RWTH AACHEN UNIVERSITY

# OpenMP on NUMA

Christian Terboven <terboven@rz.rwth-aachen.de>

20.11.2012 / Aachen, Germany

Stand: 19.11.2012

Version 2.3

# Agenda

- OpenMP Thread Binding Howto
- *Case Study*: Parallel Sparse Matrix Vector Multiplication

# OpenMP Thread Binding Howto

# Step 1: Get Info on the System Topology

- **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-ls` tool
    - `hwloc-ls`
    - 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 `KMP_AFFINTIY`
  - `KMP_AFFINITY=scatter`: Put threads far apart
  - `KMP_AFFINITY=compact`: Put threads close together
  - `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 GOMP_CPU_AFFINITY
  - `GOMP_CPU_AFFINTIY=<core_list>`: Bind threads in the order in which they are started to the cores given in the list, one thread per core.

# Parallel Sparse Matrix Vector Multiplication

# 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

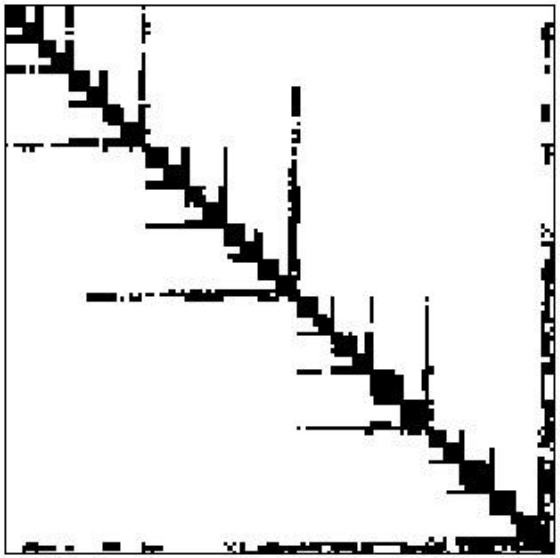
- Load Imbalance
- Data Locality on cc-NUMA architectures
- Memory Bandwidth (consumption per thread)
- Cache Effects

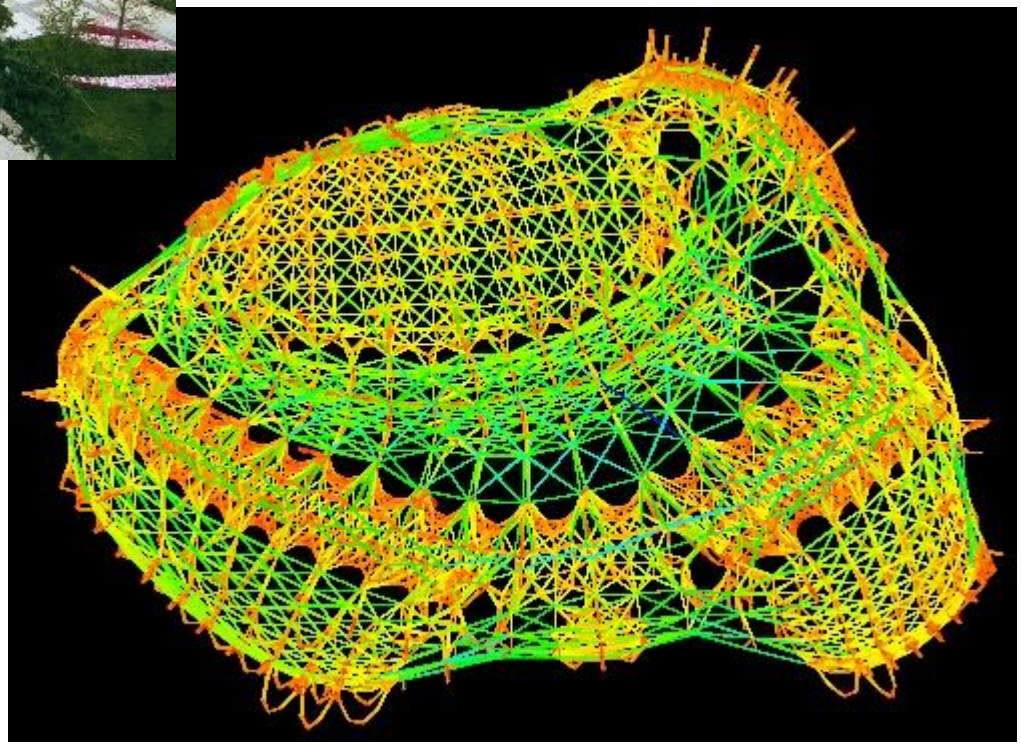
# Case Study: Sparse Matrix-Vector-Multiplication

Beijing Botanical Garden

| | |
|---|---|
| left: | original building |
| bottom-right: | lattice model |
| bottom-left: | matrix shape |

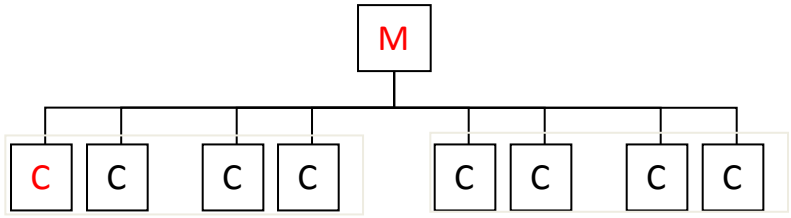
(image copyright: Beijing Botanical Garden and University of Florida, Sparse Matrix Collection)



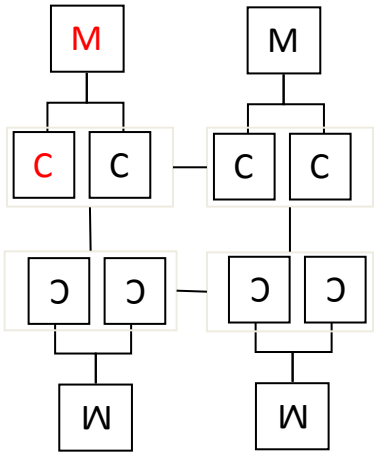


# Experiment Setup and Hardware

- **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)

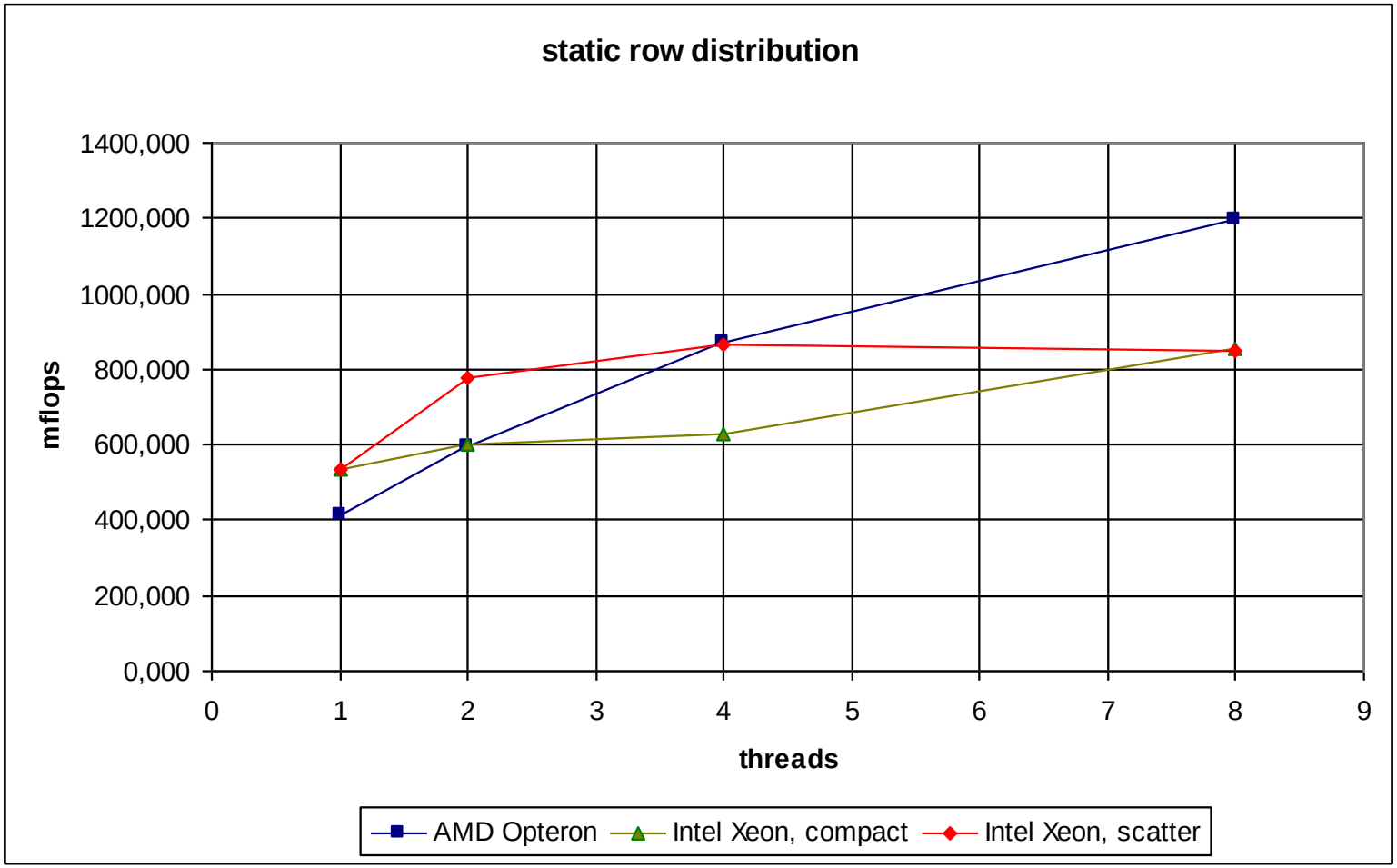



- **Matrices in CRS format**
  - „large“ means 75 MB >> caches

**→ SMXV is an important kernel in numerical codes (GMRES, …)**

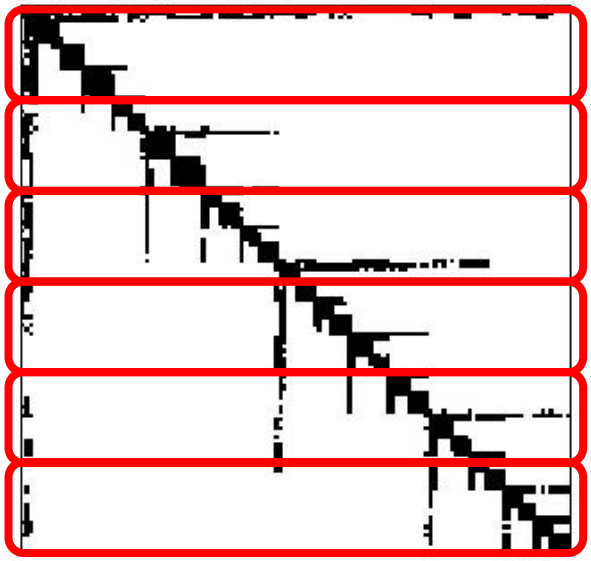
# Performance with Static Row Distribution (1)

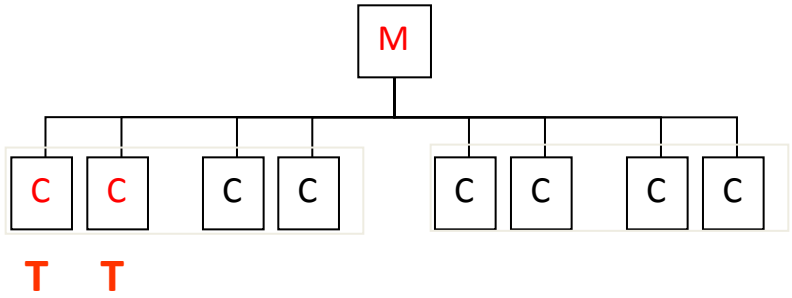
- compact: threads packed tightly; scatter: threads distributed



# Performance Analysis of (1)

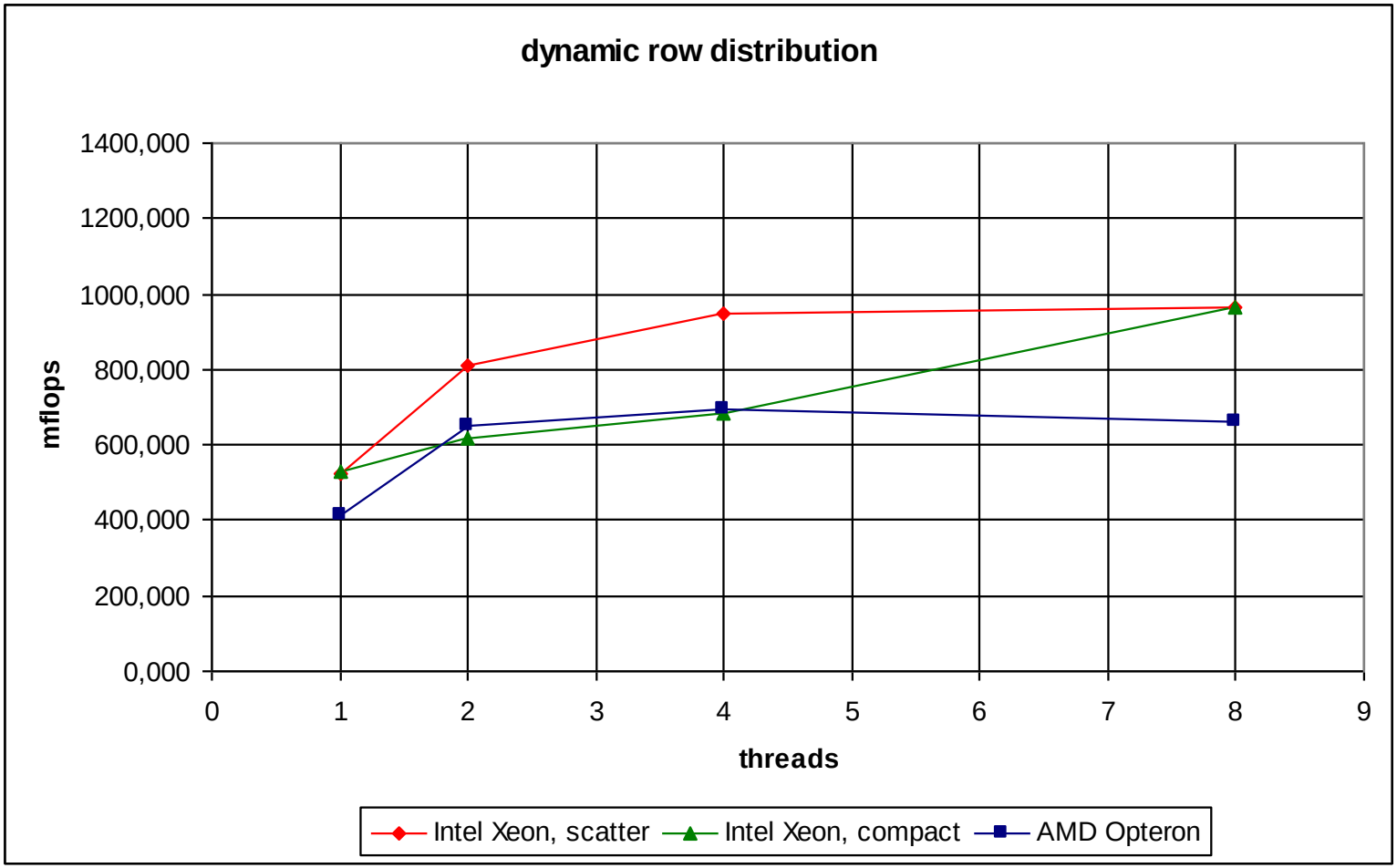
- **Load Balancing: Static data distribution is not optimal!**
- **Data Locality on cc-NUMA architectures: Static data distribution is not optimal!**
- **Memory Bandwidth: Compact thread placement is not optimal on Intel Xeon processors!**





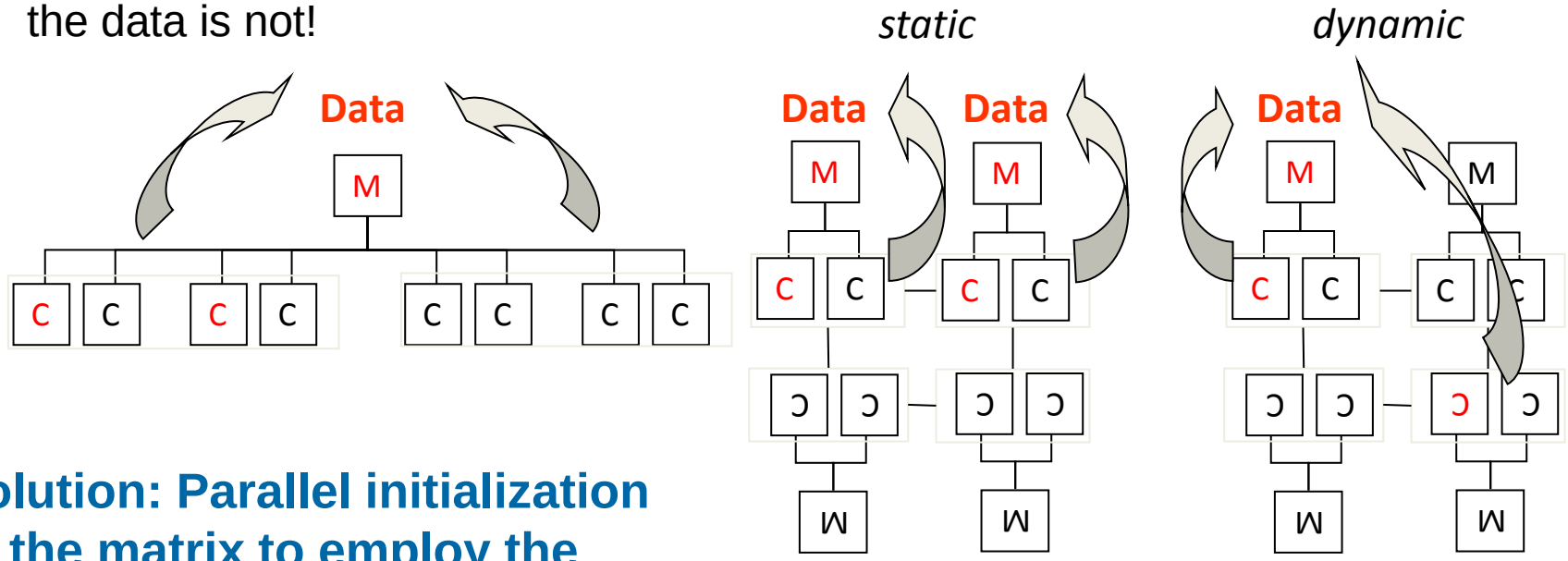
# Performance with Dynamic Row Distribution (2)

- compact: threads packed tightly; scatter: threads distributed



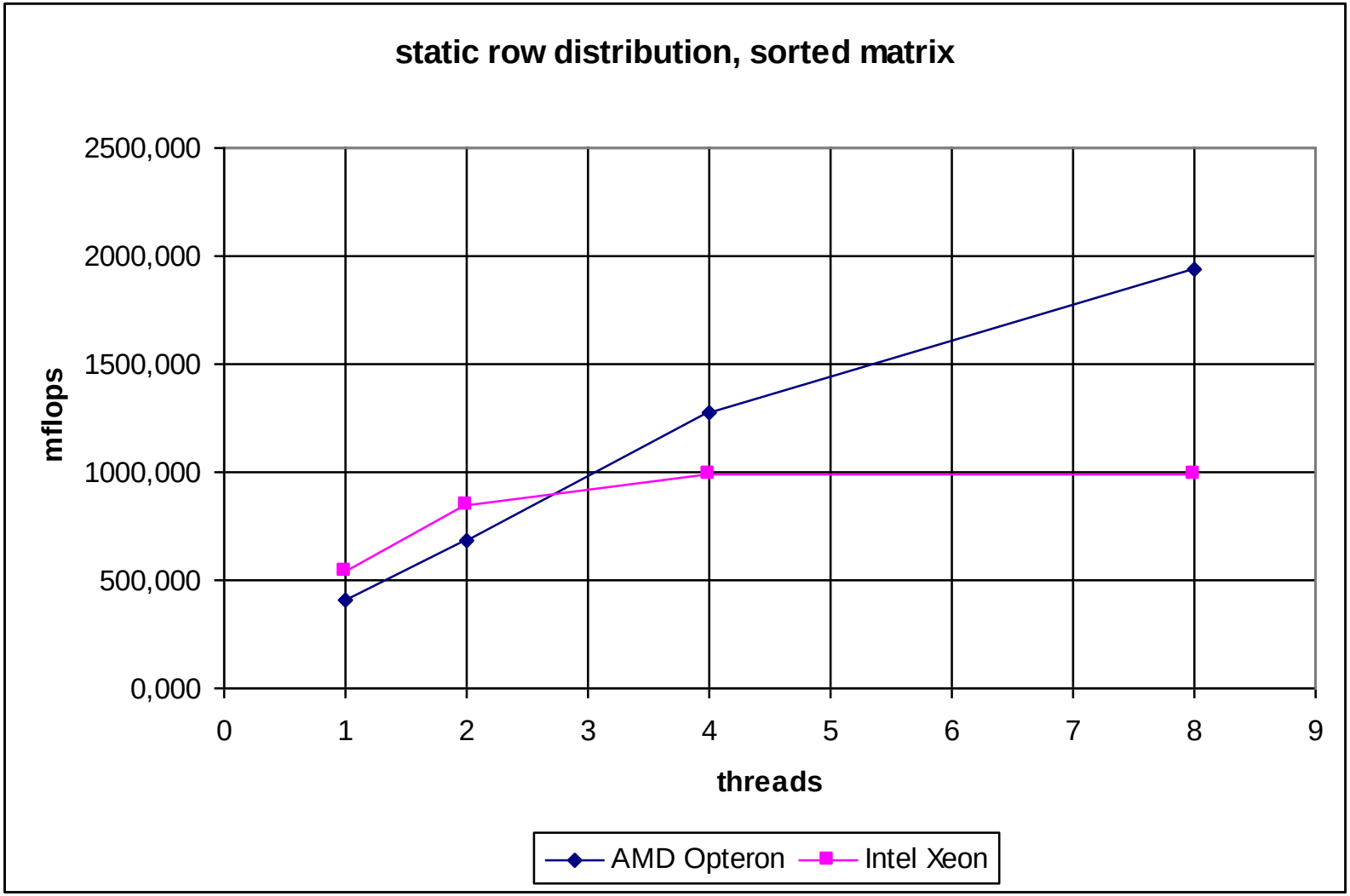
- **Why does the Xeon deliver better performance with the Dynamic row distribution, but the Operon gets worse?**

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



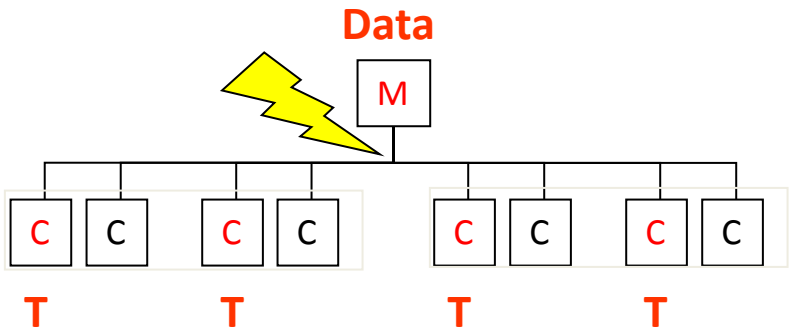

→ **Solution: Parallel initialization of the matrix to employ the first-touch mechanism of Operating Systems: Data is placed near to where it is first access from.**

# Performance with Pre-Calculated Row Distribution (3)



# 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:




- If the matrix would be smaller and fit into the cache the result would look different…

# The End

**Thank you for your attention.**