High Performance Parallel Programming (CS61064)

Day -1

Pralay Mitra

Why HPC?

- Weather forecast
- Share market forecast
- Many body interaction
- Massive Database search
- High throughput screening
- Intelligent game design
- Real time analysis
- Flexibility over the search space
- Simulation: Atoms to Planets

The first era (1940s-1960s)

Control Data Corporation (CDC) 6600

The Cray Era (1975-1990)

Cray 1, 1976

1980s

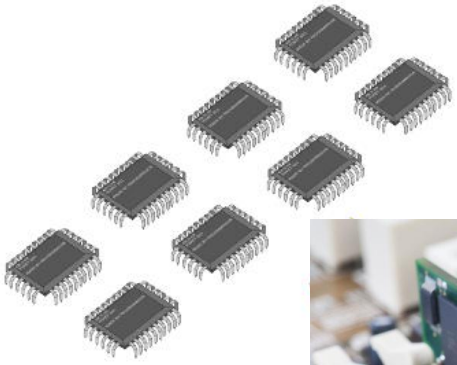
Vectors processors
Shared memory

The cluster Era (1990-2010)

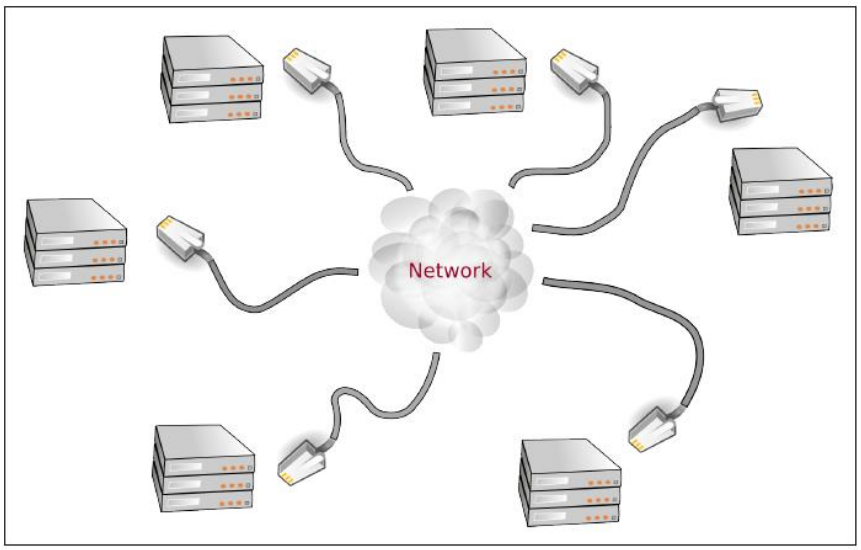
Current Scenario

- The GPGPU and Hybrid Era (2000-
- https://www.top500.org/

Shared or Private



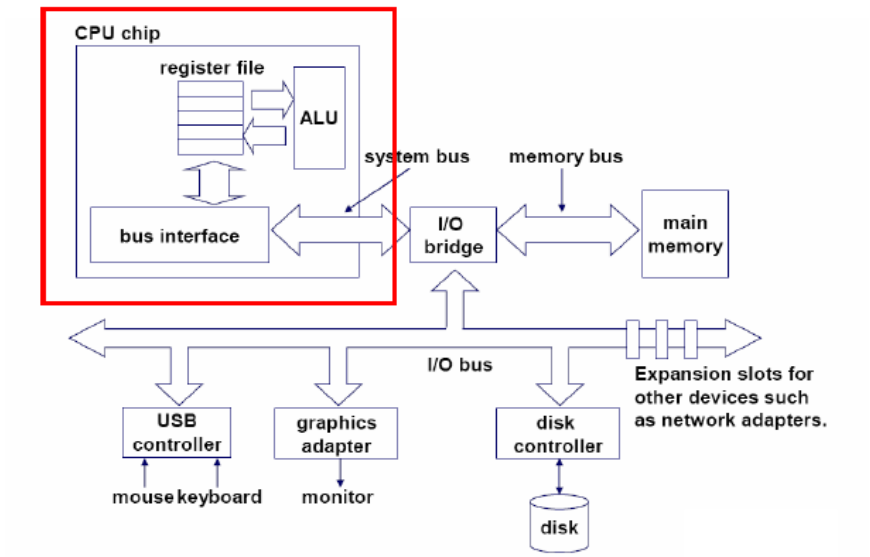
Shared or Private



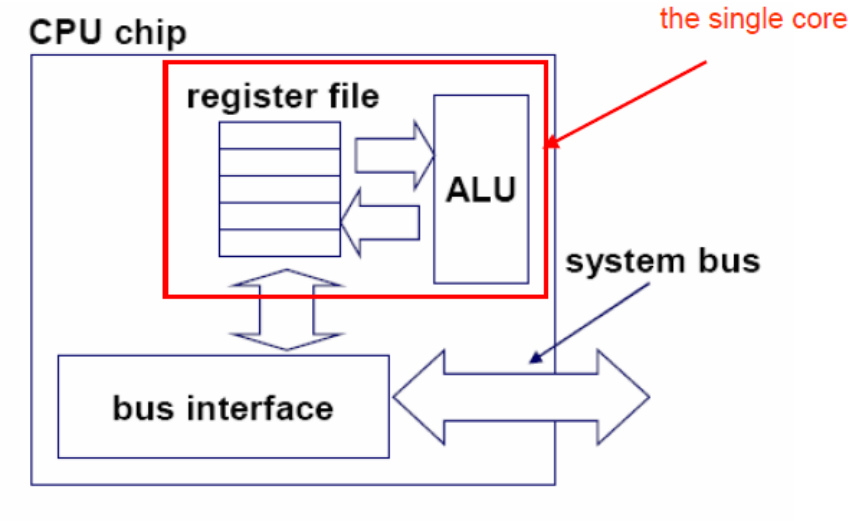
Inside the computer ….

- Core
- Processor
- Single-core
- Multi-core
- …..

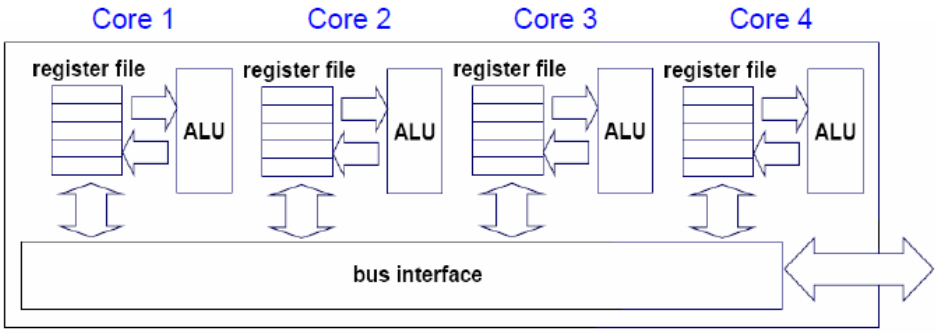
Single-core computer



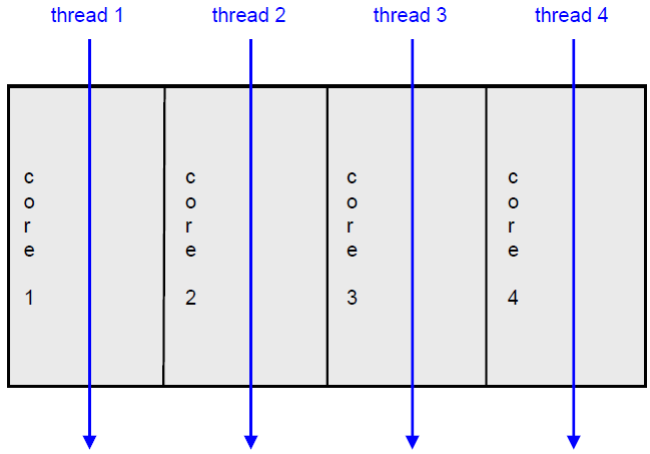
Single-core CPU



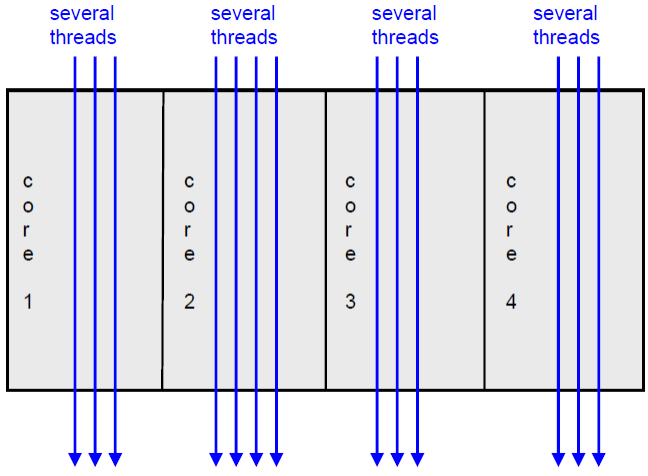
Multi-core CPU



Multi-core CPU

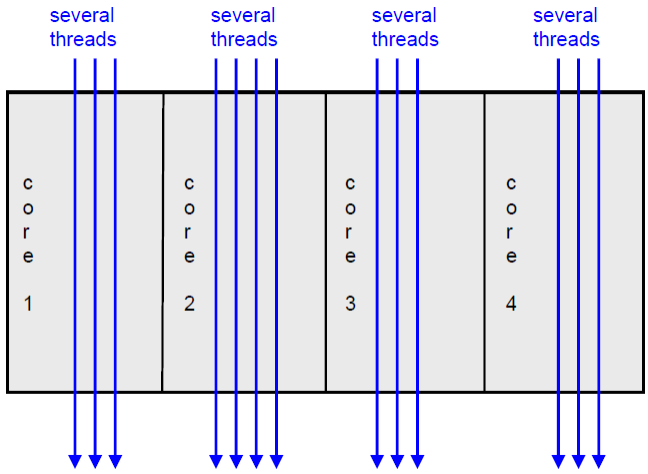


Multi-core CPU



Process vs Thread

Multi-core CPU



An operating system's perspective

Supports multi-core

Perceives each core as a separate processor

Scheduler maps threads/processes to different cores

Multi-core CPU

- ✓ Multi-core processor is a special kind of a multiprocessor where all the processors are on the same chip.
- ✓ Multi-core processors are MIMD where different cores execute different threads (Multiple Instructions), operating on different parts of memory (Multiple Data)
- ✓ Multi-core is a shared memory multiprocessor (SMP) where all cores share the same memory

Multi-core Architecture: Definition

- A multi-core architecture (or a chip multiprocessor) is a general-purpose processor that consists of multiple cores on the same die and can execute programs simultaneously

Multi-core CPU - Applications

- ✓ Database servers
- ✓ Web servers
- ✓ Compilers
- ✓ Multimedia Applications
- ✓ Scientific applications CAD/CAM
- ✓ In general, applications with Thread Level Parallelism (as opposed to instruction level parallelism)

Multi-core CPU – More Applications

- ✓Editing photo while recording a TV show through a digital video converter
- ✓Downloading software while running an anti-virus program
- ✓Anything that can be threaded today will map efficiently to multi-core
- ✓BUT: Not all --- those which are difficult to parallelize

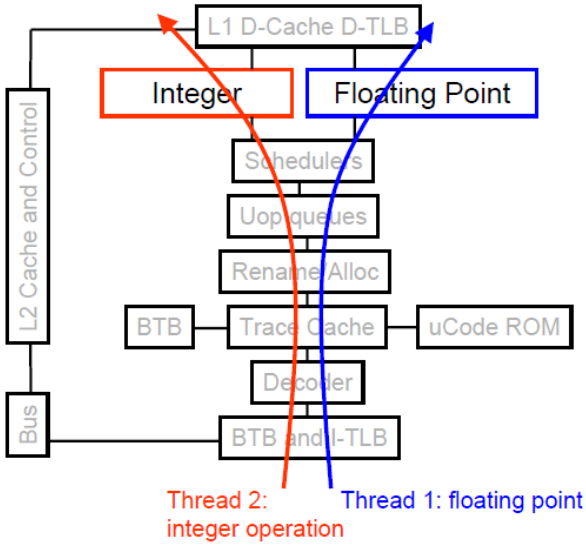
Simultaneous multithreading (SMT)

Permits multiple independent threads to execute simultaneously on the same core

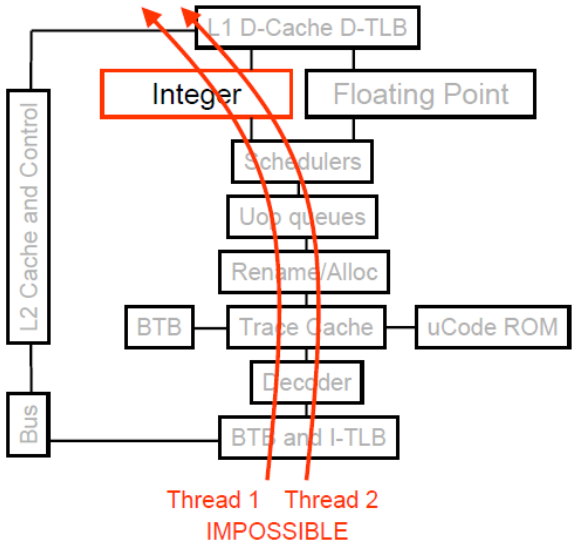
Wearing together multiple “threads” on the same core

Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

SMT processor: concurrent execution



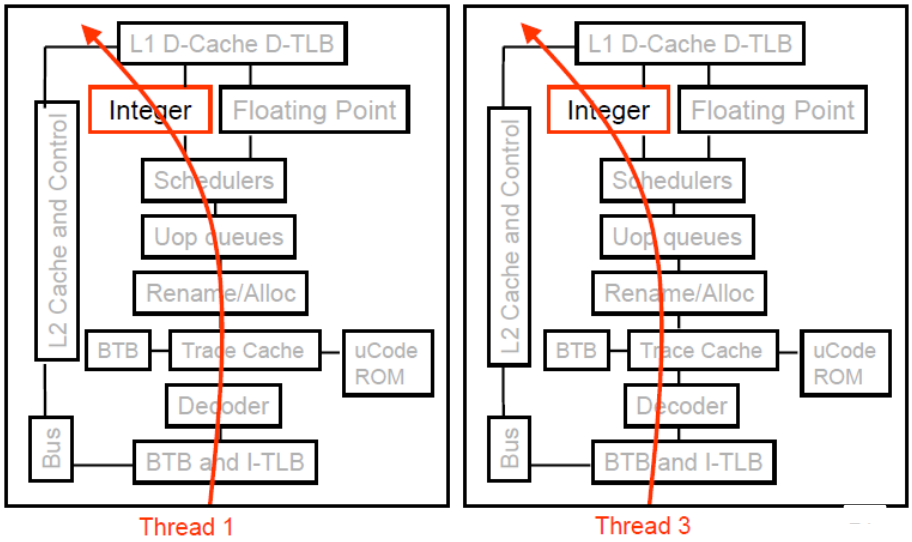
SMT processor: no concurrent execution on same FU



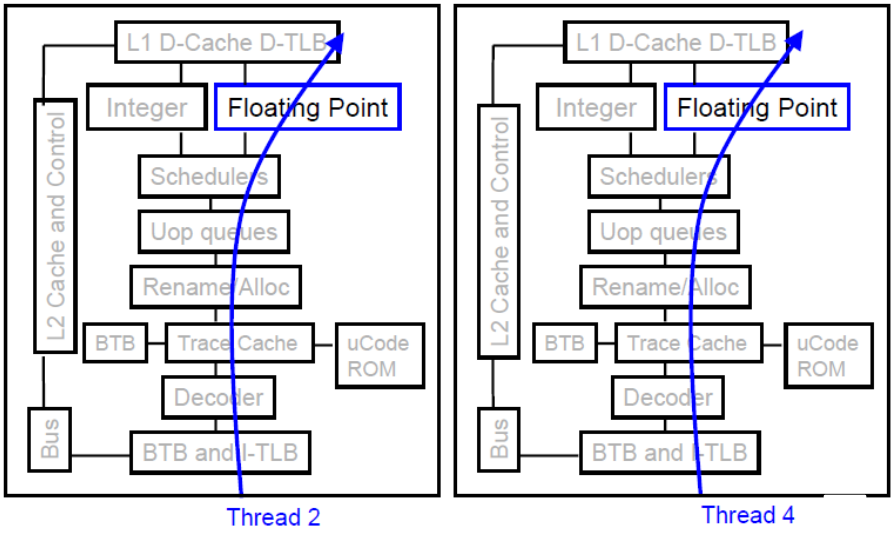
SMT processor: not a true parallel processor

- ✓Enables better threading (e.g. upto 30%)
- ✓OS and application perceives each simultaneous thread as a separate “virtual processor”
- ✓This has only a single copy of each resources
- ✓Compare to multi-core, in this case each core has its own copy of resources

Multi-core solutions



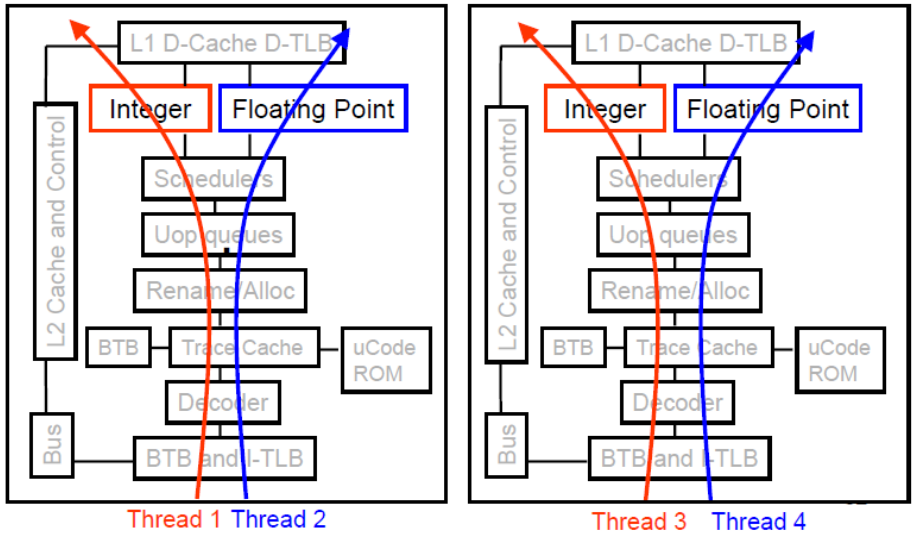
Multi-core solutions



Combining multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT
- The number of SMT threads:
  2, 4, or sometimes 8 simultaneous threads
- Intel calls them “hyper-threads”

SMT Dual-core: four threads in action



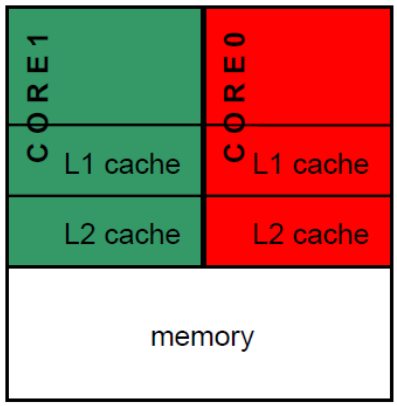
Comparison: multi-core vs SMT

- Multi-core:
  - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

The memory hierarchy

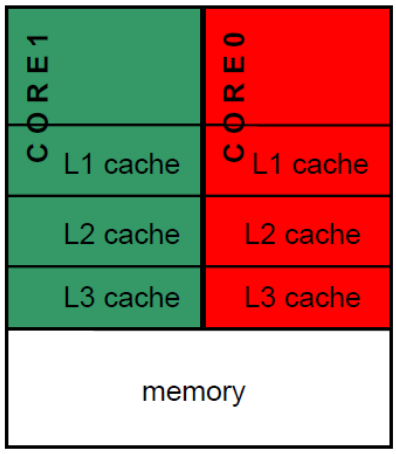
- If simultaneous multithreading only:
  - all caches shared
- Multi-core chips:
  - L1 caches private
  - L2 caches private in some architectures and shared in others
- Memory is always shared

Design with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron, AMD Athlon, Intel Pentium D



A design with L3 caches

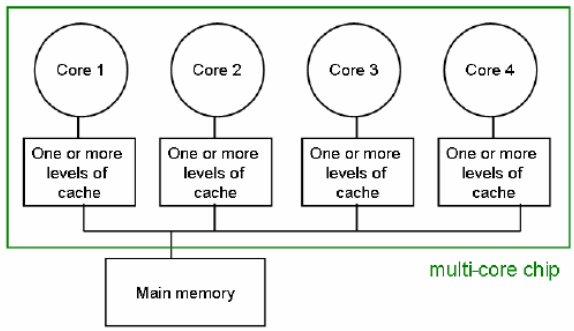
Example: Intel Itanium 2

Private vs shared caches

- Advantages of private:
  - They are closer to core, so faster access
  - Reduces contention
- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system

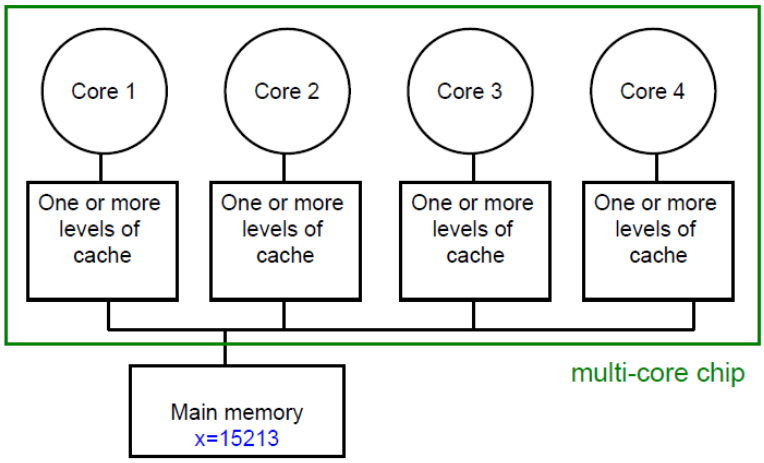
The cache coherence problem

- Since we have private caches:
  How to keep the data consistent across caches?
- Each core should perceive the memory as a monolithic array, shared by all the cores



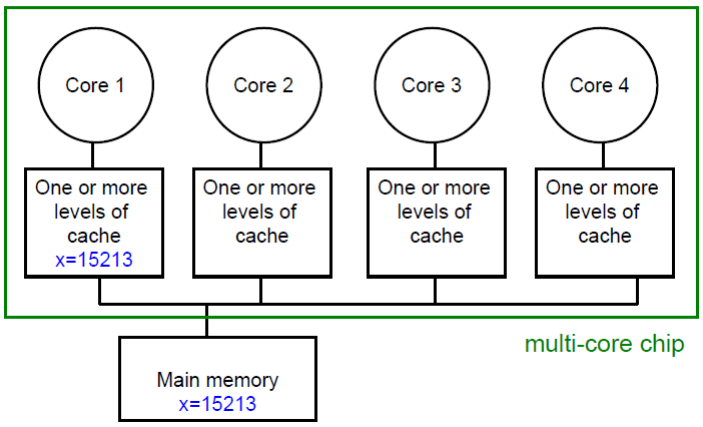
The cache coherence problem

Suppose variable x initially contains 15213



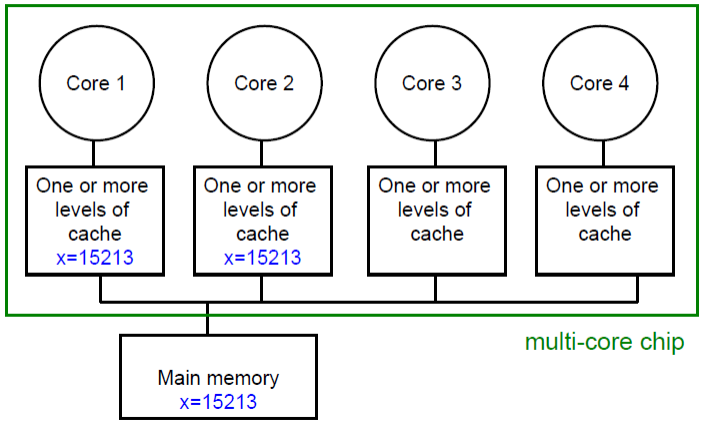
The cache coherence problem

Core 1 reads x



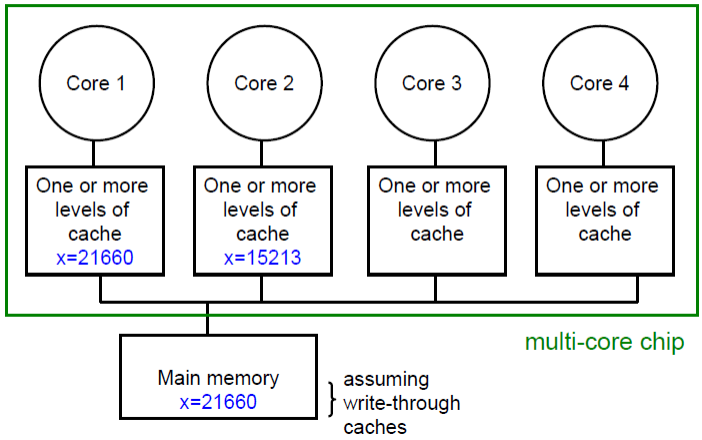
The cache coherence problem

Core 2 reads x



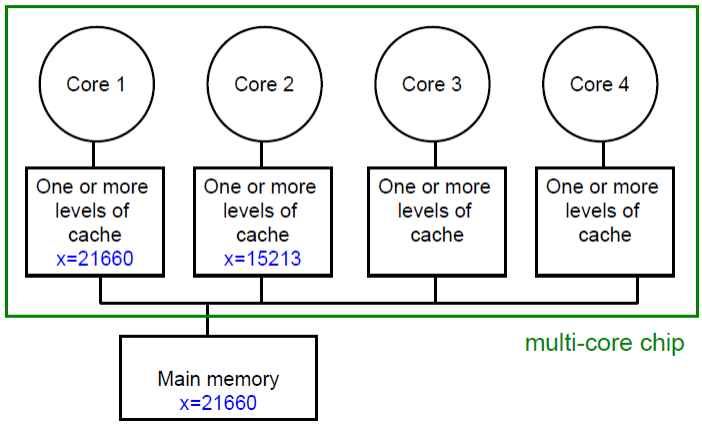
The cache coherence problem

Core 1 writes to x, setting it to 21660



The cache coherence problem

Core 2 attempts to read x… gets a stale copy



Solutions

- This is a general problem with multiprocessors, not limited just to multi-core
- There exist many solution algorithms, coherence protocols, etc.

- A simple solution:
  invalidation-based protocol with *snooping*

Programming for multi-core

- Programmers must use threads or processes
- Spread the workload across multiple cores
- Write parallel algorithms
- OS will map threads/processes to cores

Important: Safety of Thread

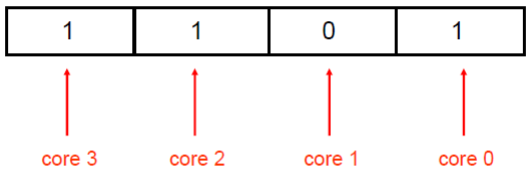
- Pre-emptive context switching: context switch can happen AT ANY TIME
- True concurrency, not just uniprocessor time-slicing
- Concurrency bugs exposed much faster with multi-core

Assigning threads to cores

- Each thread has an *affinity mask*
- Affinity mask specifies what cores the thread is allowed to run on
- Different threads can have different masks
- Affinities are inherited across fork()

Affinity masks are bit vectors

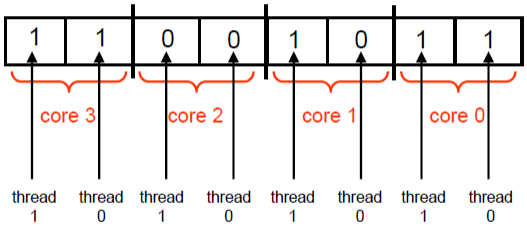
- Example: 4-way multi-core, without SMT



- Process/thread is allowed to run on cores 0,2,3, but not on core 1

Affinity masks when multi-core and SMT combined

- Separate bits for each simultaneous thread
- Example: 4-way multi-core, 2 threads per core



- Core 2 can't run the process
- Core 1 can only use one simultaneous thread

Default affinity

- Default affinity mask is all 1s: all threads can run on all processors
- Then, the OS scheduler decides what threads run on what core
- OS scheduler detects skewed workloads, migrating threads to less busy processors

Now

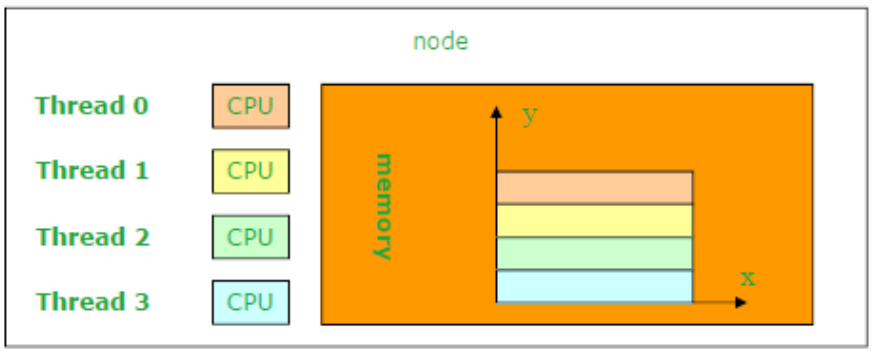
Grandness

- WRF (Weather Research Forecast) ConUS (CONtinental Usa) 2.5km 6hr benchmark
  - Single P6: ~40 hr (though theoretically ~4hr)
  - 4 nodes (128 cores): 0.6 hr
  - 64 nodes (1024 cores): 9 min

Parallel Software Models and Languages

- **Programming Models**
  - Shared Memory (OpenMP)
  - Message Passing (MPI)
  - Hardware Accelerators (CUDA, OpenCL)
  - Hybrid
- **Programming Language:**
  - C
  - C++
  - Fortran

Shared Memory



Shared Memory

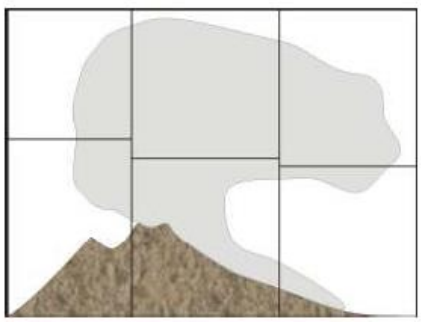
OpenMP

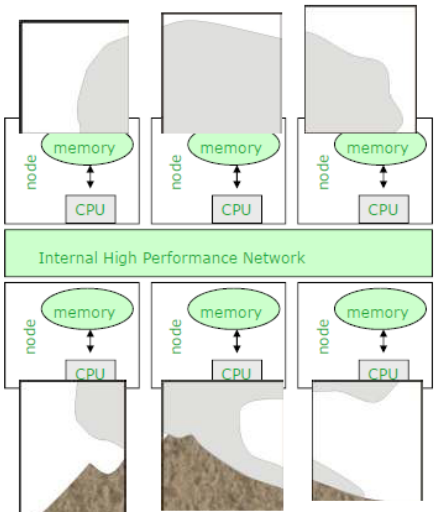
– **Main Characteristics**
- Compiler directives
- Medium grain
- Intra node parallelization
- Loop or iteration partition
- Shared memory
- Many HPC App

– **Open Issues**
- Thread creation overhead
- Memory/core affinity
- Interface with MPI

Private Memory

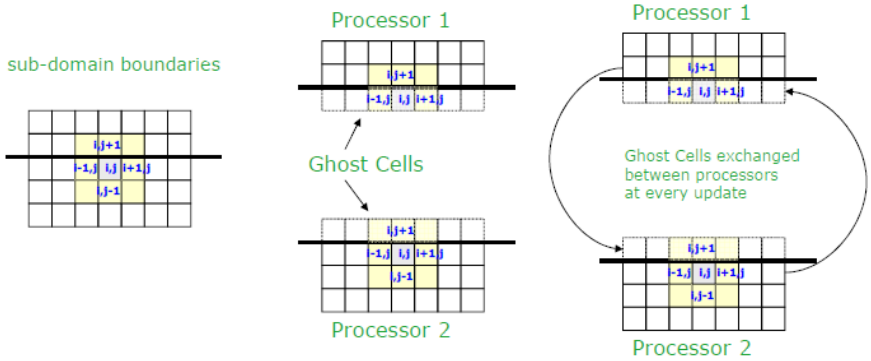
Message Passing





Private Memory

Message Passing



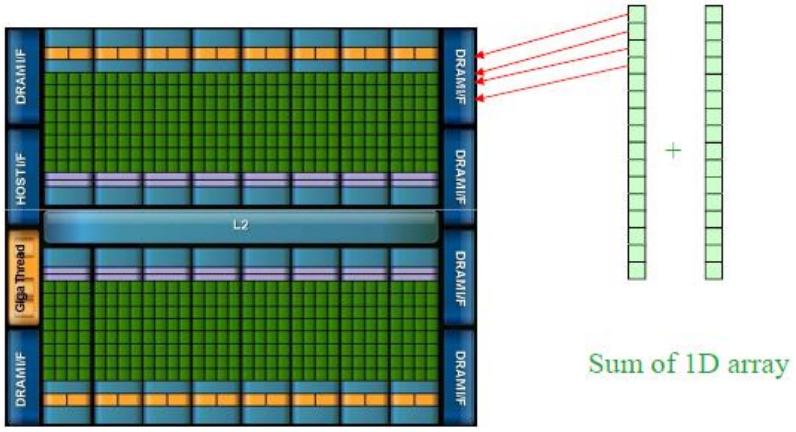
Private Memory

MPI

– **Main Characteristics**
- Library
- Coarse grain
- Inter node parallelization
- Domain partition
- Distributed memory
- Almost all HPC parallel App

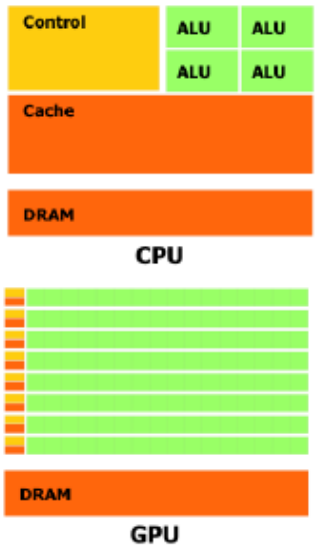
– **Open Issues**
- Latency
- OS Jitter
- Scalability

Accelerator / GPGPU

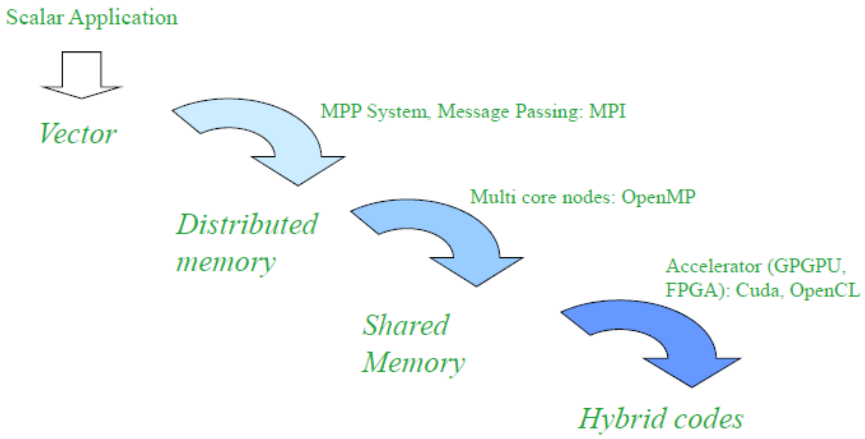


CUDA - OpenCL

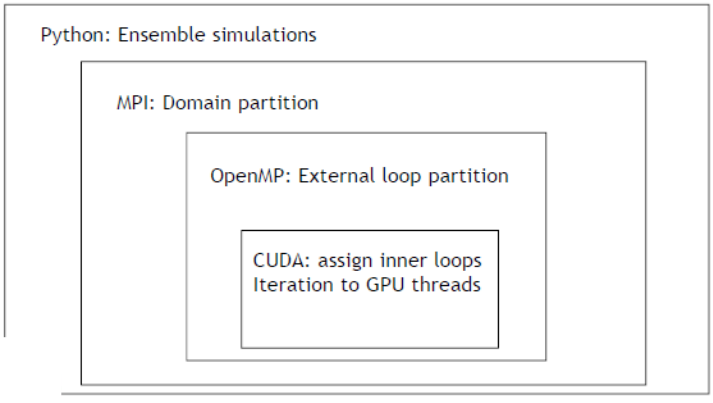
- **Main Characteristic**
  - Ad-hoc compiler
  - Fine grain
  - Offload parallelization (GPU)
  - Single iteration parallelization
  - Few HPC App
- **Open Issue**
  - Memory copy
  - Standard
  - Tools
  - Integration with other languages



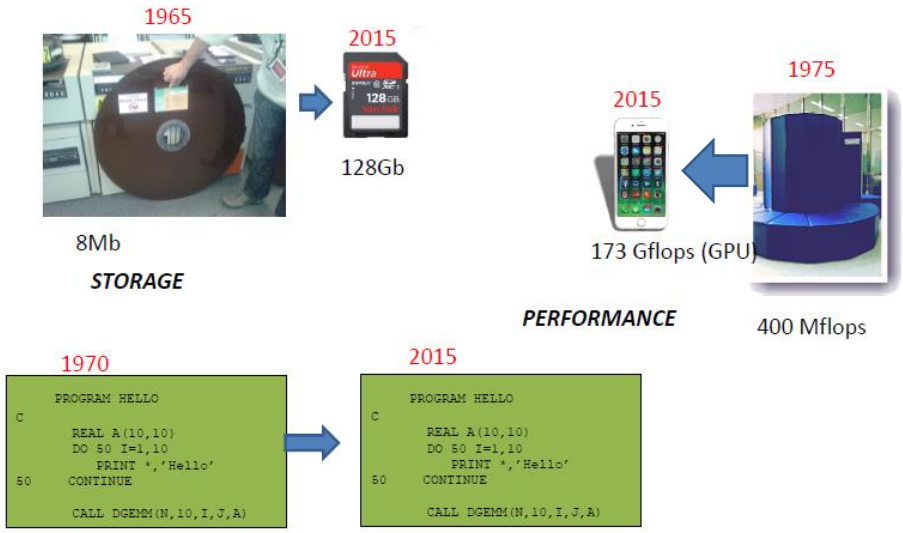
Hybrid Parallel Programming



Hybrid Parallel Programming



Advances: Hardware vs Software



Real HPC Crisis: Software

A supercomputer application and software are usually much more long-lived than a hardware

- Hardware life typically four-five years at most.
- Fortran and C are still the main programming models

Programming is stuck

- Arguably hasn't changed so much since the 70's

Software is a major cost component of modern technologies.

- The tradition in HPC system procurement is to assume that the software is free.

It's time for a change

- Complexity is rising dramatically
- Challenges for the applications on Petaflop systems
- Improvement of existing codes will become complex and partly impossible.
- The use of O(100K) cores implies dramatic optimization effort.
- New paradigm as the support of a hundred threads in one node implies new parallelization strategies
- Implementation of new parallel programming methods in existing large applications can be painful

Software Difficulties

- Legacy applications (includes most scientific applications) not designed with good software engineering principles. Difficult to parallelise programs with many global variables, for example.
- Memory/core decreasing.
- I/O heavy impact on performance, esp. for BlueGene where I/O is handled by dedicated nodes.
- Checkpointing and resilience.
- Fault tolerance over potentially many thousands of threads.
  - In MPI, if one task fails all tasks are brought down.

Summary

- HPC is only possible via parallelism and this must increase to maintain performance gains.
- Parallelism can be achieved at many levels but because of limited code scalability with traditional cores increasing role for accelerators (e.g. GPUs, MICs). The Top500 is becoming now becoming dominated by hybrid systems.
- Hardware trends forcing code re-writes with OpenMP, OpenCL, CUDA, OpenACC, etc in order to exploit large numbers of threads.
- Unfortunately, for many applications the parallelism is determined by problem size and not application code.
- Energy efficiency (Flops/Watt) is a crucial issue. Some batch schedulers already report energy consumed and in the near future your job priority may depend on predicted energy consumption.

Your first openMP program

```
#include <stdio.h>
#include <omp.h>

int main()
{
        #pragma omp parallel
        {
                printf("Hello world!\n");
        }
        return 0;
}
```

Your first openMP program

Check your system support: locate omp.h
/usr/lib/gcc/x86_64-redhat-linux/4.8.2/include/omp.h

Compilation: g++ -fopenmp first_openMP.c

Conditional Compilation:

```
#ifdef _OPENMP
        printf("Compiled with OpenMP support:%d",_OPENMP);
#else
        printf("Compiled for serial execution.");
#endif
```

Execution: ./a.out

Flags:

GNU: **-fopenmp** for Linux, Solaris, AIX, MacOSX, Windows.
IBM: **-qsmp=omp** for Windows, AIX and Linux.
Sun: **-xopenmp** for Solaris and Linux.
Intel: **-openmp** on Linux or Mac, or **-Qopenmp** on Windows
PGI: **-mp**

Your first openMP program

```
$ ./a.out
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Setting Environmental Variable

- **Know your shell**

  $ echo $SHELL

  $ /bin/bash

$ export OMP_NUM_THREADS=16

Your first openMP program

```
#include <stdio.h>
#include <omp.h>

int main()
{
        #pragma omp parallel
        {
                printf("Hello world!\n");
        }
        return 0;
}
```

Setting Environmental Variable and Executing

```
//Know your shell
$ echo $SHELL
$ /bin/bash
//Setting environmental variables
$ export OMP_NUM_THREADS=8
//Compilation
$ g++  -fopenmp  first_openMP.c
//Execution
$ ./a.out
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Directives

- **Syntactically directives are just comments**
  - **#pragma omp** *directive-name [clause[[,] clause]…] new-line*
- **Examples**
  - #pragma omp parallel
- **Clause is one of the followings**
  - if(scalar-expression)
  - private(variable-list)
  - firstprivate(variable-list)
  - default(shared | none)
  - shared(variable-list)
  - copyin(variable-list)
  - reduction(operator: variable-list)
  - num_threads(integer-expression)
- **Multiple directive names are not allowed**
  - #pragma omp parallel barrier

parallel construct

#pragma omp parallel

- Forms a team of *N* threads before starting executing parallel region
- *N* is set by *OMP_NUM_THREADS* environment or using function *omp_set_num_threads()*
- Semantics is (almost) same as serial program

Comments on *parallel* construct

- At most one **if** clause can appear on the directive (serial/parallel)
- It is unspecified whether any side effects inside the **if** expression or **num_threads** expression occur.
- Only a single **num_threads** clause can appear on the directive. The **num_threads** expression is evaluated outside the context of the parallel region, and must evaluate to a positive integer value.
- The order of evaluation of the **if** and **num_threads** clauses is unspecified.
- A nested parallel region is executed by a team composed of one thread. The default behavior may be changed by using either the runtime library function **omp_set_nested.**
- If the **num_threads** clause is present then it supersedes the number of threads requested by the **omp_set_num_threads** library function or the **OMP_NUM_THREADS** environment variable only for the parallel region it is applied to. Subsequent parallel regions are not affected by it.

Example -2

```
# include <stdio.h>
# include <omp.h>
int main ( int argc, char *argv[] ) {
 int id;
 double wtime;
 printf ( "Number of processors available = %d\n", omp_get_num_procs ( ) );
 printf ( "Number of threads =           %d\n",  omp_get_max_threads ( ) );
 wtime = omp_get_wtime ( );
 printf ( "OUTSIDE the parallel region.\n" );

 id = omp_get_thread_num ( );
 printf ( "HELLO from process %d\n Going INSIDE the parallel region:\n ", id ) ;

# pragma omp parallel \
 private ( id )  {
  id = omp_get_thread_num ( );
  printf ("  Hello from process %d\n", id );
 }
 wtime = omp_get_wtime ( ) - wtime;

 printf ( "Back OUTSIDE the parallel region.\nNormal end of execution.\nElapsed wall clock time = %f\n", wtime );
 return 0;
}
```

Example -2

Initialization:

export OMP_NUM_THREADS=16

Compilation:

g++ -fopenmp example.c

Execution:

./a.out

Example -2

```
Number of processors available = 8
Number of threads =          16
OUTSIDE the parallel region.
HELLO from process 0
Going INSIDE the parallel region:
 Hello from process 4
 Hello from process 0
 Hello from process 2
 Hello from process 14
 Hello from process 12
 Hello from process 13
 Hello from process 3
 Hello from process 7
 Hello from process 1
 Hello from process 8
 Hello from process 9
 Hello from process 10
 Hello from process 11
 Hello from process 5
 Hello from process 6
 Hello from process 15
Back OUTSIDE the parallel region.
Normal end of execution.
Elapsed wall clock time = 0.001034
```