Scalability on Manycore Machines

Claude TADONKI
MINES ParisTech – PSL Research University
Centre de Recherche Informatique
claude.tadonki@mines-paristech.fr

Sequential
84 seconds

Expected
84/84 = 1 second

Got
2 seconds

Got
25 seconds
Conceptual key factors related to scalability

Claude TADONKI

Scalability on Manycore Machines

Séminaire MATHEMATIQUES ET SYSTEMES, Mines ParisTech
May 11, 2017, Paris (France)

Code to be parallelized

Amdahl Law

Sequential Part

Parallel Programming model

Distributed memory

Shared memory

Operating System
  - Threads creation & scheduling
  - Synchronization

Hardware Mechanism
  - Resources sharing
  - Memory accesses

Tasks Scheduling
  - Load imbalance

Loss of parallel efficiency !!!!

- Processes initialization & mapping
- Data communication
- Synchronization
- Load imbalance
Speedup $\sigma(p) = \frac{T_s}{T_p}$

Efficiency (parallel) $e = \frac{\sigma(p)}{p}$

- Always keep in mind that these metrics only refer to “how go is our parallelization”.
- They normally quantify the “noisy part” of our parallelization.
- A good speedup might just come from an *inefficient sequential code*, so do *not be so happy*!
- Optimizing the reference code makes it harder to get nice speedups.
- We should also parallelize the “noisy part” so as to share its cost among many CPUs.
Amdahl’s Law illustration

<table>
<thead>
<tr>
<th>p</th>
<th>par = 95%</th>
<th>par = 90%</th>
<th>par = 75%</th>
<th>par = 50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
<td>100.00</td>
</tr>
<tr>
<td>2</td>
<td>52.50</td>
<td>55.00</td>
<td>62.50</td>
<td>75.00</td>
</tr>
<tr>
<td>4</td>
<td>28.75</td>
<td>32.50</td>
<td>43.75</td>
<td>62.50</td>
</tr>
<tr>
<td>8</td>
<td>16.88</td>
<td>21.25</td>
<td>34.38</td>
<td>56.25</td>
</tr>
<tr>
<td>16</td>
<td>10.94</td>
<td>15.62</td>
<td>29.69</td>
<td>53.12</td>
</tr>
<tr>
<td>32</td>
<td>7.97</td>
<td>12.81</td>
<td>27.34</td>
<td>51.56</td>
</tr>
<tr>
<td>64</td>
<td>6.48</td>
<td>11.41</td>
<td>26.17</td>
<td>50.78</td>
</tr>
<tr>
<td>128</td>
<td>5.74</td>
<td>10.70</td>
<td>25.59</td>
<td>50.39</td>
</tr>
<tr>
<td>256</td>
<td>5.37</td>
<td>10.35</td>
<td>25.29</td>
<td>50.20</td>
</tr>
<tr>
<td>512</td>
<td>5.19</td>
<td>10.18</td>
<td>25.15</td>
<td>50.10</td>
</tr>
</tbody>
</table>

Simulated parallel timings

Scalability on Manycore Machines
INTEL BROADWELL Intel® Xeon® Processor E5-2699 v4
Released in April 2016

- 22x2 = 44 cores
- 2.2 Ghz/core
- 3.6 GHz Boost
- Hyperthreading
- 256-bit vectors
- 256 Gb RAM
- 76.8 Gb/s
- 500 Gb disk
- 1.54 Tflops SP
- 0.78 Tflops DP
- Tflops is 1000 000 000 000 (1 billion) floating point operations per seconds
Illustrative performances with an optimized LQCD code

**LQCD performance on a 44 cores processor**

\[ D\psi(x) = A\psi(x) - \frac{1}{2} \sum_{\mu=0}^{4} \left\{ [(I_4 - \gamma_\mu) \otimes U_{x,\mu}]\psi(x + \mu) + [(I_4 + \gamma_\mu) \otimes U_{x-\mu,\mu}^\dagger]\psi(x - \mu) \right\}. \]

<table>
<thead>
<tr>
<th>#cores</th>
<th>#threads</th>
<th>t(s)</th>
<th>GFlops</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.02552</td>
<td>9.98</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0.01301</td>
<td>19.59</td>
<td>1.96</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.00679</td>
<td>37.50</td>
<td>3.76</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>0.00475</td>
<td>53.60</td>
<td>5.37</td>
</tr>
</tbody>
</table>

Optimal absolute performance on a single core and good scalability !!!

| (2 nodes) 16 | 32 | 0.00476 | 53.53 | 5.36 |
| (4 nodes) 32 | 64 | 0.00507 | 50.25 | 5.03 |

Something happened !!!

Let’s now explore and understand it.
What is the main concern?

- Speedup is just one component of the global efficiency
- We need to exploit all levels of parallelism in order to get the maximum SC performance
  - Message passing between nodes (MPI, …) [1]
  - Shared memory between cores (Pthreads, OpenMP, …) [2]
  - Vector computing inside a core (SSE, AVX, …) [3]
  - Accelerated computing beside a node (CUDA, OpenCL, …) [4]

Because of cost from explicit interprocessor communication, a scalable SMP implementation on a (manycore) compute node is a rewarding effort anyway.
Main factors against scalability on a shared memory configuration

Claude TADONKI

- Threads creation and scheduling
- Load imbalance
- Explicit mutual exclusion
- Synchronization
- Overheads of memory mechanisms
  - Misalignment (when splitting arrays)
  - False sharing
  - Bus contention
  - NUMA effects

Let’s now examine each of these aspects.
Thread creation + time-to-execution yield an overhead (usually marginal)

- Creating an pool of (always alive) threads that operate upon request is one solution

Dynamic threads migration could break some good scheduling strategies

Threads allocation without any affinity could result in an inefficient scheduling

The system might consider only part of available CPU cores

Threads scheduling regardless of conceptual priorities could be inefficient
Load imbalance or unequal execution times

- Tasks are usually distributed from static-based hypotheses
- Effective execution time is not always proportional to static complexity
- Accesses to shared resources and variables will incur unequal delays
- The execution time of a task might depend on the values of the inputs or parameters
  - Influence on the execution path following the control flow
  - Influence on the behavior because of numerical reasons
  - Constraints overhead from particular data location
  - Specific nature of data from particular instances (sparse, sorted, combinatorial complexity, ...)
- We thus need to seriously consider the choice between static and dynamic allocations
Static block allocation

- This is the most common allocation
- Each thread is assigned a predetermined block
- Assignment can be from input or output standpoint
- The need for synchronization is unlikely
- Equal chunks do no imply equal loads

Dynamic allocation with a pool of tasks

- Increasingly considered
- Thread continuously pop up tasks from the pool
- Usually organized from output standpoint
- More balanced completion times are expected (effective load balance)
- Synchronization is needed to manage the pool (some overhead is expected)

The choice depends on the nature of the computation and the influence of data accesses

Scalability on Manycore Machines

Séminaire MATHEMATIQUES ET SYSTEMES, Mines ParisTech
May 11, 2017, Paris (France)
Explicit mutual exclusion

Several threads are asking for the critical resource and typically get locked

Only one thread is selected to get the critical resource and the others remain locked

Critical resource is thus given to the requesting threads on a purely sequential basis

- Applies on critical resources sharing
- Applies on objects that cannot/should be accessed concurrently (file, single license lib, …)
- Used to manage concurrent write accesses to a common variable

- A non selected thread can choose to postpone its action and avoid being locked
- Since this yields a sequential phase, it should be used skilfully (only among the threads that share the same critical resource – strictly restricted to the relevant section of the program)
Since memory is (seamlessly) shared by all the CPU cores in a multicore processor, the overhead incurred by all relevant mechanisms should be seriously considered.
In case of a direct block distribution, some threads might receive unaligned blocks.

Threads to whom unaligned blocks are assigned will experience a slowdown.

The impact of misalignment is particularly severe with vector computing.

Always keep this in mind when choosing the number of threads and splitting arrays.
The organization of the memory hierarchy is also important for memory efficiency

Case (a):
Assigning two threads which share a lot of input data to C1 and C3 is inefficient

Case (b):
In-place computation will incur a noticeable overhead due to coherency management

Frequent thread migrations can also yield loss of cache benefit

We should care about memory organization and cache protocol
This the systematic invalidation of a duplicated cache line on every write access

The conceptual impact of this mechanism depends on the cache protocol

The magnitude of its effect depends on the level of cache line duplications

A particular attention should be paid with in place computation
The paths from L1 caches to the main memory fuse at some point (memory bus).

As the number of threads is increasing, the contention is likely to get worse.

Techniques for cache optimization can help as they reduce accesses to main memory.

Redundant computation or on-the-fly reconstruction of data are worth considering.

Figure 1: The latency to memory increases as you move up the hierarchy.
NUMA = Non Uniform Memory Access

Shared Memory Architecture – NUMA

- The whole memory is physically partitioned but is still shared between all CPU cores
- This partitioning is seamless to ordinary programs as there is a unique addressing
- A typical configuration looks like this

Scalability on Manycore Machines

Séminaire MATHEMATIQUES ET SYSTEMES, Mines ParisTech
May 11, 2017, Paris (France)
NUMA Nodes are linked by QPI links

The distances matrix between NUM nodes is displayed by issuing `numactl --hardware` command

<table>
<thead>
<tr>
<th>node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>11</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>10</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>21</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>21</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

These distances give an idea on how nodes are connected

“Local accesses” are of course faster than “remote accesses”

Links between NUMA nodes are potentially subject to heavy contention

It is important to know the topology of the processor (memory and CPU cores)

NUMA-unaware programs are likely to yield a noticeably poor scalability

Memory allocation and thread binding to specific nodes are possible within programs
NUMA considerations can be handled within programs through libraries like `libnuma`.

The library allows to:

- allocate memory on a specific node
- ask to interleave an array on all NUMA nodes
- check on which node a given memory space is allocated
- identify on which NUMA node a given core (logical id) belongs to

Such libraries should be used with flexibility in order to avoid portability issues.

An efficient explicit management of NUMA considerations can improve scalability.
$D \psi(x) = A \psi(x) - \frac{1}{2} \sum_{\mu=0}^{4} \left\{ (I_4 - \gamma_{\mu}) \otimes U_{x,\mu} \psi(x + \hat{\mu}) + (I_4 + \gamma_{\mu}) \otimes U_{x-\hat{\mu},\mu}^\dagger \psi(x - \hat{\mu}) \right\}$

<table>
<thead>
<tr>
<th>#cores</th>
<th>#threads</th>
<th>t(s)</th>
<th>GFlops</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.02552</td>
<td>9.98</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0.01301</td>
<td>19.59</td>
<td>1.96</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.00679</td>
<td>37.50</td>
<td>3.76</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>0.00475</td>
<td>53.60</td>
<td>5.37</td>
</tr>
<tr>
<td>(2 nodes) 16</td>
<td>32</td>
<td>0.00476</td>
<td>53.53</td>
<td>5.36</td>
</tr>
<tr>
<td>(4 nodes) 32</td>
<td>64</td>
<td>0.00507</td>
<td>50.25</td>
<td>5.03</td>
</tr>
</tbody>
</table>

**NUMA node 0**
- Hosts and Computes P1
- Output

**NUMA node 1**
- Hosts and Computes P3
- Output

<table>
<thead>
<tr>
<th>#cores</th>
<th>#threads</th>
<th>t(s)</th>
<th>GFlops</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.03025</td>
<td>8.42</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0.01547</td>
<td>16.47</td>
<td>1.95</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.00825</td>
<td>30.87</td>
<td>3.66</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>0.00502</td>
<td>50.72</td>
<td>6.02</td>
</tr>
<tr>
<td>(2 nodes) 16</td>
<td>32</td>
<td>0.00305</td>
<td>83.65</td>
<td>9.33</td>
</tr>
<tr>
<td>(4 nodes) 32</td>
<td>64</td>
<td>0.00209</td>
<td>121.74</td>
<td>15.43</td>
</tr>
</tbody>
</table>

**NUMA node 2**
- Hosts and Computes P0
- Output

**NUMA node 3**
- Hosts and Computes P2
- Output

+1: dependencies $i + 1$ (modulo 4)
-1: dependencies $i - 1$ (modulo 4)
Recommendations

- Identify the main performance related characteristics of the processor
- Skilfully consider threads related features at programming level
- Design a NUMA-aware memory allocation and management strategy
- Consider preventing threads migration through thread binding statements
- Do your best to reduce accesses to main memory
- Address load imbalance or unequal thread completion times
- Use good profiling tools and proceed with incremental improvements
Thanks for your attention