Experiences with OpenMP in tmLQCD

A. Deuzeman, B. Kostrzewa, C. Urbach
for the ETM collaboration

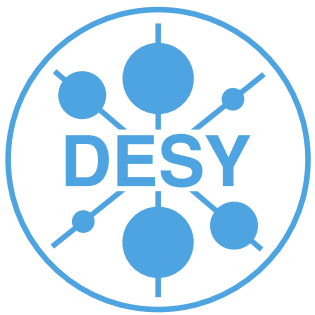
Institute of Physics
Humboldt-Universität zu Berlin

DESY, Zeuthen Site

Supported in full by FNR AFR PhD grant 2773315

Lattice 2013, Mainz
2nd August 2013

Fonds National de la
Recherche Luxembourg



Overview

Experiences from adding OpenMP to tmLQCD

Introduction

OpenMP

Shared memory parallel programming models

- Data shared between execution units (e.g. threads)
- No explicit communication (usually)
  - $\Rightarrow$ synchronization built into programming model
  - $\Rightarrow$ lower memory requirements (usually)
  - **!** need for "locking" when data is modified concurrently

OpenMP

- Simple syntax through pragmas with "directives":
  `#pragma omp parallel`
  - can be specialized with parameters, even runtime conditionals
- Most common scenarios are addressed
- Strong focus on loops
- No provision for complicated models before OpenMP 3.0
  - $\Rightarrow$ e.g.: no possiblity (within syntax) to launch independent "I/O thread"

Introduction

Basic Example

```
su3 function(args) {
  su3 accum;
  su3 U0, U1, U2;
  #pragma omp parallel for private(U0,U1,U2) shared(args,accum)
  for(int x = 0; x < VOLUME; ++x) {
    for(int mu = 0; mu < 4; ++mu) {
      U0 = get_staples(x,mu);
      [...]
    }
  }
  return accum;
}
```

+ **explicit mention of private/shared**
− **nightmare to maintain → updating private/shared prone to mistakes**

Introduction

Improved Basic Example

- **Use scoping rules to automate private / shared:**

```
su3 function(args) {
  su3 accum;
#pragma omp parallel
{
  su3 U0, U1, U2;
  #pragma omp for
  for(int x = 0; x < VOLUME; ++x) {
    for(int mu = 0; mu < 4; ++mu) {
      U0 = get_staples(x,mu);
      [...]
    }
  }
} /* OpenMP parallel closing brace */
  return accum;
}
```

+ **private/shared automatic, less overhead for multiple for loops**
− **private/shared less explicit**

Subtleties

working set size, performance, execution order

Setting up threads and assigning work has overhead

- maximize work, reduce relative size of overhead

Debugging can be challenging

- execution order not fixed (e.g.: summations) $\rightarrow$ difficult to differentiate bug and rounding
- some bugs may only show 'in production' and with very high statistics

  $\Rightarrow$ add debugging code with explicit ordering

  ! even then, errors might only show in high statistics

Subtleties

Amdahl & co.

Amdahl's law

- many threads $\rightarrow$ 2% serial function can easily turn into 25%
  - $\Rightarrow$ need to add OpenMP almost everywhere

Barriers can have substantial overhead

- Slow: computationally simple loops
- Slow: unbalanced thread workload
  - $\Rightarrow$ Use tools to find problematic areas
    - $\star$ Example: scalar product
      - $\star$ simple function $\rightarrow$ large barrier overhead
      - $\star$ scheduling: 'static' $\rightarrow$ 'guided' leads to 50% reduction in loop barrier overhead
      - $\star$ however, total time spent reduced only by 10%
  - $\Rightarrow$ Combine operations to increase workload

Subtleties

Data concurrency and race conditions

- Concurrent data access requires locking
  - ! `#pragma omp critical` is very slow

- Use `#pragma omp atomic`

```
[...]
#pragma omp atomic
derivative.d4 += [...]
[...]
```

  - ! Operation must compile into single instruction
  - ! Safety guaranteed only if multiple threads read, one thread writes
  - ! Conflicts may be unnoticeable in test programs
  - ! Conflict probability depends on total thread number

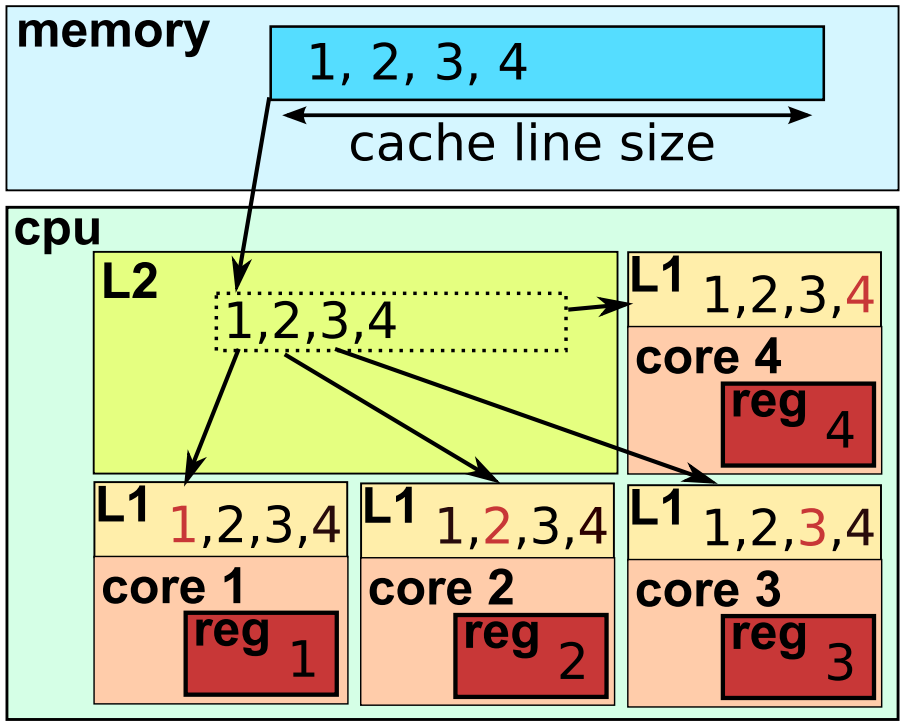
- In macros, use this syntax:

```
_Pragma("omp atomic")
```

Subtleties

False sharing

- threads update independent data, but on same cache line



- threads will invalidate each other's cache lines

! can slow multi-threaded program to less than sequential speed

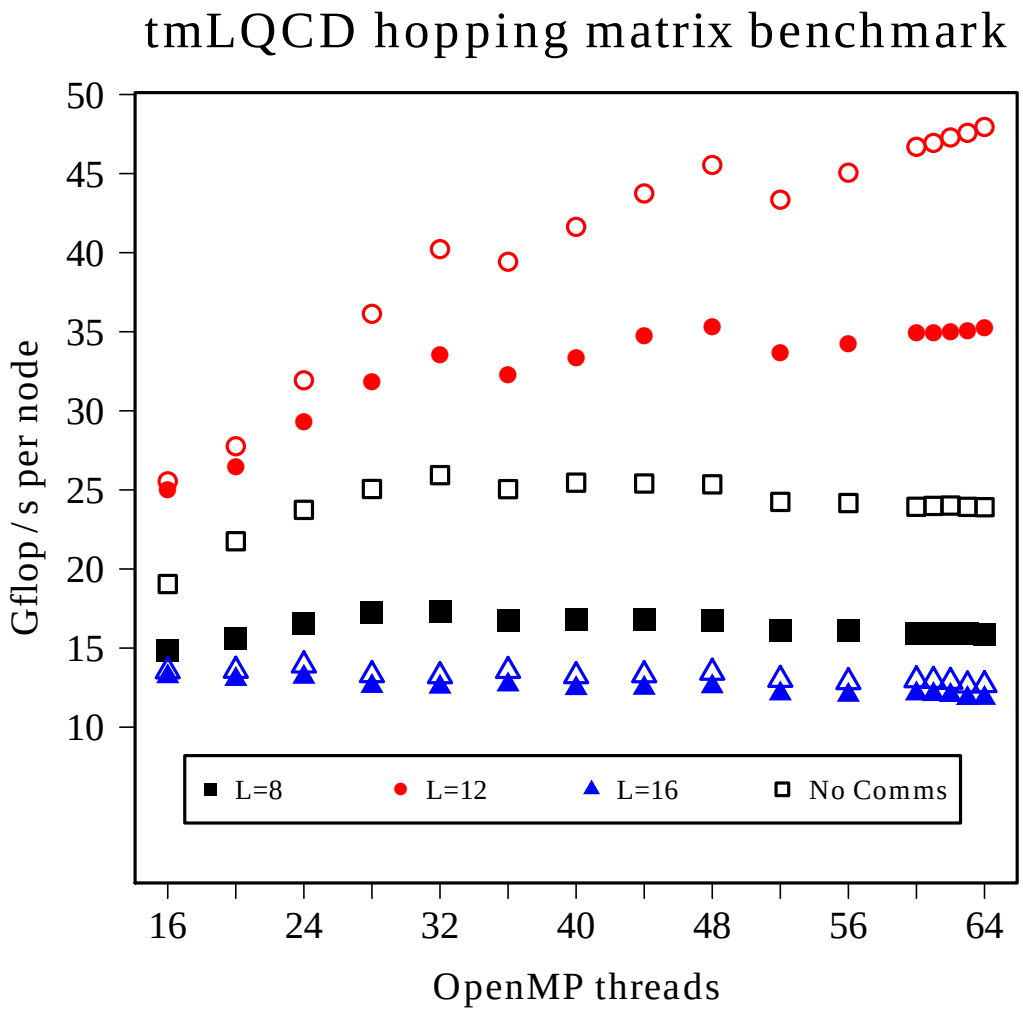
$\Rightarrow$ Add padding to ensure separate cache-lines, but total no. of cache lines limited!

Benefits and Overheads

Scaling

OpenMP scaling

- 1 process per node
- Variable number of hardware threads
- Overlapping MPI Communication (no SPI!)



- Scaling seems quite linear
- When local volume too small $\rightarrow$ use fewer threads?

Benefits and Overheads

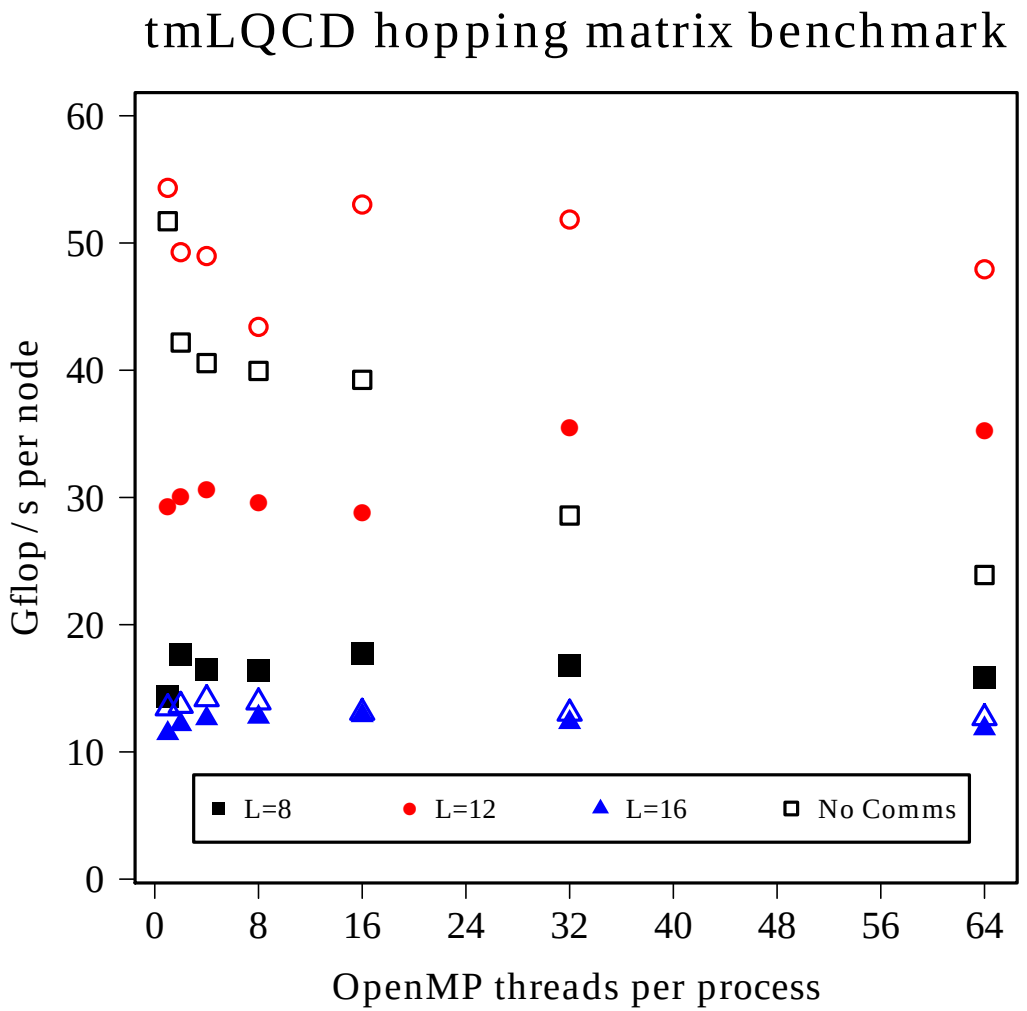
What's the point?

Full oversubscription

- Using maximum number of hardware threads

$\Rightarrow$ $N_{\text{threads}} \times N_{\text{procs}} = 64$

- Overlapping MPI Communication (no SPI!)

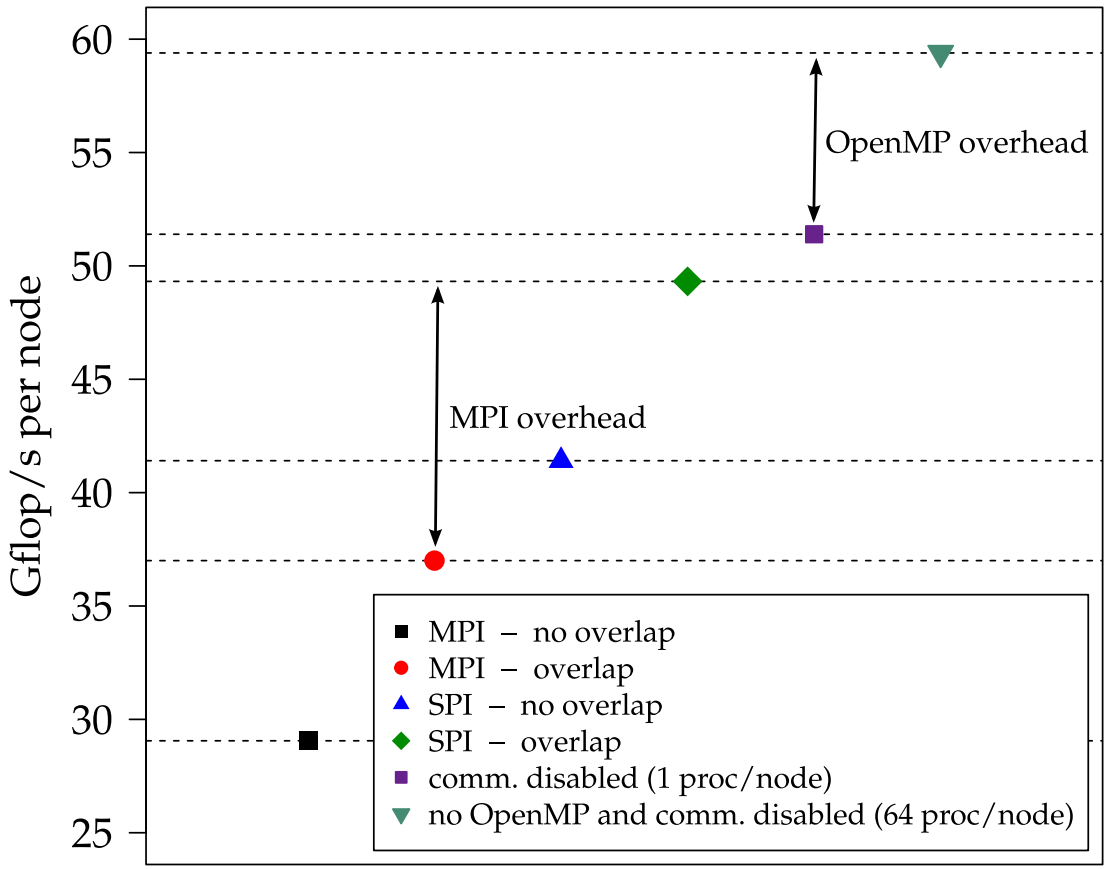


- OpenMP and MPI overheads are comparable on BG/Q
  - Hybrid MPI/OpenMP codes scale to more cores

Benefits and Overheads

Overlapping communication and computation!

- Standard non-blocking `MPI_Isend/recv` usually communicate in `MPI_WaitAll` $\Rightarrow$ effectively blocking
- One-sided MPI communication difficult and a lot of work
  - $\Rightarrow$ Do `MPI_Isend/recv` and `MPI_Waitall` in the same thread!



Improvements

Overhead reduction

Coarsen parallelism - reduce 'parallel' overhead

- employ 'orphaned' directives [a]

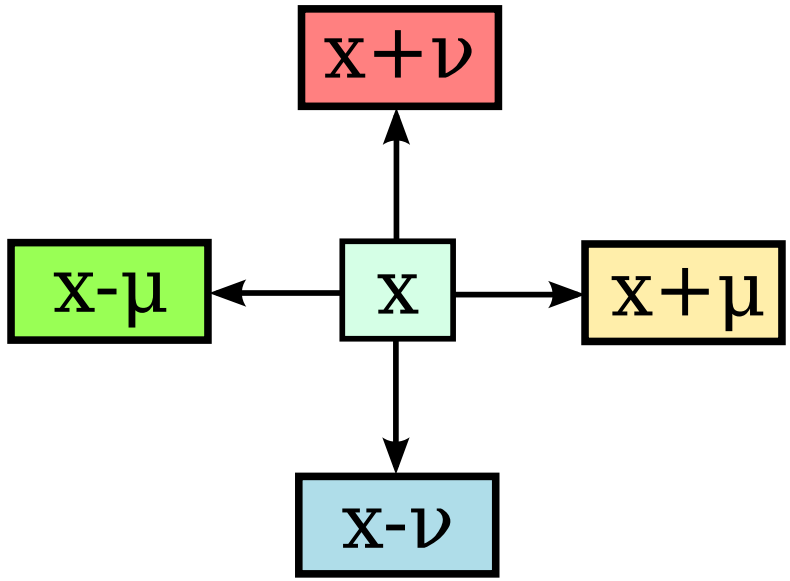
```
void complex_function(output,input) {
  #pragma omp parallel
  {
  operator1(output,input);
  operator2(output,input);
  [...]
  } /* OpenMP parallel section closing brace */
}

void operator1(output,input) {
  #pragma omp for
  for(int x = 0; x < VOLUME; ++x) { [...] }
}
```

[a] thanks to A. Deuzeman for pointing this out

Improvements

Dedicated memory for push algorithms - no locking, no concurrency issues



- \+ No locking overhead or concurrency issues
- \+ 'Obvious' for hopping matrix (half-spinor)
- \+ Keep efficient flop/byte ratio (rather than converting to pull-style)
- – Extra loop to collect the results
- – Higher memory requirements
- – Dedicated function versions and memory layout when using threads

Summary and Concluding Remarks

- Implementing good multi-threaded code is difficult
  - Benchmark and use performance tools
  - Balanced workloads lead to highest performance
  - Exploit scoping rules for maintainability
  - Eliminate false-sharing
  - Fine-tune scheduling
  - Coarse-grained parallelism
  - ...

\+ Pay-offs on CPUs with many cores and efficient threading (BG/Q)

\+ Overlapping communication and computation using usual MPI_Isend/recv

\+ For BG/Q, threading allows very efficient communication with SPI

\– Intel currently lagging behind, overheads LARGE

? Situation on Cray currently unknown

Thank you for your attention!

tmLQCD is an open-source project:
http://github.com/etmc/tmLQCD