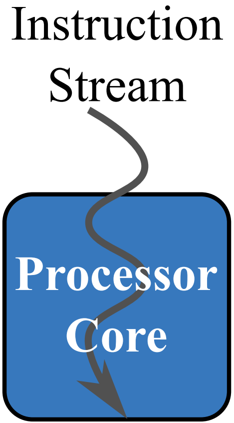
Fundamentals of Parallelism on Intel® Architecture

Week 3
Multithreading with OpenMP

May 2017

§1. Cores, Processes and Threads

Serial Processor, Serial Code

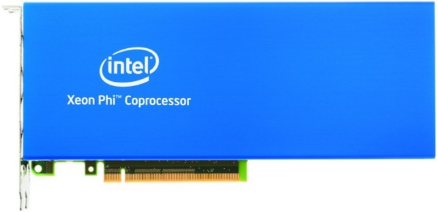


Computing Platforms

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

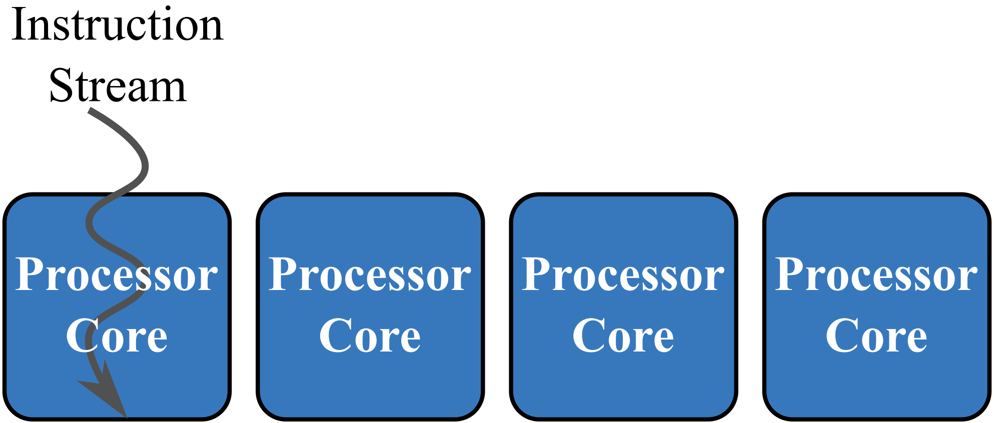
* socket and coprocessor versions

Knights Landing (KNL)

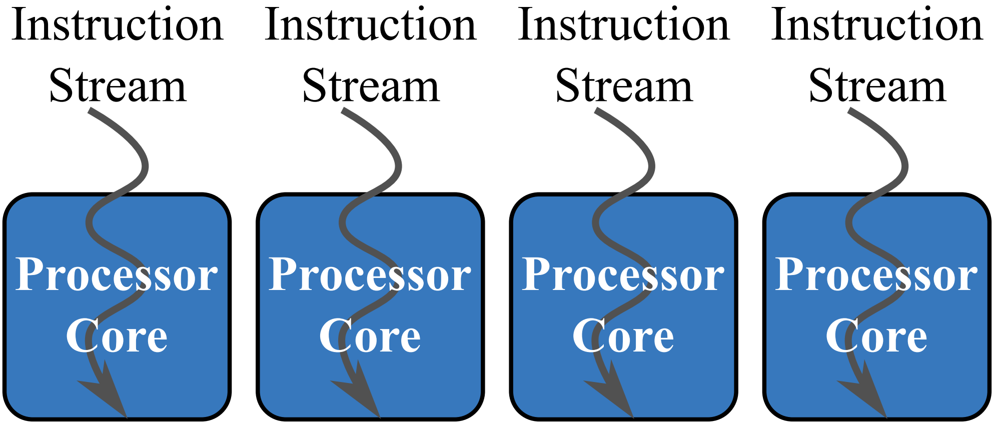
Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Parallel Processor, Serial Code

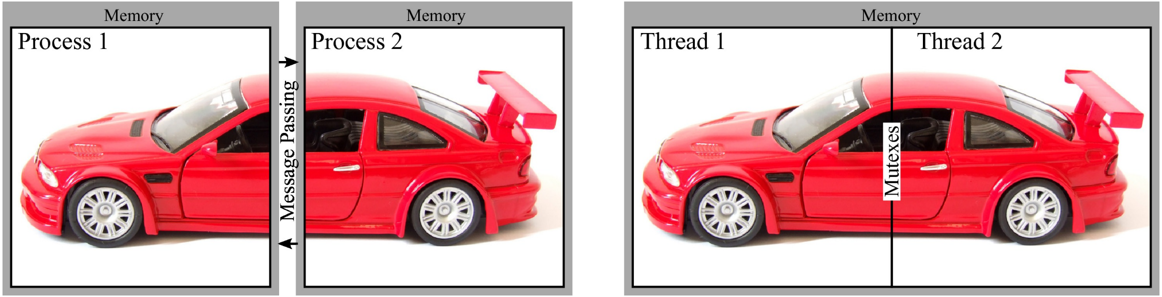


Parallel Processor, Parallel Code



Threads versus Processes

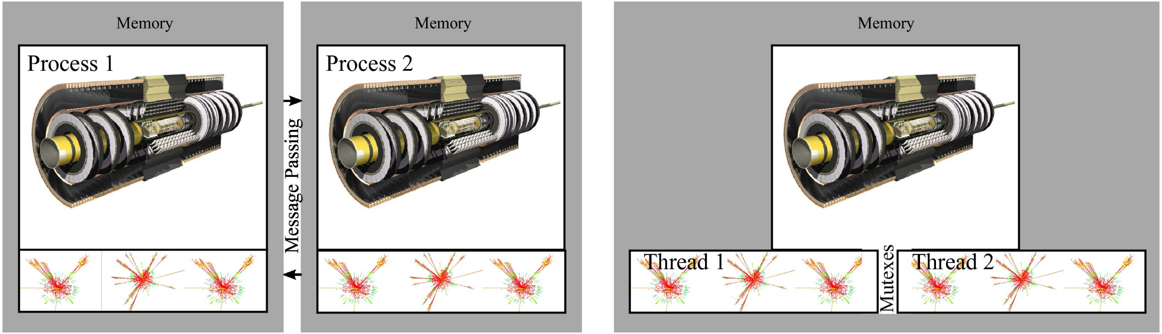
Option 1: Partitioning data set between threads/processes



Examples: computational fluid dynamics (CFD), image processing.

Threads versus Processes

Option 2: Sharing data set between threads/processes



Examples: particle transport simulation, machine learning (inference).

§2. Creating Threads

Threading Frameworks

Framework	Functionality
C++11 Threads	Asynchronous functions; only C++
POSIX Threads	Fork/join; C/C++/Fortran; Linux
Cilk Plus	Async tasks, loops, reducers, load balance; C/C++
TBB	Trees of tasks, complex patterns; only C++
OpenMP	Tasks, loops, reduction, load balancing, affinity, nesting, C/C++/Fortran (+SIMD, offload)

"Hello World" OpenMP Program

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

int main(){
  // This code is executed by 1 thread
  const int nt=omp_get_max_threads();
  printf("OpenMP with %d threads\n", nt);

#pragma omp parallel
  { // This code is executed in parallel
    // by multiple threads
    printf("Hello World from thread %d\n",
                omp_get_thread_num());
  }
}
```

- OpenMP = “Open Multi-Processing” = computing-oriented framework for shared-memory programming
- Threads – streams of instructions that share memory address space
- Distribute threads across CPU cores for parallel speedup

Compiling the "Hello World" OpenMP Program

```
vega@lyra% icpc -qopenmp hello_omp.cc
vega@lyra% export OMP_NUM_THREADS=5
vega@lyra% ./a.out
OpenMP with 5 threads
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2
Hello World from thread 4
```

OMP_NUM_THREADS controls number of OpenMP threads (default: logical CPU count)

§3. Variable Sharing

Control of Variable Sharing

Method 1: using clauses in pragma omp parallel (C, C++, Fortran):

```
int A, B; // Variables declared at the beginning of a function
#pragma omp parallel private(A) shared(B)
  {
    // Each thread has its own copy of A, but B is shared
  }
```

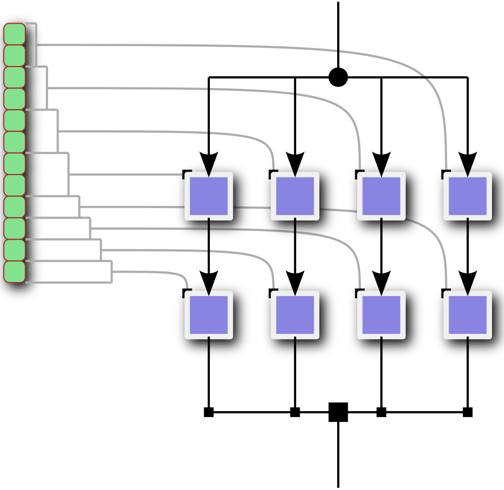
Method 2: using scoping (only C and C++):

```
int B; // Variable declared outside of parallel scope - shared by default
#pragma omp parallel
  {
    int A; // Variable declared inside the parallel scope - always private
    // Each thread has its own copy of A, but B is shared
  }
```

§4. Parallel Loops

Loop-Centric Parallelism: For-Loops in OpenMP

- Simultaneously launch multiple threads
- Scheduler assigns loop iterations to threads
- Each thread processes one iteration at a time



Parallelizing a `for`-loop.

Loop-Centric Parallelism: For-Loops in OpenMP

The OpenMP library will distribute the iterations of the loop following the `#pragma omp parallel for` across threads.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  printf("Iteration %d is processed by thread %d\n",
         i, omp_get_thread_num());
  // ... iterations will be distributed across available threads...
}
```

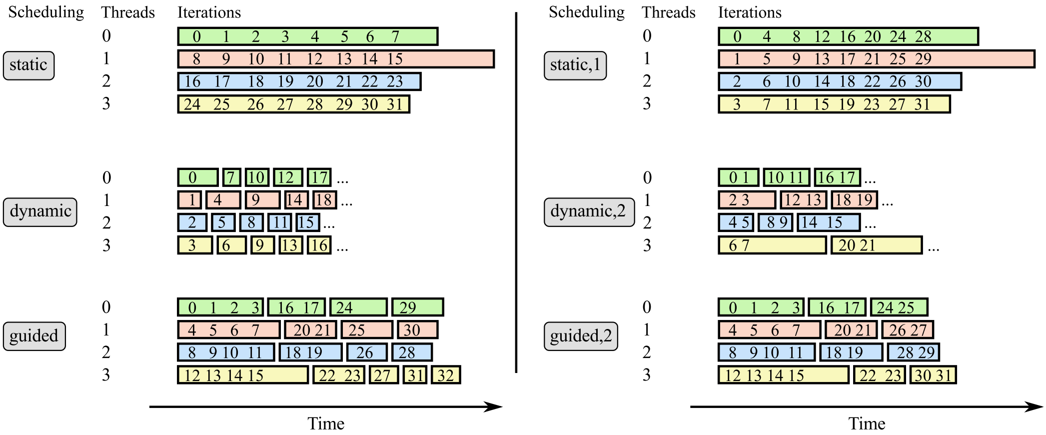
Loop-Centric Parallelism: For-Loops in OpenMP

```
#pragma omp parallel
{
  // Code placed here will be executed by all threads.

  // Alternative way to specify private variables:
  // declare them in the scope of pragma omp parallel
  int private_number=0;

#pragma omp for
  for (int i = 0; i < n; i++) {
    // ... iterations will be distributed across available threads...
  }
  // ... code placed here will be executed by all threads
}
```

Loop Scheduling Modes in OpenMP



§5. Example: Stencil Code

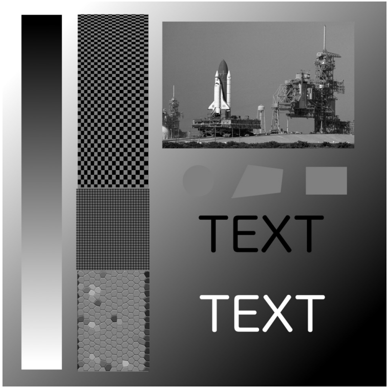
Stencil Operators

- Linear systems of equations
- Partial differential equations

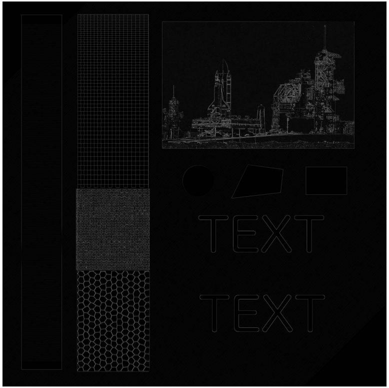
$$
\begin{aligned}
Q_{x,y} = \; & c_{00} P_{\mathrm{x-1,y-1}} + c_{01} P_{\mathrm{x,y-1}} + c_{02} P_{\mathrm{x+1,y-1}} + \\
& c_{10} P_{\mathrm{x-1,y}} + c_{11} P_{\mathrm{x,y}} + c_{12} P_{\mathrm{x+1,y}} + \\
& c_{20} P_{\mathrm{x-1,y+1}} + c_{21} P_{\mathrm{x,y+1}} + c_{22} P_{\mathrm{x+1,y+1}}
\end{aligned}
$$

Fluid dynamics, heat transfer, image processing (convolution matrix), cellular automata.

Edge Detection



$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \rightarrow$$

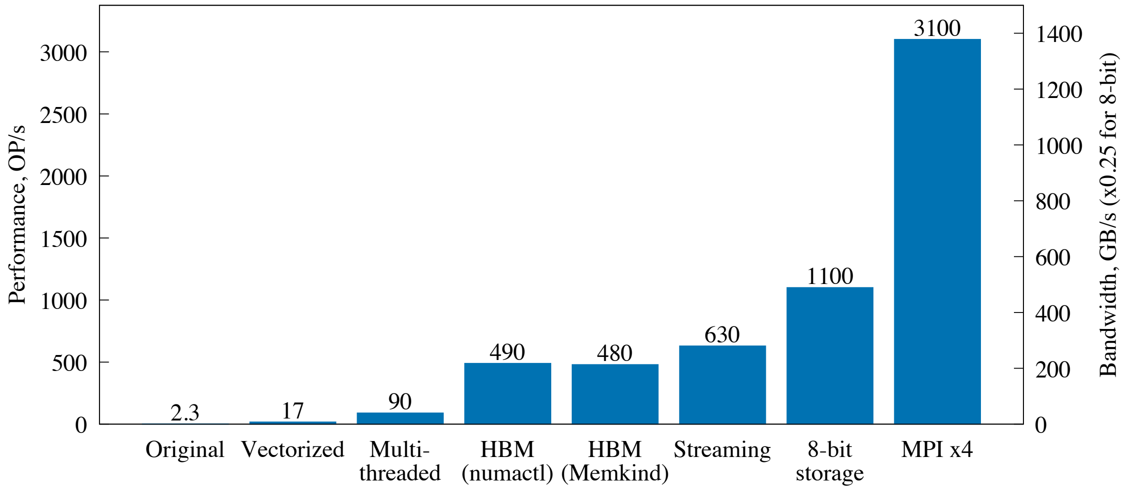


Multithreading

```
user@vega% icpc -c -qopenmp -xMIC-AVX512 stencil.cc
```

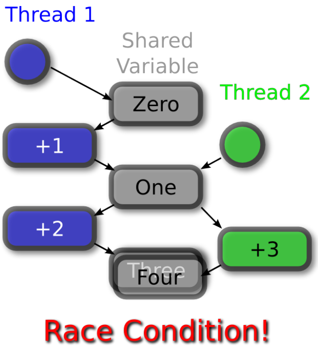
```
#pragma omp parallel for
for (int i = 1; i < height-1; i++)
#pragma omp simd
 for (int j = 1; j < width-1; j++)
  out[i*width + j] =
  -in[(i-1)*width + j-1] -   in[(i-1)*width + j] - in[(i-1)*width + j+1]
  -in[(i  )*width + j-1] + 8*in[(i  )*width + j] - in[(i  )*width + j+1]
  -in[(i+1)*width + j-1] -   in[(i+1)*width + j] - in[(i+1)*width + j+1];
```

Performance



§6. Data Races and Mutexes

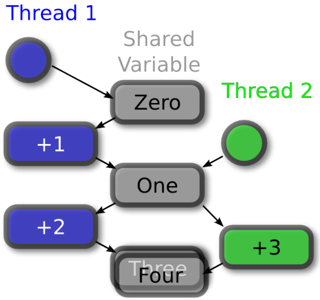
Race Conditions and Unpredictable Program Behavior



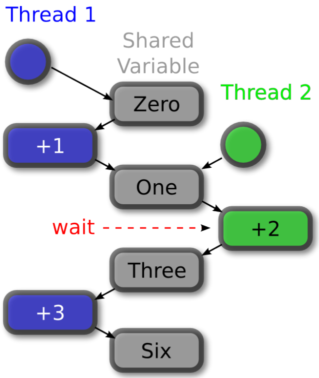
- Occurs when 2 or more threads access the same memory address, and at least one of these accesses is for writing

```c
int total = 0;
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  // Race condition
  total = total + i;
}
```

Mutexes



Race Condition!



- Mutual exclusion conditions (mutexes) protect data races by serializing code.

Mutexes in OpenMP

```
#pragma omp parallel
  {
    // parallel code
#pragma omp critical
    {
      // protected code
      // multiple lines
      // many variables
    }
  }
```

```
#pragma omp parallel
  {
    // parallel code
#pragma omp atomic
      // protected code
      // one line
      // specific operations
      // on scalars
      total += i;
  }
```

Good parallel codes minimize the use of mutexes.

§7. Parallel Reduction

Built-in reduction

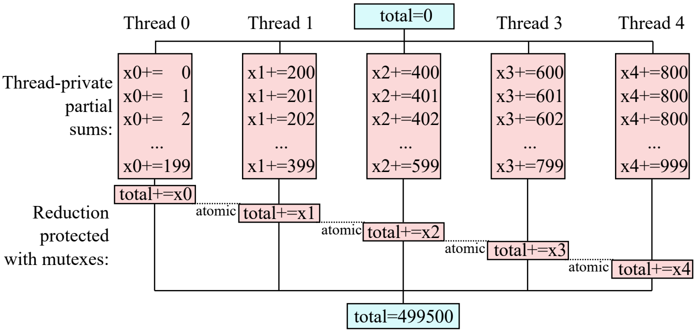
```
int total = 0;
#pragma omp parallel for reduction(+: total)
for (int i = 0; i < n; i++) {
  total += i
```

Reduction with Thread-Private Storage

```
int total = 0;
#pragma omp parallel
{
  int total_thr = 0;
#pragma omp for
  for (int i=0; i<n; i++)
    total_thr += i;

#pragma omp atomic
  total += total_thr;

}
```



§8. Example: Numerical Integration

Midpoint Rectangle Method

$$I(a,b) = \int_0^a f(x)\,\mathrm{d}x \approx \sum_{i=0}^{n-1} f\left(x_{\mathrm{i}+\frac{1}{2}}\right)\Delta x,$$

where

$$\Delta x = \frac{a}{n}, \qquad x_{\mathrm{i}+\frac{1}{2}} = \left(i + \frac{1}{2}\right)\Delta x.$$

Serial Implementation

```
const double dx = a/(double)n;
double integral = 0.0;

for (int i = 0; i < n; i++) {
  const double xip12 = dx*((double)i + 0.5);
  const double dI = BlackBoxFunction(xip12)*dx;

  integral += dI;
}
```

Unprotected Race Condition

```
const double dx = a/(double)n;
double integral = 0.0;
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  const double xip12 = dx*((double)i + 0.5);
  const double dI = BlackBoxFunction(xip12)*dx;

  integral += dI;
}
```

Excessive Use of Mutexes

```
const double dx = a/(double)n;
double integral = 0.0;
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  const double xip12 = dx*((double)i + 0.5);
  const double dI = BlackBoxFunction(xip12)*dx;
#pragma omp atomic
  integral += dI;
}
```

Built-in reduction

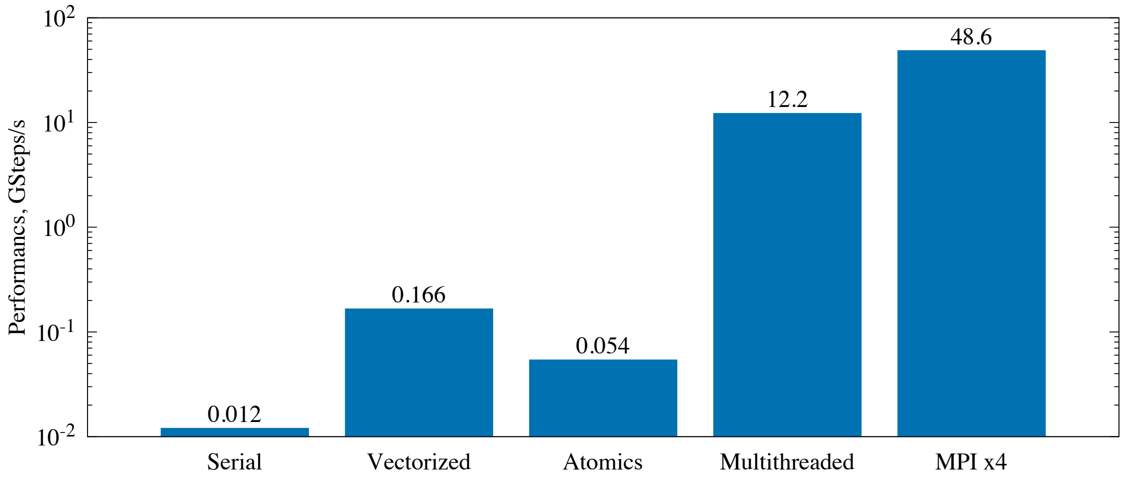
```
const double dx = a/(double)n;
double integral = 0.0;
#pragma omp parallel for reduction(+: integral)
for (int i = 0; i < n; i++) {
  const double xip12 = dx*((double)i + 0.5);
  const double dI = BlackBoxFunction(xip12)*dx;

  integral += dI;
}
```

Reduction with Thread-Private Storage

```
const double dx = a/(double)n;
double integral = 0.0;
#pragma omp parallel
{
  double integral_th = 0.0;
#pragma omp for
for (int i = 0; i < n; i++) {
    const double xip12 = dx*((double)i + 0.5);
    const double dI = BlackBoxFunction(xip12)*dx;
    integral_th += dI;
  }
#pragma omp atomic
  integral += integral_th;
}
```

Performance



§9. Learn more

OpenMP Concepts and Constructs

`#pragma omp parallel` – create threads

`#pragma omp for` – process loop with threads

`#pragma omp task/taskyield` – asynchronous tasks

`#pragma omp critical/atomic` – mutexes

`#pragma omp barrier/taskwait` – synchronization points

`#pragma omp sections/single` – blocks of code for individual threads

`#pragma omp flush` – enforce memory consistency

`#pragma omp ordered` – partial loop serialization

`OMP_*` – environment variables, omp_*() – functions

Click construct names for links to the OpenMP reference from the LLNL