% | 0.0% | 33.1% | 0.0% |
| grid_bounds_intersect | 0.0% | 0.0% | 0.0% | 30.9% | 0.0% |

# Data Sharing

**Why:** Sharing clean data (read sharing) among cores (at L2 level) has a penalty at least the first time due to coherency

**What Now:** If this metric is highlighted for your hotspot, locate the source code line(s) that is generating HITs by viewing the source.

- Look for the MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT_PS event which will tag to the next instruction after the one that generated the HIT.
- Use knowledge of the code to determine if real or false sharing is taking place. Make appropriate fixes:
  - For real sharing, reduce sharing requirements
  - For false sharing, pad variables to cache line boundaries

# Contested Accesses

**Why:** Sharing modified data among cores (at L2 level) can raise the latency of data access

**What Now:** If this metric is highlighted for your hotspot, locate the source code line(s) that is generating HITMs by viewing the source.

- Look for the MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM_PS event which will tag to the next instruction after the one that generated the HITM.
- Use knowledge of the code to determine if real or false sharing is taking place. Make appropriate fixes:
  - For real sharing, reduce sharing requirements
  - For false sharing, pad variables to cache line boundaries

# Performance Implications of Shared-Memory Software

- Data races
- Deadlocks
- Poor synchronization
- Static thread scheduling
- Scalability
- Load imbalance
- Oversubscription
- Lock contention

# Data Races - SSSP

```
do {
auto old_dst_dist = dst_data.load();
auto new_dst_dist = src_data.load() + w;
if (new_dst_dist < old_dst_dist) {
  dst_data = new_dst_dist;
  swapped = true;
}
} while (!swapped);
```

Shared variables unprotected

Application may:

- Crash Immediately
- Hang
- Run but give incorrect results
- Run and give correct results
- Run correctly 99 times but crash once (usually once you ship it to customers)

Non-determinism is always a concern in parallel programming. It may depend on how the OS decides to schedule threads.

# Data Races - SSSP

```
do {
auto old_dst_dist = dst_data.load();
auto new_dst_dist = src_data.load() + w;
if (new_dst_dist < old_dst_dist) {
  dst_data = new_dst_dist;
  swapped = true;
}
} while (!swapped);
```

Shared variables unprotected

Add a critical section

```
do {
auto old_dst_dist = dst_data.load();
auto new_dst_dist = src_data.load() + w;

if (new_dst_dist < old_dst_dist) {
  pthread_mutex_lock(&swap_mutex);
  dst_data = new_dst_dist;
  swapped = true;
  pthread_mutex_unlock(&swap_mutex);
}
} while (!swapped);
```

# Threading Problems- Deadlock

```
CRITICAL_SECTION cs1;
CRITICAL_SECTION cs2;
int x = 0;
int y = 0;
InitializeCriticalSection(&cs1); // Allocation Site (cs1)
InitializeCriticalSection(&cs2); // Allocation Site (cs2)
```

Thread #1

```
EnterCriticalSection(&cs1);
x++;
EnterCriticalSection(&cs2);
y++;
LeaveCriticalSection(&cs2);
LeaveCriticalSection(&cs1);
```

Thread #2

```
EnterCriticalSection(&cs2);
y++;
EnterCriticalSection(&cs1);
x++;
LeaveCriticalSection(&cs1);
LeaveCriticalSection(&cs2);
```

## Deadlock

1. EnterCriticalSection(&cs1); in thread #1
2. EnterCriticalSection(&cs2); in thread #2

# Poor Synchronization - SSSP

```
do {
auto old_dst_dist = dst_data.load();
auto new_dst_dist = src_data.load() + w;

if (new_dst_dist < old_dst_dist) {
  pthread_mutex_lock(&swap_mutex);
  &dst_data = new_dst_dist;
  pthread_mutex_unlock(&swap_mutex);
}
else {
swapped = true;
}
} while (!swapped);
```

## Elapsed Time: 39.869s

| | |
|---|---|
| CPU Time: | 436.670s |
| Effective Time: | 269.428s |
| Spin Time: | 167.242s |
| Overhead Time: | 0s |
| Total Thread Count: | 18 |
| Paused Time: | 1.265s |

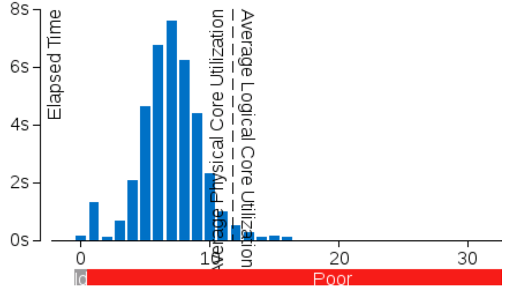
## Top Hotspots

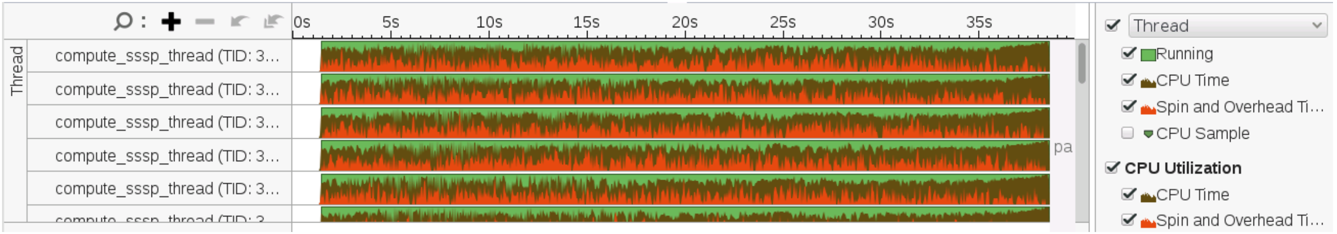
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

| Function | Module | Module | CPU Time |
|---|---|---|---|
| _L_unlock_697 | libpthread.so.0 | libpthread.so.0 | 176.103s |
| _L_lock_791 | libpthread.so.0 | libpthread.so.0 | 148.576s |

## Effective CPU Utilization Histogram

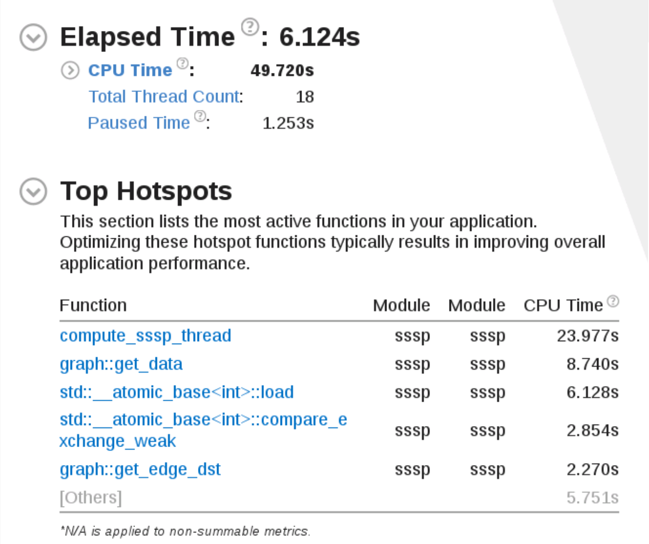
This histogram displays a percentage of the wall time the sp
CPU utilization value.



# Efficient Synchronization - SSSP

```
do {

auto old_dst_dist = dst_data.load();

auto new_dst_dist = src_data.load() + w;


if (new_dst_dist < old_dst_dist) {

swapped =
std::atomic_compare_exchange_weak(&dst_data,
&old_dst_dist, new_dst_dist);

changed |= swapped;

}

else {

swapped = true;

}

} while (!swapped);
```



## Elapsed Time: 6.124s

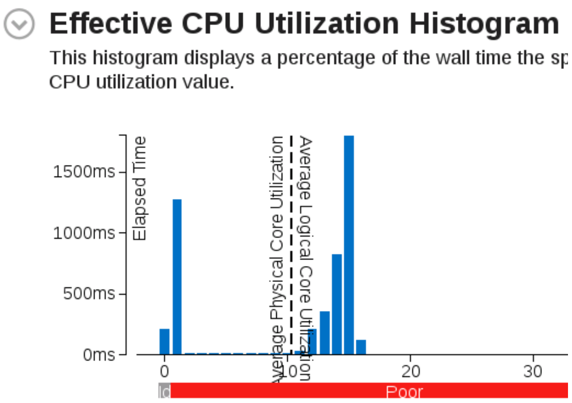
CPU Time: 49.720s
Total Thread Count: 18
Paused Time: 1.253s

## Top Hotspots

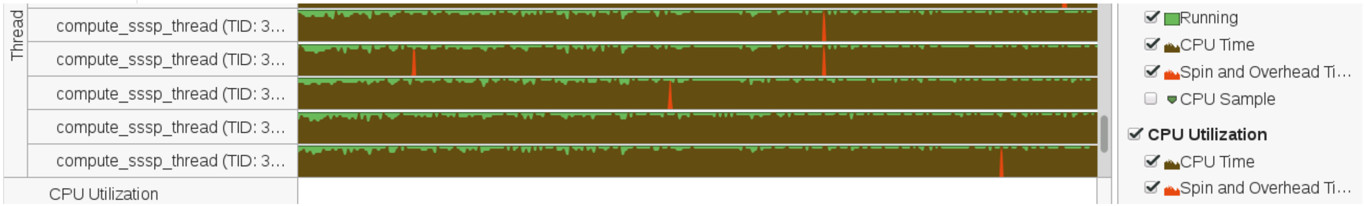
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

| Function | Module | Module | CPU Time |
|---|---|---|---|
| compute_sssp_thread | sssp | sssp | 23.977s |
| graph::get_data | sssp | sssp | 8.740s |
| std::__atomic_base<int>::load | sssp | sssp | 6.128s |
| std::__atomic_base<int>::compare_exchange_weak | sssp | sssp | 2.854s |
| graph::get_edge_dst | sssp | sssp | 2.270s |
| [Others] | | | 5.751s |

*N/A is applied to non-summable metrics.







# Efficient Synchronization - SSSP

```
do {
auto old_dst_dist = dst_data.load();
auto new_dst_dist = src_data.load() + w;

if (new_dst_dist < old_dst_dist) {
swapped =
std::atomic_compare_exchange_weak(&dst_data,
&old_dst_dist, new_dst_dist);
changed |= swapped;
}
else {
swapped = true;
}
} while (!swapped);
```

Original

**Elapsed Time: 39.869s**

| | |
|---|---|
| CPU Time: | 436.670s |
| Effective Time: | 269.428s |
| Spin Time: | 167.242s |
| Overhead Time: | 0s |
| Total Thread Count: | 18 |
| Paused Time: | 1.265s |

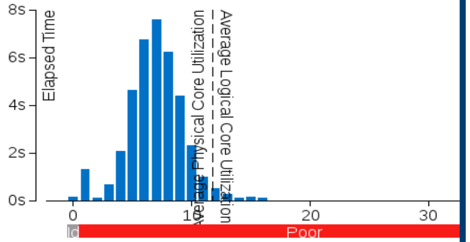
**Top Hotspots**

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

| Function | Module | Module | CPU Time |
|---|---|---|---|
| _L_unlock_697 | libpthread.so.0 | libpthread.so.0 | 176.103s |
| _L_lock_791 | libpthread.so.0 | libpthread.so.0 | 148.576s |

**Effective CPU Utilization Histogram**

This histogram displays a percentage of the wall time the sp CPU utilization value.




Efficient

**Elapsed Time: 6.124s**

| | |
|---|---|
| CPU Time: | 49.720s |
| Total Thread Count: | 18 |
| Paused Time: | 1.253s |

**Top Hotspots**

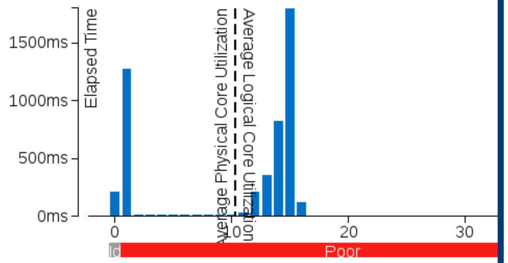
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

| Function | Module | Module | CPU Time |
|---|---|---|---|
| compute_sssp_thread | sssp | sssp | 23.977s |
| graph::get_data | sssp | sssp | 8.740s |
| std::__atomic_base<int>::load | sssp | sssp | 6.128s |
| std::__atomic_base<int>::compare_exchange_weak | sssp | sssp | 2.854s |
| graph::get_edge_dst | sssp | sssp | 2.270s |
| [Others] | | | 5.751s |

*N/A is applied to non-summable metrics.

**Effective CPU Utilization Histogram**

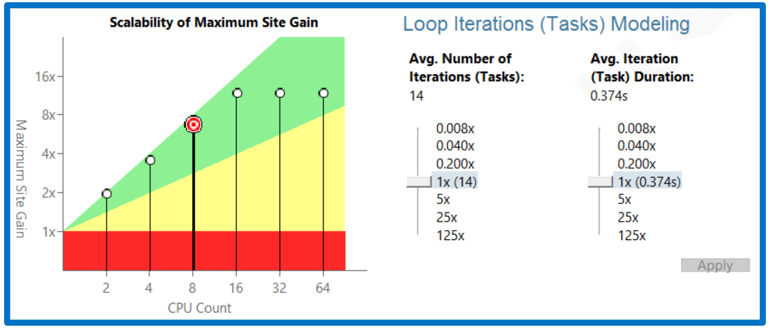
This histogram displays a percentage of the wall time the sp CPU utilization value.



# Static Thread Scheduling - SSSP

## Hardcoding thread counts or relying on inputs can have performance impacts

```
  src_node = std::atoi(argv[2]) - 1;
  initialize_sssp();

#ifdef SP2018_CS377P_PARALLEL
  num_threads = std::atoi(argv[3]);
#endif
  compute_sssp();
```




* I know this was used to teach concepts

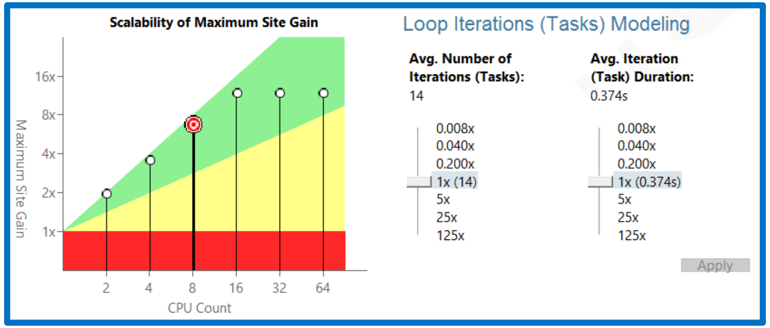
```
 …
   NUM_THREADS = 4;
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   int chunk = limit/NUM_THREADS;
   for(t=0;t<NUM_THREADS;t++){
     range *r = new range();
…
```

# Static Thread Scheduling - SSSP

## Hardcoding thread counts or relying on inputs can have performance impacts

```
  src_node = std::atoi(argv[2]) - 1;
  initialize_sssp();

#ifdef SP2018_CS377P_PARALLEL
  num_threads = std::atoi(argv[3]);
#endif
  compute_sssp();
```

* I know this was used to teach concepts




```
…
  NUM_THREADS = 4;  NUM_THREADS = get_num_procs();
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  int chunk = limit/NUM_THREADS;
  for(t=0;t<NUM_THREADS;t++){
    range *r = new range();
…
```

Use dynamic processor identification or scalable runtime library like OpenMP or Threading Building Blocks

```
#pragma omp parallel for
```

# Scalability is Not a Given - SSSP

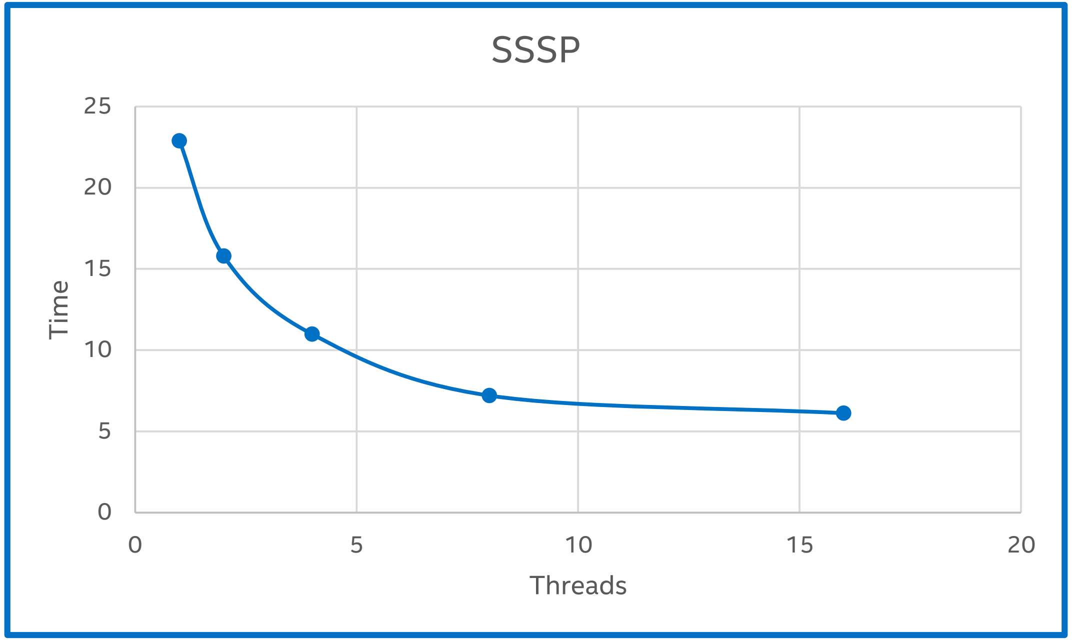
**Elapsed Time: 22.947s**
CPU Time: 21.470s
Total Thread Count: 3
Paused Time: 1.257s

**Elapsed Time: 15.791s**
CPU Time: 25.830s
Total Thread Count: 4

**Elapsed Time: 10.978s**
CPU Time: 30.200s
Total Thread Count: 6

**Elapsed Time: 7.239s**
CPU Time: 34.140s
Total Thread Count: 10

**Elapsed Time: 6.124s**
CPU Time: 49.720s
Total Thread Count: 18
Paused Time: 1.253s

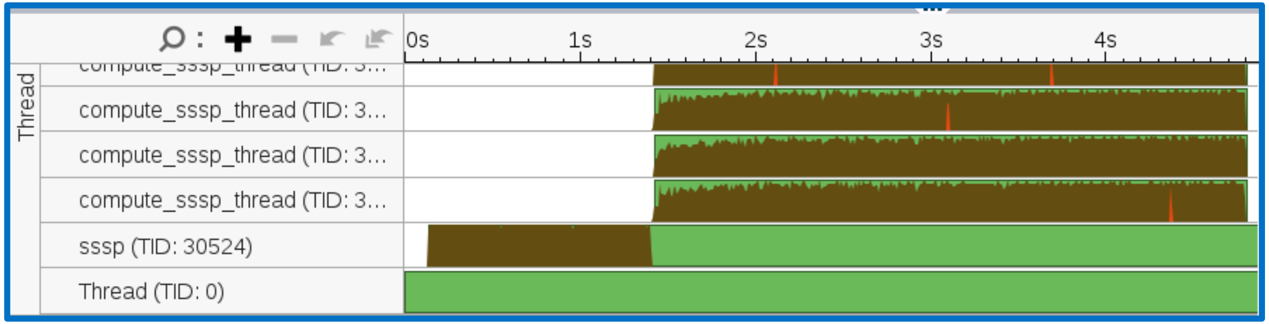


# Scalability is Not a Given - SSSP

**Strong Scaling** - Solution time scales with the number of processors for a fixed *total* problem size.

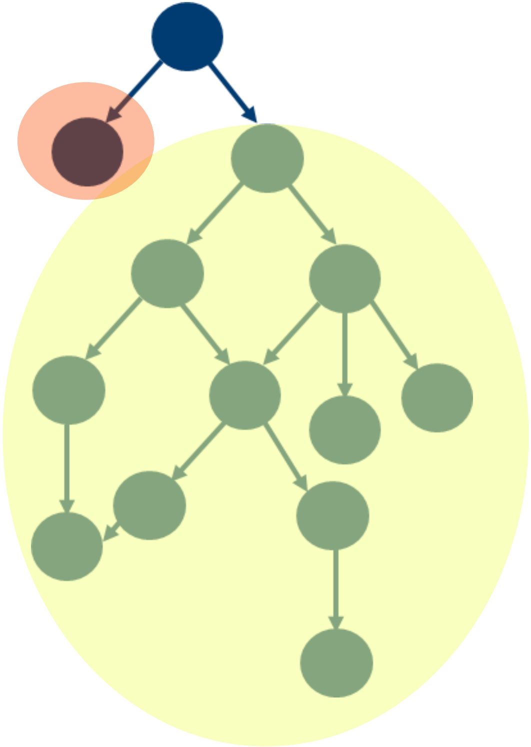
**Weak Scaling** - Solution time scales with the number of processors for a fixed *total* problem size *per processor* - i.e. scales if the problem size also scales

**Serial Time** will always be a limiting factor as well.



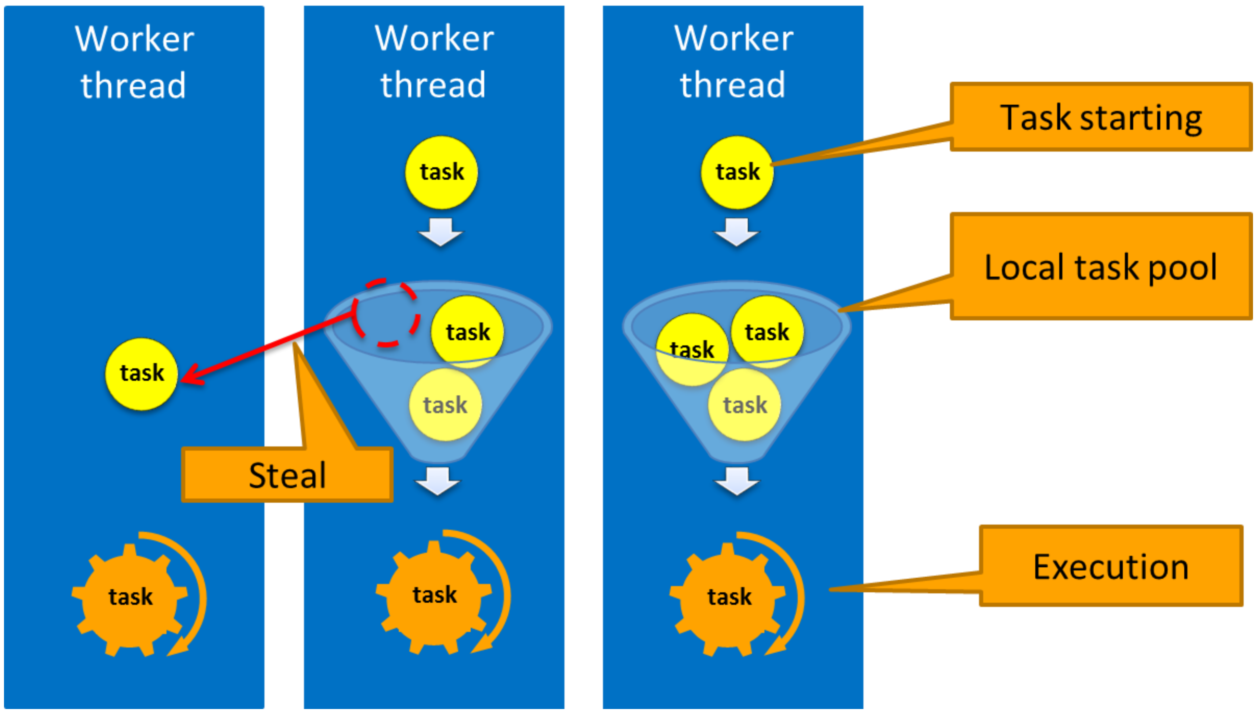
# Load Balancing

- Work should be divided among threads evenly
- What is even?
  - Loop Iterations? Elements to process?
- Intelligent parallelism uses dynamic workload balancing
  - Work stealing and/or dynamic chunking



Should we divide work by subgraphs?

# Load Balancing – Work Stealing



# Example – Calculating Prime Numbers

```
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD msBegin = timeGetTime();

#pragma omp parallel for
    for(int p = 3; p <= limit; p += 2) {
        if (IsPrime(p)) Tick();
    }
    DWORD msDuration = timeGetTime() - msBegin;

    printf("MS: %d\n", msDuration);
    printf("primes = %d\n", primes);
    return primes != correctCount;
}
```

- Is 7 prime?
- Is 76853341 prime?

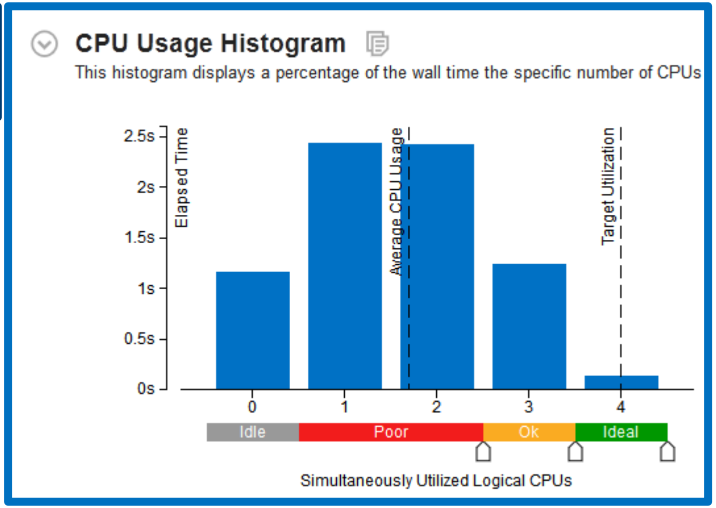
Static Scheduling/Chunking:

- Check 1-10000
- Check 10001-20000
- Check 20001-30000
- …

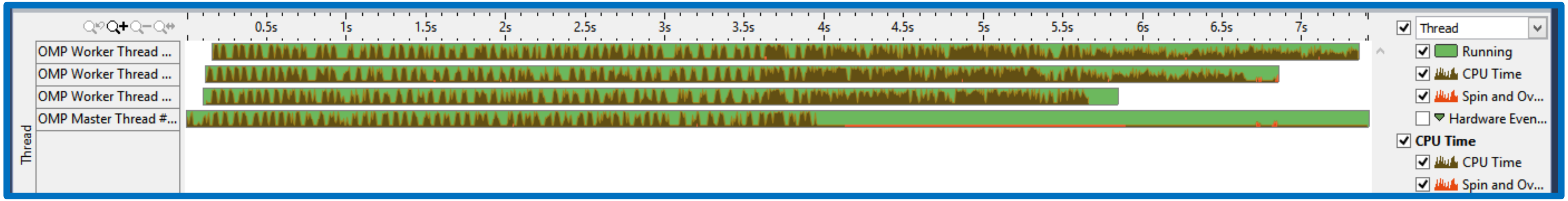
# Example – Calculating Prime Numbers

```
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD msBegin = timeGetTime();

#pragma omp parallel for
    for(int p = 3; p <= limit; p += 2) {
        if (IsPrime(p)) Tick();
    }
    DWORD msDuration = timeGetTime() - msBegin;

    printf("MS: %d\n", msDuration);
    printf("primes = %d\n", primes);
    return primes != correctCount;
}
```

OpenMP uses Static Scheduling



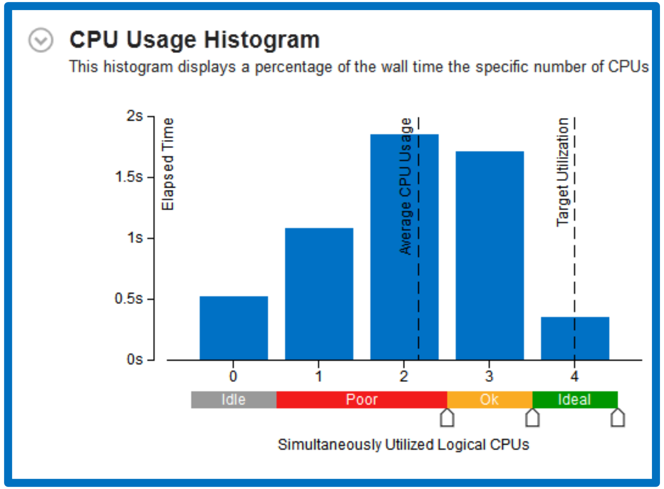

Load Imbalance



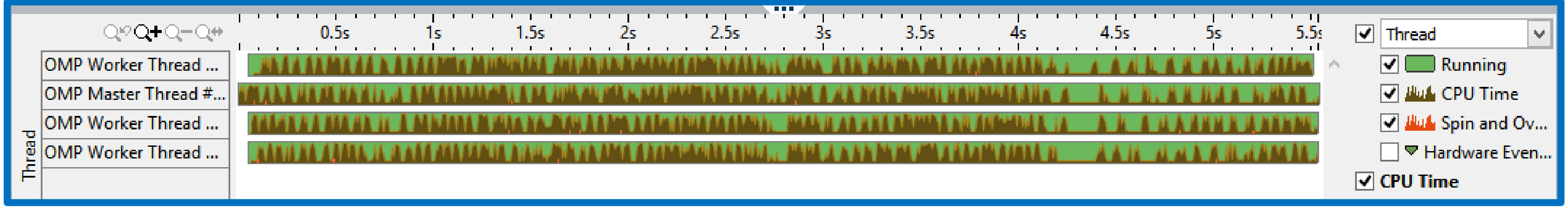
# Example – Calculating Prime Numbers

```
int _tmain(int argc, _TCHAR* argv[])
{
    DWORD msBegin = timeGetTime();

#pragma omp parallel for schedule (dynamic, 1000)
    for(int p = 3; p <= limit; p += 2) {
        if (IsPrime(p)) Tick();
    }
    DWORD msDuration = timeGetTime() - msBegin;

    printf("MS: %d\n", msDuration);
    printf("primes = %d\n", primes);
    return primes != correctCount;
}
```

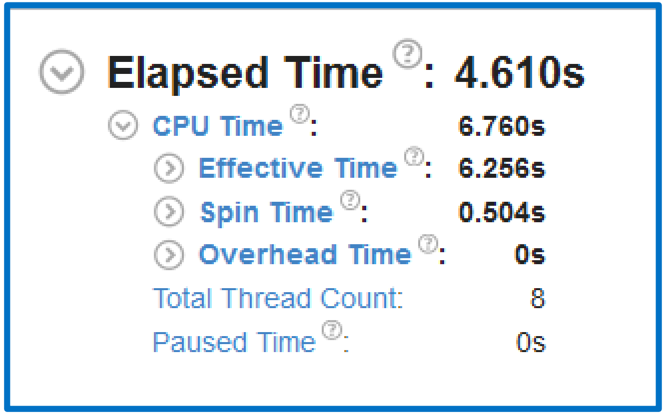
Switch to Dynamic Scheduling

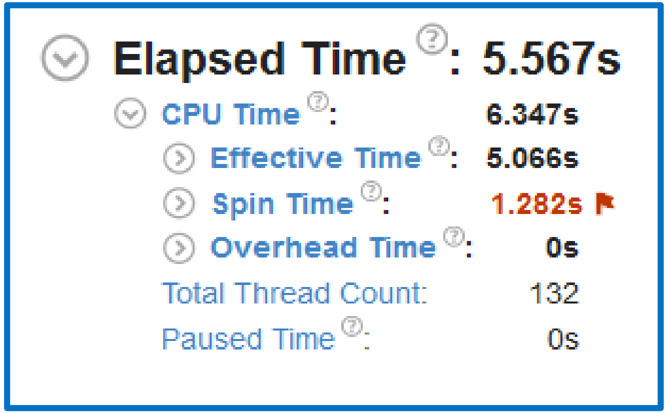


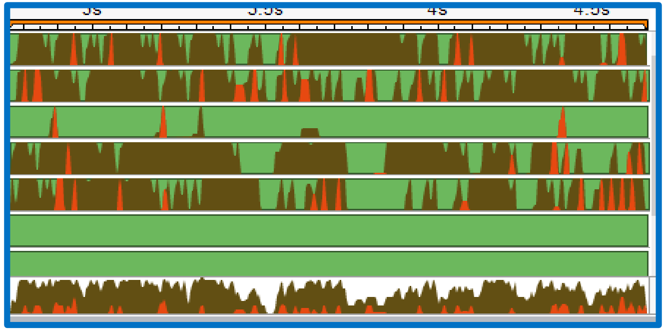

More Balanced Threads

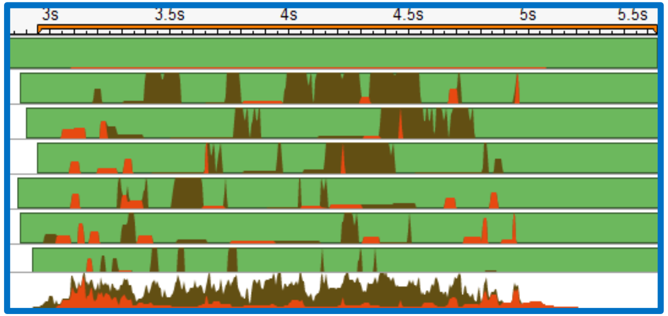


# Oversubscription



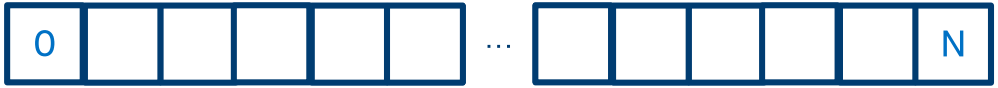






Common Causes:
- Nested Parallelism
- Manual Threading
- Library Usage

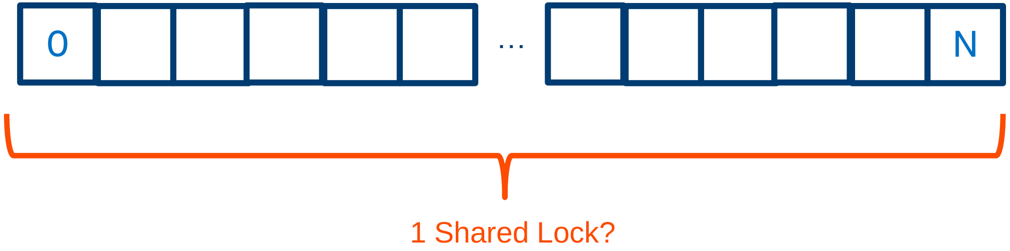
# Lock Contention

- Acquiring and releasing a lock isn't free – it has overhead
- Threads waiting for a lock also impacts performance
- How do we balance these?

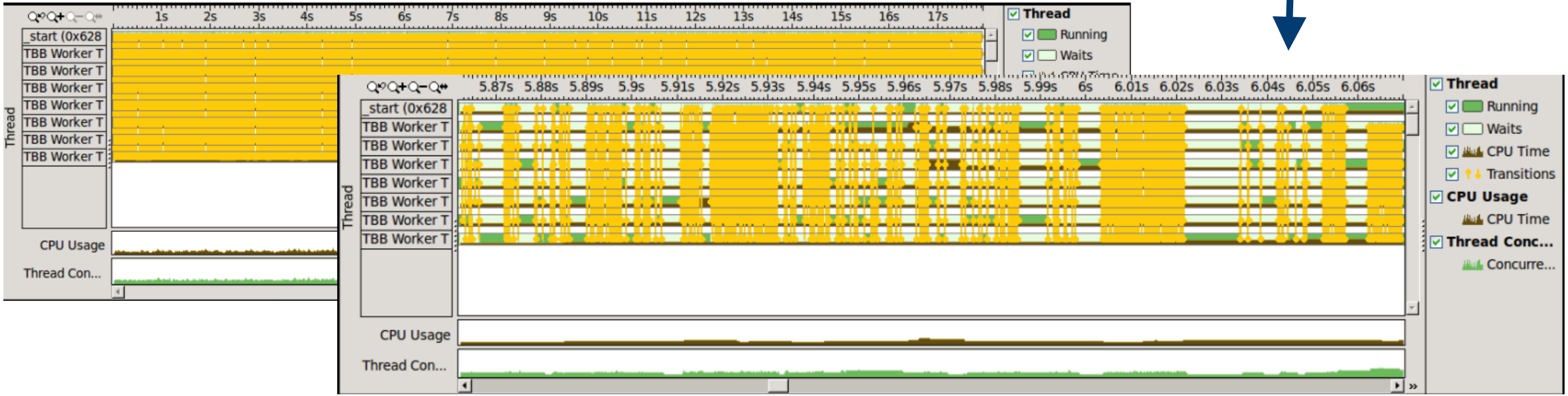



- Imagine an array that multiple threads read and write

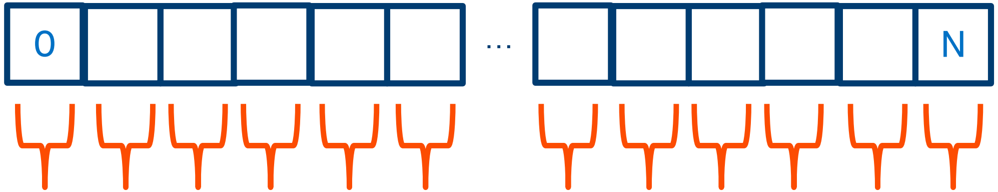
# Lock Contention




Dense lock contention and stalls



# Lock Contention




1 Lock per Element?

## Lots of locking overhead

## What can we do?

- Adjust lock granularity
- Using lock free or thread safe data structures
  - tbb::atomic<int> primes;
  - tbb::concurrent_vector<int> all_primes;
- Local storage and reductions

# Summary

- Programming for shared memory is difficult
- Correctness and performance issues are unique
- Issues are from hardware and software
  - Data sharing
  - Contested accesses
  - Deadlock
  - Data races
  - Poor synchronization
  - Static thread scheduling
  - Scalability
  - Load imbalance
  - Oversubscription
  - Lock contention

Software

# Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.



**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804