OpenMP in the Real World

Christian Terboven, Dieter an Mey
{terboven, anmey}@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University, Germany

OpenMP Tutorial
June 3rd, TU Dresden, Germany
Nested Parallelization
- FIRE: Pattern Recognition
- NestedCP: Computation of Critical Points
- Dynamic Thread Balancing in FLOWER

OpenMP and C++
- DROPS: Navier-Stokes Solver

CMP / CMT Architectures

OpenMP on Windows

Conclusion
FIRE: Image Retrieval System

- FIRE = Flexible Image Retrieval Engine
  - Compare the performance of common features on different databases
  - Analysis of correlation of different features

*Thomas Deselaers and Daniel Keysers, RWTH I6: Chair for Human Language Technology and Pattern Recognition*
FIRE: Image Retrieval System

\[ D(Q, X) := \sum_{m=1}^{M} w_m \cdot d_m(Q_m, X_m) \]

- Q: query image, X: set of database images
- Q_m, X_m: m-th feature of Q and X
- d_m: distance measure, w_m: weighting coefficient
- Return the k images with lowest distance to query image

- Well-suited for Shared-Memory parallelization: Data Mining in a large image database!

- Three levels to exploit parallelism:
  - Process multiple query images in parallel
  - Process database comparison for one query image in parallel
  - Computation of distances might be parallelized as well
How can Nested OpenMP improve the scalability?
- Scalability on outer level is limited because of output sync.
- OpenMP overhead increases with the number of threads.
- Dataset might better fit to the number of threads.

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Sun Fire E25K, 72 dual-core UltraSPARC-IV processors</th>
</tr>
</thead>
<tbody>
<tr>
<td># Threads</td>
<td>Only outer level</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>3.88</td>
</tr>
<tr>
<td>8</td>
<td>6.98</td>
</tr>
<tr>
<td>16</td>
<td>12.46</td>
</tr>
<tr>
<td>32</td>
<td>25.97</td>
</tr>
<tr>
<td>144</td>
<td></td>
</tr>
</tbody>
</table>
ScaleMP: virtual Shared-Memory Architecture

- **ScaleMP – Versatile SMP™ Architecture**
  - Aggregation of multiple x86 boards into one single system image
  - Cache coherent connection through InfiniBand
  - Modified IB stack and BIOS, virtualization, caching strategies
  - Aggregation of all I/O resources to the OS

- Technology by [www.ScaleMP.com](http://www.ScaleMP.com)

- **Maximum (end of 2008):**
  - 16 boards with 32 processors / 128 cores
  - up to 1 TB of main memory

- **Our system:**
  - 13 boards with 2x Intel Xeon E5450 (dual-core, 2.50 GHz)
 Nested Parallelization improves speedup by reducing the total overhead. Best effort: Speedup of 66.96 on 104 cores.

Explicit Thread Binding via KMP_AFFINITY env. var. (Intel Compiler)
NestedCP: Parallel Critical Point Extraction

- VR in Aachen: Analysis of large-scale flow simulations
  - Feature extraction from raw data
  - Interactive analysis in virtual environment (e.g. a cave)
- Critical Point: Point in the vector field with zero velocity

Andreas Gerndt, Virtual Reality Center, RWTH Aachen
NestedCP: Addressing Load Imbalance

- Algorithmic sketch of Critical Point extraction:
  - Loop over the time steps of unsteady datasets
  - Loop over the blocks of multi-block datasets
  - Loop checking the cells within the block for CP

- The time needed to check a cell may vary considerably!
NestedCP: Addressing Load Imbalance

- Solution in OpenMP is rather simple:

```c
#pragma omp parallel for num_threads(nTimeThreads) \ schedule(dynamic,1)
for (cutT = 1; curT <= maxT; ++curT)
{
    #pragma omp parallel for num_threads(nBlockThreads) \ schedule(dynamic,1)
    for (curB = 1; curB <= maxB; ++curB)
    {
        #pragma omp parallel for num_threads(nCellThreads) \ schedule(guided)
        for (curC = 1; curC <= maxC; ++curC)
        {
            findCriticalPoints(curT, curB, curC);
        }
    }
}
```
NestedCP: Addressing Load Imbalance

- The achievable speedup heavily depends on the dataset
- No load imbalance → almost perfect scalability

- Speedup on Sun Fire E25k, 72 dual-core UltraSPARC-IV processors, execution with 128 threads:
  - Without load balancing: 10.3 (static scheduling)
  - With load balancing: 33.9 (dynamic scheduling)
  - Sun extension to guided sched.: 55.3 (weight factor = 20)
FLOWer: A Navier-Stokes solver

- FLOWer: Navier-Stockes solver, German Aerospace Center
- PHOENIX: a small scale prototype of the space launch vehicle HOPPER (take off horizontally, place cargo in orbit, glide back to earth)
  - MPI + OpenMP / autoparallelization \(\rightarrow\) hybrid parallel program
  - DTB library used to automatically adjust number of threads to improve load balance of MPI
FLOWer: MPI parallelization is not balanced.

Process 5 has too much work to do.
FLOWer: Dynamic thread balancing

Sun Fire E25K
23 MPI procs start with 2 threads each

- Warm-up phase (1-12) artificially vary number or threads per process
- Steering phase (13-30) increase number of threads of busy procs
- Production phase (31-) freeze thread numbers + “nexttouch”

Nested Parallelization
OpenMP and C++
CMP / CMT Architectures
OpenMP on Windows
Conclusion
FLOWer: Dynamic thread balancing

Sun Fire 25K, ~ 65 Mflop/s per thread = 3% of peak performance
(high MPI communication overhead)

Sun Fire 25K, ~ 65 Mflop/s per thread = 3% of peak performance
(high MPI communication overhead)

„nexttouch“ mechanism applied
Agenda

- Nested Parallelization
  - FIRE: Pattern Recognition
  - NestedCP: Computation of Critical Points
  - Dynamic Thread Balancing in FLOWER

- OpenMP and C++
  - DROPS: Navier-Stokes Solver

- CMP / CMT Architectures

- OpenMP on Windows

- Conclusion
DROPS: A Navier-Stokes Solver in C++

- Numerical Simulation of two-phase flow
- Modeled by instationary and non-linear Navier-Stokes equation
- Level Set function is used to describe the interface between the two phases
- Written in C++: is object-oriented, uses nested templates, uses STL types, uses compile-time polymorphism, ...
- (Adaptive) Tetrahedral Grid Hierarchy
- Finite Element Method (FEM)

Example:
Silicon oil drop in D$_2$O (fluid/fluid)
OpenMP in the Real World

03.06.2009 – C. Terboven

Parallelization: Naive Approach w/ OpenMP

PCG(const MatrixCL& A, VectorCL& x, const VectorCL& b,
    const PreCon& M, int& max_iter,
    double& tol)
{
    VectorCL p(n), z(n), q(n), r(n);

    for (int i=1; i<=max_iter; ++i) {
        q = A * p;
        double alpha = rho / (p*q);
        x += alpha * p;
        r -= alpha * q;
    }
}

Option 1: Replace operator calls

y_Ax_par(&q.raw()[0],
    A.num_rows(), A.raw_val(),
    A.raw_row(), A.raw_col(),
    Addr( p.raw()));

Option 2: Place parallelization inside operator calls

Problems of both options:

- Code Changes
- Parallelization not applicable to complex expressions
- Parallelization may introduce additional overhead
Possible problem: Temporaries

```cpp
laperf::vector<double> x(dim), a(dim), b(dim);
x = (a * 2.0) + b;

ideal code for this vector-type operation:

```cpp
for(int i = 0; i < dim; ++i)
    x[i] = a[i] * 2.0 + b[i];
```

but in C++ it translates to:

```cpp
laperf::vector<double> _t1 = operator*(a, 2.0);
laperf::vector<double> _t2 = operator+(_t1, b);
x.operator=(_t2);
```

→ two temporary vector copies and unnecessary overhead
→ impossible to implement efficient parallelization
→ bad placement of temporaries on cc-NUMA architectures

Solution: Parallelization in Expression Template Mechanism.
Handling of cc-NUMA architectures

- All x86-based multi-socket system will be cc-NUMA!
  - Current Operating Systems apply first-touch placement
  - If cc-NUMA is ignored, the speedup will be zero, typically

- STL provides the concept of an allocator to encapsulate memory management
  - build on the same concept to optimize for cc-NUMA

- Two possible allocators:
  - May even be plugged into your own data types
  - `dist_allocator`: Distribute data according to OpenMP schedule type (same scheduling as in computation)
  - `chunked_allocator`: Distribute data according to explicitly precalculated scheme to improve load balancing
typedef SparseMatrixBaseCl<...> MatrixCL;
typedef VectorBaseCL<double> VectorCL;

PCG(const MatrixCL& A, VectorCL& x, const VectorCL& b,
    const PreCon& M, int& max_iter, double& tol)
{
    VectorCL q(n), p(n), r(n);
    [...]
    for (int i=1; i<=max_iter; ++i) {
        [...]
        q = A * p;
        double alpha = rho / (p*q);
        x += alpha * p;
        r -= alpha * q;
        [...]
    }
DROS: Parallel Iteration Loop of CG-type solver

```
typedef SparseMatrixBaseCl<..., MyAllocator> MatrixCL;
typedef VectorBaseCL<double, OpenMPInternalPar> VectorCL;

PCG(const MatrixCL& A, VectorCL& x, const VectorCL& b,
    const PreCon& M, int& max_iter, double& tol)
{
    VectorCL q(n), p(n), r(n);
    [...]  
    for (int i=1; i<=max_iter; ++i) {
        [...]  
        q = A * p;
        double alpha = rho / (p*q);
        x += alpha * p;
        r -= alpha * q;
        [...]  
    }
}
```

Expression Templates allow parallelization of whole line. Here: Complete Parallel Region inside operator calls.
Library-parallelized GMRES solver on cc-NUMA machine:

- cc-NUMA architecture provides good memory bandwidth
- Allocator concept successful, TBB Tasks have no affinity
OpenMP in the Real World

03.06.2009 – C. Terboven

OpenMP and C++

○ A clean C++ object-oriented coding style may be very helpful for OpenMP parallelization:
  – Encapsulation prohibits unintended data dependencies
  – Encapsulation may improve data locality (think ccNUMA)

○ But: OpenMP’s C++ support is limited:
  – Non-POD types not well supported (e.g. with reductions)
  – Pragmas are not language constructs: You might have to restructure / extend your code

○ Nevertheless: We like the combination … and it can be used successfully!
Agenda

- Nested Parallelization
  - FIRE: Pattern Recognition
  - NestedCP: Computation of Critical Points
  - Dynamic Thread Balancing in FLOWER

- OpenMP and C++
  - DROPS: Navier-Stokes Solver

- CMP / CMT Architectures

- OpenMP on Windows

- Conclusion
<table>
<thead>
<tr>
<th>Metric \ Server</th>
<th>SF V40z</th>
<th>FSC RX200 S4</th>
<th>Sun T5120</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Chip</td>
<td>AMD Opteron 875 2.2 GHz</td>
<td>Intel Xeon 5450 3.00 GHz</td>
<td>UltraSPARC T2 1.4 GHz</td>
</tr>
<tr>
<td># sockets</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td># cores</td>
<td>8 (dual-core)</td>
<td>8 (quad-core)</td>
<td>8 (octo-core)</td>
</tr>
<tr>
<td># threads</td>
<td>8</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>Accumulated L2 $</td>
<td>8 mb</td>
<td>16 mb</td>
<td>4 mb</td>
</tr>
<tr>
<td>L2 $ Strategy</td>
<td>Separate per core</td>
<td>Shared by 2 cores</td>
<td>Shared by 8 cores</td>
</tr>
<tr>
<td>Technology</td>
<td>90 nm</td>
<td>45 nm</td>
<td>65 nm</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>35.2 GFLOPS</td>
<td>96 GFLOPS</td>
<td>11.2 GFLOPS</td>
</tr>
<tr>
<td>Dimension</td>
<td>3 units</td>
<td>1 unit</td>
<td>1 unit</td>
</tr>
</tbody>
</table>

Note: Here we compare machines of different ages – which can be seen as unfair! For example newer Opteron-based machines provide similar settings in 1 unit...
Measuring Memory Bandwidth

- Do not look at the CPU performance only – the memory subsystem’s performance is crucial for your HPC application!

```c
long long *x, *xstart, *xend, mask;
for (x = xstart; x < xend; x++) *x ^= mask;
```

- Each loop iteration: One load + One store

- We ran this kernel with multiple threads working on private data at a time using OpenMP (large memory footprint >> L2)

- Explicit processor binding to control the thread placement
  - Linux: `taskset` command
  - Solaris: `SUN_MP_PROCBIND` environment variable
Selected results: 1 thread

2x Clovertown, 2.66 GHz
1 thread: 3.970 GB/s

4x Opteron 875, 2.2 GHz
1 thread: 3.998 GB/s

1x Niagara 2, 1.4 GHz
1 thread: 1.254 GB/s
Memory Bandwidth: Dual-Socket Quad-Core Clovertown

1 thread: 3.970 GB/s

2 threads: 3.998 GB/s

2 threads: 6.871 GB/s

2 threads: 4.661 GB/s

8 threads: 8.006 GB/s

Limited Scalability!
Memory Bandwidth: Quad-Socket Dual-Core Opteron

1 thread: 3.998 GB/s

2 threads: 4.674 GB/s
2 threads: 8.210 GB/s
2 threads: 4.335 GB/s

Good Scalability (if you do it right)!

ccNUMA!
Memory Bandwidth: 8-Core CMT Niagara-2

1 thread: 1.254 GB/s
2 threads: 2.405-2.455 GB/s
4 threads: 4.182-4.645 GB/s
8 threads: 7.367 GB/s

4 threads: 4.997 GB/s
8 threads: 9.470-9.828 GB/s
16 threads: 12.459 GB/s
32 threads: 11.395 GB/s

Good Scalability (but slow with one thread)!
What does that mean for my application?

- Ouch, these boxes all behave differently...

- Let's look at a Sparse Matrix Vector multiplication:

<table>
<thead>
<tr>
<th>Sparse MatVec</th>
<th>SF V40z (Opteron)</th>
<th>FSC RX200 S4 (Xeon)</th>
<th>SF ??? (Niagara2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>2.17</td>
<td>9.34</td>
<td>0.98</td>
</tr>
<tr>
<td>Large</td>
<td>1.47</td>
<td>0.91</td>
<td>1.76</td>
</tr>
</tbody>
</table>

- Which architecture is best-suited depends on:
  - Can your application profit from big caches?
  - Can your application profit from shared / separate caches?
  - Can your application profit from a high clock rate?
  - Is your application memory bound anyway?
  - ... and more factors ...
Agenda

- Nested Parallelization
  - FIRE: Pattern Recognition
  - NestedCP: Computation of Critical Points
  - Dynamic Thread Balancing in FLOWER

- OpenMP and C++
  - DROPS: Navier-Stokes Solver

- CMP / CMT Architectures

- OpenMP on Windows

- Conclusion
Case Study: KegelToleranzen (WZL)

- Contact analysis simulation of Bevel Gears
  - Written in Fortran, using Intel Fortran 10.1 compiler
  - Very cache-friendly → runs at high Mflop/s rates

Bevel Gear Pair

Differential Gear

Laboratory for Machine Tools and Production Engineering, RWTH Aachen
Case Study: KegelToleranzen (WZL)

- Comparing Linux and Windows Server 2008:

  ![Performance of KegelToleranzen](chart)

  - Linux 2.6: 4x Opteron dual-core, 2.2 GHz
  - Windows 2003: 4x Opteron dual-core, 2.2 GHz
  - Linux 2.6: 2x Harpertown quad-core, 3.0 GHz
  - Windows 2008: 2x Harpertown quad-core, 3.0 GHz

  Performance gain for the user: Speedup of 5.6 on one node. Even better from starting point (desktop: 220 MFlop/s).
Agenda

- Nested Parallelization
  - FIRE: Pattern Recognition
  - NestedCP: Computation of Critical Points
  - Dynamic Thread Balancing in FLOWER

- OpenMP and C++
  - DROPS: Navier-Stokes Solver

- CMP / CMT Architectures

- OpenMP on Windows

- Conclusion
Conclusion

- OpenMP is often a good alternative to multiple MPI processes per node, as parallelization might require less work!
  - Adding levels of parallelism can help to increase scalability!

- Tools for performance analysis and verification are critical ingredients of the program development environment.

- OpenMP is useful for multi-core architectures!
  - But: Current support for C++ is limited.
  - But: Support for architecture aspects is still missing.
The End

Thank you for your attention!