Scalable OpenMP Programming

Dieter an Mey
Center for Computing and Communication
RWTH Aachen University, Germany

www.rz.rwth-aachen.de
anmey@rz.rwth-aachen.de
Overview

- Why OpenMP
- Short OpenMP Introduction
- OpenMP on NUMA Machines
- OpenMP on Clusters
- Conclusion
Overview

• Why OpenMP
• Short OpenMP Introduction
• OpenMP on NUMA Machines
• OpenMP on Clusters
• Conclusion
Why OpenMP?

- Large codes mainly in C++ and Fortran and some C
- Software lifetime measured in decades
- MPI is there to stay on clusters
  - Cannot always be applied easily – if at all
  - Scalability may be limited due to underlying problem (geometry etc.)
  - "MPI only" may not be appropriate for "many cores"
    => MPI + OpenMP (hybrid)
- OpenMP is the alternative and the supplement to MPI
- Scalability of OpenMP limited by current machinery
- So far scalability explored on
  - Sun Fire E25K (144 cores UltraSPARC IV)
  - Sun UltraSPARC T2 (64 threads in one "Niagara 2" chip)
  - Intel Cluster OpenMP
  - ScaleMP "Virtual SMP"
**Keywords and Remarks**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Hits</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>612</td>
<td>since 1994</td>
</tr>
<tr>
<td>OpenMP</td>
<td>150</td>
<td>since 1997 with some 28 hits in our own chapter about OpenMP</td>
</tr>
<tr>
<td>threads</td>
<td>109</td>
<td>frequently in the context of OpenMP, 57 in our chapter about OpenMP</td>
</tr>
<tr>
<td>C++</td>
<td>87</td>
<td>since 1983</td>
</tr>
<tr>
<td>Fortran</td>
<td>69</td>
<td>since 1957</td>
</tr>
<tr>
<td>Chapel</td>
<td>49</td>
<td>with some 22 hits in Zima's chapter about Chapel</td>
</tr>
<tr>
<td>UPC</td>
<td>30</td>
<td>since 2001</td>
</tr>
<tr>
<td>Co-array Fortran</td>
<td>27</td>
<td>since 1998</td>
</tr>
<tr>
<td>hybrid MPI/OpenMP</td>
<td>~26</td>
<td>hard to count</td>
</tr>
<tr>
<td>C</td>
<td>~20</td>
<td>hard to count</td>
</tr>
<tr>
<td>HPF</td>
<td>11</td>
<td>since 1993</td>
</tr>
<tr>
<td>X10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Fortress</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>5</td>
<td>since 1995</td>
</tr>
<tr>
<td>Titanium</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>posix threads</td>
<td>2</td>
<td>1995, Linux since 2003</td>
</tr>
</tbody>
</table>

Overview

• Why OpenMP
• **Short OpenMP Introduction**
• OpenMP on NUMA Machines
• OpenMP on Clusters
• Conclusion
What is OpenMP?

- OpenMP is the API for **portable shared-memory parallel programming** in C/C++ and Fortran.
- It consists of compiler **directives**, **runtime calls** and **environment variables**.
- The parallelism has to be expressed **explicitly** by the programmer. Still, it can be combined with autoparallelization.
- **Parallel regions** are executed by a **team** of threads.
- Work can be distributed among the threads of by **worksharing constructs**, like the **parallel loop construct**, which provides powerful **loop scheduling** mechanisms.
- **Tasks** (code plus data environment) can be queued by a **task construct** and their execution by any thread of the team can be deferred. (well suited for recursions, while loops ....)
- **Nested parallelism** is supported.
Memory Model of OpenMP

- OpenMP: Shared-Memory model
  - All threads share a common address space (shared memory)
  - Threads can have private data as well (explicit user control)

- Relaxed memory consistency
  - Temporary View ("Caching"): Memory consistency is guaranteed only after synchronization points, namely implicit and explicit flushes
    - Each OpenMP barrier includes a flush
    - Exit from worksharing constructs include barriers by default
    - Entry to and exit from critical regions include a flush
    - Entry to and exit from lock routines (OpenMP API) include a flush
**Example: Calculate $\pi$ by Numerical Integration**

\[
\Pi = \int_{0}^{1} \frac{4}{1 + x^2} \, dx
\]

double \( f(double \, x) \) {return (double)4.0/((double)1.0 + (x*x));}

```c
void computePi() {
    double h = (double)1.0 / (double)n;
    double sum = 0, x;

    #pragma omp parallel private(x) shared(h,n) num_threads(4) // "fork"
    {
        printf ( "my threadid %d\n", omp_get_thread_num() );
        #pragma omp for reduction(+:sum) schedule(static)
        for (int i = 1; i <= n; i++) {
            x = h * ((double)i - (double)0.5);
            sum += f(x);
        } // implied barrier
        #pragma single
        {
            pi = h * sum;
            printf("\n\npi is approximately %.16f\n", pi);
        } // implied barrier
    } // "join"
}
```
Heat Flow Simulation with FEM - ThermoFlow60

Thomas Haarmann, Wolfgang Koschel, Jet Propulsion Laboratory, RWTH Aachen University

- simulation of the heat flow in a rocket combustion chamber
- Finite Element Method
- OpenMP Parallelization
  - 30000 lines of Fortran
  - 200 OpenMP directives, 69 parallel loops,
  - 1 main parallel region, "orphaning"

Speedup: ~40 with 68 threads on a Sun Fire 15K (72 UltraSPARC III single core)
Nested OpenMP for Critical Point Computation

Samuel Sarholz, Andreas Gerndt, Computing and Communication Center, RWTH Aachen University

- Analysis of complex and accurate fluid dynamics simulations
- Extraction of Critical Points for VR (Location with velocity = 0)
- 25-100% efficiency with 128 threads on Sun Fire E25K (72 UltraSPARC IV dual core) depending on data set

```c
// Loop over time levels
#pragma omp parallel for num_threads(nTimeThreads) schedule(dynamic,1)
for (curT=1; curT<=maxT; ++curT) {
  // Loop over Blocks
  #pragma omp parallel for num_threads(nBlockThreads) schedule(dynamic,1)
  for (curB=1; curB<=maxB; ++curB) {
    // Loop over Cells
    #pragma omp parallel for num_threads(nCellThreads) schedule(guided)
    for (curC=1; curC<=maxC; ++curC) {
      FindCriticalPoints (curT, curB, curC); // highly adaptive algorithm (bisectioning)
      // huge load imbalances
    }
  }
}
```

Scalable OpenMP Programming – D. an Mey
Overview

• Why OpenMP
• Short OpenMP Introduction
• OpenMP on NUMA Machines
• OpenMP on Clusters
• Conclusion
The Earth is Flat

OpenMP is hardware agnostic
It has no notion of data locality

=>

The Affinity Problem:
How to maintain or improve the nearness of threads and their most frequently used data

Or:
Where to run threads?
Where to place data?
Operating System Aspects

- Today, operating systems are aware of the HW architecture and
  - group (OS-) processors according to their locality
  - launch processes to less loaded "processor groups" (Solaris: locality groups, Linux: NUMA nodes)
  - try to keep new threads close to their master – if advantageous
  - try to avoid moving threads around
  - try to allocate data close to the thread which initializes them (first touch policy)

- But what if
  - program behavior is unpredictable or changes over time?
  - the machine is overloaded such that multiple users' jobs interfere?
  - Frequently data is initialized at the beginning of the program by the initial thread, but later on used by multiple threads!
  - Things are getting more complicated with nested parallelization...
Automatic Migration

- In an optimal case the operating system **automatically** detects which thread accesses which data most frequently.
- It may **replicate data** which is read by multiple threads.
- It may **migrate data** which is modified and used by threads residing on remote locality groups.
- (HW counters may assist the OS to make decisions on migration)

- Automatic migration had been implemented in the IRIX operating system for the SGI Origin systems.
  - Users complained about the high overhead which was involved in automatic migration (TLB shoot-down).

- Automatic migration was also implemented in the Sun's WildFire project, which worked well but was not productized.
User Control of Affinity

- The OS may do a reasonable good job,
  - if the machine is not overloaded
  - and the **first touch policy** has been carefully taken into account
  - and the **program does not change** its behaviour with respect to locality.

- There are possibilities for additional **user control**
  - Explicit **binding of threads** to processors by
    - environment variables,
    - commands,
    - system calls
  - **Control Memory allocation**
    - By carefully **first touching** data by the thread which later uses them
    - Change **default memory allocation strategy** (Linux, Solaris)
    - or **explicit migration of pages** (Linux, Solaris)
      (unfortunately Linux and Solaris use a different approach)
Sun Fire V40z (w/ dualcore AMD Opteron Chip)

4 AMD Opteron 875
dual core processors
2.2 GHz

Cache-coherent
HyperTransport
Connections
Sparse-Matrix-Vector-Multiplication as part of the Navier Stokes Solver DROPS (C++)

Performance of a cc-Numa system is very sensitive to data placement.
Everything under control?

- In principle, yes
  - if threads are always explicitly bound and
  - data is always explicitly migrated to where it is used
- No portability
- What if multiple jobs or processes (hybrid MPI/OpenMP) bind to the same processors?
  - On Solaris, we provide a library to bind to empty processors
  - Still tedious
- How about nested OpenMP?
  - Typically, threads are organized in a pool by the runtime system and may be allocated variably, thus losing data affinity!
  - Always explicitly binding and migrating may be too costly
OpenMP nested, here: 4x2 threads

```c
!
$omp parallel private(me) num_threads(4)
  me = omp_get_thread_num()
  CALL stream(a(1,me),b(1,me),c(1,me))
$omp end parallel
...
subroutine stream (a,b,c)
  double precision a(*),b(*),c(*)
...
$omp parallel do num_threads(2)
  do 50 j = 1,n
      c(j) = a(j) + x*b(j)
  50     continue
$omp end parallel do
...
```

Depending on the placement of threads and data, the memory bandwidth varies between 2.5 GB/s and 11.8 GB/s.

With Sun Studio on Solaris, SUNW_MP_THR_AFFINITY=TRUE helps.
OpenMP nested

Thread tree

Typically, threads are organized in a pool by the runtime system and may be allocated variably, thus losing data affinity!

Depending on the placement of threads and data, the memory bandwidth varies between 2.5 GB/s and 11.8 GB/s.
NPB Benchmark BT-MZ Class B

Time

Speedup

- MPI+OMP, SGI Altix
- nested OMP, SGI Altix
- nested OMP, SunFire

H. Jin, Nasa Ames
Overview

• Why OpenMP
• Short OpenMP Introduction
• OpenMP on NUMA Machines
• OpenMP on Clusters
• Conclusion
OpenMP on Clusters

- Multiple Approaches (based on MPI, on DSM ...) so far not very successful or uncomplete.

- Intel Cluster OpenMP on Commodity Infiniband Cluster
  - Based on TreadMarks (twin pages, sending diffs, ...)
  - Integrated in commercial compiler (C++ and F95)
  - Profits from OpenMP's memory model (relaxed consistency, temporary view of shared data, consistency enforced at well defined synchronization points.)
  - Need to explicitly mark some shared variables (sharable directive)

- ScaleMP – Versatile SMP™ Architecture
  - Aggregation of multiple x86 boards into one larger system
  - Cache coherent connection through InfiniBand
  - Modified IB stack and BIOS, caching strategies
  - Single system image, virtual SMP machine
  - Aggregation of all I/O resources to the OS
  - Affinity matters!
## EPCC OpenMP Micro-Benchmarks


<table>
<thead>
<tr>
<th></th>
<th>Tigerton</th>
<th>Opteron</th>
<th>CLOMP</th>
<th>ScaleMP (MEG)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARALLEL FOR</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 threads</td>
<td>1.31</td>
<td>1.36</td>
<td>723.77</td>
<td>264.83</td>
</tr>
<tr>
<td>16 threads</td>
<td>5.01</td>
<td>7.17</td>
<td>4342.82</td>
<td>717.77</td>
</tr>
<tr>
<td><strong>BARRIER</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 threads</td>
<td>0.75</td>
<td>0.58</td>
<td>598.82</td>
<td>144.45</td>
</tr>
<tr>
<td>16 threads</td>
<td>2.55</td>
<td>2.64</td>
<td>4062.67</td>
<td>429.35</td>
</tr>
<tr>
<td><strong>REDUCTION</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 threads</td>
<td>1.56</td>
<td>2.05</td>
<td>932.18</td>
<td>298.06</td>
</tr>
<tr>
<td>16 threads</td>
<td>5.68</td>
<td>25.77</td>
<td>4686.00</td>
<td>801.91</td>
</tr>
</tbody>
</table>

Overhead in microseconds [us].

2 Nodes

Binding: 1 Thread/board for CLOMP and ScaleMP(MEG)
8 Threads/board for CLOMP and ScaleMP(MEG)
## Stream Benchmark

<table>
<thead>
<tr>
<th># threads</th>
<th>Tigerton</th>
<th>Opteron (*)</th>
<th>CLOMP</th>
<th>ScaleMP (RWTH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2080.78</td>
<td>1882.24</td>
<td>3321.08</td>
<td>2674.13</td>
</tr>
<tr>
<td>2</td>
<td>4033.88</td>
<td>3665.35</td>
<td>6495.34</td>
<td>5330.22</td>
</tr>
<tr>
<td>4</td>
<td>7008.31</td>
<td>6674.57</td>
<td>10031.07</td>
<td>10439.76</td>
</tr>
<tr>
<td>8</td>
<td>7156.36</td>
<td>9029.56</td>
<td>10344.97</td>
<td>17478.77</td>
</tr>
<tr>
<td>16</td>
<td>7508.01</td>
<td>8787.33</td>
<td>10473.24</td>
<td>18666.49</td>
</tr>
</tbody>
</table>

Bandwidth in MB/s. Scattered Binding. 2 Nodes

(*) We see better performance on our 4-socket Opteron machine running Solaris
Sparse Matrix-Vector-Multiplication [Mflop/s]
Apply Suitable Strategy!

parallel loop over #rows, dynamic loop schedule

parallel loop over #nonzeros, static partitions
FIRE: Image Retrieval System
Scales on ScaleMP

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

ScaleMP (MEG)

<table>
<thead>
<tr>
<th># threads</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>25</td>
</tr>
<tr>
<td>32</td>
<td>35</td>
</tr>
</tbody>
</table>

- Fire
- linear speedup
SHEMAT on ScaleMP

- Simulation of Coupled Flow, Heat Transfer and Transport Interaction
- BiCGStab Solver with ILU0 Preconditioner
- Nested Parallelization with OpenMP
- Explicit binding of all threads in all inner parallel regions

**Graph:**
- SGI F1200 vSMP: 4 nodes with 2x4 cores
- Dell 140 vSMP: 4 nodes with 2x2 cores
- Harpertown: 1 nodes with 2x4 cores
- ScaleMP: ~ 3x on 4 nodes
Overview

• Why OpenMP
• Short OpenMP Introduction
• OpenMP on NUMA Machines
• OpenMP on Clusters
• Conclusion
Conclusion

- Scalable applications may need multiple levels of parallelization
- OpenMP suitable for a growing number of cores per node
- Combining MPI and OpenMP is getting more popular
- OpenMP on Clusters an alternative, if MPI is too hard to apply.

- Thread/Data Affinity is essential for OpenMP performance on ccNUMA machines and even more on Clusters
- OpenMP is hardware agnostic
- Need for control of thread and data placement
- Need for data migration, explicite and/or automatic for irregular, adaptive problems
- May profit from coherency, but there are alternatives as well
For those who want to get a quick impression of OpenMP, I have a little OpenMP demo …

It does not scale very far on my laptop though …
Making OpenMP ccNUMA-ready

Our proposal:

- Do not describe the HW architecture within OpenMP
- Describe the structure of a (nested) OpenMP program as a series of thread trees.
- Pass a description of the tree to the runtime system upfront
- Maybe give some guidance like "place threads of a team scattered or compact"
- Let the runtime system take care for the mapping of threads onto the HW (analogous to MPI topology concept)
- Guarantee threadprivate persistence as long as the tree remains constant.
- Provide first touch or random placement as a general strategy
- Use data migration ("next touch") to move data to threads
- Did not make it into OpenMP3.0 (released in May 2008)
### FIRE: Image Retrieval System

Nested OpenMP improves scalability

<table>
<thead>
<tr>
<th>Speedup of FIRE</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>4</td>
<td>---</td>
</tr>
<tr>
<td>8</td>
<td>---</td>
</tr>
<tr>
<td>16</td>
<td>14.8</td>
</tr>
<tr>
<td>32</td>
<td>29.6</td>
</tr>
<tr>
<td>72</td>
<td>56.5</td>
</tr>
<tr>
<td>144</td>
<td>---</td>
</tr>
</tbody>
</table>
Simulating the Flow through the Human Nose
TFS on Solaris

SUNW_MP_THR_AFFINITY=TRUE

Thread affinity + processor binding + data migration improved the performance by ~25% on a Sun Fire E 25K

<table>
<thead>
<tr>
<th>Before</th>
<th>Improved thread affinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>#threads</td>
<td>Speed-up</td>
</tr>
<tr>
<td>64</td>
<td>20</td>
</tr>
<tr>
<td>121</td>
<td>20</td>
</tr>
</tbody>
</table>