TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

Manycore Optimization: Multitasking

John Cazes

February 16, 2018

Multitasking

Basic Concepts: A task is an active program and related resources

- *Multiprocessing*: concurrently running tasks are processes
  - processes do not share memory by default
- *Multithreading*: concurrently running tasks are threads
  - threads do share memory by default
- *speedup*: runtime of serial versus parallel $\frac{T_s}{T_p}$
- *efficiency*: speedup versus number of tasks $\frac{T_s}{pT_p}$
  - Typically less than 1
- Process/memory *affinity*: pinning tasks to sets of cores and memory
- *Contention*: tasks competing for bandwidth and ownership of data
- *Dependencies*: execution of code by one task needs results from another (similar to dependency analysis in vectorization)
- *Granularity*: amount of work per task

Granularity: Load Balancing vs Overhead

Granularity is the amount of work per task

- It is important to balance granularity for parallel efficiency
- Too much granularity can lead to load imbalance
  - Some tasks could be idle for long periods, waiting for others to complete
- Too little granularity can lead to large overhead
  - Parallelizing has overhead
    - Partitioning problem
    - Task startup
    - Scheduling work to tasks
    - Communicating/sharing data between tasks
    - Lock management and synchronization

Load Imbalance Example

A loop with 10 iterations, divided among 9 threads

```
for(j=0;j<10;j++){...}
```

- Each iteration takes 1 thread 1 day
- 9 iterations are done the 1st day by 9 threads
- 1 iteration is done the 2nd day by 1 thread (9 threads are idle for a day :( )
- Negligible communication overhead but poor load balance
- $Speedup = 10/2 = 5$ vs $Speedup = 10$ in perfect balance

Parallel Overhead Example

A loop with 10 iterations, divided among 10 threads

```
for(j=0;j<10;j++){...}
```

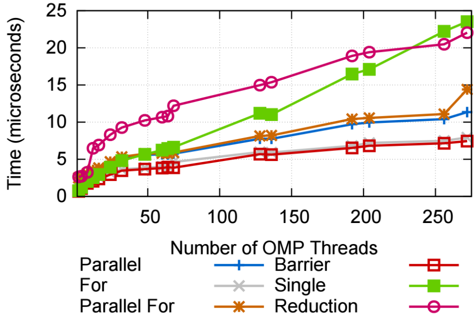
- Setup of parallelization requires 1ms
- Each iteration takes 1 thread 0.1ms
- 10 iterations are done in parallel in 0.1 ms
- Total parallel time is 1.1ms $\rightarrow$ serial would have run in 1ms!
- $Speedup = 1/1.1$

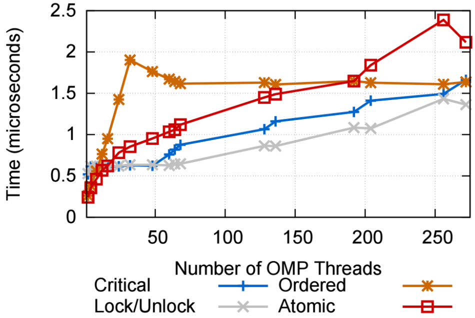
OpenMP

We will focus on OpenMP (OMP) here

- MPI considerations are similar on Manycore to Multicore
  - One caveat to that: Typically use less than 68 MPI tasks per node
  - MPI has a larger memory footprint than OpenMP
  - MPI composed of processes
- OpenMP is very common in scientific applications
  - OpenMP composed of threads
  - shared memory multithreading API for C,C++,Fortran
- Other approaches to parallelization exist:
  - multiprocessing, e.g. Python multiprocessing module
  - launchers
  - pthreads

OMP Overhead: EPCC Benchmark on KNL





Impact of Overhead and Imbalance on Performance

Measure in FLOPs lost

- KNL: 3.264 TF theoretical peak $\rightarrow$ $\sim$2 TF (e.g. HPL)
- Overhead for `parallel` directive: $0.58\mu s \rightarrow 11\mu s$
- $\frac{2TFLOPs}{s}(0.58 \rightarrow 11)\mu s = (1.2 \rightarrow 22)MFLOP$
- Cost of parallelization needs to be worthwhile $\equiv$ sufficient work per task
- Short loops or small amount of work is rarely worth parallelizing
- Any reduction in overhead you can get is worth it!

Overhead reduction: Thread Pools

OMP minimizes overhead of thread creation *automatically* (GCC & Intel)

- Thread pool is created the first time a parallel construct is encountered
- Thread creation can take thousands of microseconds
- Threads from pool are reused for subsequent parallel work
- If new threads are needed they are created and added to pool
- Thread pool creation overhead occurs only in first call

```
#pragma parallel for // thread creation overhead
    for(j=0;j<10;j++){...}
    .
    .
#pragma parallel for // no thread creation overhead
    for(k=0;k<100;k+){...}
```

Loop Modifications for Parallel Performance

- Loops are a common place for parallelization
  - Divide iterations among threads
  - Iterations must be independent
- Loop modifications can make parallelization possible
  - Remove dependencies between iterations
- Loop modifications can improve granularity
  - Can improve load balancing
  - Can reduce overhead

General tip for good loop granularity in loops

Parallelize over loops with high iteration counts relative to the number of threads.

- Overhead is less significant: Time to setup versus time spent on work is small
- Load imbalance is less significant: Time for “remainder" iterations versus time all threads are working is smaller

Loop fission

Can enable parallelization by removing dependencies

Later iteration needs result from earlier iteration (RAW dependency):

```
for (j=1;j<N;j++){
     a[j]=b[j-1];
     b[j]+=1;
}
```

Loop fission can remove dependency and enable parallelization:

```
#pragma parallel for
for (j=1;j<N;j++){
     b[j]+=1;
}
```

```
#pragma parallel for
for (j=1;j<N;j++){
     a[j]=b[j-1];
}
```

Also can improve cache reuse

Loop Fusion

Increases granularity and reduces overhead from parallelization and loop

```
#pragma parallel for
    for (j=0;j<N;j++)
         a[j]=b[j]*c[j];
#pragma parallel for
     for (j=0;j<N;j++)
         d[j]=e[j]*f[j];
```

Loop fusion

```
#pragma parallel for
    for (j=0;j<N;j++){
         a[j]=b[j]*c[j];
         d[j]=e[j]*f[j];
    }
```

Also more data is in flight which is good if BW is not saturated

Merge nested loops

Increases iteration count → better load balancing

```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        d[i][j]=e[i]*f[j];
```

Merge

```
for (k=0;k<N*N;k++){
    i=k/N;
    j=k%N;
    d[i][j]=e[i]*f[j];
}
```

OpenMP has a Directive for this: `collapse(#loops)`

```
#pragma omp parallel for collapse(2)
    for (i=0;i<N;i++)
        for (j=0;j<N;j++)
            d[i][j]=e[i]*f[j];
```

Reduce overhead

OpenMP `parallel` has some overhead (even with thread pools)

ok

```
for (rep=0;rep<M;rep++)
   #pragma omp parallel for
       for (j=0;j<N;j++)
            A[j]+=x[j];
```

better

```
#pragma omp parallel
{
 for (rep=0;rep<M;rep++)
   #pragma omp for
     for (j=0;j<N;j++)
          A[j]+=x[j];
}
```

Load balancing with OpenMP Scheduling

OpenMP has 3 *scheduling* types

```
#pragma omp parallel for schedule(scheduletype, chunksize)
 for(k=0;k<N*N;k++)
      work(k);
```

- static
  - default scheduler
  - same iterations for every thread
  - very little overhead
  - no load balancing
- dynamic
  - assign work in chunksize (default=1)
  - when thread completes, new work is assigned
  - overhead from synchronization
- guided
  - similar to dynamic
  - initially gives large chunks of iterations to tasks
  - gradually reduces chunks of iterations to min of chunksize
  - sometimes overhead less than dynamic

Load balancing with OpenMP Scheduling

Scheduling impacts overhead and cache reuse

- Static has least overhead, dynamic most
  - larger chunksize can reduce overhead
- Dynamic may spoil cache reuse
  - After first iteration is performed by a thread the data may already be in the local cache for next iteration
  - Other cores may not have data in cache
- If a different thread on a different core is assigned the next iteration it may not have the data in it's local cache
- Larger chunk size can mitigate this effect
  - recall default chunksize is 1: Use with caution!

Load balancing with OpenMP `nowait`

`nowait` clause

- parallel constructs have implicit barrier at end where all threads must wait for other threads to complete
- `nowait` allows completed threads to advance and do more work
- usually need to use dynamic to benefit

```
#pragma omp parallel
{
#pragma omp for nowait
for (j=0;j<N;j++)
    A[j]+=f(x[j]);

#pragma omp for
for (j=0;j<N;j++)
    B[j]+=g(x[j]);
}
```

Cache Coherence, Memory Consistency, & Performance in Manycore

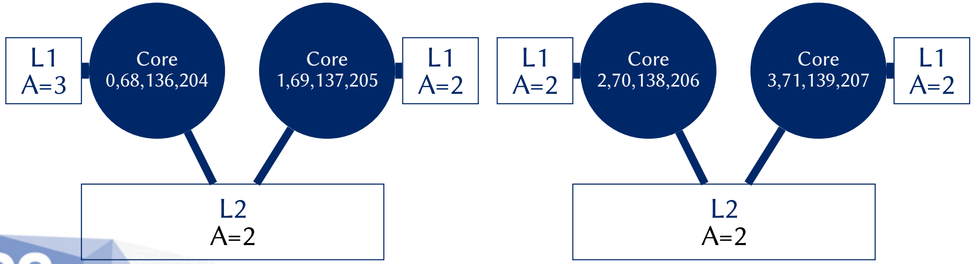
Shared data is a major performance limiter to parallelism

- If algorithm looks like it should scale but does not often due to this
- Data is kept consistent between cores' local caches by cache coherence and memory consistency
  - Recall KNL core has private L1 cache and shares L2 cache with one other core
  - Recall the 4 hardware threads on each core share an L1
- Consistency requires cycles and bandwidth
- Consistency can force serialization in otherwise parallel code
  - Can be explicit in program (e.g. `atomic`)
  - Can be transparent in program through *cache coherence protocol*

Parallel Contention: True Sharing

Major limitation to scalability!

- If core0 thread changes A's value, core1 thread must wait to change A
- KNL hyperthreads, e.g. 0,68,132,204, share L1 so not bad
- cores on local tile share L2 so not so bad
- cores on remote must request A over mesh



True Sharing Mitigation

Performance Improvement Factors

- Frequent writes to shared variables main bottleneck
- Frequent reads are OK
- `#pragma omp atomic` should be avoided (serializes code)
- Make one copy of value per thread if possible (even if redundant): `private` clause
- OpenMP has efficient `reduction` directive for many operators: +,*,-,| etc.

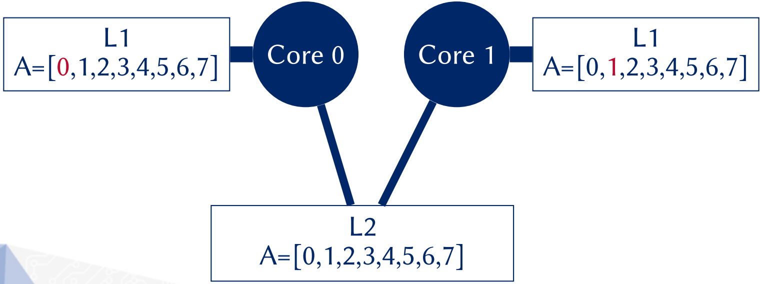
```
#pragma omp parallel for
for(i=0;i<N;i++)
    #omp atomic
    a+=i;

int tmp, result;
#pragma omp parallel private(tmp) shared(result)
{
#pragma omp for
for(i=0;i<N;i++) tmp+=i;
#pragma omp critical
    result+=tmp
}
```

```
#pragma omp parallel for reduction(a,+)
for (i=0;i<N;i++)
    a+=i;
```

Parallel Contention: False Sharing

Can happen easily but sometimes hard to detect

- Cache lines hold multiple elements (64B or 8 doubles on KNL)
- Different data on same cache line may be modified by multiple threads
- Entire cache lines are invalidated by modification, not individual data elements
- Shared cache line is forwarded or written to L2/memory constantly to maintain coherency



False Sharing Mitigation

Strategies

- Allocate data used by each thread contiguously (definitely do not interleave)
- Watch out for sequentially declared variables (may be land on same cache line)
- Chunksize - make multiple of cache line
- Spreading, add a dimension to array so each thread has own cache line
  - Reduces cache reuse → increases memory bandwidth but reduces cache coherency traffic

```
//contiguous b.x and b.y probably on same cache line
struct b {
    int x;
    int y;
};

//a access is interleaved: bad chunksize
double a[N];
#pragma omp parallel for schedule(static,1)
    for (i=0;i<N;i++)
        a[i]+=i
```

```
//each thread has a separate cache line: good chunksize
double a[N];
#pragma omp parallel for schedule(static,8)
    for (i=0;i<N;i++)
        a[i]+=i

// each thread has separate cache line: spreading
double b[N][8];
#pragma omp parallel for
    for (i=0;i<N;i++)
        b[i][0]+=i
```

Detecting Data Contention

Can be difficult: profilers are a good way to go

- If algorithm will not scale
- Looks for heavy cache traffic with using, for example `perf stat -d ./executable`
- `perf` is a light-weight tool built into newer (2.6.32+) Linux distros
- Grants access to hardware performance monitoring counters (PMCs)
- Try Intel's Vtune - also uses PMCs

perf stat -d ./executable

Example on KNL

```
Performance counter stats for 'numactl -C 0,2 ./sharing':

      11437.839149      task-clock (msec)         #    1.822 CPUs utilized
               185      context-switches          #    0.016 K/sec
                 1      cpu-migrations            #    0.000 K/sec
               408      page-faults               #    0.036 K/sec
    17,007,721,754      cycles                    #    1.487 GHz                      (33.34%)
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
     2,027,617,926      instructions              #    0.12  insns per cycle          (50.00%)
       333,745,966      branches                  #   29.179 M/sec                    (49.99%)
           590,178      branch-misses             #    0.18% of all branches          (50.01%)
   <not supported>      L1-dcache-loads
        21,916,918      L1-dcache-load-misses     #    1.916 M/sec                    (33.35%)
        22,675,535      LLC-loads                 #    1.983 M/sec                    (33.33%)
   <not supported>      LLC-load-misses

       6.278683864 seconds time elapsed
```

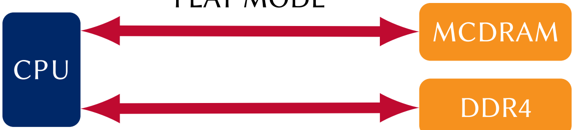
High BW Memory and Parallel Performance

- For BW bound MCDRAM is better
  - More threads = higher BW requirements
- For latency bound DDR can be better
- Cache mode is easy to use but Flat is more performant
  - Cache mode has extra memory latency due to caching mechanisms
  - Flat mode DDR best for latency sensitive
  - Flat mode MCDRAM best for BW sensitive
- Crossover between modes can occur as more threads are added and BW saturates

	DDR	MCDRAM
Capacity	100GB	16 GB
Bandwidth	90GB/s	450GB/s
Latency	125ns	150ns

CACHE MODE

FLAT MODE



Aside: OpenMP + Vectorization

With OpenMP 4.0: #pragma omp simd

- New portable directive
- Tells compiler to vectorize a loop
- more portable than intrinsics (vendor agnostic)
- can specify even a function as simd safe
- Warning: Inhibits compiler safety analysis!

```
#pragma omp parallel for simd
for(i=0;i<N;i++)
    a[i] += c[i]

#pragma omp declare simd
double vectorized_function(double *a, double *b){...}

#pragma omp simd
for(i=0;i<N;i++) c[i] += vectorized_function(a[i], b[i]);
```

Aside: Super-linear Speedup/Slowdown

Speed-up exceeds number of cores

- Usually happens when problem is decomposed small enough to fit in each core's cache
- Serial problem size did not fit in cache - streamed from memory and back, possibly many times
- Basically improved cache line reuse
- Be careful analyzing speed-up with datasets that are too small!

Slowdown for larger problem sizes even with more threads

- Little cache reuse
- Memory accesses are dominating
- Hide latency

License