Parallel architectures in detail

Shared memory architectures
Multicore issues
ccNUMA-aware programming
Network topologies

J. Eitzinger (RRZE)
G. Hager (RRZE)
R. Bader (LRZ)
Anatomy of a cluster system

*Strongly hierarchical topology!*

---

**Core**

- Chip/socket
- Shared-memory compute node
- Distributed-memory cluster

---

(C) 2019 RRZE, LRZ

Parallel architectures
Before we start:

Taxonomy of memory access in parallel computers

- **NORMA** ("No Remote Memory Access")
  - Access to memory of other processor only via explicit message passing; software handshake required
  - This does not rule out remote DMA (data transfer without processor intervention)
  - MPP systems like IBM Blue Gene, Clusters beyond node boundary

- **NUMA** ("Non-Uniform Memory Access")
  - Access to memory of other processor via hardware mechanisms. Usage of special libraries, highly tuned MPI implementation without SW handshake
  - Cray X2 and Cray XT/XE

- **ccNUMA** ("cache-coherent NUMA")
  - SGI Altix (NUMAlink), AMD Opteron (HyperTransport), Intel Core i7(++) (QuickPath)
  - May be complex to optimize bandwidth-bound programs (see later)

- **UMA** ("Uniform Memory Access")
  - Flat memory, uniform access from all CPUs
  - Multi-core chips, NEC SX
  - Cache coherent despite the absence of "cc" in the name
Shared Memory architectures
Shared Memory Architectures

- Suitable for **medium-grained parallelism**
- Available in a variety of flavours ranging from off-the-shelf PCs to large-scale supercomputers
- But: **significant differences in implementations!**
  - Shared memory means shared bandwidth. Or does it not?
  - Hardware support for parallelization is a luxury, but it pays
  - Compilers have to support parallelization directives (GNU suite does not)
  - Shared caches degrade performance
- Varying OS kernel support for shared memory architectures
Shared Memory Architectures: UMA – memory bus

- **Architecture: switch/bus arbitrates memory access**
  - Special protocol ensures cross-CPU cache data consistency

![Diagram of UMA architecture](image)

- **CPU 1**
  - Cache
- **CPU 2**
  - Cache
- **CPU 3**
  - Cache
- **CPU 4**
  - Cache
- **Switch/Bus**
- **Memory**
Shared Memory Architectures:
UMA – memory bus

- **Worst case:** bus system provides single bandwidth to multiple processors
  - Only one CPU at a time can use the bus and access memory at any one time
    - No need to provide for faster memory
  - Collisions occur frequently, causing one or more CPUs to wait for “bus ready” (contention)
  - Even worse: Shared memory bus in current multi-core chips further reduce available bandwidth/peak FLOP balance!
  - Still, price/performance ratio is good for certain applications
Shared Memory Architectures: UMA - Memory Crossbar Switch

- **Best case:** *crossbar switch* provides separate data path to memory for each CPU
  - Can saturate full memory bandwidth of every CPU concurrently
  - Bus contention occurs only if same memory module is accessed by more than one CPU
  - Memory interleaving is a must
  - Example: NEC SX-9 node
    - 16 CPUs, 256 GB/sec each, 4096 GB/sec node memory bandwidth

- **Reality:** *Compromises are made*
  - E.g. 2 of 4 CPUs can use full bandwidth concurrently
  - May be advantageous not to use all CPUs on a node
  - Example: Altix 4700 (HLRBII) Compute Blade
    - 2 sockets (4 cores), 8.5 GB/sec each, 8.5 GB/sec node bandwidth
Shared Memory Architectures: UMA Node Layout

- **Examples:**
  - Single socket nodes, e.g., RRZE Woody cluster (4-core “desktop” chips)
  - NEC vector systems
  - Your dual-core laptop computer, your quad-core smartphone

- **Advantages**
  - Cache Coherence (see below) is "easy" to implement
  - Easy to optimize memory access

- **Disadvantages**
  - Memory bandwidth and price (!) often limit scalability (2 – 8 cores per UMA node)
Shared Memory Architectures: ccNUMA Node Layout

- **ccNUMA**:  
  - Single address space although physically distributed memory through proprietary hardware concepts (e.g. NUMALink in SGI systems)

- **Advantages**:  
  - Memory bandwidth: scalable  
  - Systems with up to 1024 CPUs are available

- **Disadvantages**:  
  - Cache Coherence hard to implement / expensive  
  - Performance depends on access to local or remote memory

- **Standard in HPC nodes today**  
  - Intel Haswell++: ccNUMA on the socket (optional – “CoD” mode)
Shared memory computers: Some realistic examples

- Dual CPU Intel Xeon node

- Dual Intel “Core2” QC node

- Dual Intel Haswell/Skylake node
A modern shared memory node

- AMD “Epyc” dual-socket system
- 24 cores per socket
- 6 cores per “Zeppelin die”, 4 dies per socket
- Shared L3 cache for core triplets
  - Two “CCX” per Zeppelin die
- AMD “Infinity Fabric” between dies
- Eight ccNUMA domains
- Two DDR4 memory channels per ccNUMA domain
Shared memory computers
Cache coherence

- Data in cache is only a copy of data in memory
  - Multiple copies of same data on multiprocessor systems
  - Cache coherence protocol/hardware ensure consistent data view
  - Without cache coherence, shared cache lines can become clobbered:

```
P1
  C1
    A1, A2

P2
  C2
    A1, A2

Bus

Memory

P1
  Load A1
  Write A1=0

P2
  Load A2
  Write A2=0

Write-back to memory leads to incoherent data

A1, A2  A1, A2  A1, A2

C1 & C2 entry can not be merged to:

A1, A2```
Cache coherence protocol must keep track of cache line (CL) status

- P1
  - Load A1
  - Write A1 = 0:
    1. Request exclusive access to CL
    2. Invalidate CL in C2
    3. Modify A1 in C1

- P2
  - Load A2
  - Write A2 = 0:
    1. Request exclusive CL access
    2. CL write back + Invalidate
    3. Load CL to C2
    4. Modify A2 in C2

C2 is exclusive owner of CL
Cache coherence can cause substantial overhead
- may reduce available bandwidth
- “False sharing” when multiple cores modify same CL frequently

Different implementations
- **Snoop**: On modifying a CL, a CPU must broadcast its address to the whole system
- **Directory, “snoop filter”**: Chipset (“network”) keeps track of which CLs are where and filters coherence traffic

Directory-based ccNUMA can reduce pain of additional coherence traffic

But always take care:

Multiple processors should never write frequently to the same cache line ("false sharing")!
Multi-core issues

Shared vs. separate caches
Case study: OpenMP vector triad
Shared vs. separate caches

- **Shared cache**
  - Fast communication path between cores
    - Can also reduce synchronization overhead
  - Less coherence overhead between cores connected to the same cache
  - More cache for sequential applications
  - Shared bandwidth → potential bottleneck

- **Separate caches**
  - No cache bandwidth bottleneck
  - More overhead for cache coherence
  - Single-threaded workloads leave a lot of cache unused

Dominant today
The vector triad benchmark
A “swiss army knife” for microbenchmarking

Simple streaming benchmark:

double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

do j=1,NITER
    do i=1,N
        A(i) = B(i) + C(i) * D(i)
    enddo
    if (.something.that.is.never.true.) then
        call dummy(A,B,C,D)
    endif
enddo

- Report performance for different N
- Choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all current architectures!
\( A(:) = B(:) + C(:) \times D(:) \) on one Sandy Bridge core (3 GHz)

What about multiple cores?
Do the bandwidths scale?

L1D cache (32k)
L2 cache (256k)
L3 cache (20M)
Memory

Parallel architectures
Every core runs its own, independent triad benchmark

double precision, dimension(:,), allocatable :: A,B,C,D

!$OMP PARALLEL private(i,j,A,B,C,D)
allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
endo
!$OMP END PARALLEL

→ pure hardware probing, no impact from OpenMP overhead
Throughput vector triad on Sandy Bridge socket (3 GHz)

Saturation effect in memory

Scalable BW in L1, L2, L3 cache
The OpenMP-parallel vector triad benchmark

OpenMP work sharing in the benchmark loop

double precision, dimension(:), allocatable :: A,B,C,D

allocate(A(1:N),B(1:N),C(1:N),D(1:N))

A=1.d0; B=A; C=A; D=A

!$OMP PARALLEL private(i,j)
do j=1,NITER
 !$OMP DO
do i=1,N
   A(i) = B(i) + C(i) * D(i)
endo
 !$OMP END DO
if(.something.that.is.never.true.) then
 call dummy(A,B,C,D)
endif
endo

!$OMP END PARALLEL
OpenMP vector triad on Sandy Bridge socket (3 GHz)

- Sync overhead grows with the number of threads.
- L1 core limit.
- Bandwidth scalability across memory interfaces.
OpenMP synchronization (barrier) overhead
Welcome to the multi-/many-core era
Synchronization of threads may be expensive!

Threads are synchronized at **explicit** AND **implicit** barriers. These are a main source of overhead in OpenMP programs.

On x86 systems there is no hardware support for synchronization!

• Next slide: Test **OpenMP** Barrier performance…
• for different compilers
• and different topologies:
  ➢ shared cache
  ➢ shared socket
  ➢ between sockets
• and different thread counts
  ➢ 2 threads
  ➢ full domain (chip, socket, node)
### Thread synchronization overhead on IvyBridge-EP

**Barrier overhead in CPU cycles**

<table>
<thead>
<tr>
<th></th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 Threads</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared L3</td>
<td>599</td>
<td>425</td>
</tr>
<tr>
<td>SMT threads</td>
<td>612</td>
<td>423</td>
</tr>
<tr>
<td>Other socket</td>
<td>1486</td>
<td>1067</td>
</tr>
</tbody>
</table>

- **Strong topology dependence!**

<table>
<thead>
<tr>
<th></th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full domain</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Socket (10 cores)</td>
<td>1934</td>
<td>1301</td>
</tr>
<tr>
<td>Node (20 cores)</td>
<td>4999</td>
<td>7783</td>
</tr>
<tr>
<td>Node + SMT</td>
<td>5981</td>
<td>9897</td>
</tr>
</tbody>
</table>

- **Strong dependence on compiler, CPU and system environment!**
- **OMP_WAIT_POLICY=ACTIVE** can make a big difference
ccNUMA and its impact on high performance programming
Memory Locality Problems

- ccNUMA:
  - whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)

- How do we make sure that memory access is always as "local" and "distributed" as possible?

  - Page placement is implemented in units of OS pages (4 KiB, today often 2 MiB)
Bandwidth map created with likwid-bench. All cores used in one NUMA domain, memory is placed in a different NUMA domain.

Test case: simple copy \( A(:) = B(:) \), large arrays.
A more complicated system

- Example: AMD “Epyc” 2-socket system (8 chips, 2 sockets, 48 cores)
  - BW in Gbyte/s

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU node</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM node</td>
<td>0</td>
<td>32.4</td>
<td>21.4</td>
<td>21.8</td>
<td>21.9</td>
<td>10.6</td>
<td>10.6</td>
<td>10.7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>21.5</td>
<td>32.4</td>
<td>21.9</td>
<td>21.9</td>
<td>10.6</td>
<td>10.5</td>
<td>10.7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>21.8</td>
<td>21.9</td>
<td>32.4</td>
<td>21.5</td>
<td>10.6</td>
<td>10.6</td>
<td>10.8</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>21.9</td>
<td>21.9</td>
<td>21.5</td>
<td>32.4</td>
<td>10.6</td>
<td>10.6</td>
<td>10.6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10.6</td>
<td>10.7</td>
<td>10.6</td>
<td>10.6</td>
<td>32.4</td>
<td>21.4</td>
<td>21.9</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>10.6</td>
<td>10.6</td>
<td>10.6</td>
<td>10.6</td>
<td>21.4</td>
<td>32.4</td>
<td>21.9</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>10.6</td>
<td>10.7</td>
<td>10.6</td>
<td>10.6</td>
<td>21.9</td>
<td>21.9</td>
<td>32.3</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>10.7</td>
<td>10.6</td>
<td>10.6</td>
<td>10.6</td>
<td>21.9</td>
<td>21.9</td>
<td>21.4</td>
</tr>
</tbody>
</table>
"Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that first touches it!

- Except if there is not enough local memory available
- this might be a problem, see later

Caveat: "touch" means "write", not "allocate"

Example:

```c
double *huge = (double*)malloc(N*sizeof(double));
// memory not mapped yet
for(i=0; i<N; i++) // or i+=PAGE_SIZE
    huge[i] = 0.0; // mapping takes place here!
```

It is sufficient to touch a single item to map the entire page
Memory Locality Problems

- **Locality of reference** is key to scalable performance on ccNUMA
  - Less of a problem with distributed memory (MPI) programming, but see below

- **What factors can destroy locality?**

- **MPI programming:**
  - processes lose their association with the CPU the mapping took place on originally
  - OS kernel tries to maintain strong affinity, but sometimes fails

- **Shared Memory Programming** (OpenMP,...):
  - threads losing association with the CPU the mapping took place on originally
  - improper initialization of distributed data

- **All cases:**
  - Other agents (e.g., OS kernel) may fill memory with data that prevents optimal placement of user data
Programming for ccNUMA
Coding for Data Locality

- Up to now, all technical actions were devoted to keeping threads/processes where they are
  - usually sufficient with MPI programs

- In OpenMP the programmer must ensure that memory pages get mapped locally in the first place (and then prevent migration)
  - rigorously apply the "Golden Rule":
    
    **A memory page gets mapped into the local memory of the processor that first touches it!**

  - i.e. we have to take a closer look at initialization code
  - some non-locality at domain boundaries may be unavoidable
Simplest case: explicit initialization

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

A = 0.d0

!$OMP parallel do
do i = 1, N
   B(i) = function ( A(i) )
end do
!$OMP end parallel do

!$OMP parallel do
!$OMP do schedule(static)
do i = 1, N
   A(i) = 0.d0
end do
!$OMP end do
!$OMP end parallel
```

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
   A(i) = 0.d0
end do
!$OMP end do
...

!$OMP do schedule(static)
do i = 1, N
   B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```
Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so "localize" arrays before I/O

```fortran
integer, parameter :: N=10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
   A(i)=0.d0
end do
!$OMP end do
!$OMP single
READ(1000) A
!$OMP end single
!$OMP end parallel
```

```fortran
integer, parameter :: N=10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
   B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so "localize" arrays before I/O.
Coding for Data Locality

- Required condition: OpenMP loop schedule of initialization must be the same as in all computational loops
  - best choice: **static**! Specify explicitly on all NUMA-sensitive loops, just to be sure…
  - imposes some constraints on possible optimizations (e.g. load balancing) → see exercises

- **How about global objects?**
  - better not use them
  - if communication vs. computation is favorable, might consider properly placed copies of global data

- In C++, **STL allocators** provide an elegant solution
Coding for Data Locality:
Placement of static arrays or arrays of objects

- Speaking of C++: Don't forget that constructors tend to touch the data members of an object. Example:

```cpp
class D {
    double d;
public:
    D(double _d=0.0) throw() : d(_d) {}
    inline D operator+(const D& o) throw() {
        return D(d+o.d);
    }
    inline D operator*(const D& o) throw() {
        return D(d*o.d);
    }
    ...}
};

→ placement problem with
D* array = new D[1000000];
```
Solution: Provide overloaded `D::operator new[]`

```c++
void* D::operator new[](size_t n) {
    char *p = new char[n]; // allocate

    size_t i, j;
    #pragma omp parallel for private(j) schedule("")
    for (i = 0; i < n; i += sizeof(D))
        for (j = 0; j < sizeof(D); ++j)
            p[i + j] = 0;
    return p;
}

void D::operator delete[](void* p) throw() {
    delete [] static_cast<char*>(p);
}
```

- Placement of objects is then done automatically by the C++ runtime via "placement new"
template <class T> class NUMA_Allocator {
public:
    T* allocate(size_type numObjects, const void *localityHint=0) {
        size_type ofs,len = numObjects * sizeof(T);
        void *m = malloc(len);
        char *p = static_cast<char*>(m);
        int i,pages = len >> PAGEBITS;
        #pragma omp parallel for schedule(static) private(ofs)
        for(i=0; i<pages; ++i) {
            ofs = static_cast<size_t>(i) << PAGEBITS;
            p[ofs]=0;
        }
        return static_cast<pointer>(m);
    }
...  
};

Application:
vector<double,NUMA_Allocator<double> > x(1000000)
Diagnosing Bad Locality

- If your code is cache-bound, you might not notice any locality problems
- Otherwise, bad locality **limits scalability at very low CPU numbers** (whenever a node boundary is crossed)
  - if the code makes good use of the memory interface
  - but there may also be a general problem in your code...

- Consider using **hardware performance counters**
  - likwid-perfctr
    Example for Intel Nehalem (Core i7):

    ```bash
    likwid-perfctr -g MEM -C N:0-7 ./a.out
    ```

  - hpcutils
  - Intel Amplifier (a.k.a. VTune)
  - ...
How about libraries?

- Libraries like MKL/SCSL often use local work storage to make sure data is local (and also well aligned to caches and bank-structure of caches).
- This applies to those routines parallelized:
  - BLAS3, FFTs, LAPACK calling BLAS-routines.
- Make sure you understand and provide sufficient work storage when required.

- Libraries are a problem in general:
  - No documentation about thread scheduling within library.
  - Still the user has to take care of NUMA placement of arrays.
    - Example: BLAS DGEMV.
The curse and blessing of interleaved placement: OpenMP STREAM triad on 4-socket (48 core) Magny Cours node

- **Parallel init**: Correct parallel initialization
- **LD0**: Force data into LD0 via `numactl -m 0`
- **Interleaved**: `numactl --interleave <LD range>`

---

![Bandwidth Chart](chart.png)

- **Bandwidth [Mbyte/s]**
- **# NUMA domains (6 threads per domain)**

(C) 2019 RRZE,LRZ
Summary of parallel architectures

- **Bottlenecks** do exist from the chip level up to the highly parallel machine

- **Node topology** is the first issue to be aware of
  - Affinity matters
  - Bandwidth bottlenecks, synchronization overhead

- **ccNUMA** is here to stay
  - Non-local access & contention
  - Observe the “Golden Rule” of first-touch page placement
  - Simple first check for ccNUMA problems: Run code with interleaved placement
  - Modern Linux kernels and distros: NUMA balancing (i.e., automatic page migration) may let you get away without even knowing…
Network Topologies
Network topologies: Bisection bandwidth

- Network bisection bandwidth $B_b$ is a general metric for the data transfer “capability” of a system
- Sum of the bandwidths of all connections cut when splitting the system into two equal parts
- More meaningful metric when comparing systems: Bisection BW per core or per node, $B_b/N$

- Bisection BW depends on
  - Bandwidth per link
  - Network topology
Network topologies:  
Bus

- Bus can be used by one connection at a time
- Bandwidth is shared among all devices
- Bisection BW is constant:  
  \[ \frac{B_b}{N} \sim \frac{1}{N} \]
- Collision detection, bus arbitration protocols must be in place
- Examples: PCI bus, memory bus of multi-core chips, diagnostic buses

**Advantages**
- Low latency
- Easy to implement

**Disadvantages**
- Shared bandwidth, not scalable
- Problems with failure resiliency (one defective agent may block bus)
- Fast buses for large N require large signal power
A non-blocking crossbar can mediate a number of connections between a group of input and a group of output elements.

This can be used as a 4-port non-blocking switch (fold at the secondary diagonal).

Switches can be cascaded to form hierarchies (common case).

Crossbars can also be used directly as interconnects in computer systems:
- Example: Scalable UMA memory access – see later
- (Historic) example: Hitachi SR8000
“Fat tree” switch hierarchies

- **“Fully non-blocking”**
  - N/2 end-to-end connections with full bandwidth
  - \( B_b/N = \text{const.} = B/2 \)
  - Sounds good, but see next slide

- **“Oversubscribed”**
  - Spine does not support N/2 full BW end-to-end connections
  - \( B_b/N = \text{const.} = B/2k \), where k is the oversubscription factor
  - Intelligent resource management is crucial
Interconnect Examples

- Ethernet
  - 1 Gbit/s and 10 Gbit/s variants
  - < 40% of all Top500 entries (November 2012) use Ethernet (mostly 1 Gbit)

- InfiniBand
  - Dominant high-performance “commodity” interconnect
  - 45% of all Top500 entries (November 2012)
    - DDR: 20 Gbit/s per link and direction
    - QDR: 40 Gbit/s per link and direction (used in RRZE’s Lima & Emmy)

- Omni-Path
  - Intel product
  - 100 Gbit/s per link and direction (used in RRZE’s Meggie cluster)

- Custom Interconnects
  - E.g. Cray, IBM BlueGene, Fujitsu K Computer
Meshes

- Fat trees can become prohibitively expensive in large systems
- Compromise: **Meshes**
  - n-dimensional Hypercubes
  - **Toruses** (2D / 3D)
  - Many others (including hybrids)
- Each node is a “router”
- Direct connections only between direct neighbors
- This is not a non-blocking crossbar!
  - Intelligent resource management and routing algorithms are essential
- **Toruses** are used in very large systems: **IBM Blue Gene**
  - \( B_b \sim N^{(d-1)/d} \quad \Rightarrow \quad B_b/N \to 0 \) for large \( N \)
  - Sounds bad, but those machines show good scaling for many codes
  - Well-defined and predictable bandwidth behavior!
Meshes

- **Advantages of toroidal/cubic meshes**
  - Limited cabling required
  - Cables can be kept short

- **Meshes can come in all shapes and sizes**
  - Driven by current technology and marketing …