VSC Seminar

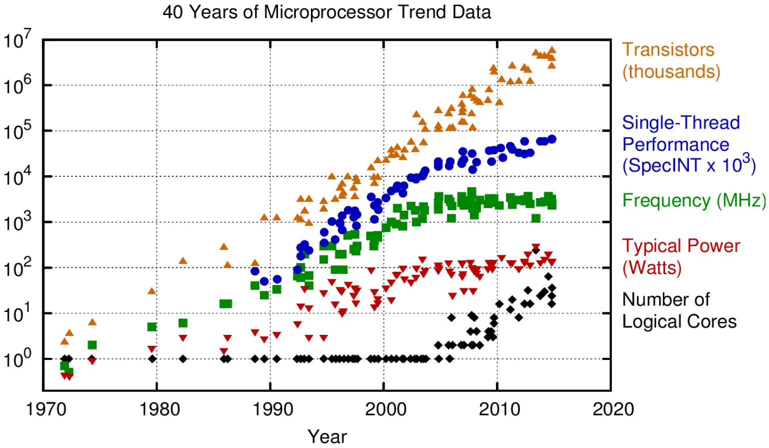
Modern Multi-Core Architectures for Supercomputing

Josef Weinbub

December 11, 2015

Christian Doppler Laboratory for High Performance TCAD
Institute for Microelectronics, TU Wien
www.iue.tuwien.ac.at/hptcad/

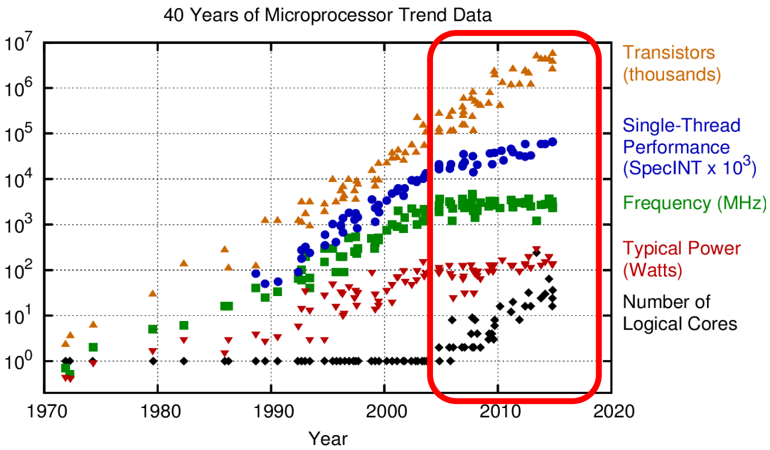
Why Parallel Computing?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

© Karl Rupp / www.karlrupp.net

Why Parallel Computing?



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

© Karl Rupp / www.karlrupp.net

Outline

Part A - Hardware

Modern Processors

Parallel Computers

Part B - Software

Shared-Memory Parallel Programming

Vectorization

Distributed and Hybrid Parallel Programming

Summary

Outline

Part A - Hardware

Modern Processors

Parallel Computers

Part B - Software

Shared-Memory Parallel Programming

Vectorization

Distributed and Hybrid Parallel Programming

Summary

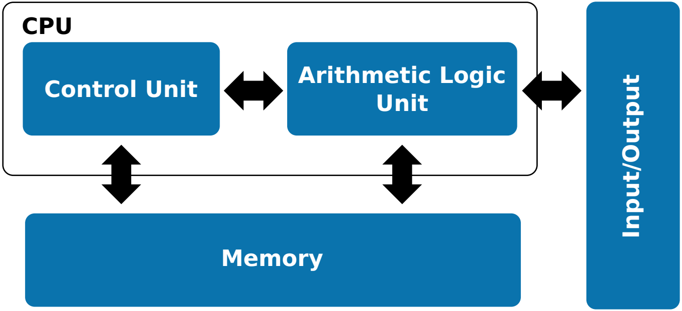
Part A - Hardware

- Modern Processors
  - Stored Program Computer Architecture
  - Memory Hierarchies:
    Memory Gap, Caches, Prefetch
  - Pipelining
  - Multi-Core Processors
  - Multi-Threaded Processors
- Parallel Computers
  - Basics
  - Shared-Memory Computers:
    Cache Coherence, UMA, ccNUMA
  - Distributed-Memory Computers
  - Hierarchical (Hybrid) Systems
  - Networks

Part A - Hardware

- **Modern Processors**
  - **Stored Program Computer Architecture**
  - **Memory Hierarchies:**
    Memory Gap, Caches, Prefetch
  - **Pipelining**
  - **Multi-Core Processors**
  - **Multi-Threaded Processors**
- **Parallel Computers**
  - **Basics**
  - **Shared-Memory Computers:**
    Cache Coherence, UMA, ccNUMA
  - **Distributed-Memory Computers**
  - **Hierarchical (Hybrid) Systems**
  - **Networks**

Stored Program Computer Architecture



- Program feeds control unit
- Stored in memory together with data
- Data required by arithmetic unit

Stored Program Computer Architecture

Two Big Issues

- Memory bottleneck
  - Instructions and data must be fed to control and arithmetic unit
  - Speed of memory interface poses limitation to compute performance
  - Commonly known as von Neumann bottleneck
- Inherently sequential architecture
  - A single instruction with a single/group of operands from memory
  - Single instruction single data (SISD)

However ..

- No widespread alternative within reach
- So, we better make the best out of it

Stored Program Computer Architecture

Two Big Issues

- Memory bottleneck
  - Instructions and data must be fed to control and arithmetic unit
  - Speed of memory interface poses limitation to compute performance
  - Commonly known as von Neumann bottleneck
- Inherently sequential architecture
  - A single instruction with a single/group of operands from memory
  - Single instruction single data (SISD)

However ..

- No widespread alternative within reach
- So, we better make the best out of it

Stored Program Computer Architecture

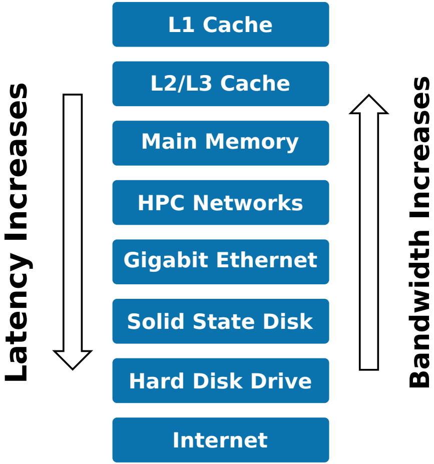
Two Big Issues

- Memory bottleneck
  - Instructions and data must be fed to control and arithmetic unit
  - Speed of memory interface poses limitation to compute performance
  - Commonly known as von Neumann bottleneck
- Inherently sequential architecture
  - A single instruction with a single/group of operands from memory
  - Single instruction single data (SISD)

However ..

- No widespread alternative within reach
- So, we better make the best out of it

Memory Hierarchies



Latency Increases

L1 Cache

L2/L3 Cache

Main Memory

HPC Networks

Gigabit Ethernet

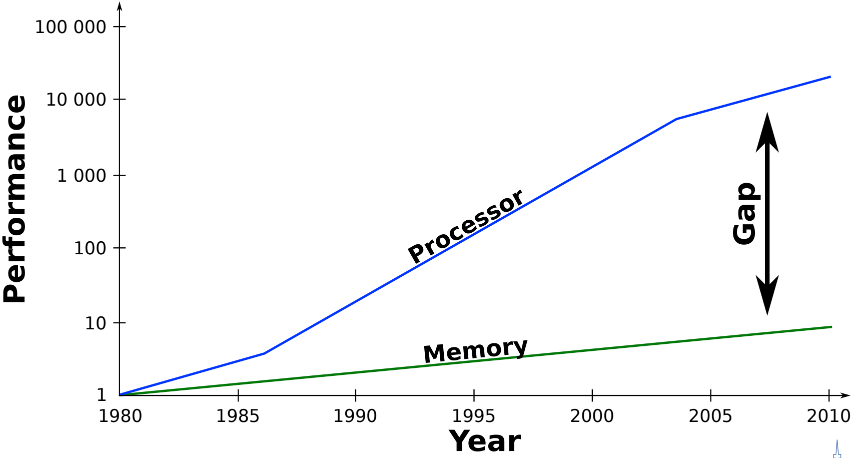
Solid State Disk

Hard Disk Drive

Internet

Bandwidth Increases

Memory Hierarchies - Memory Gap



Memory Hierarchies - Caches

- **Low-capacity, high-speed memory**
- **Integrated on CPU-die**
- **Replicate data lines in main memory**
- **Required to buffer slow main memory access**
- **GFlops/s per core vs memory bandwidth GBytes/s**
- **Insufficient to continuously feed arithmetic units**
- **Additional problem: Memory latency ($\sim$100 cycles)**
- **Caches can alleviate effects of memory gap**

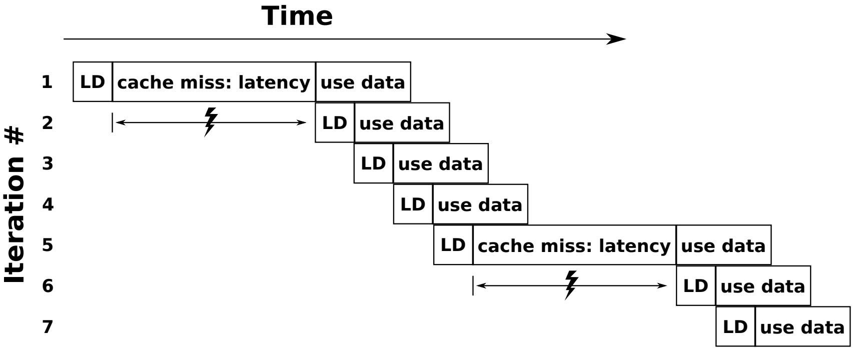
Memory Hierarchies - Caches

- **Usually two-three cache levels: L1-L3**
- **L1 is normally split into two parts**
  - **L1I: for instructions**
  - **L1D: for data**
- **Outer cache levels are typically unified**
- **Data load request into register:**
  - **Is available in cache: cache hit**
  - **Is not available in cache: cache miss**
- **Cache miss: data fetch from outer cache levels or in worst case from main memory**
- **No cache space available: algorithm evicts old data**

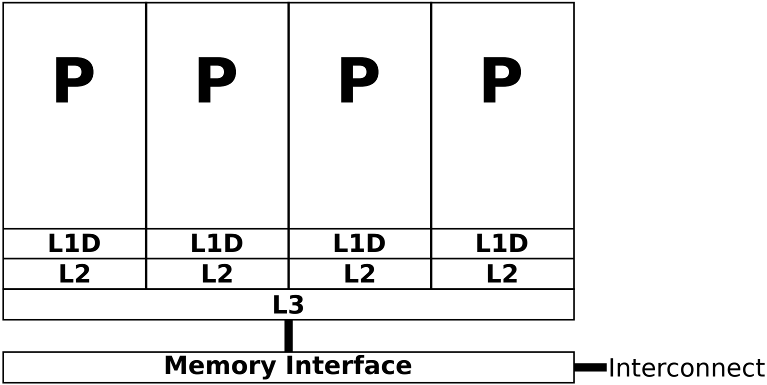
Memory Hierarchies - Caches

- Usually two-three cache levels: L1-L3
- L1 is normally split into two parts
  - L1I: for instructions
  - L1D: for data
- Outer cache levels are typically unified
- Data load request into register:
  - Is available in cache: cache hit
  - Is not available in cache: cache miss
- Cache miss: data fetch from outer cache levels or in worst case from main memory
- No cache space available: algorithm evicts old data

Memory Hierarchies - Caches



Memory Hierarchies - Caches



- **Shared caches in a multi-core setting**
- **Intel Nehalem**

Memory Hierarchies - Prefetch

- Provide caches with data ahead of time
- Idea is to hide latency
- Software prefetching by compilers
- Interleaving special instructions with software pipelined instruction stream
- ”Touch” cache lines in advance
- Requires asynchroneous memory operations
- Hardware prefetcher identifies patterns to *read ahead*
- Hardware and software prefetchers are used in today’s processors

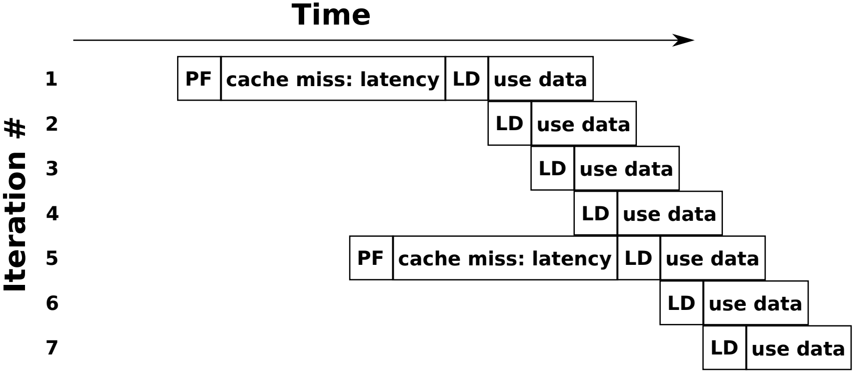
Memory Hierarchies - Prefetch

- Both techniques can only *do so much*
- Significant burden on memory subsystem
- Support of certain number of *outstanding prefetch operations* (i.e. pending prefetch requests)
- One prefetch for each cache line transfer
- Application with many operations on the cache line will require less
- Application with heavy BW demand can overstretch prefetching
- Tip: provide long continuous data streams

Memory Hierarchies - Prefetch

- Both techniques can only *do so much*
- Significant burden on memory subsystem
- Support of certain number of *outstanding prefetch operations* (i.e. pending prefetch requests)
- One prefetch for each cache line transfer
- Application with many operations on the cache line will require less
- Application with heavy BW demand can overstretch prefetching
- Tip: provide long continuous data streams

Memory Hierarchies - Prefetch



Time

Iteration #

1 PF | cache miss: latency | LD | use data

2 LD | use data

3 LD | use data

4 LD | use data

5 PF | cache miss: latency | LD | use data

6 LD | use data

7 LD | use data

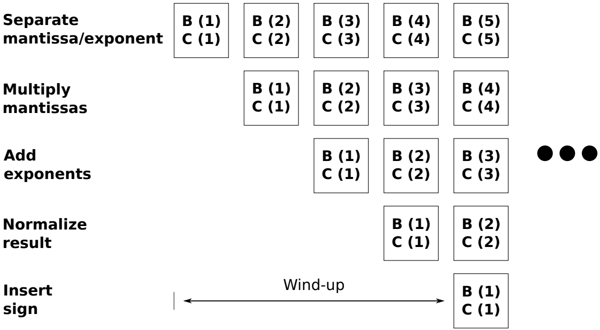
Pipelining

- **Pipelining is similar to an assembly line**
- **”Workers” are specialized on a single task**
- **Forward object to next worker**
- **Goal is to optimize the work among workers**

Pipelining

- Complex operations require more than one cycle
- Apply pipeline complex: ”fetch-decode-execute”
- Each stage can operate independently
- Idea: break down complex tasks into simple ones
- Allows for higher clock rate due to simple tasks

Pipelining



Separate mantissa/exponent	B (1) C (1)	B (2) C (2)	B (3) C (3)	B (4) C (4)	B (5) C (5)	
Multiply mantissas		B (1) C (1)	B (2) C (2)	B (3) C (3)	B (4) C (4)	
Add exponents			B (1) C (1)	B (2) C (2)	B (3) C (3)	●●●
Normalize result				B (1) C (1)	B (2) C (2)	
Insert sign					B (1) C (1)	

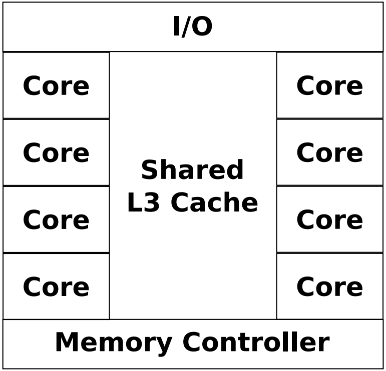
- Timeline for simplified floating-point multiplication
- Executes vector product: A(:)=B(:)*C(:)
- One result per cycle; four-cycle wind-up phase
- Modern pipeline lengths between 10 and 35

Pipelining

Issues

- Inefficient if short and tight loops are used
- Operations with long latencies (e.g. square root)
- Pipeline bubble when low degree of pipelining
- Compiler aims to optimally utilize pipeline
- Stalls if load operation does not deliver data on time for arithmetic operation

Multi-Core Processors



I/O

Core

Core

Core

Core

Shared
L3 Cache

Core

Core

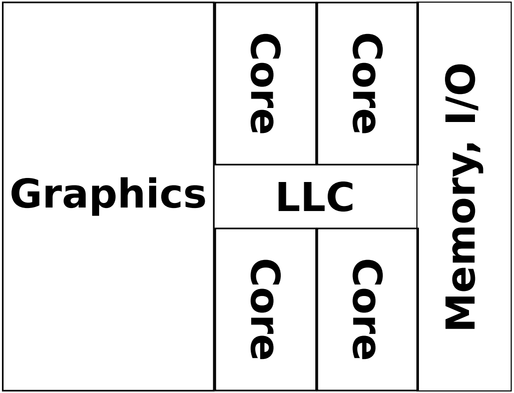
Core

Core

Memory Controller

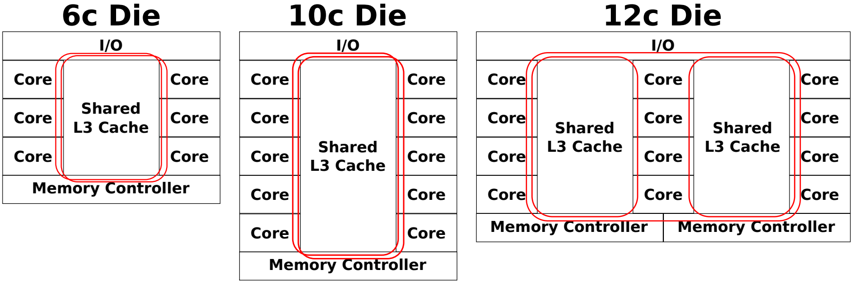
- Intel Haswell E
- Desktop, Core i7-5960X, no graphics

Multi-Core Processors



- Intel Skylake
- Desktop, Core i7-6700K, with graphics

Multi-Core Processors



6c Die

I/O

Core | Shared L3 Cache | Core

Core | Core

Core | Core

Memory Controller

10c Die

I/O

Core | Shared L3 Cache | Core

Core | Core

Core | Core

Core | Core

Core | Core

Memory Controller

12c Die

I/O

Core | Shared L3 Cache | Core | Shared L3 Cache | Core

Core | Core | Core

Core | Core | Core

Core | Core | Core

Memory Controller | Memory Controller

- **Intel Ivy Bridge EP, Configurations**
- **@12c: Note the additional bus, memory controller, and splitted L3 cache**

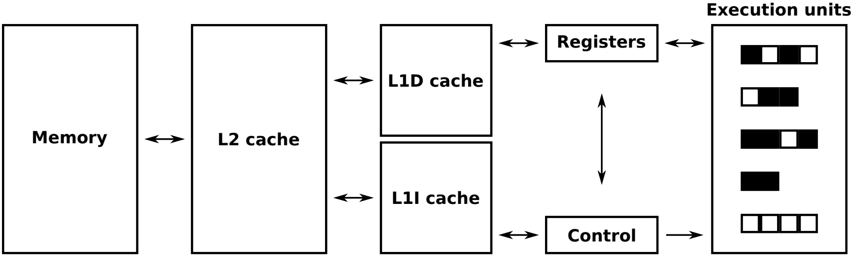
Multi-Threaded Processors

- All modern processors are heavily pipelined
- However, often the pipelines cannot be efficiently used
- Make use of additional cores on modern processors
- Threading capabilities
  - Simultaneous Multithreading (SMT)
  - Hyperthreading (Intel’s version of SMT)

Multi-Threaded Processors

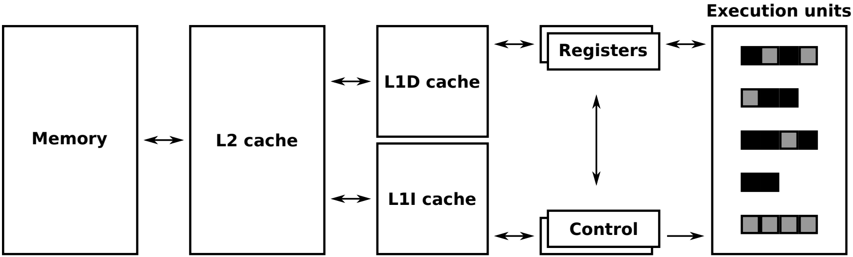
- Main idea: increase number of independent instructions in pipeline
- Architectural state is present multiple times (i.e. registers, stack and instruction pointers)
- Execution units are not multiplied
- CPU appears to be composed of several cores (i.e. logical cores)
- Potential to fill bubbles in pipelines as threads share execution resources

Multi-Threaded Processors



- Pipelined microprocessor without SMT
- White blocks in pipelines denote bubbles

Multi-Threaded Processors



- Pipelined microprocessor with two-way SMT
- Two threads share caches and pipelines but retain respective architectural state

Part A - Hardware

- Modern Processors
  - Stored Program Computer Architecture
  - Memory Hierarchies:
    Memory Gap, Caches, Prefetch
  - Pipelining
  - Multi-Core Processors
  - Multi-Threaded Processors
- **Parallel Computers**
  - **Basics**
  - **Shared-Memory Computers:**
    Cache Coherence, UMA, ccNUMA
  - **Distributed-Memory Computers**
  - **Hierarchical (Hybrid) Systems**
  - **Networks**

Basics

- Parallel computing solves a problem in cooperative way
- Uses several compute elements
- All modern (supercomputer) architectures depend on parallelism
- Also: check your phone - Nexus 6P (8 cores) ..
- Parallel computing importance will continue to rise
- Dominating concepts are:
  - SIMD: Single Instruction, Multiple Data (e.g. GPUs)
  - MIMD: Multiple Instruction, Multiple Data (e.g. Parallel computers)

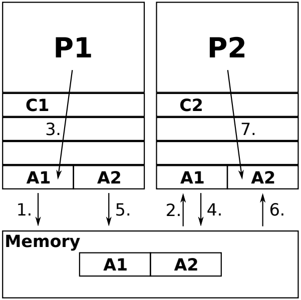
Shared-Memory Computers

- **System with number of CPUs**
- **Work on common, physical address space**
- **Transparent to the programmer, however:**
- **Uniform Memory Access (UMA)**
- **cache-coherent Nonuniform Memory Access (ccNUMA)**
- **Regardless: require cache coherence**

Shared-Memory Computers - Cache Coherence

- **Cache coherence is required in all cache-based multi-processor systems**
- **Copies of cache lines can reside in different caches**
- **Imagine: One cache line gets modified ..**
- **Cache coherence protocols ensure consistent view of memory at all times**
- **Coherence traffic can hurt application performance if the same cache line is frequently modified by different processors (i.e. false sharing)**
- **Implemented in the hardware (CPU or chipsets)**

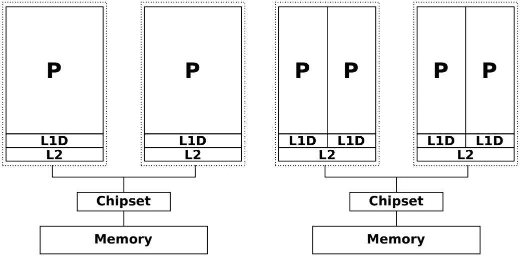
Shared-Memory Computers - Cache Coherence



MESI

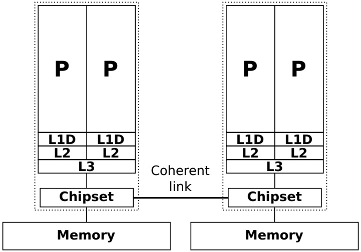
- **M: modified**
- **E: exclusive**
- **S: shared**
- **I: invalid**

Shared-Memory Computers - UMA



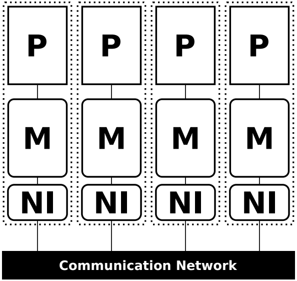
- Both are UMA!
- Latency and bandwidth same for processors/memory
- Aka symmetric multiprocessing (SMP)
- Limited scalability: memory bus contention
- E.g. typical single multi-core processor chips are UMA

Shared-Memory Computers - ccNUMA



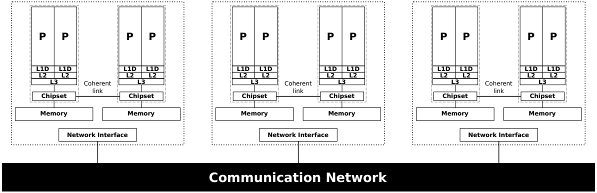
- **Memory is physically distributed but logically shared**
- **Better scalability: Small number of CPUs compete for shared memory bus**
- **Physical layout similar to distributed-memory case**
- **Intel’s UltraPath (UPI)/Qickpath Interconnect (QPI)**
- **E.g. 2- to 4-socket HPC node setup (VSC-2/3 uses 2)**

Distributed-Memory Computers



- No Remote Memory Access (NORMA)
- Today no system available like that anymore
- However, can be seen as programming model
- Communication over the network (e.g. MPI)

Hierarchical (Hybrid) Systems



- Large-scale parallel clusters (VSC-2/3)
- A mix of distributed- and shared-memory type
- Even more anisotropic than multi-core & ccNUMA
- Network an additional layer of communication
- Success due to price vs performance
- Might also contain accelerators/co-processors

Networks

- Networks connect individual compute components
- Primary importance to overall performance
- Bandwidth and latency are of importance
- However, latency effects affect most applications
- Large number of compute components:
  Topology becomes important!
- Simple and cheap: Gigabit Ethernet networks
  ($\sim 110\, MB/s$, $\sim 40 - 50\, \mu s$)
- Professional clusters: Infiniband networks
  ($\sim 1000\, MB/s$, $\sim 4 - 10\, \mu s$)

Networks

- **Buses**
  - **Shared medium**
  - **Used by one communicating device at a time**
  - **E.g. PCI (1-30 GByte/s), CPU-memory (10-50 GByte/s)**
  - **Issues: Blocking (bandwidth)**
- Switched and fat-tree networks
  - Subdivision of communicating devices into groups
  - Fat-tree: fatter links for higher tiers, e.g. VSC-3
  - Issues: Doesn’t scale well (cables, active components)
- Others: mesh networks, torus networks, hybrids
  - Computing elements located at *grid intersections*
  - No direct connections between non-neighbors
  - Requires certain routing circuits to control flow
  - E.g. Cray’s Gemini/Aries (1-50 GByte/s), ccNUMA (40 GByte/s)

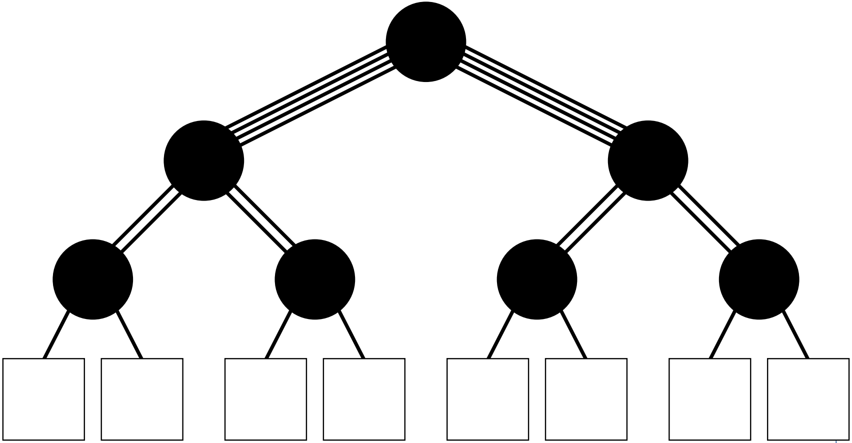
Networks

- **Buses**
  - **Shared medium**
  - **Used by one communicating device at a time**
  - **E.g. PCI (1-30 GByte/s), CPU-memory (10-50 GByte/s)**
  - **Issues: Blocking (bandwidth)**
- **Switched and fat-tree networks**
  - **Subdivision of communicating devices into groups**
  - **Fat-tree: fatter links for higher tiers, e.g. VSC-3**
  - **Issues: Doesn’t scale well (cables, active components)**
- **Others: mesh networks, torus networks, hybrids**
  - **Computing elements located at *grid intersections***
  - **No direct connections between non-neighbors**
  - **Requires certain routing circuits to control flow**
  - **E.g. Cray’s Gemini/Aries (1-50 GByte/s), ccNUMA (40 GByte/s)**

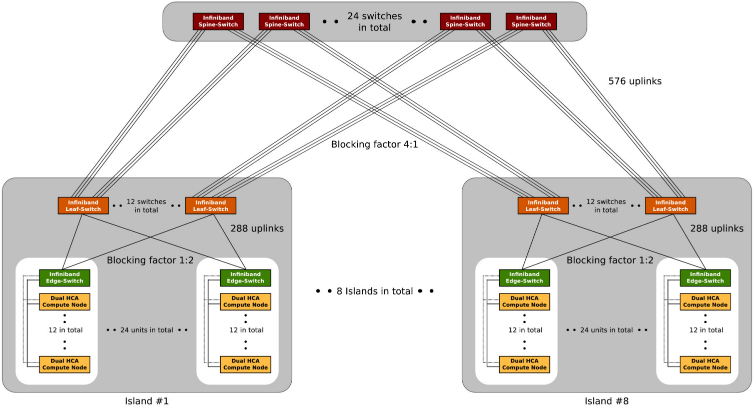
Networks

- **Buses**
  - **Shared medium**
  - **Used by one communicating device at a time**
  - **E.g. PCI (1-30 GByte/s), CPU-memory (10-50 GByte/s)**
  - **Issues: Blocking (bandwidth)**
- **Switched and fat-tree networks**
  - **Subdivision of communicating devices into groups**
  - **Fat-tree: fatter links for higher tiers, e.g. VSC-3**
  - **Issues: Doesn’t scale well (cables, active components)**
- **Others: mesh networks, torus networks, hybrids**
  - **Computing elements located at *grid intersections***
  - **No direct connections between non-neighbors**
  - **Requires certain routing circuits to control flow**
  - **E.g. Cray’s Gemini/Aries (1-50 GByte/s), ccNUMA (40 GByte/s)**

Networks - Fat Tree



Networks - Fat Tree



- **VSC-3's fat tree network structure**

Outline

Part A - Hardware

- Modern Processors
- Parallel Computers

Part B - Software

- Shared-Memory Parallel Programming
- Vectorization
- Distributed and Hybrid Parallel Programming

Summary

Software

- Shared-Memory Parallel Programming
  - Data Locality
    Locality of Access, Placement Pitfalls, C++ Issues
  - OpenMP
    Basics, Efficiency
  - Alternatives
- Vectorization
  - Intrinsics vs Pragmas
  - Automatic Vectorization
- Distributed and Hybrid Parallel Programming
  - Distributed Parallel Programming
  - Hybrid Parallel Programming
    Potential Benefits and Drawbacks

Software

- Shared-Memory Parallel Programming
  - Data Locality
    Locality of Access, Placement Pitfalls, C++ Issues
  - OpenMP
    Basics, Efficiency
  - Alternatives
- Vectorization
  - Intrinsics vs Pragmas
  - Automatic Vectorization
- Distributed and Hybrid Parallel Programming
  - Distributed Parallel Programming
  - Hybrid Parallel Programming
    Potential Benefits and Drawbacks

Shared-Memory Parallel Programming

- Program consists of threads of control with
  - Shared variables
  - Private variables
  - Thread communication via read/write shared data
  - Threads coordinate by synchronizing on shared data
- Threads can be dynamically created and destroyed
- Other programming models:
  - Distributed-memory
  - Hybrid
  - ..

Shared-Memory Parallel Programming

Processes

- Independent execution units
- Own state and own address space
- Interaction with inter-process communication
- A process may contain several threads

Threads

- All threads within a process share address space
- Own state but global and heap data are shared
- Interaction via shared variables

State

- Instruction pointer
- Register file (one per thread)
- Stack pointer (one per thread)

Shared-Memory Parallel Programming

Processes

- Independent execution units
- Own state and own address space
- Interaction with inter-process communication
- A process may contain several threads

Threads

- All threads within a process share address space
- Own state but global and heap data are shared
- Interaction via shared variables

State

- Instruction pointer
- Register file (one per thread)
- Stack pointer (one per thread)

Shared-Memory Parallel Programming

Processes

- Independent execution units
- Own state and own address space
- Interaction with inter-process communication
- A process may contain several threads

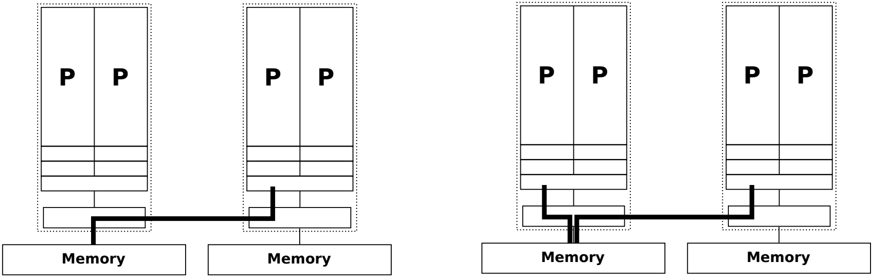
Threads

- All threads within a process share address space
- Own state but global and heap data are shared
- Interaction via shared variables

State

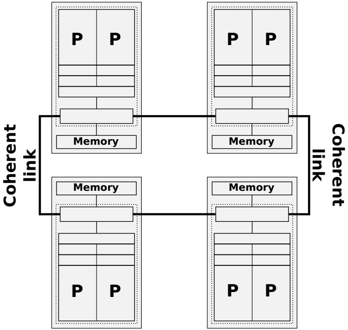
- Instruction pointer
- Register file (one per thread)
- Stack pointer (one per thread)

Data Locality



- Locality problem: Data in nonlocal memory
- Contention problem: Concurrent local&remote access
- Memory-bound code must use proper page placement
- Pinning must be used to ensure locality

Data Locality - Locality of Access



- ccNUMA system with four locality domains
- Zero, one, or two hops required
- Contention problem cannot be eliminated
- No interconnect turns ccNUMA into UMA

Data Locality - Locality of Access

```
double precision, allocatable, dimension(:) :: A, B, C, D
allocate(A(N), B(N), C(N), D(N))
! initialization
do i=1,N
  B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
enddo
...
do j=1,R
  !$OMP PARALLEL DO
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  !$OMP END PARALLEL DO
  call dummy(A,B,C,D)
enddo
```

- **Vector triad code in Fortran**
- **No first touch policy - scaling will be bad ..**
- **.. if data does not fit in the cache!**

Data Locality - Locality of Access

```
! initialization
!$OMP PARALLEL DO
do i=1,N
  B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
enddo
!$OMP END PARALLEL DO
```

- **Better: memory pages reside in respective domains**

Data Locality - Locality of Access

```fortran
integer,parameter:: N=1000000
double precision :: A(N),B(N)




! executed on single LD
READ(1000) A
! contention problem
!$OMP PARALLEL DO
do i = 1, N
  B(i) = func(A(i))
enddo
!$OMP END PARALLEL DO
```

→

```fortran
integer,parameter:: N=1000000
double precision :: A(N),B(N)
!$OMP PARALLEL DO
do i=1,N
  A(i) = 0.d0
enddo
!$OMP END PARALLEL DO
! A is mapped now
READ(1000) A
!$OMP PARALLEL DO
do i = 1, N
  B(i) = func(A(i))
enddo
!$OMP END PARALLEL DO
```

- **Serial is OK, but first first-touch in parallel!**

Data Locality - Locality of Access

- Proper first-touching not always possible
- Think of dynamically partitioned parallel loops:
  Usually used in poorly load-balanced scenarios
- Successive parallel loops:
  Threads should get always same partition
- Cannot avoid global variables?
  Make local first-touched copies

Data Locality - Placement Pitfalls

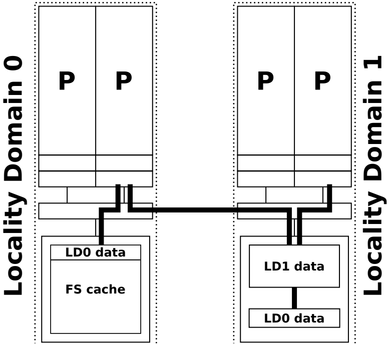
- **ccNUMA offers superior scalability for memory-bound code**
- **UMA easier to handle and no locality handling**
- **However, ccNUMA is/will be the likely scenario**
- **Placement optimizations not always possible**
- **Dynamic loop scheduling**

Data Locality - Placement Pitfalls

```
! initialization
!$OMP PARALLEL DO SCHEDULE(STATIC,512)
do i=1,N
  A(i) = 0; B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
enddo
!$OMP END PARALLEL DO
...
do j=1,R
  !$OMP PARALLEL DO SCHEDULE(RUNTIME)
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  !$OMP END PARALLEL DO
  call dummy(A,B,C,D)
enddo
```

- **Hope for statistically even distribution of access**
- **Vector triad to fathom impact of random access**
- **Round Robin static initialization**

Data Locality - Placement Pitfalls



- Operating systems reserve file I/O caches for reuse
- Might force nonlocal data placement
- FS buffer cache can be a remnant from other job
- Option: sweep memory - allocate all memory

Data Locality - C++ Issues

- C and Fortran straightforward in handling data locality
- C++’s object oriented features offers some problems
  - Arrays of objects
  - Standard Template Library

Data Locality - C++ Issues

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

- **Compare**

  `double * array = new double(n);` with

  `D * array = new D(n);`
- **Constructor initializes immediately**
- **Placement with locality domain which issued `new`**
- **Obvious option (don’t default initialize) not always possible/desirable**

Data Locality - C++ Issues

```
void* D::operator new[](size_t n) {
  char *p = new char[n];
  // allocate
  size_t i,j;
  #pragma omp parallel for private(j) schedule(runtime)
  for(i=0; i<n; i += sizeof(D))
    for(j=0; j<sizeof(D); ++j)
      p[i+j] = 0;
  return p;
}
```

- **Provide overload for `new` operator**
- **Manual parallel first touch**

Data Locality - C++ Issues

```
template <class T> class NUMA_Allocator {
...
// allocate raw memory including page placement
pointer allocate(size_type numObjects, const void *localityHint=0)
  size_type len = numObjects * sizeof(value_type);
  char *p = static_cast<char*>(std::malloc(len));
  if(!omp_in_parallel()) {
    #pragma omp parallel for schedule(runtime) private(ofs)
    for(size_type i=0; i<len; i+=sizeof(value_type)) {
      for(size_type j=0; j<sizeof(value_type); ++j)
        p[i+j]=0;
    }
  }
  return static_cast<pointer>(m);
} ... };
```

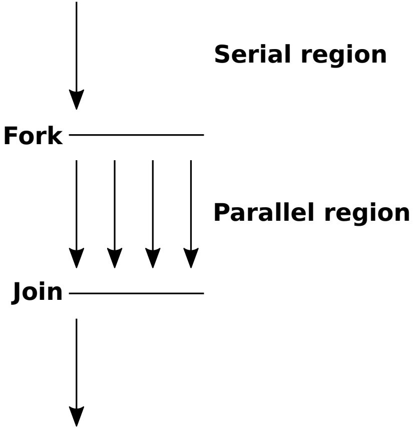
- **Default STL allocator is not NUMA-aware**
- **You can provide your own!**

OpenMP

- **POSIX threads not very convenient for HPC**
- **Majority is loop-centric ..**
- **OpenMP is a directive-based language**
- **Central entity is a thread**
- **Non-OpenMP compiler would simply ignore**
- **Each well-written OpenMP program is a serial program**
- **VSC-3 currently supports**
  OpenMP 3.1 (GNU GCC 4.8.2) and
  OpenMP 4.0 (Intel Compilers 16.0.0)

```
#pragma omp parallel for
for(int i = 0; i < N; i++)
  std::cout << i << std::endl;
```

OpenMP



- **Fork-join model**

OpenMP

- **Parallel regions are indicated with `parallel` pragma**

C/C++

```
#pragma omp parallel
{
  // parallel region
  do_work_package(omp_get_thread_num(), omp_get_num_threads());
}
```

Fortran

```
!$OMP PARALLEL
  ! parallel region
  call do_work_package(omp_get_thread_num(),omp_get_num_threads())
!$OMP END PARALLEL
```

OpenMP

- **Controlling an application's number of threads**

Environment variable

```
$ export OMP_NUM_THREADS=4
$ ./a.out
```

Set during run time

```
omp_set_num_threads(4)
```

OpenMP

Data scoping

- All variables before parallel region are accessible
- Explicitly control data scoping:
  - `private(...)`
  - `shared(...)`
  - `firstprivate(...)`
  - ...
- All variables within parallel region are private

OpenMP

```
#include <omp.h>
main ()  {
  int nthreads, tid;
  /* Fork a team of threads with each thread having a
     private tid variable */
  #pragma omp parallel private(tid)
  {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    /* Only master thread does this */
    if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

OpenMP

Loop Parallelization and Scheduling

- **Loop parallelization is most used mechanism**
- **Different scheduling techniques available**
  - `schedule(type[, chunk])`
  - `static`
  - `dynamic`
  - `guided`
- `collapse`**: nested loops can be collapsed into one large iteration space**
- `nowait`**: threads do not wait at the end of the parallel region**
- **...**

OpenMP

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000
main () {
  int i, chunk;
  float a[N], b[N], c[N];
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
  #pragma omp parallel shared(a,b,c,chunk) private(i)
  {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

- **Can be merged into one scope**

OpenMP

Synchronization

- **Sometimes this is not avoidable**
- **`critical`: only one thread executes the part**
- **`atomic`: faster but only available for few executions**
- **`barrier`: all threads will wait here for each other**
- **Should be avoided at all costs!**

OpenMP

Reductions

- **Expression support is restricted**
- **C++: `x++, ++x, x = x op expr, ..`**

```
double precision :: r,s
double precision, dimension(N) :: a

call RANDOM_SEED()
!$OMP PARALLEL DO PRIVATE(r) REDUCTION(+:s)
do i=1,N
  call RANDOM_NUMBER(r) ! thread safe
  a(i) = a(i) + func(r) ! func() is thread safe
  s = s + a(i) * a(i)
enddo
!$OMP END PARALLEL DO
```

OpenMP

Tasking

- **Not every problem can be loop-wise parallelized**
- **`std::list<..>`: not easily random accessible**
- **OpenMP provides the task mechanism**
- **Allows to process elements of a list in parallel**

```
void increment_list_items(node* head) {
  #pragma omp parallel {
    #pragma omp single {
      for(node* p = head; p; p = p->next) {
       #pragma omp task
       process(p); // p is firstprivate by default
}}}}
```

Thread Pinning

- **Operating system assigns threads to cores**
- **Assignment might change during execution**
- **Assignment might be not optimal**
- **Consider pinning your threads**
- **E.g. Likwid provides, among others, pinning**

www.github.com/RRZE-HPC/likwid

OpenMP - Efficient Programming

Performance Profiling

- Rarely do applications scale nicely
- Tune your parallel implementation
- Start with the most time consuming parts
- But how to identify those?

Intel VTune Amplifier

- Commercial performance profiler with GUI
- Should be used in conjunction with Intel Compiler
- Publicly available if you are an open source dev

http://software.intel.com/en-us/intel-vtune-amplifier-xe

Performance Profiling

- **Rarely do applications scale nicely**
- **Tune your parallel implementation**
- **Start with the most time consuming parts**
- **But how to identify those?**

Intel VTune Amplifier

- **Commercial performance profiler with GUI**
- **Should be used in conjunction with Intel Compiler**
- **Publicly available if you are an open source dev**

http://software.intel.com/en-us/intel-vtune-amplifier-xe

Run Serial if Parallel Doesn’t Fly

- If the workload per thread is too low
- Use OpenMP’s `IF` clause to switch serial execution
- Or limit the number of threads for a region

OpenMP - Efficient Programming

Avoid Implicit Barriers

- **Most worksharing constructs have implicit barriers**
- **Sometimes this is not required - use `nowait`**

```
!$OMP PARALLEL
!$OMP DO
do i=1,N
  A(i) = func1(B(i))
enddo
!$OMP END DO NOWAIT
! still in parallel region here. do more work:
!$OMP CRITICAL
CNT = CNT + 1
!$OMP END CRITICAL
!$OMP END PARALLEL
```

OpenMP - Efficient Programming

Minimize Number of Parallel Regions

- **Parallelizing inner loops: too much overhead**
- **Always aim for parallelizing the most outer loop**
- **Here, move parallel region to include `j`-loop**
- **Also gets rid of the reduction**

```
double precision :: R
R = 0.d0
do j=1,N
  !$OMP PARALLEL DO REDUCTION(+:R)
  do i=1,N
    R = R + A(i,j) * B(i)
  enddo
  !$OMP END PARALLEL DO
  C(j) = C(j) + R
enddo
```

OpenMP - Efficient Programming

Avoid Trivial Load Imbalance

- **Typical problem with heavily nested loops**
- **With increasing thread numbers, workload per thread will drop**
- **Use collapse to increase iteration space (here `MxN`)**
- **Increases the workload by thread and thus scaling**

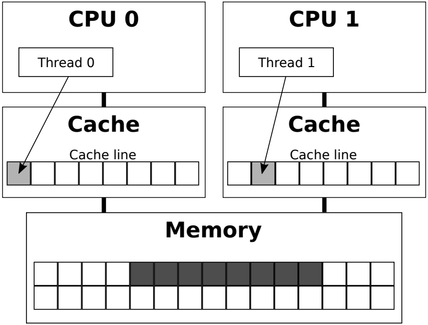
```
double precision, dimension(N,N,N,M) :: A
!$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:res) COLLAPSE(2)
do l=1,M
  do k=1,N
    do j=1,N
      do i=1,N
        res = res + A(i,j,k,l)
enddo ; enddo ; enddo ; enddo
!$OMP END PARALLEL DO
```

OpenMP - Efficient Programming

Avoid Dynamic/Guided Loop Scheduling

- **All scheduling mechanisms (except `static`) introduce overhead**
- **If possible, use `static`**
- **If data set is highly imbalanced, go for `dynamic` or `guided`**

OpenMP - Efficient Programming



False Sharing

- Threads of different processors
- Modify variables of the same cache line
- Cache line invalidated and forces memory update

OpenMP - Efficient Programming

False Sharing

- `sum_local` **dimensioned to number of threads**
- **Small enough to fit in single cache line**
- **Threads modify adjacent elements!**
- **Use *array padding* (insert additional data to ensure not fitting into a cache line)**
- **Or: use private variables and reduction**

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)  {
  int me = omp_get_thread_num();
  sum_local[me] = 0.0;
  #pragma omp for
  for (i = 0; i < N; i++)
    sum_local[me] += x[i] * y[i];
  #pragma omp atomic
  sum += sum_local[me];
}
```

Alternatives

- Since 1990: POSIX Threads (C/C++/Fortran)
- Predominant in today’s HPC: OpenMP (C/C++/Fortran)
- (C/)C++ specific libraries:
  - Intel Threading Building Blocks
  - Cilk/Intel CilkPlus
  - Boost Thread
  - STL Thread
  - Qt Qthread
  - (Charm++) ..

Software

- Shared-Memory Parallel Programming
  - Data Locality
    Locality of Access, Placement Pitfalls, C++ Issues
  - OpenMP
    Basics, Efficiency
  - Alternatives
- **Vectorization**
  - **Intrinsics vs Pragmas**
  - **Automatic Vectorization**
- Distributed and Hybrid Parallel Programming
  - Distributed Parallel Programming
  - Hybrid Parallel Programming
    Potential Benefits and Drawbacks

Vectorization

- **SIMD: Single Instruction, Multiple Data**
- **Different form of parallelization**
- **Instead of using threads/processes**
- **Dedicated SIMD execution hardware units are used**
- **Auto vectorization (compilers) vs. manual vectorization (intrinsics)**
- **Unrolling of a loop combined with packed SIMD instructions**
- **Packed instructions operate on more than one data element at a time**
- **Parallelism!**

Vectorization

- SIMD: Single Instruction, Multiple Data
- Different form of parallelization
- Instead of using threads/processes
- Dedicated SIMD execution hardware units are used
- Auto vectorization (compilers) vs. manual vectorization (intrinsics)
- Unrolling of a loop combined with packed SIMD instructions
- Packed instructions operate on more than one data element at a time
- Parallelism!

Vectorization

- Vectorization: convert scalar algorithm to vector algorithm
- Requires some programming effort but usually pays off
- Major research is done in automatic vectorization
- Automatic vectorization starts with ”-O2” or higher
- Compiler log reports on vectorization results

Vectorization

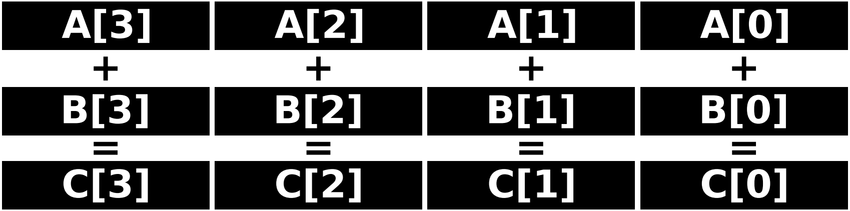
Vector Addition

- `a,b,c` are integer arrays

```
for(i=0;i<=MAX;i++)
  c[i]=a[i]+b[i];
```

A[0]	unused	unused	unused
+	+	+	+
B[0]	unused	unused	unused
=	=	=	=
C[0]	unused	unused	unused

Vectorization



A[3] A[2] A[1] A[0]

+ + + +

B[3] B[2] B[1] B[0]

= = = =

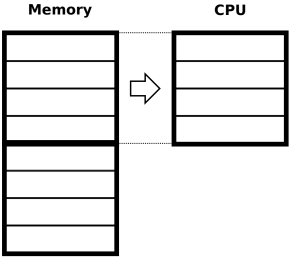
C[3] C[2] C[1] C[0]

Vectorization

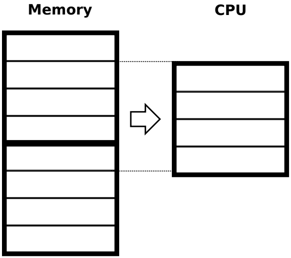
General Requirements/Tips for Vectorization

- **Loop must be countable: Size known at beginning**
- **Single entry, single exit: No `breaks`**
- **Straight-line code: No branching in loop**
- **No function calls, except intrinsics and inlined**
- **Innermost loop of a nest**
- **Favor inner loops with unit stride**
- **Use array notations over pointers**
- **Use aligned data structures**

Vectorization



4 Byte aligned data access



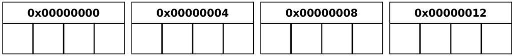
4 Byte misaligned data access

Data alignment

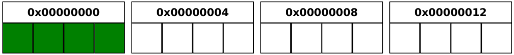
- **A CPU does not read one Byte at a time**
- **It accesses it in 2,4,8,16,.. chunks at a time**
- **If data is misaligned: overhead!**

Vectorization - Data Alignment

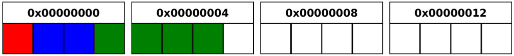
Four word-sized memory cells in a 32-bit computer



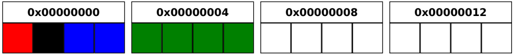
`int` (4 Byte) - aligned



`char` (1 Byte), `short` (2 Byte), `int` (4 Byte) - misaligned



Properly aligned memory using *padding*



Vectorization

General Obstacles To Vectorization

- **Non-contiguous memory access:**
  Non-adjacent data requires separate loading

```
for(int i=0; i<SIZE; i+=2)
  b[i] += a[i] * x[i];
```

- **Data dependencies:**
  Variable is written in one iteration and read in subsequent iteration (read-after-write dependency)

```
A[0]=0;
for(j=1; j<MAX; j++)
  A[j]=A[j-1]+1;
```

Intrinsics vs Pragmas

Intrinsics

- **Use of intrinsic vectorized functions using C style API**
- **Provide access to instructions without using assembly E.g. Compute the absolute value of packed 16-bit integers in `a` and store it in `dest`.**

```
__m128i _mm_abs_epi16 (__m128i a)
```

- **`__m128i`: represents register content, can hold: sixteen 8-bit, eight 16-bit, four 32-bit, or two 64-bit integers.**

Intrinsics vs Pragmas

Pragmas

- **Help the compiler!**
  E.g. `ivdep`: Discard assumed data dependencies

```
#pragma ivdep
for (i = 0; i < N; i++) {
  a[i] = a[i+k] + 1;
}
```

AVX

- Advanced Vector Extensions
- Supported by Intel and AMD
- 128/256/512 bit SIMD registers
- Support for compute intensive floating-point calculations

AVX

- Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, ..
- VSC-3 supported

AVX-2

- Haswell, Broadwell, Skylake, ..

AVX-512

- Xeon Phi (Knights Landing), Skylake Xeon

Software

- Shared-Memory Parallel Programming
  - Data Locality
    Locality of Access, Placement Pitfalls, C++ Issues
  - OpenMP
    Basics, Efficiency
  - Alternatives
- Vectorization
  - Intrinsics vs Pragmas
  - Automatic Vectorization
- **Distributed and Hybrid Parallel Programming**
  - **Distributed Parallel Programming**
  - **Hybrid Parallel Programming**
    Potential Benefits and Drawbacks

Distributed and Hybrid Parallel Programming

Distributed Parallel Programming

- **Enable communication between processes**
- **Thus enables parallel programming**
- **Processes run on separated, interconnected nodes**
- **But also on local node**

Hybrid Parallel Programming

- **Mix distributed parallel programming with node parallelism**
- **E.g. shared-memory programming, accelerators ..**
- **Communication required for inter-node/socket communication**
- **On node/socket: switch to node parallelism**

Distributed and Hybrid Parallel Programming

Distributed Parallel Programming

- Enable communication between processes
- Thus enables parallel programming
- Processes run on separated, interconnected nodes
- But also on local node

Hybrid Parallel Programming

- Mix distributed parallel programming with node parallelism
- E.g. shared-memory programming, accelerators ..
- Communication required for inter-node/socket communication
- On node/socket: switch to node parallelism

Distributed-Memory Parallel Programming

MPI

- Since the availability of parallel computers
- Which programming model is most appropriate
- Explicit message passing is tedious but most flexible
- Message passing interface (MPI) is the de-facto standard (for over 25 years)
- MPI provides communication but also parallel file I/O
- MPI is standardized
- Free open source and commercial implementations available

Distributed-Memory Parallel Programming

Required Information

- **Which process is sending the message?**
- **Where is the data on the sending process?**
- **What kind of data is being sent?**
- **How much data is there?**
- **Which process is going to receive the message?**
- **Where should the receiving process store the message?**
- **What amount of data is the receiver prepared to accept?**

Distributed-Memory Parallel Programming

Blocking point-to-point communication

```
std::vector<double> data(N);
MPI_Send(&(data[0]), N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

Nonblocking point-to-point communication

```
std::vector<double> data(N);
MPI_Request request;
MPI_ISend(&(data[0]), N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &request);
```

Collective communication

```
std::vector<double> data(N);
MPI_Bcast(&(data[0]), N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Hybrid Parallel Programming

Basic Idea

- MPI for inter process communication
- Process spawns several threads to do the actual work
- MPI code is augmented with, e.g., OpenMP directives
- Two basic hybrid programming approaches
  - Vector mode
  - Task mode

Vector Mode

- **All MPI calls are outside of shared-memory regions**
- **Existing MPI code can be easily extended towards hybrid**

```
do iteration=1,MAXITER
  !$OMP PARALLEL DO PRIVATE(..)
  do k = 1,N
    ! Standard 3D Jacobi iteration here ...
  enddo
  !$OMP END PARALLEL DO
  ! halo exchange
  ...
  do dir=i,j,k
    call MPI_Irecv( halo data from neighbor in -dir direction )
    call MPI_Isend( data to neighbor in +dir direction )
    call MPI_Irecv( halo data from neighbor in +dir direction )
    call MPI_Isend( data to neighbor in -dir direction )
  enddo
  call MPI_Waitall( )
enddo
```

Hybrid Parallel Programming

Task Mode

- Most general hybrid programming approach
- MPI communication happens inside shared-memory region
- Needs to be tailored to thread-safety requirements of the utilized MPI library
- Approach for, e.g., decoupling computation and communication
- Task mode provides very high flexibility
- But increases code complexity and size
- Mapping problem: how to assign processes/threads

Hybrid Parallel Programming

```
!$OMP PARALLEL PRIVATE(iteration,threadID,k,j,i,...)
threadID = omp_get_thread_num()
do iteration=1,MAXITER ...
  if(threadID .eq. 0) then
    ! Standard 3D Jacobi iteration; updating BOUNDARY cells ...
    do dir=i,j,k ! After updating BOUNDARY cells do halo exchange
      call MPI_Irecv( halo data from -dir) ...
    enddo
    call MPI_Waitall( )
  else ! not thread ID 0
    ! Remaining threads perform; update of INNER cells 2,...,N-1
    ! Distribute outer loop iterations manually:
    chunksize = (N-2) / (omp_get_num_threads()-1) + 1
    my_k_start = 2 + (threadID-1)*chunksize
    my_k_end = min((2 + (threadID-1+1)*chunksize-1), (N-2))
    do k = my_k_start , my_k_end ! INNER cell updates
      do j = 2, (N-1)
       do i = 2, (N-1) ...
    enddo; enddo; enddo; endif ! thread ID
  !$OMP BARRIER
enddo
!$OMP END PARALLEL
```

Hybrid Parallel Programming - Benefits/Drawbacks

General Remarks

- No general rule whether, e.g., MPI-OpenMP pays off relative to pure MPI
- General rule of thumb: go pure MPI first
- Pure MPI forces one to think about data locality (not the case for shared-memory)
- Certainly of interest to couple MPI with accelerators and co-processors: distributed GPU computing

Hybrid Parallel Programming - Benefits/Drawbacks

Benefits

- Potential re-use of shared-cache
- Introduce additional parallelism to MPI limited applications
- Additional options for load-balancing
- Overlapping communication with computation
- Potential for reducing MPI overhead at domain decomposition

Drawbacks

- Additional level of parallelism complicates development
- OpenMP: not forced upon data locality investigations impact performance

Outline

Part A - Hardware

- Modern Processors
- Parallel Computers

Part B - Software

- Shared-Memory Parallel Programming
- Vectorization
- Distributed and Hybrid Parallel Programming

Summary

Summary

Hardware

- Gets more and more heterogeneous
- NUMA issues already in single-socket settings

Software

- Use all the parallelism the system gives you
- Use all the parallelism which fits the problem

Reference

- G. Hager, G. Wellein: *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, ISBN: 9781439811924, 2010.