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

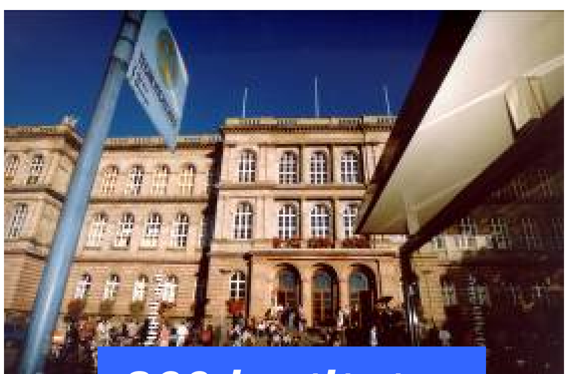
- Why OpenMP
- Short OpenMP Introduction
- OpenMP on NUMA Machines
- OpenMP on Clusters
- Conclusion

RWTH Aachen University: Key Figures WT 06/07

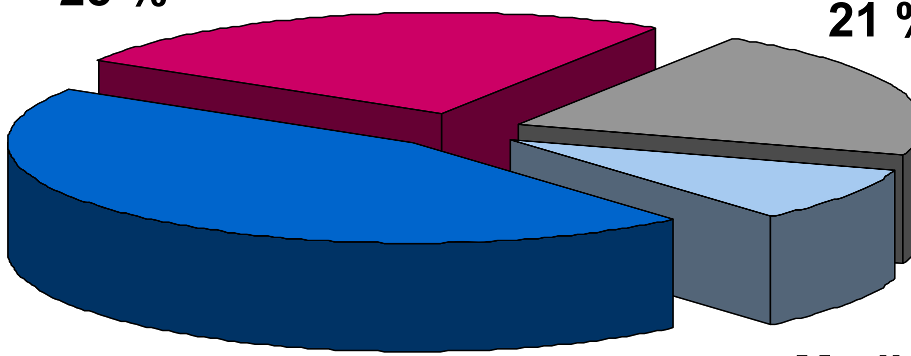
30180 students

428 professorships

3600 academic staff



260 institutes

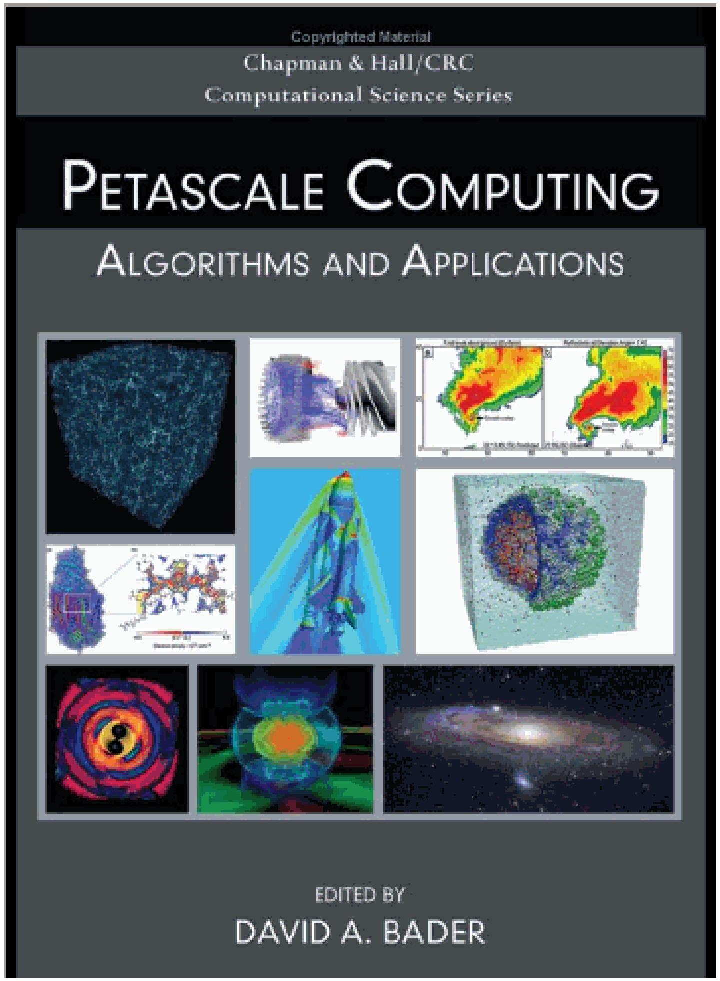


Why OpenMP?

- 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"

Statistics from the "First Petascale Book"



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

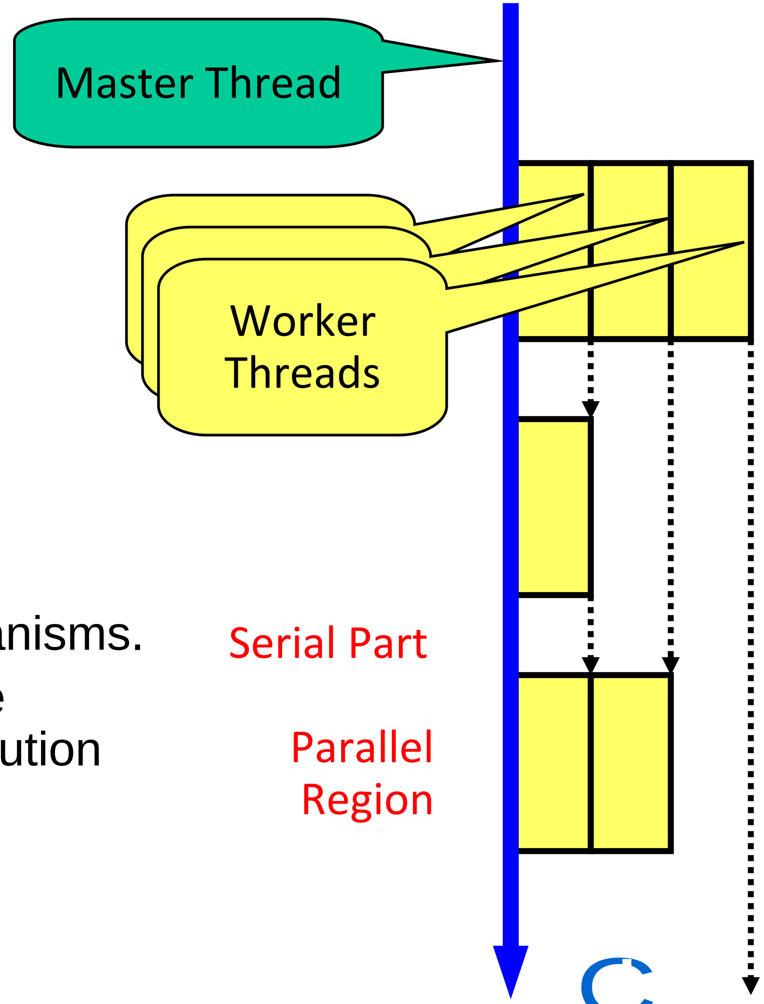
Petascale Computing: Algorithms and Applications (Chapman & Hall/Crc Comp. Sci. Ser.)
edited by David A. Bader , 2007, 528 pages, 24 contributions, 90 contributors

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.

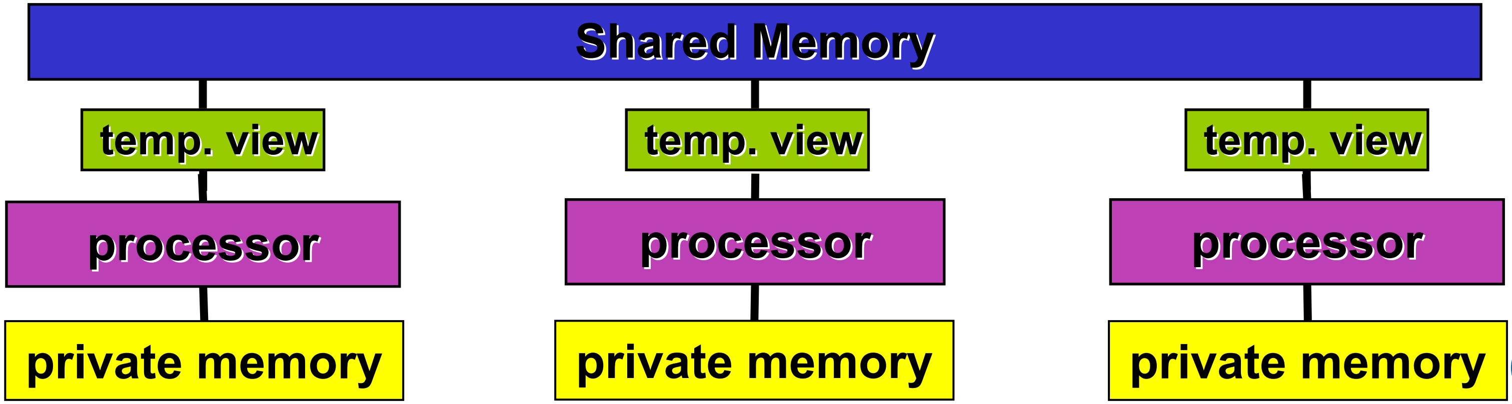


Center for Computing and Communication

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 `flush`es
    - 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 π 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));}

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 omp single
       {
          pi = h * sum;
          printf("\n\npi is approximately %.16f\n", pi);
       }  // implied barrier
 }  // "join"
}
```

Parallel Region

All threads

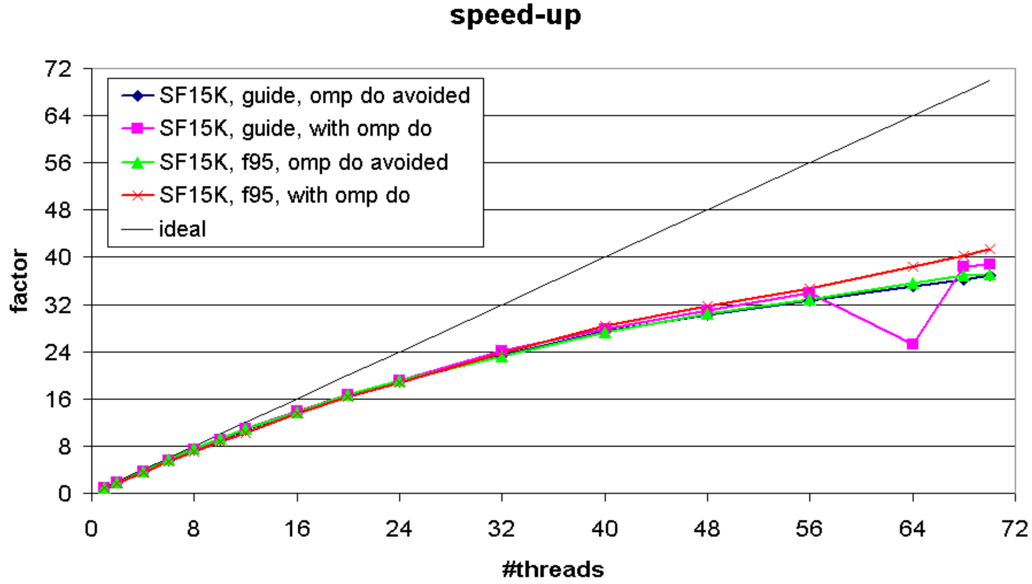
Work-sharing

One thread

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 Parallelelization
  - 30000 lines of Fortran
  - 200 OpenMP directives, 69 parallel loops,
  - 1 main parallel region, "*orphaning*"

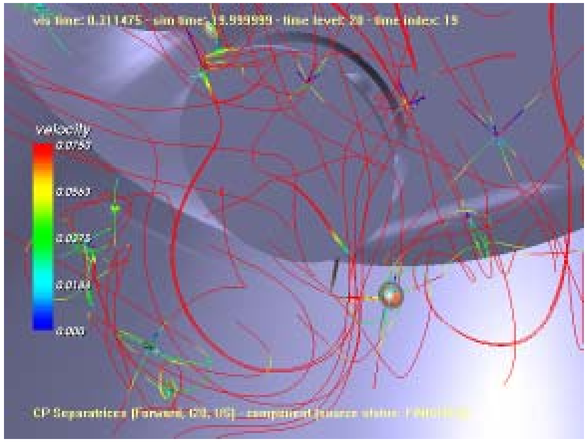


Speedup: ~40 with 68 threads on a Sun Fire 15K (72 UltraSPARC III single core)

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**




```
// 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
```

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)

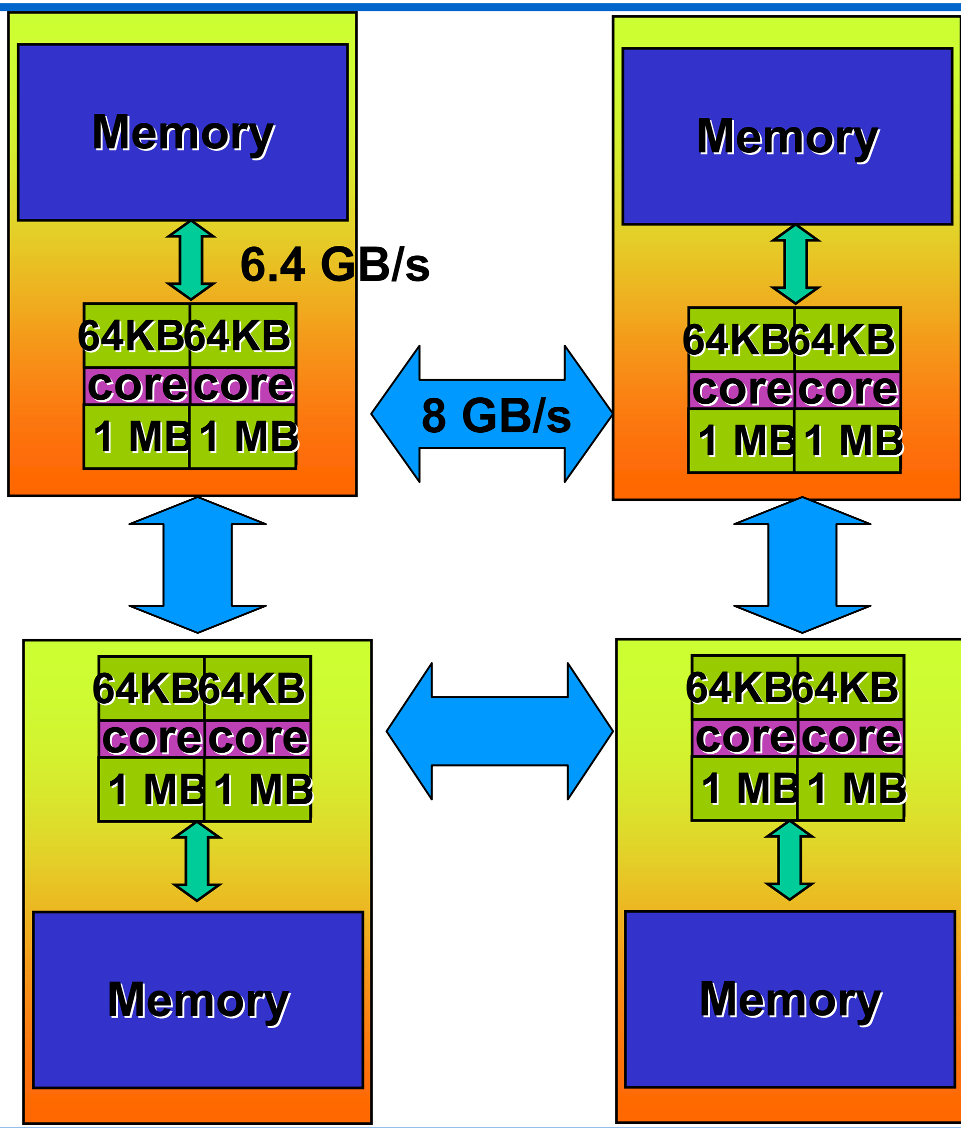
- Automatic migration had been implemented in the IRIX operating system for the SGI Origin systems.
  - Users complained about the high overhead which was involved in automatic migration (TLB shoot-down)

- 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)

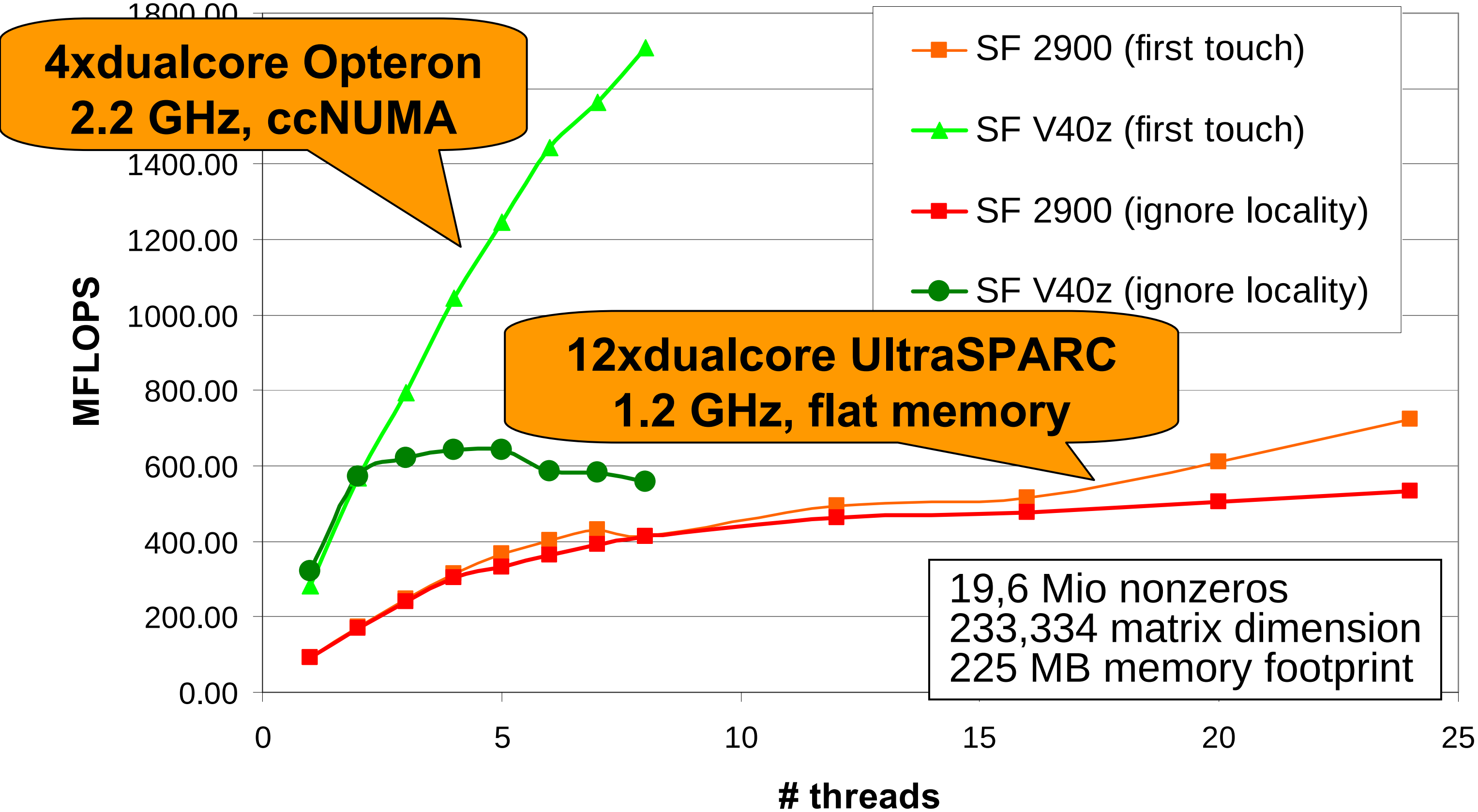
Sun Fire V40z (w/ dualcore AMD Opteron Chip)

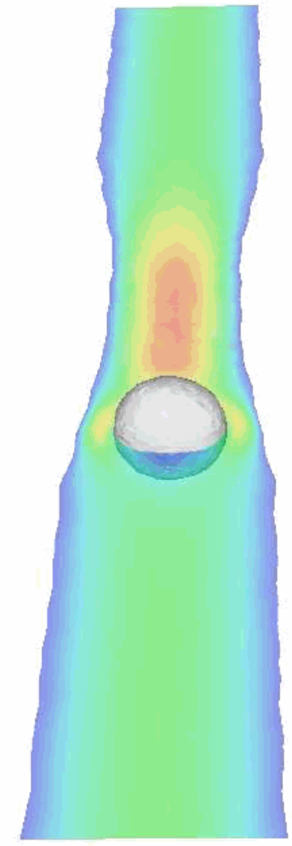


4 AMD Opteron 875 dual core processors 2.2 GHz

Cache-coherent HyperTransport Connections

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





IWOMP 2005

Performance of a cc-Numa system is very sensitive to data placement.

RWTH Center for Computing and Communication

Everything under control ?

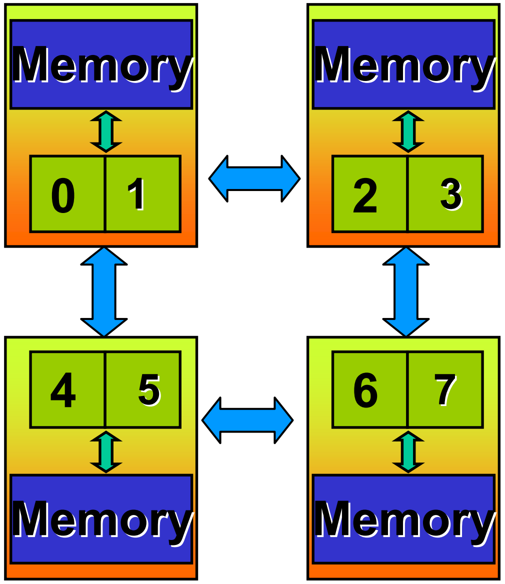
- In principle, yes
  - if threads are always explicitly bound and
  - data is always explicitly migrated to where it is used
- no portability

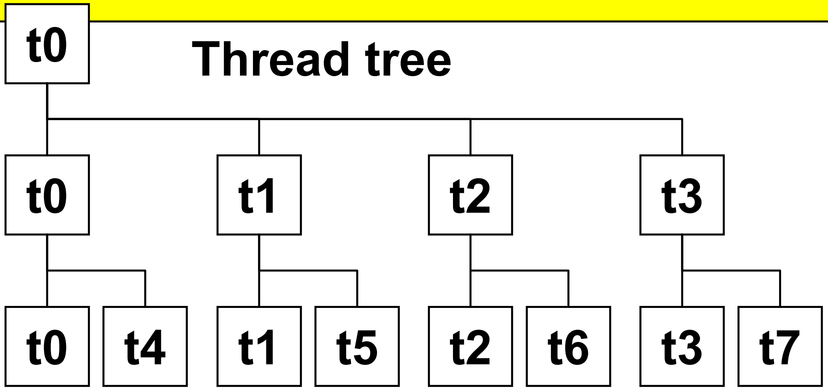
- what if multiple jobs or processes (hybrid MPI/OpenMP) bind to the same processors?
  - On Solaris, we provide a library to bind to empty processors
  - Still tedious

- How about nested OpenMP ?
  - Typically, threads are organized in a pool by the runtime system and may be allocated variably, thus loosing data affinity!
  - Always explicitly binding and migrating may be too costly

OpenMP nested, here: 4x2 threads

```
!$omp parallel private(me) num_threads(4)
      me = omp_get_thread_num()
      CALL stream(a(1,me),b(1,me),c(1,me))
!$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
      ...
```

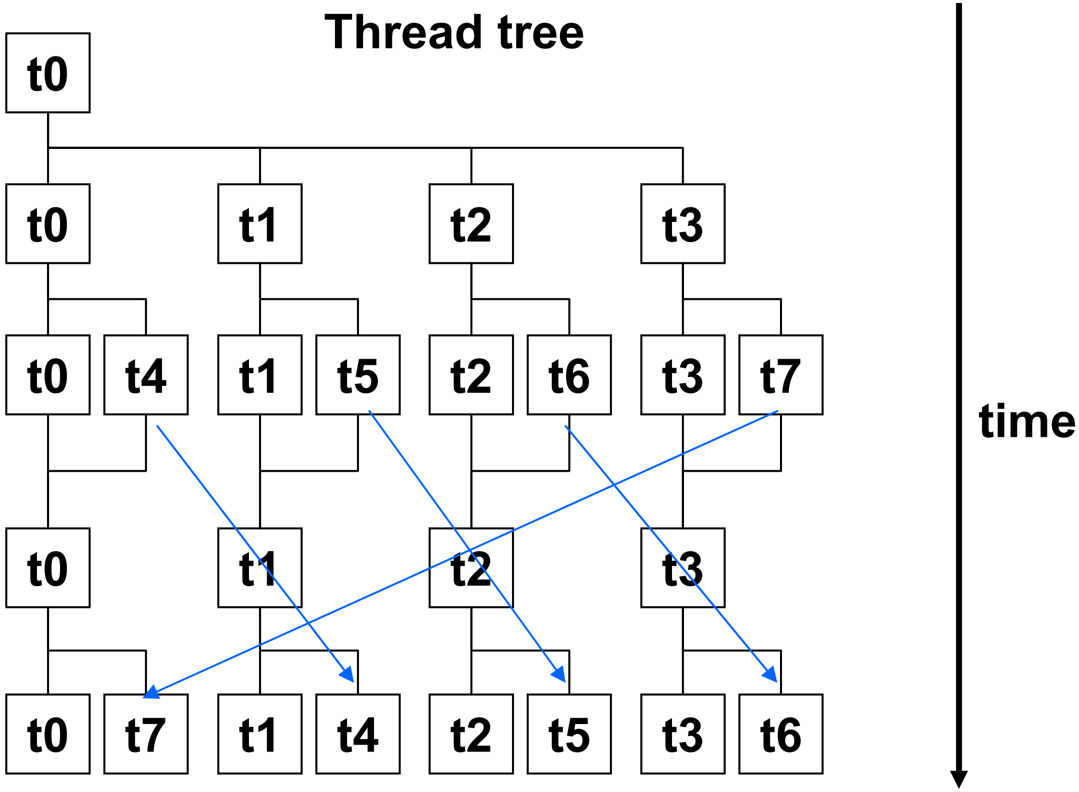


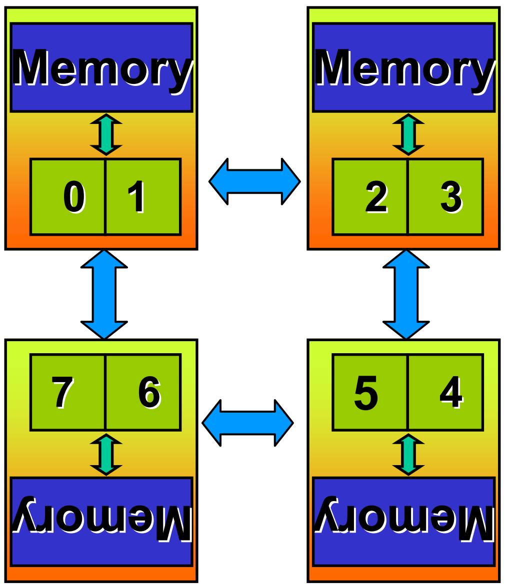


Depending on the placement of threads and data, the memory bandwidth varies between ***2.5 GB/s and 11.8 GB/s***

With Sun Studio on Solaris SUNW_MP_THR_AFFINITY=TRUE helps.

OpenMP nested

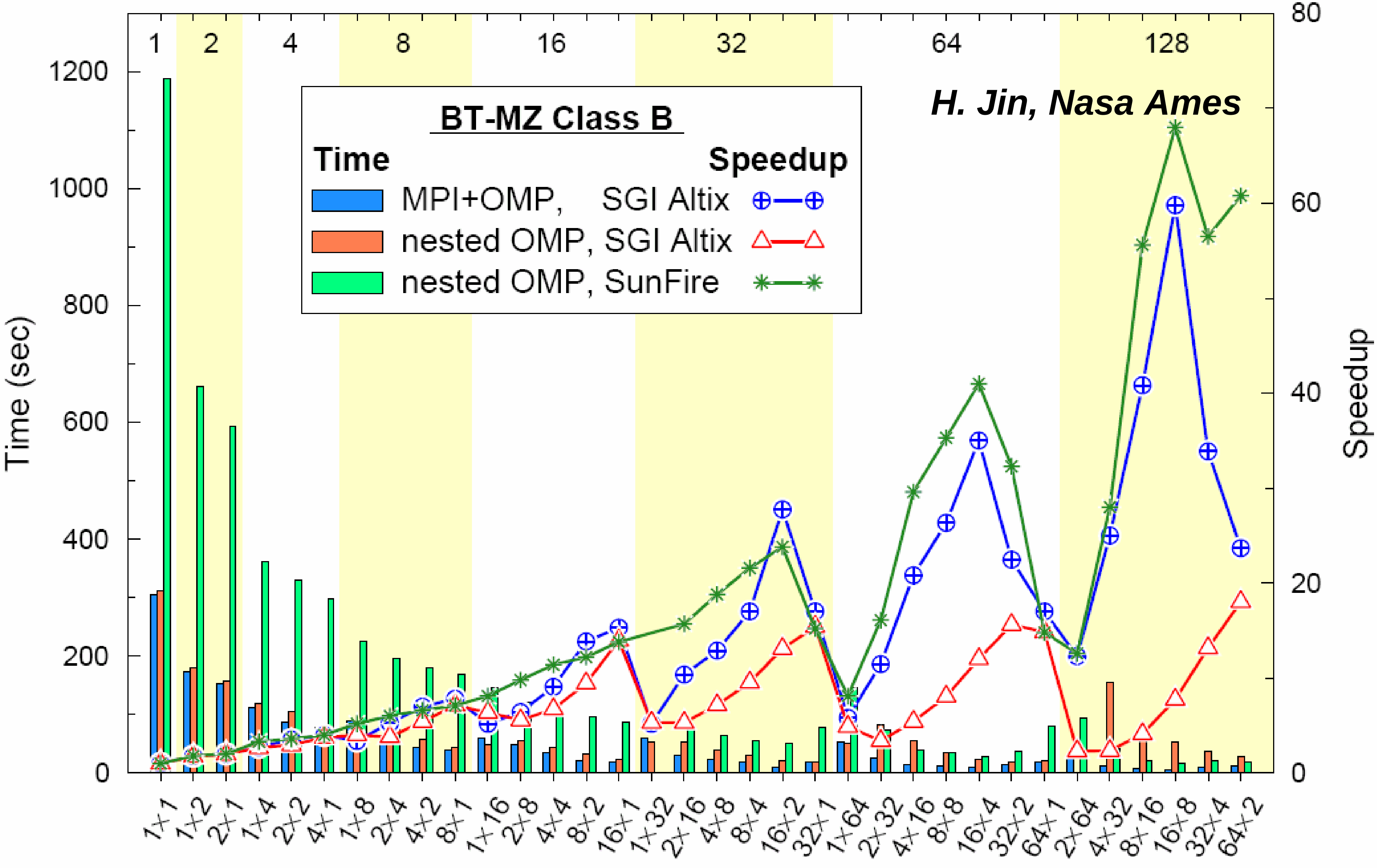




Typically, threads are organized in a pool by the runtime system and may be allocated variably, thus loosing data affinity!

*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



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 explicitely 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

J. M. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. 1999.

	Tigerton	Opteron	CLOMP	ScaleMP (MEG)
PARALLEL FOR				
2 threads	**1.31**	**1.36**	**723.77**	**264.83**
16 threads	**5.01**	**7.17**	**4342.82**	**717.77**
BARRIER				
2 threads	**0.75**	**0.58**	**598.82**	**144.45**
16 threads	**2.55**	**2.64**	**4062.67**	**429.35**
REDUCTION				
2 threads	**1.56**	**2.05**	**932.18**	**298.06**
16 threads	**5.68**	**25.77**	**4686.00**	**801.91**

About three orders of magnitude

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

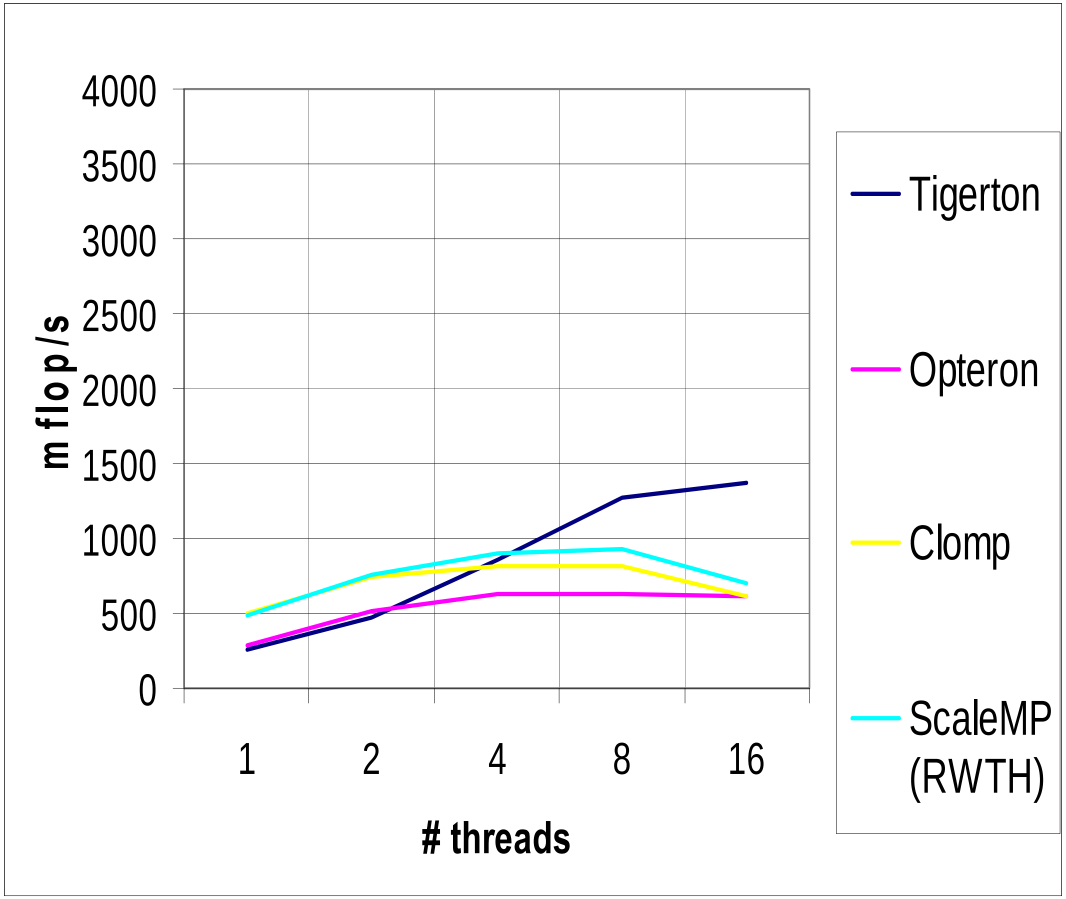
# threads	Tigerton	Opteron (*)	CLOMP	ScaleMP (RWTH)
1	**2080.78**	**1882.24**	**3321.08**	**2674.13**
2	**4033.88**	**3665.35**	**6495.34**	**5330.22**
4	**7008.31**	**6674.57**	**10031.07**	**10439.76**
8	**7156.56**	**9629.90**	**10344.97**	**17478.77**
16	**7508.01**	**8787.33**	**10473.24**	**18666.49**

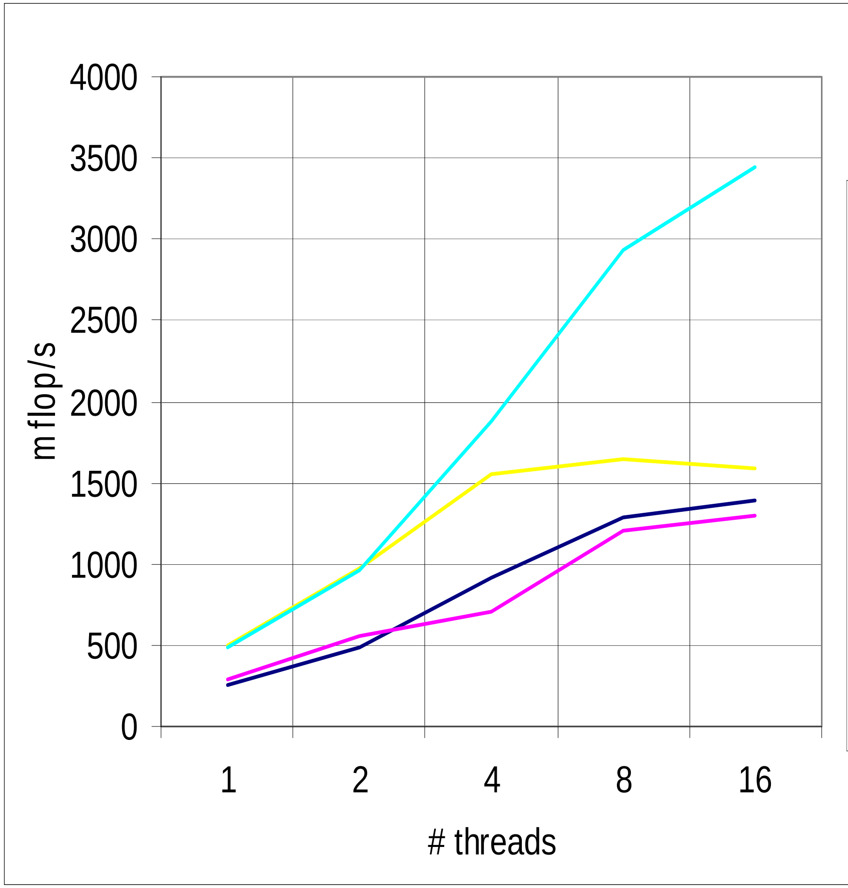
Higher Memory Bandwidth

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

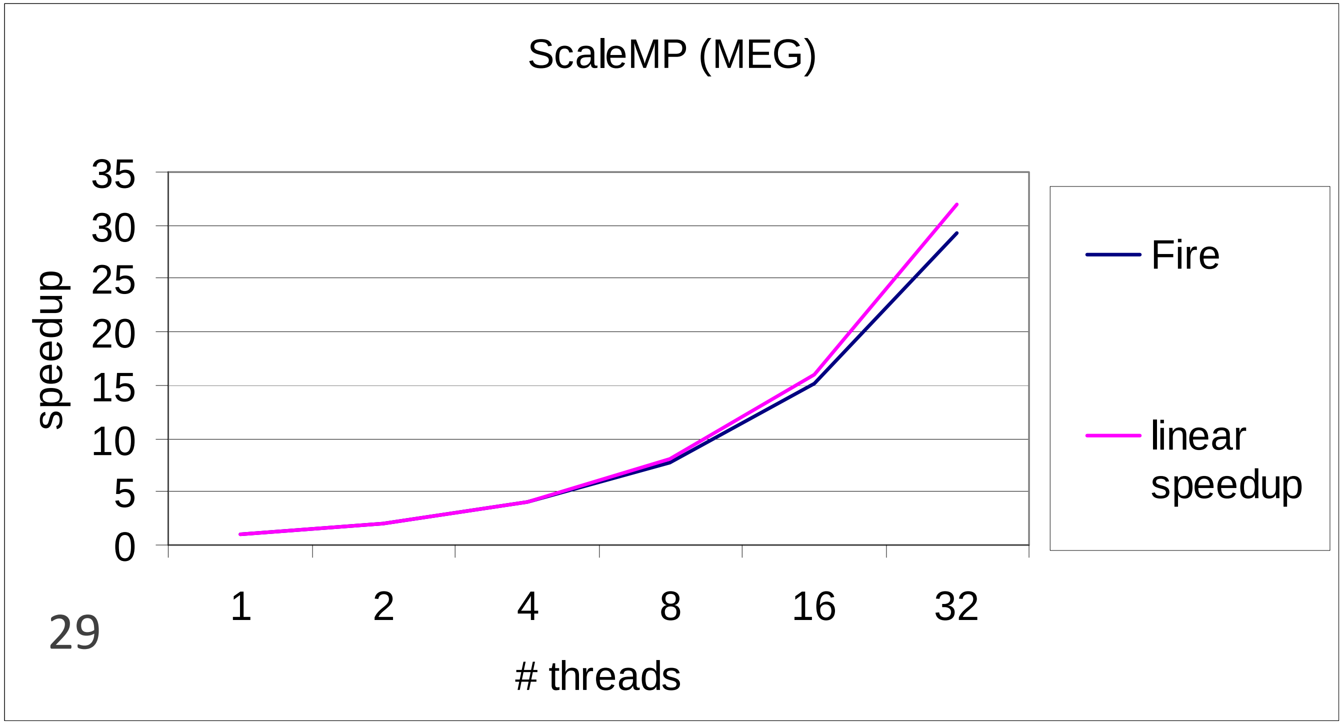
RWTH Center for Computing and Communication

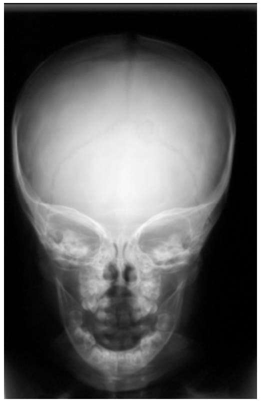
FIRE: Image Retrieval System Scales on ScaleMP

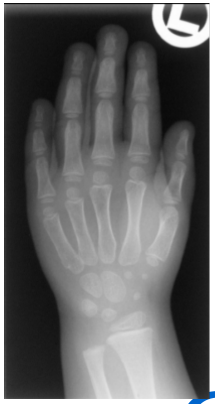
FIRE = Flexible Image Retrieval Engine

- 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

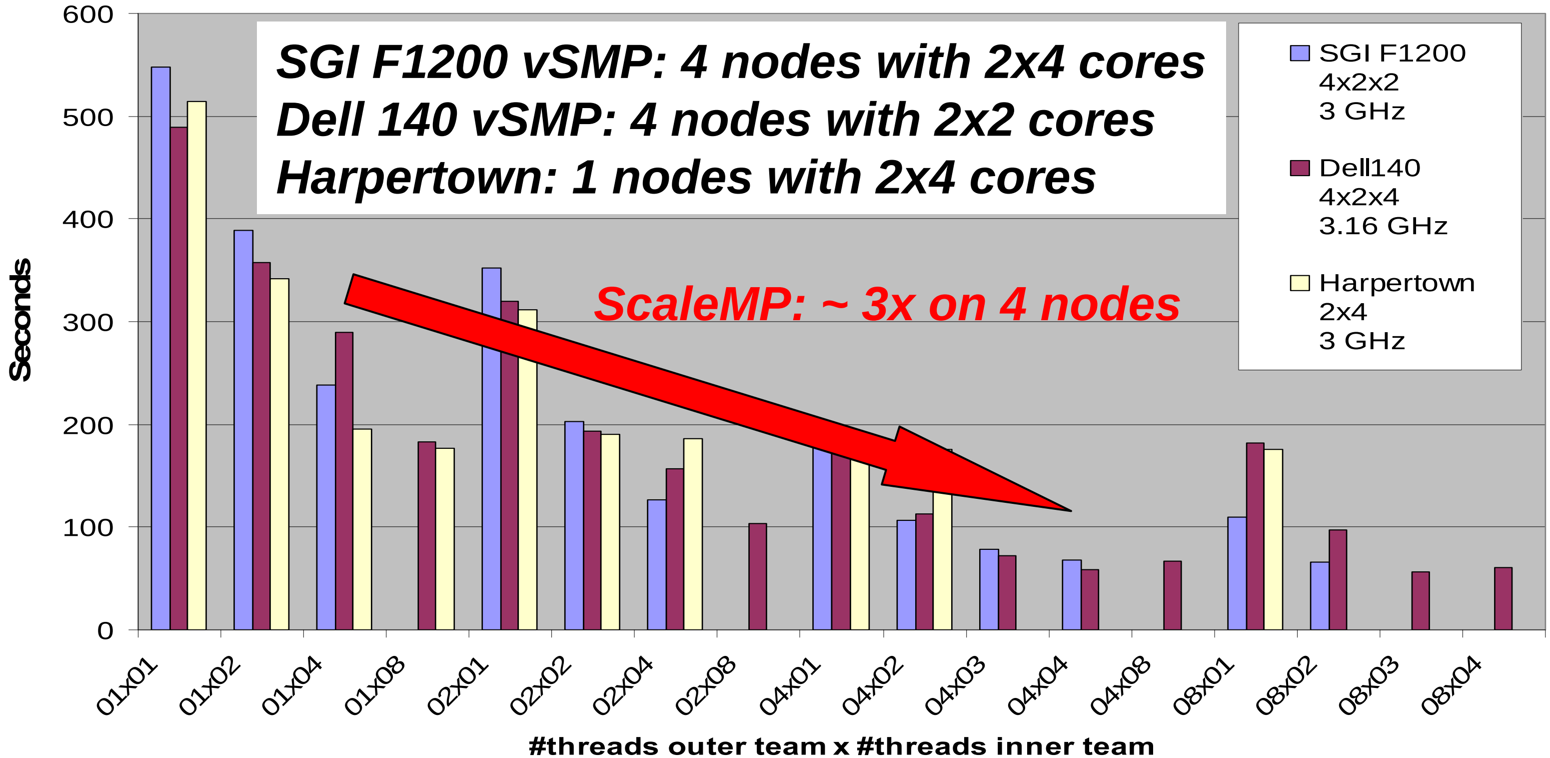






SHEMAT on ScaleMP

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



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

Thank you for your attention!

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)

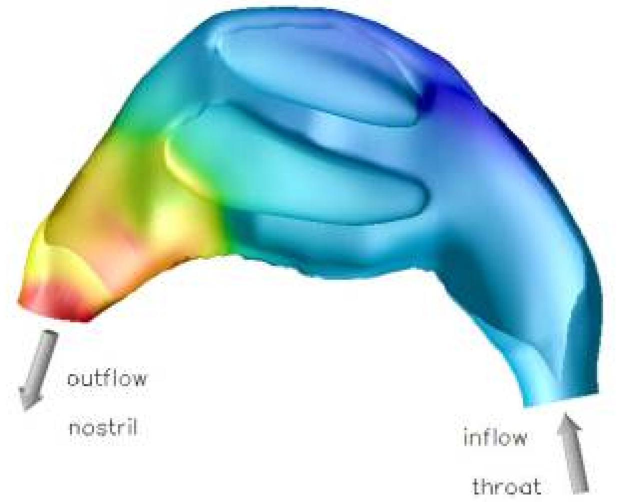
FIRE: Image Retrieval System
Nested OpenMP improves scalability

Speedup of FIRE	Sun Fire E25K, 72 dual-core UltraSPARC-IV processors		
# Threads	Only outer level	Only inner level	Nested OpenMP
4	---	3.8	---
8	---	7.6	---
16	14.8	14.1	15.4
32	29.6	28.9	30.6
72	56.5	---	67.6
144	---	---	133.3

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



Before		**Improved thread affinity**		
#threads	Speed-up	#threads	Speed-up	Strategy (best effort)
64	**20**	64	**25**	thread balancing 2-11 threads per team, static schedule, 16 threads in outer team
121	**20**	128	**27**	block grouping, 16 threads in outer team