Productivity and Performance with Multi-Core Programming

Christian Terboven <terboven@rz.rwth-aachen.de>
Center for Computing and Communication
RWTH Aachen University

04.07.2012, CE Seminar, TU Darmstadt, Darmstadt, Germany
05.07.2012, Lecture, TU Darmstadt, Darmstadt, Germany

Rechen- und Kommunikationszentrum (RZ)
**Motivation**

- **Data Locality and Data Access Optimization are key to performance**
  - Consider some additional instructions to avoid (remote) memory access …
  - … and parallelize your code such that the parallelization does not hinder further development!

Rechen- und Kommunikationszentrum (RZ)
Agenda

- Case Study: Optimizing SMXV on a NUMA Architecture
- Strategies to deal with NUMA Architectures
- Exploiting Object-Oriented Abstractions
- Patterns for Task-Parallelism
- Tasking Behavior on NUMA Architectures
- Application Case Studies
- Summary and Future Work
Case Study: Optimizing SMXV on a NUMA Architecture
Review: Performance Aspects

- **Performance Measurements**
  - Runtime (real wall time, user cpu time, system time)
  - FLOPS: number of floating point operations (per sec)
  - Speedup: performance gain relative to one core/thread
  - Efficiency: Speedup relative to the theoretical maximum

- **Performance Impacts**
  - Load Imbalance
  - Data Locality on cc-NUMA architectures
  - Memory Bandwidth (consumption per thread)
  - (Cache Effects)
Experiment Setup and Hardware

- **Intel Xeon 5450 (2x4 cores) – 3,0 GHz; 12MB Cache**
  - L2 cache shared between 2 cores
  - flat memory architecture (FSB)

- **AMD Opteron 875 (4x2 cores) - 2,2 GHz; 8MB Cache**
  - L2 cache not shared
  - ccNUMA architecture (HT-links)

- **Matrices in CRS format**
  - „large“ means 75 MB >> caches

→ **SMXV is an important kernel in numerical codes (GMRES, …)**
Performance with Static Row Distribution (1)

- compact: threads packed tightly; scatter: threads distributed

![Graph showing performance with static row distribution](graph)

- The graph illustrates the performance (in mflops) of different processors: AMD Opteron, Intel Xeon, compact, and Intel Xeon, scatter, under static row distribution.

- The x-axis represents the number of threads, while the y-axis represents the performance in mflops.

- The graph shows that as the number of threads increases, the performance also increases, with Intel Xeon, scatter showing the highest performance and compact showing the lowest.
Performance Analysis of (1)

- Load Balancing: Static data distribution is not optimal!

- Data Locality on cc-NUMA architectures: Static data distribution is not optimal!

- Memory Bandwidth: Compact thread placement is not optimal on Intel Xeon processors!
- compact: threads packed tightly; scatter: threads distributed

![Graph showing performance with dynamic row distribution](image)

- Intel Xeon, scatter
- Intel Xeon, compact
- AMD Opteron
Performance Analysis of (2)

Why does the Xeon deliver better performance with the Dynamic row distribution, but the Operon gets worse?

→ Data Locality. The Opteron is a cc-NUMA architecture, the threads are distributed along the cores over all sockets, but the data is not!

→ Solution: Parallel initialization of the matrix to employ the first-touch mechanism of Operating Systems: Data is placed near to where it is first access from.
Performance with Pre-Calculated Row Distribution

static row distribution, sorted matrix

<table>
<thead>
<tr>
<th>threads</th>
<th>mflops</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>500,000</td>
</tr>
<tr>
<td>2</td>
<td>1,000,000</td>
</tr>
<tr>
<td>3</td>
<td>1,500,000</td>
</tr>
<tr>
<td>4</td>
<td>2,000,000</td>
</tr>
<tr>
<td>5</td>
<td>2,500,000</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

- AMD Opteron
- Intel Xeon
Final Performance Analysis

- By exploiting data locality the AMD Opteron is able to reach about 2000 mflops!
- The Intel Xeon performance stagnates at about 1000 mflops, since the memory bandwidth of the fronside bus is exhausted with using four threads already:

![Diagram showing data locality and cache utilization]

- If the matrix would be smaller and fit into the cache the result would look different…
All measurements exclude the „warm up“ phase.
Strategies to deal with NUMA Architectures
NUMA Strategies: Overview

- **First Touch:** Modern operating systems (i.e. Linux >= 2.4) decide for a physical location of a memory page during the first page fault, when the page is first „touched“, and put it close to the CPU causing the page fault.

- **Explicit Migration:** Selected regions of memory (pages) are moved from one NUMA node to another via explicit OS syscall.

- **Next Touch:** The binding of pages to NUMA nodes is removed and pages are put where the next „touch“ comes from. Well-supported in Solaris, expensive to implement in Linux.

- **Automatic Migration:** Not supported by current operating systems.

- **OpenMP V4 - Places + Binding Policies:** Powerful support for Thread Binding and Thread Placement to exploit first touch policy.
Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:

→ Intel MPI’s `cpuinfo` tool

→ module switch openmpi intelmpi

→ `cpuinfo`

→ Delivers information about the number of sockets (= packages) and the mapping of processor ids used by the operating system to cpu cores.

→ `hwlocs`’ `hwloc-1s` tool

→ `hwloc-1s`

→ Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to cpu cores and additional info on caches.
Step 2: Decide for Binding Strategy

- Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.
  - Putting threads far apart, i.e. on different sockets
    - May improve the aggregated memory bandwidth available to your application
    - May improve the combined cache size available to your application
    - May decrease performance of synchronization constructs
  - Putting threads close together, i.e. on two adjacent cores which possibly shared some caches
    - May improve performance of synchronization constructs
    - May decrease the available memory bandwidth and cache size

- If you are unsure, just try a few options and then select the best one.
Step 3: Implement Binding Strategy

**Intel C/C++/Fortran Compiler**

→ Use environment variable \texttt{KMP\_AFFINITY}

→ \texttt{KMP\_AFFINITY=scatter}: Put threads far apart

→ \texttt{KMP\_AFFINITY=compact}: Put threads close together

→ \texttt{KMP\_AFFINITY=<core\_list>}: Bind threads in the order in which they are started to the cores given in the list, one thread per core.

→ Add “,verbose” to print out binding information to stdout.

**GNU C/C++/Fortran Compiler**

→ use environment variable \texttt{GOMP\_CPU\_AFFINITY}

→ \texttt{GOMP\_CPU\_AFFINITY=<core\_list>}: Bind threads in the order in which they are started to the cores given in the list, one thread per core.
OpenMP V4: Places + Binding Policies (1/2)

- **Define OpenMP Places**
  - set of OpenMP threads running on one or more processors
  - can be defined by the user

- **Define a set of OpenMP Thread Affinity Policies**
  - SPREAD: spread OpenMP threads evenly among the places
  - CLOSE: pack OpenMP threads near master thread
  - MASTER: collocate OpenMP thread with master thread

- **Goals**
  - user has a way to specify where to execute OpenMP threads for
  - locality between OpenMP threads / less false sharing / memory bandwidth
Example‘s Objective:

→ separate cores for outer loop
→ near cores for inner loop

Outer Parallel Region uses SPREAD, inner uses COMPACT

→ spread creates partition
→ compact binds threads within respective partition

Example

→ initial

→ spread 4

→ compact 4
Exploiting Object-Oriented Abstractions
Object-Oriented and Parallel Programming

- Compute intense core of many PDE solvers consists of Krylov subspace methods. Variations exist among different programs and throughout the development process.

- Object-Oriented Programming is mainstream since the 90s, Parallel Programming is just about to enter mainstream.
  → Reasons for OO: Encapsulation and Modularity → Reusability

- Parallelization is often decoupled from ongoing development.
  → Use of OO techniques to introduce and optimize parallelization
  → Use of OO techniques to investigate parallelization approaches
  → Use of OO techniques to hide complex architecture details from application / algorithm developer
C++: Iteration Loop of CG-style solver

MatrixCL A(rows, cols, nonzeros);
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;
    [...]

- Code excerpt from C++ Navier-Stokes solver DROPS pretty much resembles notation found in math text books.
- Goals: Hide the parallelization as much as possible with as little overhead as possible to not hinder development.
Aspects of OpenMP

- **OpenMP**
  - Supports Fortran, C and C++
  - Explicit Parallelization via Parallel Regions:
    - `pragma + structured block`
  - Worksharing
  - Task-based parallelization

- **Each Worksharing construct has an implicit Barrier associated (can be skipped)**

- **Intel + Sun + Others: Implementation via Thread Pool → Threads are not terminated**
MatrixCL A(rows, cols, nonzeros);
VectorCL q(n), p(n), r(n);
[…]
for (int i = 1; i <= max_iter; ++i) {
   […]
    q = A * p;
    double alpha = rho / 
    x += alpha * p;
    r -= alpha * q;
   […]

Option 1: Replace operator calls by loops:
#pragma omp parallel for
for (int r = 0; r < numRows, r++)
{
    double sum = 0.0; size_t nz;
    size_t rb = Arow[r];
    size_t re = Arow[r + 1];
    for (nz = rb, nz < re, nz++)
        sum += Aval[nz] * x[Acol[nz]];
    y[r] = sum;
}

- Refactoring code into OpenMP loops breaks OO paradigm.
- Code changes can easily break the parallelization! (races)
C++: Parallel Matrix & Vector class

- Extend existing abstractions to introduce parallelism!
- DROPS: Numerical parts are implemented via Matrix (CRS) and Vector class, or descendents of those.

→ Use Adapter design pattern to replace those with parallel ones

- Problems:
  - Temporaries in complex expressions.
  - Overhead introduced by the Parallel Region in every operator call.
Problem: The compiler translates the user code

\[ x = (a \times 2.0) + b; \]

into the following code

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

Solution: Use Expression Templates to transform this into

```cpp
LB<OpAdd, LB<OpMul, vector, double>, vector>
    expr( LB<OpAdd, LB<OpMul, vector, double>, vector>(
        LB<OpMul, vector, double>(a, 2.0), b
    )
);

template<typename TExpr>
    vector::operator=( TExpr expr ) {
    #pragma omp parallel for
    for( size_t i = 0; i < dim; ++i )
        this[i] = expr[i];
}
```

which can be efficiently parallelized with OpenMP.
Handling of cc-NUMA architectures

- All x86-based multi-socket systems will be cc-NUMA!
  - Current Operating Systems apply first-touch placement

- Thread binding is a necessity on cc-NUMA system: Hardware is examined automatically at application startup (Singleton design pattern).

- Policy-based approach: Apply the Strategy design pattern to influence the internal workings of a target class.

- Provide simple-to-choose-from options for the user:
  - DistributedPolicy: Distribute data according to OpenMP schedule type (same scheduling as in computation, default)
  - ChunkedPolicy: Distribute data according to explicitly precalculated scheme to improve load balancing (special cases)
C++: Parallel Iteration Loop of CG-style solver

MatrixCL A(rows, cols, nonzeros);
VectorCL<OpenMPParallelization> 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 the parallelization of whole lines.
- Parallelization is completely invisible, algorithmic modifications do not break the parallelization!

Incremental approach: (i) Go Parallel, (ii) cc-NUMA, (iii) ...
Intel Compiler: hardly any difference in the programming language.

Abstractions are for free, if done the right way.

Two-socket Intel Nehalem (X5570) @ 2.93 GHz, Intel 11.0 compiler.
Example: OpenMP Loop without Pragma (1/4)

- Using C++0x features to parallelize the following loop and hide the OpenMP pragma:

```cpp
double dStart, dEnd;
for (int rep = 0; rep < iNumRepetitions; rep++)
{
    dStart = omp_get_wtime();

    for (int i = 0; i < iNumElements; i++)
    {
        vec[i] = compute(vec[i], iNumIterations);
    }
    dEnd = omp_get_wtime();
}
```
Example: OpenMP Loop without Pragma (2/4)

- Using C++0x features to parallelize a loop and hide the OpenMP pragma:

```c
double dStart, dEnd;
for (int rep = 0; rep < iNumRepetitions; rep++)
{
    dStart = omp_get_wtime();
#pragma omp parallel for shared(iNumElements, vec, iNumIterations)
    for (int i = 0; i < iNumElements; i++)
    {
        vec[i] = compute(vec[i], iNumIterations);
    }
    dEnd = omp_get_wtime();
}
```

- Regular OpenMP parallelization
Example: OpenMP Loop without Pragma (3/4)

- Using C++0x features to parallelize a loop and hide the OpenMP pragma:

```c++
double dStart, dEnd;
for (int rep = 0; rep < iNumRepetitions; rep++)
{
    dStart = omp_get_wtime();
    omp_pfor(0, iNumElements, [&](int i)
    /* for (int i = 0; i < iNumElements; i++) */
    {
        vec[i] = compute(vec[i], iNumIterations);
    });
    dEnd = omp_get_wtime();
}
```

- No OpenMP pragma. Loop body as well as the iteration space is passed as an argument to the `omp_pfor` lambda function.
Example: OpenMP Loop without Pragma (4/4)

- This is how the lambda function is implemented:

```cpp
template<typename F>
void omp_pfor(int start, int end, F x)
{
    #pragma omp parallel for
    for(int __i = start; __i < end; __i++)
    {
        x(__i);
    }
}
```

- Done.
Patterns for Task-Parallelism
How to create Tasks

- When using task-based parallelism …
  - … you may not care about affinity and just create tasks, or …
  - … you may wonder how the runtime executes your tasks on NUMA.
  - According to the OpenMP spec the runtime has a great degree of freedom in determining when and where to execute task

- Two performance-critical optimization goals
  - Load Balancing: Evenly distribute the work among the threads / cores
  - Data Locality: Minimize remote memory accesses on NUMA machines

- Two general patterns to create tasks
  - Single-Producer Multiple-Executors
  - Parallel-Producer Multiple-Executors
Single-Producer Multiple-Executors

- Implementation in OpenMP
  - Parallel Region + Single Construct
  - Hence the code region is executed by one thread only, which is responsible for creating the tasks

- Often requires only little changes to code and data structures
  - No data races in the producer code
  - Implicit barrier at the end of the Single Construct

- Expectations
  - Overhead of task creation increasing with the number of threads
  - Problematic on NUMA architectures
Parallel-Producer Multiple-Executors

- Implementation in OpenMP
  - Any OpenMP Worksharing construct + Implicit Barrier, or
  - Control of task creation via Thread id

- Careful examination of all data structures necessary

- Expectations
  - Task creation is „parallelized“ – overhead increases very slowly with the number of threads
  - Task stickiness: The probability that the task creator executes a task is higher than any other thread executing this task
Load Balancing vs. Data Affinity (1/3)

- Artifical benchmark executes 128,000 work packages
  - distributed among the NUMA nodes
  - each performs some work on an inner array
  - to simulate load imbalance the inner array size increases
Load Balancing vs. Data Affinity (2/3)

- Results on a 4-socket Bull system, Intel Nehalem-EX, 2.0 GHz
  
  → MIN, MAX and STDV work done per thread
  
  → Percentage of local iterations

<table>
<thead>
<tr>
<th>Tasking</th>
<th>MIN</th>
<th>MAX</th>
<th>STDV</th>
<th>local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>97.7 %</td>
<td>100.4 %</td>
<td>0.5</td>
<td>79 %</td>
</tr>
<tr>
<td>GNU</td>
<td>81.5 %</td>
<td>114.2 %</td>
<td>5.6</td>
<td>80 %</td>
</tr>
<tr>
<td>Oracle</td>
<td>76.0 %</td>
<td>161.5 %</td>
<td>17.7</td>
<td>60 %</td>
</tr>
<tr>
<td>PGI</td>
<td>83.4 %</td>
<td>106.6 %</td>
<td>5.0</td>
<td>82 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Worksharing</th>
<th>MIN</th>
<th>MAX</th>
<th>STDV</th>
<th>local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel (static)</td>
<td>6.1 %</td>
<td>193.9 %</td>
<td>56.0</td>
<td>100 %</td>
</tr>
<tr>
<td>Intel (dynamic)</td>
<td>83.1 %</td>
<td>110.0 %</td>
<td>5.2</td>
<td>3.1 %</td>
</tr>
</tbody>
</table>
Load Balancing vs. Data Affinity (3/3)

- Results on a 16-socket Bull BCS system, Intel Nehalem-EX, 2.0 GHz

  → MIN, MAX and STDV work done per thread

  → Percentage of local iterations

<table>
<thead>
<tr>
<th>Tasking</th>
<th>MIN</th>
<th>MAX</th>
<th>STDV</th>
<th>local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>84.4 %</td>
<td>121.3 %</td>
<td>8.5</td>
<td>80 %</td>
</tr>
<tr>
<td>GNU</td>
<td>66.9 %</td>
<td>271.0 %</td>
<td>41.3</td>
<td>69 %</td>
</tr>
<tr>
<td>Oracle</td>
<td>0.1 %</td>
<td>566.6 %</td>
<td>152.9</td>
<td>29 %</td>
</tr>
<tr>
<td>PGI</td>
<td>25.0 %</td>
<td>199.8 %</td>
<td>27.8</td>
<td>79 %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Worksharing</th>
<th>MIN</th>
<th>MAX</th>
<th>STDV</th>
<th>local</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel (static)</td>
<td>1.6 %</td>
<td>198.5 %</td>
<td>57.3</td>
<td>100 %</td>
</tr>
<tr>
<td>Intel (dynamic)</td>
<td>8.6 %</td>
<td>522.4 %</td>
<td>149.0</td>
<td>0.8 %</td>
</tr>
</tbody>
</table>
Overhead of Task Creation

Based on the EPCC OpenMP benchmark suite we created 2 tests:

- Single-Creator vs. Parallel-Creator

```c
#pragma omp parallel private(j)
#pragma omp single
for (j=0; j<innerreps* \ omp_get_num_threads(); j++)
#pragma omp task
delay(delaylength);
```

Results on the 16-sockets machine, in us:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>sin-pro</td>
<td>0.12</td>
<td>1.89</td>
<td>187.7</td>
<td>199.6</td>
<td>492.6</td>
<td>2440.6</td>
<td>2432.1</td>
<td>5656.2</td>
</tr>
<tr>
<td></td>
<td>par-pro</td>
<td>0.12</td>
<td>0.32</td>
<td>0.93</td>
<td>0.17</td>
<td>0.72</td>
<td>1.11</td>
<td>1.32</td>
<td>1.69</td>
</tr>
<tr>
<td>GNU</td>
<td>sin-pro</td>
<td>1.8</td>
<td>1.8</td>
<td>14.6</td>
<td>209.8</td>
<td>785.7</td>
<td>1361.5</td>
<td>11938.5</td>
<td>40555.7</td>
</tr>
<tr>
<td></td>
<td>par-pro</td>
<td>1.7</td>
<td>2.0</td>
<td>2.9</td>
<td>34.8</td>
<td>324.2</td>
<td>401.8</td>
<td>1870.3</td>
<td>9908.2</td>
</tr>
<tr>
<td>Oracle</td>
<td>sin-pro</td>
<td>0.09</td>
<td>3.52</td>
<td>37.0</td>
<td>176.3</td>
<td>643.7</td>
<td>1534.8</td>
<td>3571.8</td>
<td>7440.8</td>
</tr>
<tr>
<td></td>
<td>par-pro</td>
<td>0.09</td>
<td>1.89</td>
<td>6.0</td>
<td>18.4</td>
<td>88.0</td>
<td>235.5</td>
<td>505.1</td>
<td>1183.1</td>
</tr>
<tr>
<td>PGI</td>
<td>sin-pro</td>
<td>0.69</td>
<td>5.3</td>
<td>4.8</td>
<td>14.1</td>
<td>62.6</td>
<td>161.0</td>
<td>367.8</td>
<td>892.6</td>
</tr>
<tr>
<td></td>
<td>par-pro</td>
<td>0.02</td>
<td>2.7</td>
<td>2.4</td>
<td>3.5</td>
<td>6.9</td>
<td>23.6</td>
<td>51.4</td>
<td>357.4</td>
</tr>
</tbody>
</table>
Tasking Behavior on NUMA Architectures
STREAM with Tasks

- Arrays with a dimension of 256,435,456 double elements
  → 1.96 GB memory consumption per array, 5.87 GB for the TRIAD kernel

- We compared four versions
  → workshare: static-init for-loop. For t threads each array is divided into t parts which are evenly distributed among the NUMA nodes. Threads are scattered across the NUMA nodes in the very same way. (reference)
  → tasks: task-init parallel-producer. A Parallel-Producer pattern with tasks is used in both the data distribution and the STREAM operation.
  → Is the pattern reliable?
Results: STREAM TRIAD on 4-sockets

- Intel Compiler
- GNU Compiler
- Oracle Compiler
Results: STREAM TRIAD on 16-sockets

- Intel Compiler
- Oracle Compiler
SMXV with Tasks

- **Matrix with** $N = 2,017,169$ and $nnz = 283,073,458$
  - memory footprint of approx. 3.2 GB
  - static data distribution (via worksharing)

- **Work distribution critical for performance (load balance + locality)**
  - Optimal distribution is hard, if not impossible, to realize with for worksharing construct
  - One row per task can be expected to incur high overhead, thus

  $$chunk\_size(tasks) = \begin{cases} 
  \lfloor N/tasks \rfloor, & \text{if } N\%tasks = 0 \\
  \lfloor N/tasks \rfloor + 1, & \text{otherwise}
  \end{cases}$$
Results: SMXV on 16-sockets

- Single-Producer

![Graph showing GFLOPS vs. # Tasks]

- Oracle Studio 12.2
- GNU 4.6
- Intel 12.2
Results: SMXV on 16-sockets

- Parallel-Producer
Application Case Studies
Flexible Image Retrieval Engine (FIRE)

- FIRE = Flexible Image Retrieval Engine (written in C++)
- Compare the performance of common features on different databases
- Analysis of correlation of different features
- Nested Parallelization with OpenMP

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

**How can Nested OpenMP improve the scalability?**

→ Some synchronization for output ordering on the higher level

→ OpenMP implementation overhead increases over-linear with the number of threads

→ Dataset might better fit to the number of threads

<table>
<thead>
<tr>
<th># Threads</th>
<th>Speedup of FIRE</th>
<th>Sun Fire E25K, 72 dual-core UltraSPARC-IV processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Only outer level</td>
<td>Only inner level</td>
</tr>
<tr>
<td>4</td>
<td>---</td>
<td>3.8</td>
</tr>
<tr>
<td>8</td>
<td>---</td>
<td>7.6</td>
</tr>
<tr>
<td>16</td>
<td>14.8</td>
<td>14.1</td>
</tr>
<tr>
<td>32</td>
<td>29.6</td>
<td>28.9</td>
</tr>
<tr>
<td>72</td>
<td>56.5</td>
<td>---</td>
</tr>
<tr>
<td>144</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
Tasking (done the right way) is even better

- Tasking avoids the thread synchronization in inner parallel region
- Tasking allows for finer parallelization with better load balancing
Critical Point Extraction (NestedCP)

- Algorithm for 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 blocks

The time needed to check different cells may vary considerably!
Addressing the Load Imbalance

```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);
        }
    }
}
```
Nested OpenMP improves Scalability

- Sun Fire E25K with 72 dual-core UltraSPARC-IV with 128 threads
- „Bad“ dataset

<table>
<thead>
<tr>
<th>Without load-balancing static scheduling</th>
<th>10.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic / Guided</td>
<td>33.9</td>
</tr>
<tr>
<td>Time = 4, Block = 4, Cell = 32</td>
<td></td>
</tr>
<tr>
<td>Dynamic / Guided (Sun extension)</td>
<td>55.3</td>
</tr>
<tr>
<td>Weight factor = 20</td>
<td></td>
</tr>
</tbody>
</table>

„Good“ dataset

![Graph showing speedup vs. number of processes]

**Productivity and Performance with Multi-Core Programming**
Christian Terboven | Rechen- und Kommunikationszentrum
Tasking (done the right way) is even better

- Tasking avoids the thread synchronization in inner parallel region
- Tasking allows for finer parallelization with better load balancing

→ Again, Tasking outperforms the Worksharing implementation
→ However, the behavior differs significantly between OpenMP implementations
  → Best speedup of Tasking implementation with Intel: 127
  → Best speedup of Tasking implementation with GNU: 100
  → With Oracle, the Tasking implementation does not scale beyond 32 thr.
KegelSpan

- Laboratory for Machine tools and Production Engineering: 3D simulation of the bevel gear cutting process
- Computes key values (i.a. chip thickness) to enable tool load and tool wear analysis
- Objective: Optimization of manufacturing parameters to reduce production costs
- Parallelization of module *intersect*
  - **GPGPU**: CUDA, OpenCL, PGI Accelerator, OpenACC
  - **CPU**: OpenMP, OpenMP + Auto-Vectorization
Performance and Productivity

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>OpenCL</th>
<th>PGI Acc</th>
<th>OpenACC</th>
<th>Vec + Omp</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated 1st time effort**</td>
<td>30 days</td>
<td>40 days</td>
<td>25 days</td>
<td>25 days</td>
<td>8 days</td>
<td>5 days</td>
</tr>
<tr>
<td>Estimated 2nd time effort***</td>
<td>4.5 days</td>
<td>5 days</td>
<td>1.5 days</td>
<td>1.5 days</td>
<td>3.5 days</td>
<td>0.5 days</td>
</tr>
</tbody>
</table>

preliminary results

Preliminary results show significant speedup improvements for Single and Double Precision computations.

- CUDA (NVIDIA Fermi C2050): smem
- OpenCL (NVIDIA Fermi C2050): smem
- PGI Acc (NVIDIA Fermi C2050): simple
- OpenACC (NVIDIA Fermi C2050): simple
- OpenMP (12 threads, 2xIntel Westmere EP 3.06GHz): vec
- OpenMP (12 threads, 2xIntel Westmere EP 3.06GHz): simple
- Serial (Intel Westmere EP 3.06GHz): simple
Summary and Future Work
Think Parallel. But think about Data first.

Optimizing for modern (NUMA) systems is really about optimizing data locality.

Object-oriented abstractions can be exploited to hide parallelization from the user (as much as the users wishes for), lowering the burden of parallelization and optimization.

C++: Expression Templates can be used to implement parallelization very efficiently, but with some coding work. FORTRAN works as well.

Use the right level of abstraction: Threads vs. Tasks.

Tasks, if done right, can deliver performance on par with Worksharing.

Use the right programming paradigm for your job…
Next Steps

- Tasking implementations do not deliver reliable performance – Intel is best, with noticeable different to others.
  - Talking to OpenMP implementers to improve task scheduling on NUMA
  - Working inside the OpenMP Language Committee to allow the programmer to provide scheduling hints to the OpenMP runtime

- Thread Binding + first touch works well for many applications.
  - Migration necessary when the data access pattern changes over time
  - Working inside the OpenMP Language Committee to add support for data affinity (thread affinity almost done)

- Abstractions: OpenMP is pretty compatible with C++0x.
  - Explore more abstractions and patterns
End.

- Thank you for your attention.

- **aiXcelerate 2012 Workshop**
  
  - Monday, October 8th – Wednesday, October 10th, 2012:
    
    *Hybrid and SMP Programming*
    
    - with speakers from Intel, ScaleMP, …
  
  - Thursday, October 11th – Friday, October 12th, 2012:
    
    *OpenACC Programming*
    
    - with speakers from Nvidia, …

http://www.rz.rwth-aachen.de/aixcelerate