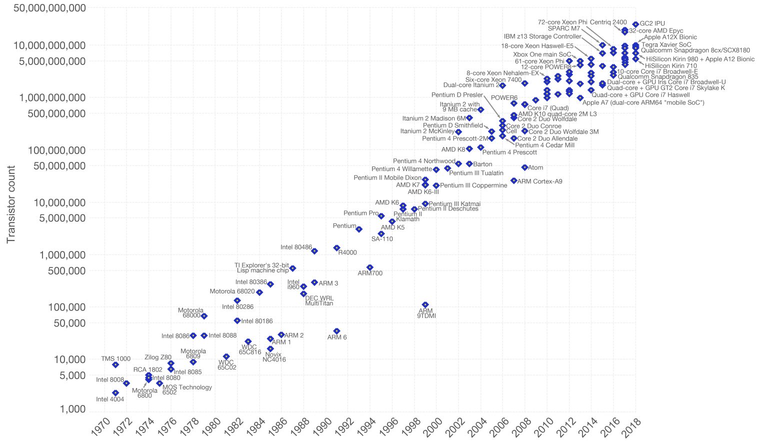
# Applied Mathematics 225

## Unit 0: Introduction and OpenMP programming

Lecturer: Chris H. Rycroft

## Moore’s law

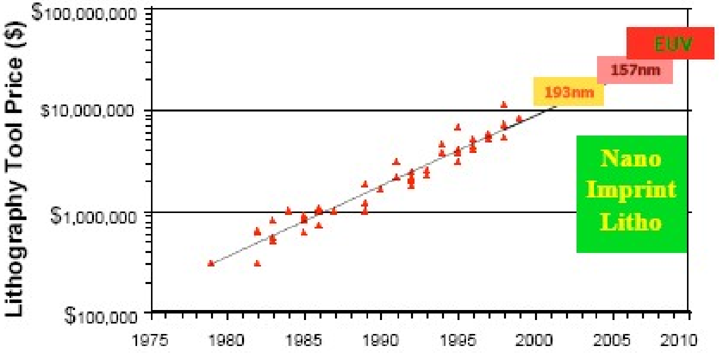
*“The transistor density of semiconductor chips will double roughly every 18 months”*



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

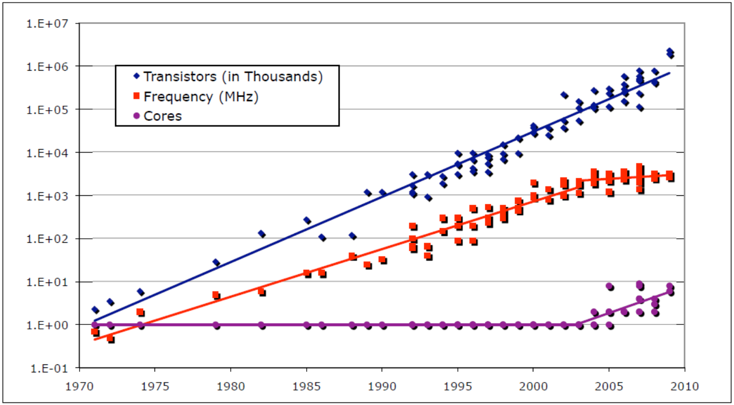
## Moore's second law

- There is exponential growth in the cost of tools for chip manufacturing
- Power density scales like the cube of clock frequency



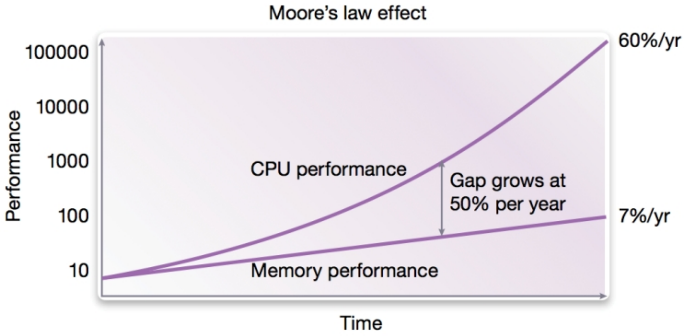
## Consequence

Serial processors are not getting faster. Instead, the emphasis is on parallelism via multi-core processors.



## A related problem: scaling of memory performance

Improvements in memory access time are significantly slower than the transistor count



## Important questions for scientific computing

Multi-core chips are now standard—even a smartphone has a dual or quad core chip

But many classic textbook algorithms date from times before these considerations were important

How well can numerical algorithms exploit parallelism?

Do we need to think differently about algorithms to address memory access limitations?

## Example: BLAS and LAPACK

BLAS: Basic Linear Algebra Subprograms
(`http://www.netlib.org/blas/`)

LAPACK: Linear Algebra PACKage
(`http://www.netlib.org/lapack/`)

- Highly successful libraries for linear algebra (*e.g.* solving linear systems, eigenvalue computations, *etc.*)
- Installed by default on most Linux and Mac computers
- Forms the back-end for many popular linear algebra platforms such as MATLAB and NumPy
- Key advances: refactor basic matrix operations to limit memory usage

We will examine BLAS and LAPACK in Unit 2 of the course

## C++ and Python comparison

Computer demo: Ridders' method for one-dimensional root finding.

## Quick note[1]

The rest of this unit is heavy on computer science principles and programming

It is not especially indicative of the tone of the rest of the course

Next week will see a shift into mathematics

[1]Thanks to Georg Stadler (NYU) for allowing me to adapt material from his HPC17 course.

## Basic CS terms

- compiler: translates human code into machine language
- CPU/processor: central processing unit performs instructions of a computer program, *i.e.*, arithmetic/logical operations, input/output
- core: individual processing unit in a "multicore" CPU
- clock rate/frequency: indicator of speed at which instructions are performed
- floating point operation (flop): multiplication–add of two floating point numbers, usually double precision (64 bits, $\sim 16$ digits of accuracy)
- peak performance: fastest theoretical flop/s
- sustained performance: flop/s in actual computation
- memory hierarchy: large memories (RAM/disc/solid state) are slow; fast memories (L1/L2/L3 cache) are small

## Memory hierarchies

Computer architecture is complicated. We need a basic performance model.

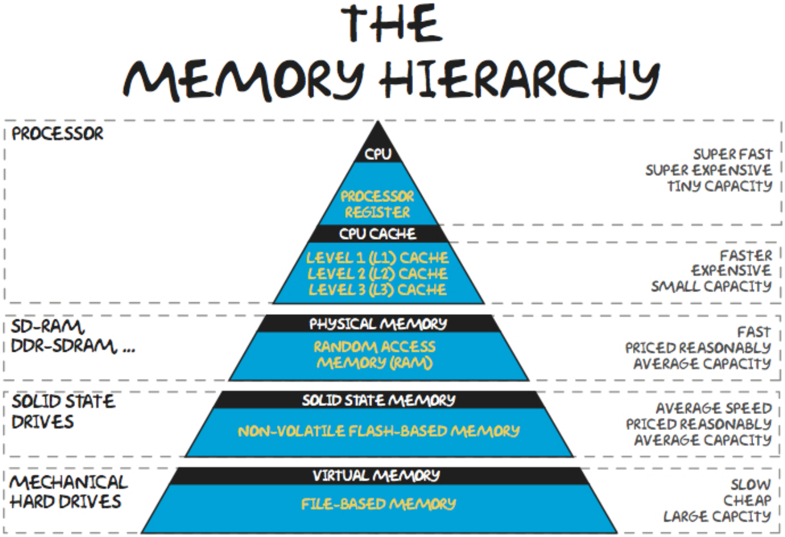
- Processor needs to be “fed” with data to work on
- Memory access is slow; memory hierarchies help
- This is a single processor issue, but it’s even more important on parallel computers

More CS terms:

- latency: time it taks to load/write data from/at a specific location in RAM to/from the CPU registers (in seconds)
- bandwidth: rate at which data can be read/written (for large data); in (bytes/second)

Bandwidth grows faster than latency

## Memory hierarchies



CPU: $O(1\text{ ns})$, L2/L3: $O(10\text{ ns})$, RAM: $O(100\text{ ns})$, disc: $O(10\text{ ms})$

## Memory hierarchies

To decrease memory latency

- Eliminate memory operations by saving data in fast memory and reusing them, *i.e.*, temporal locality: access an item that was previously accessed
- Exploit bandwidth by moving a chunk of data into the fast memory, *i.e.*, spatial locality: access data nearby previous accesses
- Overlap computation and memory access (pre-fetching; mostly figured out by the compiler, but the compiler often needs help)

More CS terms:

- cache-hit: required data is available in the cache $\Longrightarrow$ fast access
- cache-miss: required data is not in cache and must be loaded from main memory (RAM) $\Longrightarrow$ slow access

## Programming in C++

- Developed by Bjarne Stroustrup in 1979 as a successor to the C programming language (1972)
- Efficient and flexible similar to C, but also with features for high-level program organization[2]
- Compiled language—input program is converted into machine code
- Shares many features with other languages (*e.g.* for loops, while loops, arrays, *etc.*), but provides more direct access and control of memory, which is useful for this course

[2]B. Stroustrup, *Evolving a language in and for the real world: C++ 1991–2006*. [Link]

## C++ compilers

- **The GNU Compiler Collection (GCC)** – free, widely-used compiler that is available by default on most Linux computers, and can be installed on many different systems. The GCC C++ compiler is called g++.
- **Clang** – another free compiler project that is the back-end for C++ on Apple systems via Xcode. (For compatibility, if you type g++ on Macs, you are actually using Clang.)
- **Intel compiler** – proprietary compiler that sometimes gives a small (*e.g.* 5%–10%) performance boost
- **Portland (PGI) compiler**
- **Microsoft Visual C++**

## Good references

- *The C++ Programming Language, 4th edition* by Bjarne Stroustrup, 2013.
- `http://www.cplusplus.com` – extensive documentation and language tutorial.
- `http://en.cppreference.com/w/` – very nice, but more designed as a reference.
- Chris, Nick, and Eder: they *love* C++! They'll talk about it for hours!

## Evolving standards

- C++98 – original standardized version from ANSI[3]/ISO[4] committees
- C++11 – many useful features like `auto` keyword and `nullptr` added
- C++14, C++17, C++20, . . .

Trade-offs in the choice of standard:

- Newer versions provide more flexibility and fix small issues with the original version
- Older versions are more widely supported and inter-operable with different systems

Chris's preference (mainly borne out of developing software libraries) is to use the original C++98 standard for maximum compatibility

[3]American National Standards Institute

[4]International Organization for Standardization

## Basic command-line compilation

To compile a program `hello_world.cc` into `hello_world`:

```
g++ -o hello_world hello_world.cc
```

To enable optimization, pedantically enforce ANSI C++98 syntax, and switch on all warnings:

```
g++ -O3 -Wall -ansi -pedantic -o hello_world \
    hello_world.cc
```

## Quick C++ example #1

```cpp
#include <cstdio>

int main() {
  puts("Hello world!");
}
```

## Quick C++ example #1 (annotated)

```cpp
// Include system header with
// input/output functions
#include <cstdio>

// Main program is defined
// as a function called "main"
int main() {

  // Call system function
  // to print a string
  puts("Hello world!");
}
```

## Quick C++ example #2

```
#include <cstdio>
int main() {

  // Variables must explicitly declared with a type
  int a=1,b;

  // Single-precision and double-precision
  // floating point numbers
  float c=2.0;
  double d=3.4;

  // Arithmetic
  b=3*(a++);

  // Formatted print
  printf("%d %d %g %g\n",a,b,c,d);
}
```

## Quick C++ example #3

```cpp
#include <cstdio>
#include <cmath>

int main() {

  // Standard math functions
  // are in the <cmath> header
  double a,b,c;
  a=sqrt(1.2);
  b=4*atan(1.0);
  c=tanh(5.0);

  // Formatted print
  printf("%g %g %g\n",a,b,c);
}
```

## Quick C++ example #4[5]

```
#include <cstdio>

int main() {

  // Implement Fizz Buzz children’s game
  for(int i=1;i<20;i++) {
    if(i%3==0) puts(i%5==0?"Fizz Buzz":"Fizz");
    else {
      if(i%5==0) puts("Buzz");
      else printf("%d\n",i);
    }
  }
}
```

[5] https://en.wikipedia.org/wiki/Fizz_buzz

## Quick C++ example #5

```
#include <cstdio>

int main() {

  // Simple array construction
  double v[32];
  v[3]=4.;

  // A pointer to a double
  double* w;

  // Assign pointer. Note v itself is a pointer to the start
  // of the array.
  w=v+3;
  printf("%p %g\n",w,*w);

  // For-loop with pointers
  for(w=v;w<v+32;w++) *w=1.;

  // Out of bounds. May cause segmentation fault error. But
  // may not. With great power comes great responsibility.
  v[40]=0.;
}
```

## C++ and Python comparison

Computer demo: Timing comparison for Ridders' method in Python and C++

## C++/Python timing results (on Mid 2014 MacBook Pro)

```
altair:% python ridders_array.py
Time: 26.1 s (total)
Time: 26.0999 microseconds (per value)

altair:% ./ridders_array
Time: 0.237 s (total)
Time: 0.236984 microseconds (per value)
```

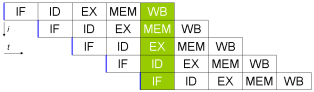
## C++ version is about 110 times faster

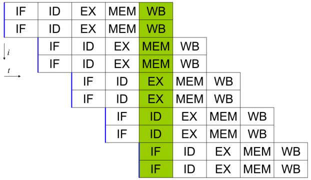
- In-class poll showed most people expected roughly a $20\times$ to $50\times$ speedup.
- Relative slowness of Python is well-documented and is due to many reasons: interpreted language, dynamic typing, *etc.*[6]
- Many considerations in language choice:
  - Python offers great flexibility
  - Many Python library routines (*e.g.* NumPy) are in compiled code and are much faster
  - Extra speed not required for many tasks; need to weigh the time of the programmer against the time of computation
- Compiled languages are a good choice for critical code bottlenecks

[6]Good article suggested by W. Burke: https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

# Levels of parallelism

- Parallelism at the bit level (64-bit operations)
- Parallelism by pipelining (overlapping of execution of multiple instructions); several operators per cycle
- Multiple functional units parallelism: ALUs (algorithic logical units), FPUs (floating point units), load/store memory units, . . .



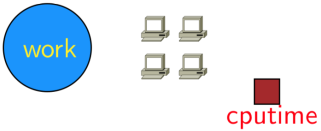


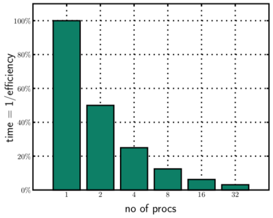
All of the above assume single sequential control flow

- process/thread level parallelism: independent processor cores, mulitcore processors; parallel control flow

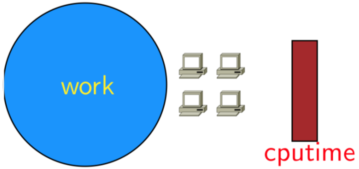
# Strong versus weak scaling

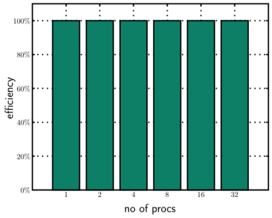
Strong scalability





Weak scalability





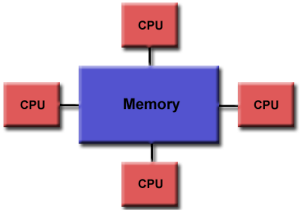
## Load (im)balance in parallel computations

In parallel computations, the work should be distributed evenly across workers/processors

- Load imbalance: idle time due to insufficient parallelism or unequal-sized tasks
- Initial/static load balancing: distribution of work at beginning of computation
- Dynamic load balancing: workload needs to be re-balanced during computation. Imbalance can occur, *e.g.*, due to adaptive mesh refinement

## Shared memory programming model (the focus of this course)

- Program is a collection of control threads that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- Danger: more than one processor core reads/writes to a memory location – a race condition

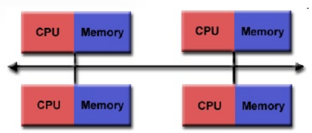


Programming model must manage different threads and avoid race conditions

OpenMP: Open Multi-Processing is the application interface that supports shared memory parallelism, `http://www.openmp.org/`

## Distributed memory programming model (for comparison)[7]

- Program is run as a collection of named processes; fixed at start-up
- Local address space; no shared data
- Logically shared data is distributed (*e.g.* every processor only has access to a chunk of rows of a matrix)
- Explicit communication through send/receive pairs



Programming model must accommodate communication

MPI: Message Passing Interface (different implementations: LAM, OpenMPI, Mpich), `http://www.mpi-forum.org/`

[7] For full details, see COMPSCI 205 offered this semester.

## Hybrid distributed/shared programming model

- Pure MPI approach splits the memory of a multicore processor into independent memory pieces, and uses MPI to exchange information between them
- Hybrid approach uses MPI across processors and OpenMP for processor cores that have access to the same memory. This often results in optimal performance.
- A similar hybrid approach is also used for hybrid architectures, *e.g.* computers that contain CPUs and GPUs

## OpenMP introduction

- Built into all modern versions of GCC, and enabled with the `-fopenmp` compiler flag.
- Clang has OpenMP support. Unfortunately, Apple's custom version of Clang doesn't.
- On the Mac, you can obtain an OpenMP-capable compiler via the package management systems MacPorts[8] and Homebrew[9]
- Excellent online tutorial at `https://bisqwit.iki.fi/story/howto/openmp/`
- Standard C++ but with additional `#pragma` commands to denote areas that require multithreading

[8]`http://www.macports.org`

[9]`https://brew.sh`

## Quick OpenMP example #1

```
#include <cstdio>

int main() {

#pragma omp parallel
  {
    // Since this is within a parallel block,
    // each thread will execute it
    puts("Hi");
  }
}
```

## Quick OpenMP example #2

```
#include <cstdio>

// OpenMP header file with specific
// thread-related functions
#include "omp.h"

int main() {

#pragma omp parallel
  {
    // Variables declared within a
    // parallel block are local to it
    int i=omp_get_thread_num(),
        j=omp_get_max_threads();

    printf("Hello from thread %d of %d\n",i,j);
  }
}
```

## Quick OpenMP example #3

```
#include <cstdio>
#include <cmath>

int main() {
  double a[1024];

  // Since each entry of the array can
  // be filled in separately, this loop
  // can be parallelized
#pragma omp parallel for
  for(int i=0;i<1024;i++) {
    a[i]=sqrt(double(i));
  }
}
```

## A practical OpenMP example

Computer demo: Extending the Ridders' method code to use multithreading

## An important point

By default, OpenMP programs run with all available threads on the machine

Some multicore workstations might have, *e.g.*, 64 threads available. You probably don't want all of them—often you should aim for a happy medium depending on the size of the workload (this will be explored on Homework 1)

Option 1: Run your program with

```
OMP_NUM_THREADS=4 ./openmp_example3
```

Option 2: Explicitly control with the `num_threads` keyword:

```
#pragma omp parallel for num_threads(4)
  for(int i=0;i<1024;i++) {
    a[i]=sqrt(double(i));
  }
```

## A numerical example: finite-difference simulation of the diffusion equation

Consider the diffusion equation

$$\frac{\partial u}{\partial t} = b\frac{\partial^2 u}{\partial x^2}$$

for the function $u(x, t)$ and diffusion constant $b > 0$. Discretize as $u_j^n \approx u(hj, n\Delta t)$ for timestep $\Delta t$ and grid spacing $h$. Explicit finite-difference scheme is

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = b\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}$$

or

$$u_j^{n+1} = u_j^n + \nu(u_{j+1}^n - 2u_j^n + u_{j-1}^n)$$

where $\nu = b\Delta t/h^2$. Stability achieved for $\nu < 1/2$.

## A numerical example: finite-difference simulation of the diffusion equation

Computer demo: Multithreading the diffusion equation simulation

## Summing numbers – a race condition

- A pitfall of shared memory parallel programming is the race condition, where two threads access the same memory, leading to unpredictable behavior
- The code below is legitimate if interpreted serially, but is unpredictable if run with multiple threads, due to conflicts between the loading/storage of c

```
#include <cstdio>

int main() {
    unsigned int c=0;
#pragma omp parallel for
    for(unsigned int i=0;i<1024;i++) {
        c=i*i+c;
    }
    printf("Sum=%u\n",c);
}
```

## A more subtle race condition

```
#include <cstdio>

int main() {
    int c[4096],d;

    // Fill table with square numbers
#pragma omp parallel for
    for(int i=0;i<4096;i++) {
        d=i*i;
        c[i]=d;
    }

    // Print out discrepancies
    for(int i=0;i<4096;i++)
    if(c[i]!=i*i) printf("%d %d\n",i,c[i]);
}
```

- d is shared among all threads. Its value will be continually overwritten. Values in the c array will be inconsistent.
- Practical tip: if you suspect problems, compare to the serial version with one thread

## Summing numbers – solution #1

```
#include <cstdio>

int main() {
    unsigned int c=0;
#pragma omp parallel for
    for(unsigned int i=0;i<1024;i++) {
        int d=i*i;
#pragma omp atomic
        c+=d;
    }
    printf("Sum=%u\n",c);
}
```

- OpenMP `atomic` keyword ensures the following statement is executed as an indivisible unit.
- Only works for very simple statements
- Fast, but not as fast as a regular operation

## Summing numbers – solution #2

```
#include <cstdio>

int main() {
    unsigned int c=0;
#pragma omp parallel for
    for(unsigned int i=0;i<1024;i++) {
        int d=i*i;
#pragma omp critical
        {
            if(i%100==0) printf("Processing %d\n",i);
            c+=d;
        }
    }
    printf("Sum=%u\n",c);
}
```

- OpenMP `critical` keyword marks a statement or block to only be processed by one thread at a time
- Unlike `atomic` it works for general blocks of code
- Comes with a performance penalty—threads will stand idle waiting for the block to become free

## Summing numbers – solution #3

```
#include <cstdio>

int main() {
    unsigned int c=0;
#pragma omp parallel for reduction(+:c)
    for(unsigned int i=0;i<1024;i++) {
        c+=i*i;
    }
    printf("Sum=%u\n",c);
}
```

- The `reduction` keyword marks a variable for accumulation across threads
- Cleanest solution for this scenario

## An illustrative example – happy numbers

- For a given positive number *n*, repeat the following process: replace *n* by the sum of the square of its digits.[10] If the process ends in 1, the number is happy. Otherwise it is sad.
- For example

  $$97 \to 9^2 + 7^2 = 130 \to 1^2 + 3^2 + 0^2 = 10 \to 1^2 + 0^2 = 1$$

  and hence 97 is a happy number
- It can be shown that all sad numbers end in a cycle involving 4
- Key point: the number of iterations varies depending on *n*. Could lead to load imbalance.
- OpenMP `schedule(dynamic)` option allows cases to be passed out dynamically to threads, instead of the cases being assigned *a priori*

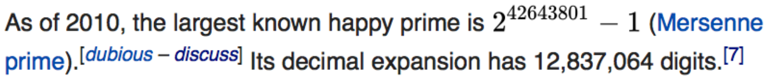
[10] In base 10.

## Happy number calculation

Computer demo: OpenMP dynamic for-loop calculation of happy numbers

## A challenge

From Wikipedia:



As of 2010, the largest known happy prime is $2^{42643801} - 1$ (Mersenne prime).[dubious – discuss] Its decimal expansion has 12,837,064 digits.[7]

- Problem 3 on HW1 involves constructing a representation of Mersenne primes
- Optional challenge: fix Wikipedia!

## A performance subtlety: false sharing

Computer demo: memory organization affects thread performance