Introduction to Programming with OpenMP

Outline

1) Parallell programming models
2) About OpenMP
3) OpenMP Directives
4) Data Scope
5) Runtime Library Routines and Environment Variables
6) SIMD using OpenMP
7) Using OpenMP

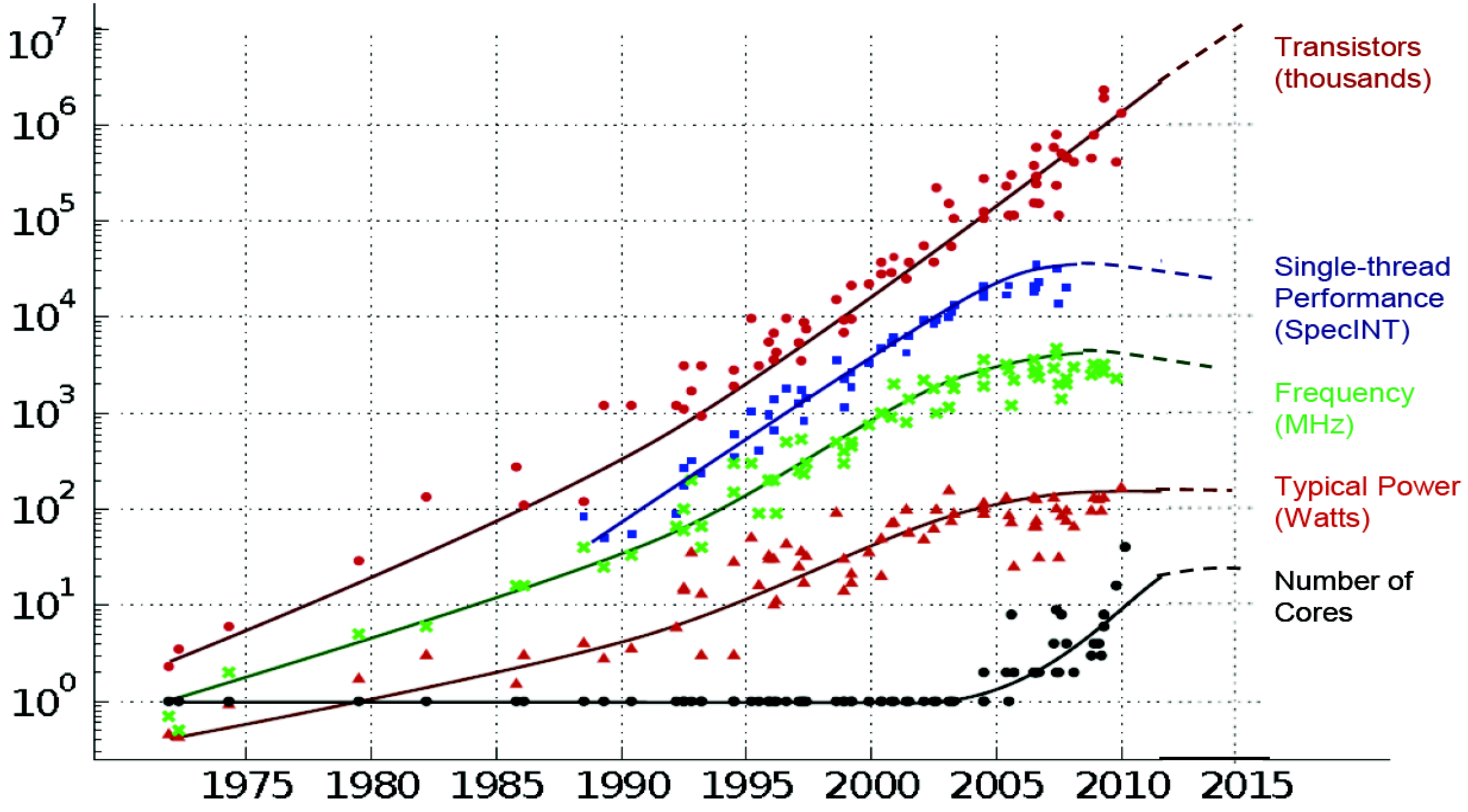
Parallel Programming

- Traditionally, a serial program views the problem at hand as follows:
    - Break the problem into discrete instructions
    - Execute instructions sequentially
        - May use hardware level parallelism (automatic) e.g.
            - Instruction Level Parallelism
            - out of order execution
            - hyperthreading
    - Execute on a single processor
    - In modern architectures a finite number of instructions can be executed by a CPU safely.

Parallel Programming– contd..

- In a parallel program
  - Break the problem into discrete parts which can run independent of each other
  - Each part can be sequential in nature
  - Instructions from each part run on different compute units
  - There should be some means of global process to coordinated and control the workflow

35 YEARS OF MICROPROCESSOR TREND DATA



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Source: karlrupp.net

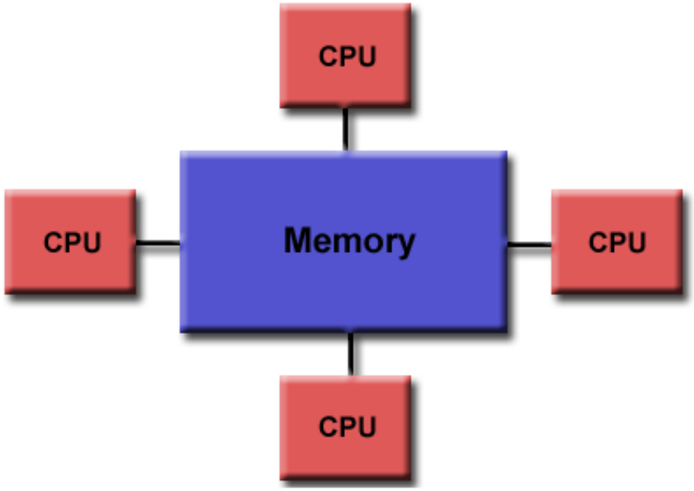
Parallel Architectures

For the memory as a compute resource, a machine can be seen as:

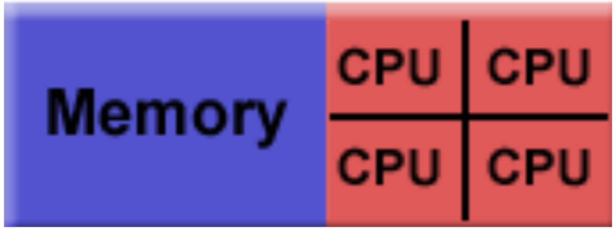
Shared memory architecture

- *Uniform Memory Access (UMA)*

- *Non-Uniform Memory Access (NUMA)*



CC-UMA

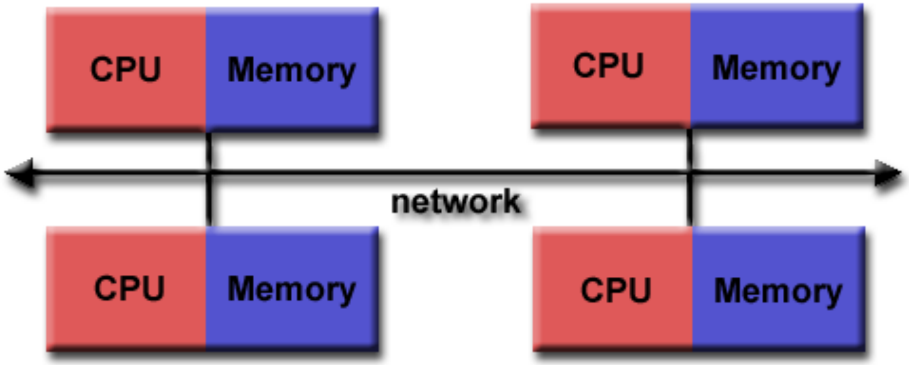


CC-NUMA

Source: LLNL Tutorial

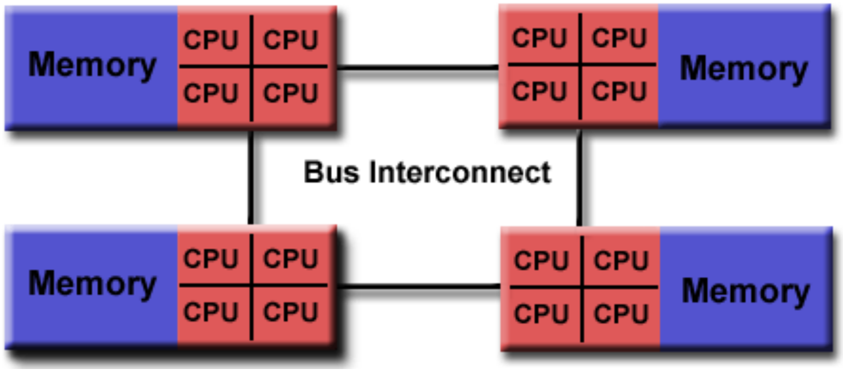
Parallel Architectures – contd..

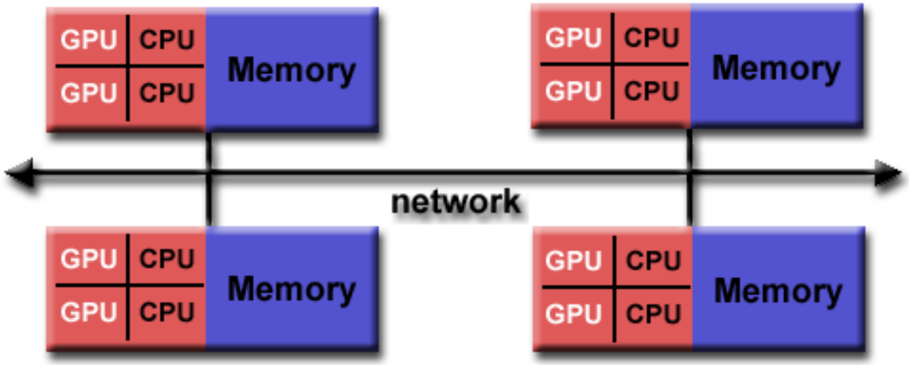
Distributed memory architecture



Source: LLNL Tutorial

Hybrid (distributed – shared) Memory architecture





Parallel Programming Models

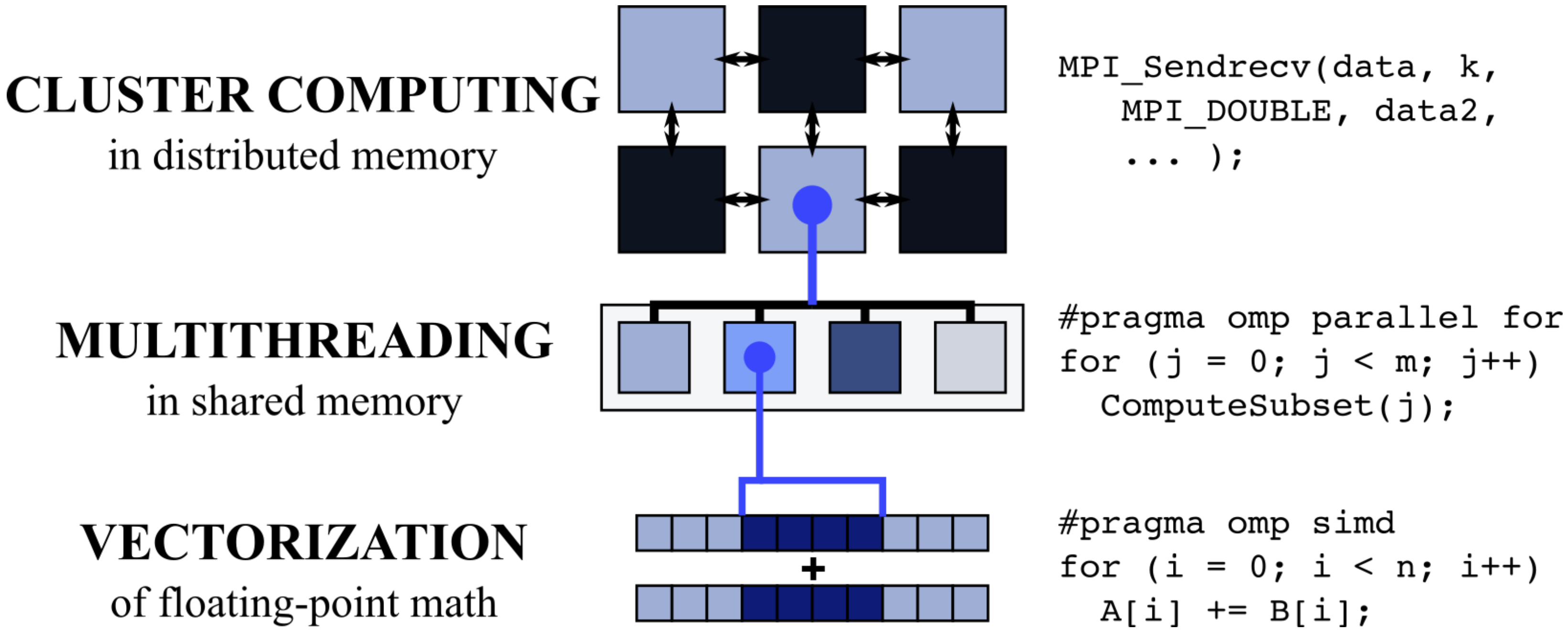
Depending on the architecture at hand, following are the commonly used programming models:

- Shared memory model
  - Without threads (e.g. using lock and semaphores)
  - Thread model (e.g. pthreads, OpenMP)
- Distributed Memory Model (Message Passing Interface MPI)
- Data parallel model (e.g. Partitioned Global memory address space PGAS)
- Hybrid (e.g. MPI+OpenMP or MPI+CUDA)

Parallel programming layers

How to get the maximum out of the modern microprocessor architecture?

- MPI across the nodes
- Multithreading on node
- Vectorization on core



CLUSTER COMPUTING
in distributed memory

```
MPI_Sendrecv(data, k,
   MPI_DOUBLE, data2,
   ... );
```

MULTITHREADING
in shared memory

```
#pragma omp parallel for
for (j = 0; j < m; j++)
  ComputeSubset(j);
```

VECTORIZATION
of floating-point math

```
#pragma omp simd
for (i = 0; i < n; i++)
  A[i] += B[i];
```

Source: Colfax

About OpenMP

OpenMP is an API for writing multithreaded applications to run on shared memory architecture

- Industry-standard shared memory programming (SMP) model
- Developed in 1997
- Simplifies the writing of multithreaded programs in Fortran, C and C++
- OpenMP Architecture Review Board (ARB) determines additions and updates to standard
- Latest version = 4.5 (5.0 rollout soon)

Advantages to OpenMP

- Parallelize small parts of application, one at a time (beginning with ***most time-critical*** parts)
- Can express simple or complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, so code is easy to read
- Single source code for OpenMP and non-OpenMP
  - Compilers simply ignore OpenMP directives if not compiled with it

OpenMP Programming Model

Application Programmer Interface (API) is a combination of

- Directives
- Runtime library routines
- Environment variables

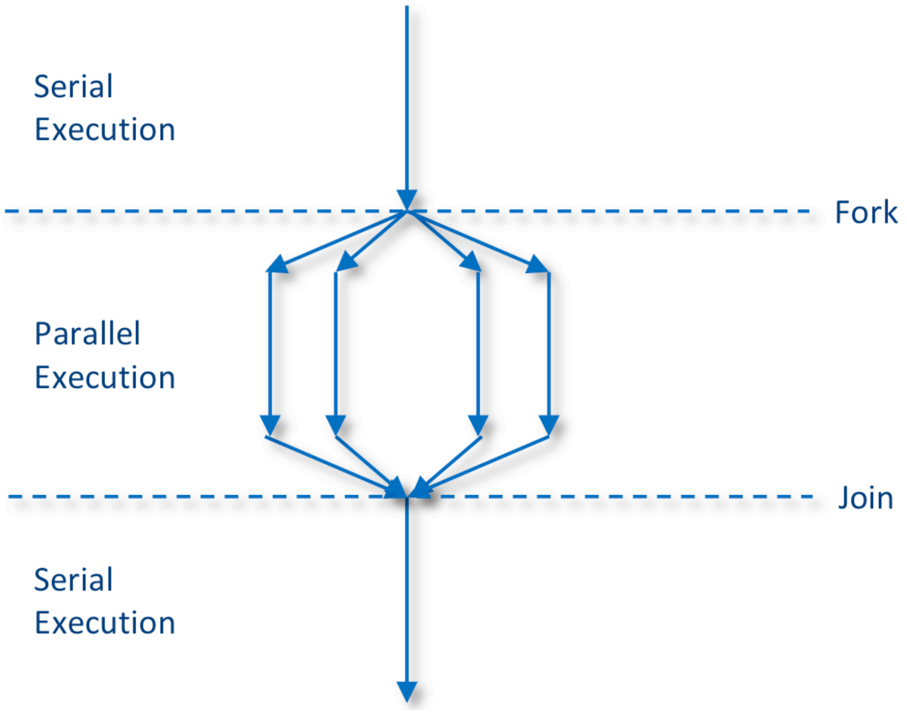
API falls into three categories

- Expression of parallelism (flow control)
- Data sharing among threads (communication)
- Synchronization (coordination or interaction)

Parallelism in OpenMP

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model

Fork join model

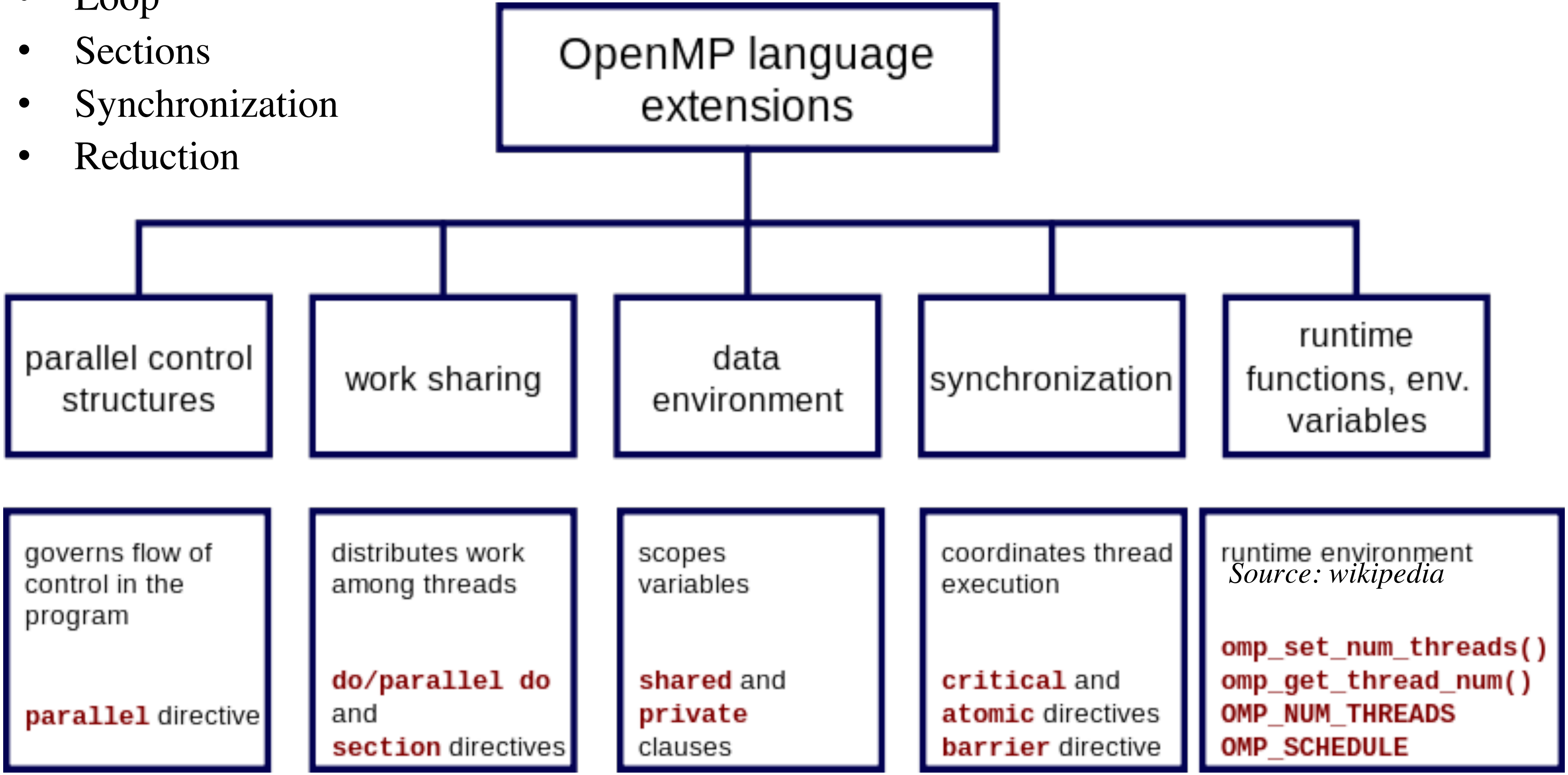


Source: Intel in-house training

OpenMP Directives

OpenMP Directives

- Syntax overview
- Parallel
- Loop
- Sections
- Synchronization
- Reduction



OpenMP language extensions

parallel control structures

work sharing

data environment

synchronization

runtime functions, env. variables

governs flow of control in the program

parallel directive

distributes work among threads

do/parallel do and **section** directives

scopes variables

shared and **private** clauses

coordinates thread execution

critical and **atomic** directives **barrier** directive

runtime environment

omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

Source: wikipedia

Syntax Overview: C/C++

- Basic format

```
#pragma omp directive-name [clause] newline
```

- All directives followed by newline
- Uses pragma construct (pragma = Greek for “thing done”)
- Case sensitive
- Directives follow standard rules for C/C++ compiler directives
- Use curly braces (not on pragma line) to denote scope of directive
- Long directive lines can be continued by escaping newline character with \

OpenMP Directives: Parallel

A block of code executed by multiple threads

C syntax:

```
#pragma omp parallel private(list)\
  shared (list)
{
   /* parallel section */
}
```

Simple Example in C

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

int main (int argc, char *argv[]) {
	int tid;
	printf(“Hello world from threads:\n”);
	#pragma omp parallel private(tid)
	{
		tid = omp_get_thread_num();
		printf(“<%d>\n”, tid);
	}
	printf(“I am sequential now\n”);
	return 0;
}
```

Accessing Exercises

All material in Github. Clone it:

```
git clone https://github.com/ADACS-Australia/HPC-Workshop.git
```

OpenMP Exercises, Solutions and Demos in:

```
cd HPC-Workshop/Introduction_to_OpenMP
```

Exercises and Solution directories contain:

- C and Fortran source codes
- Makefile and job submission scripts.

Exercise 1: OpenMP HelloWorld

Run an OpenMP hello world program in directory Exercise1

```
> cd HPC-Workshop/Introduction_to_OpenMP/Exercises/ex1/c
```

To build a executable run:

```
> make
```

To submit the a job to scheduler:

```
> sbatch hello-omp.slurm
```

Examine the output in `slurm-xxxxx.out`

Notice the thread order?

Output (Simple Example)

Output 1

```
Hello world from threads:
<0>
<1>
<2>
<3>
<4>
I am sequential now
```

Output 2

```
Hello world from threads:
<1>
<2>
<0>
<4>
<3>
I am sequential now
```

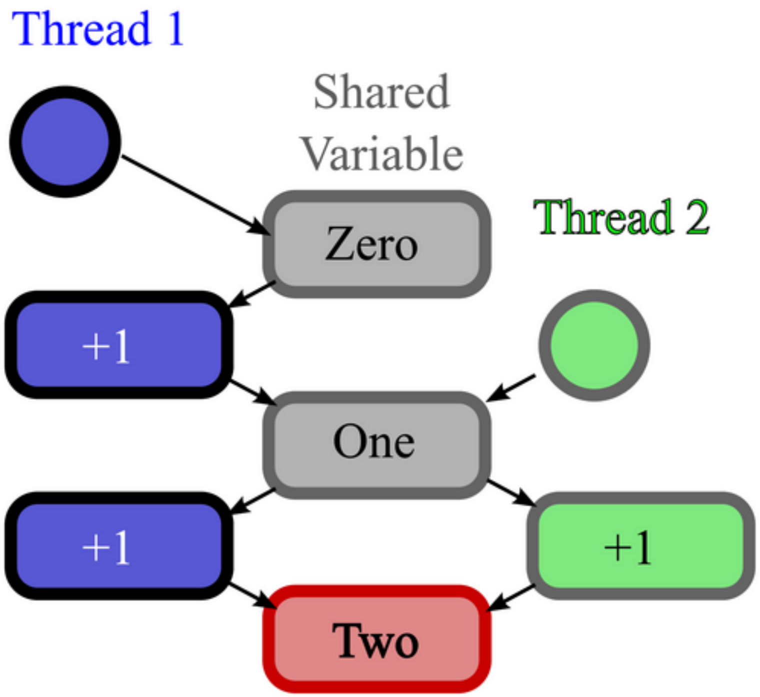
Order of execution is scheduled by OS!!!!!!

OpenMP Directives

Synchronization

Synchronization

- Sometimes order of thread execution matters
  - Maybe one part depends on another part being completed
  - Maybe only one thread need execute a section of code



Race Condition!

Synchronization

Solution: Synchronization directives

Mutexes

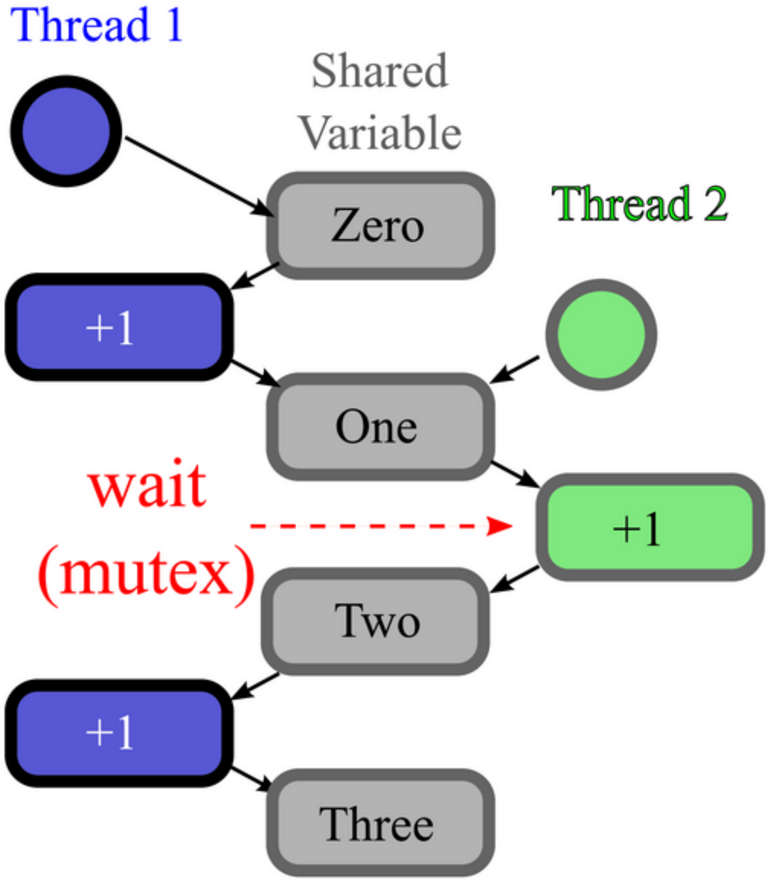
Critical , Atomic

Explicit Synchronization

Barrier

Worksharing

Single, Master



Source: Colfax

Synchronization: Mutexes

Critical

- One of the two supported Mutexes
- Specifies section of code that must be executed by only one thread at a time
- Syntax:

```
#pragma omp critical (name)
```

- Names are global identifiers – critical regions with same name are treated as same region

Synchronization: Mutexes

Task farming example:

If x and y are tabulated in separate list then it is important to protect each list on thread level. ---

```
int read_list(float* a, int type)
void work(int index, float* a)

void main(){

#pragma omp parallel shared(x, y)\
 private(ix_next, iy_next)
   {
    #pragma omp critical (xaxis)
     ix_next = read_list(&x,0);
     work(ix_next, &x);

    #pragma omp critical (yaxis)
     iy_next = read_list(&y,1);
     work(iy_next, &y);
   }
}
```

Synchronization: Mutexes

- Atomic
  - Specifics updating a variable atomically
  - Faster then Critical but limited in scope i.e. supported operations
  - Syntax: C/C++

```
#pragma omp atomic (read | write | capture | update)
         x=x+1
```

Faster than critical but supports limited arithmetic operations

Synchronization: Explicit

Barrier

- Synchronizes all threads
- a thread waits on a barrier unit all hit it
- All threads resume execution together

Syntax: `#pragma omp barrier`

Sequence of worksharing and barrier regions encountered must be the same for every thread

Worksharing Constructs

Single

- Enclosed code is to be executed by only one thread, whichever reaches it first.
- Useful for thread-unsafe sections of code (e.g., I/O)
- Syntax: `#pragma omp single`

Worksharing Construct

Master

- Enclosed code is to be executed by the master thread only, usually thread 0.
- All other threads skip this block of code.
- There is no implied barrier with this call

Syntax:

```
#pragma omp master
```

Worksharing construct

Sections

- Non-iterative work
- OS allocates a section to thead
- Thread runs section once
- Each **section** directives needs to be nested within **sections** directive

Syntax:

```
#pragma omp sections
  {
    #pragma omp section
    /* first section */
    #pragma omp section
    /* next section */
  }
```

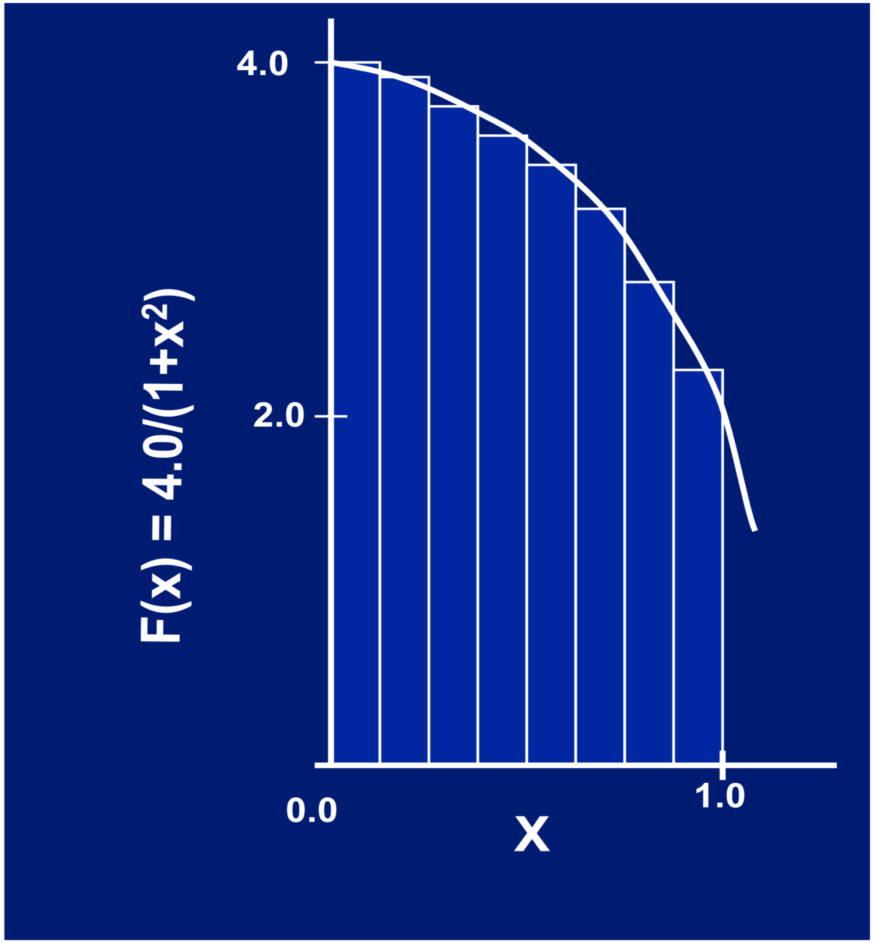
Sections: Simple Example

```
#include <omp.h>
#define N     1000
int main () {
  int i;
  double a[N], b[N];
  double c[N], d[N];
  /* Some initializations
*/
  for (i=0; i < N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
  }

  contd …
```

```
contd …
#pragma omp parallel \
shared(a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i < N; i++)
          c[i] = a[i] + b[i];
      #pragma omp section
        for (i=0; i < N; i++)
         d[i] = a[i] * b[i];
     }  /* end of sections */
   }  /* end of parallel section */
return 0;
}
```

Exercise 2:
Calculating value of Pi (Reimann sum)



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \Pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \Pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Source: Tim Mattson's presentation from SC08

Serial version

Algorithm:

- Decide the width (delta x) of each rectangle, lets call it the **step**
- Decide how many steps to take in total
- Evaluate the height of each rectangle by evaluating $f(x) = \frac{4}{1+x^2}$
- `Pi = step * (sum of heights)`

How to run:

```
> cd HPC-Workshop/Introduction_to_OpenMP/Exercises/ex2/c

> make

> sbatch pi-serial.slurm
```

Profiling Pi-serial

- Using Allinea MAP
  - Code links to the MAP's sampler library
  - Needs to be compiled with `-g` debug flag
  - Runs as a normal code and produces a profile
- You will need to download Allinea Forge's remote client to see the profile

https://developer.arm.com/products/software-development-tools/hpc/downloads/download-arm-forge/older-versions-of-remote-client-for-arm-forge

How to run a profile of `pi-serial`?

```
sbatch pi-serial-prof.slurm
```

Investigate profile and identify hotspot(s)

Data scoping

Variable scope in OpenMP Directives

Variable Scope

- By default, all variables are shared *except*
  - Certain loop index values – **private by default**
  - Local variables and value parameters within subroutines called within parallel region – **private**
  - Variables declared within lexical extent of parallel region – **private**

Default Scope Example

```
void caller(int *a, int n) {
int i,j,m=3;
#pragma omp parallel for
for (i=0; i<n; i++) {
   int k=m;
   for (j=1; j<=5; j++) {
      callee(&a[i], &k, j);
   }
}

void callee(int *x, int *y, int z)
{
   int ii;
   static int count;
   count++;
   for (ii=1; ii<z; ii++) {
      *x = *y + z;
   }
}
```

Var	Scope	Comment
a	shared	Declared outside parallel construct
n	shared	same
i	private	Parallel loop index
j	shared	Sequential loop index
m	shared	Declared outside parallel construct
k	private	Automatic variable/parallel region
x	private	Passed by value
*x	shared	(actually a)
y	private	Passed by value
*y	private	(actually k)
z	private	(actually j)
ii	private	Local stack variable in called function
count	shared	Declared static (like global)

Variable Scope

- Good programming practice: explicitly declare scope of all variables
- This helps you as programmer understand how variables are used in program
- Reduces chances of data race conditions or unexplained behavior – Also use thread synchronization

Variable Scope: Shared

Syntax: `shared(`*`list`*`)`

One instance of shared variable, and each thread can read or modify it

WARNING: watch out for multiple threads simultaneously updating same variable, or one reading while another writes

Example

```
counter =0;
#pragma omp parallel for shared(a)firstprivate(counter)
for (i = 0; i < N; i++) {
  a[i] += i;
  counter ++; // may be use omp critical or atomic here??
}
```

Variable Scope: race condition

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
  if (a[i] == b[i]) {
    n_eq++;
  }
}
```

- **All variables shared by default here except index of for loop**
- **n_eq will not be correctly updated**
- Instead, put **n_eq++**; in critical block (slow) or introduce private variable **my_n_eq**, then update **n_eq** in critical block after loop (faster)

Variable Scope: Private

Syntax: private(*list*)

Gives each thread its own copy of variable

Example

```
#pragma omp parallel private(i, my_n_eq)
    {
    my_n_eq=0;
    #pragma omp for
      for (i = 0; i < N; i++) {
       if (a[i] == b[i])  my_n_eq++;
      }
      #pragma omp critical (update_sum)
      {
        n_eq+=my_n_eq;
      }
    }
```

Exercise 3
(use OpenMP – first attempt)

Introduce OpenMP to alleviate hot spot in serial version of Pi

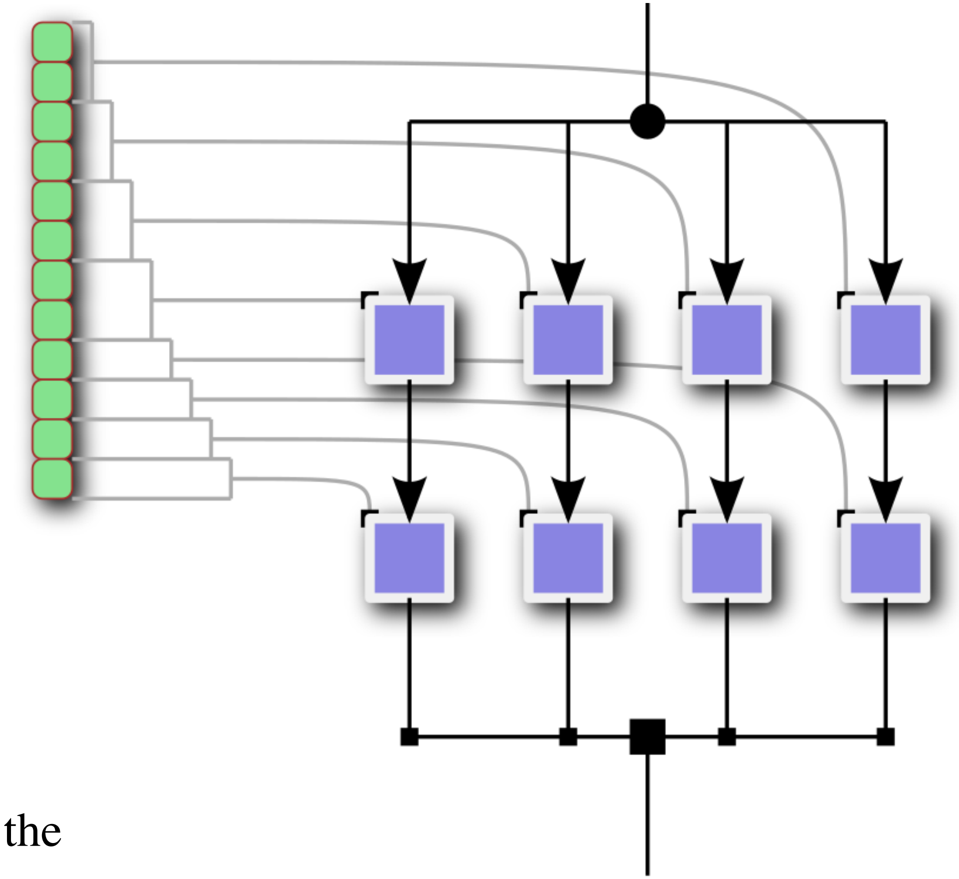
Edit pi-omp-v1.c in vi editor

Hints:

- Focus on hot spots
- Use `pragma omp parallel` where you want to commence work sharing
- Synchronization may be required in places which have dependencies, (hint: `pi = (step * sum)` may need your attention if sum is a private variable to each thread)
- Use `omp_get_wtime()` function for time profiling.

Worksharing : Loop

Iterations of the loop following the directive are executed in parallel



Syntax:

```
#pragma omp for schedule(type
[,chunk]) private(list) \
shared(list) nowait
      {
         /* for loop */
      }
```

Type = {static, dynamic, guided, runtime}

If nowait specified, threads do not synchronize at the end of loop

Worksharing: Loop Scheduling

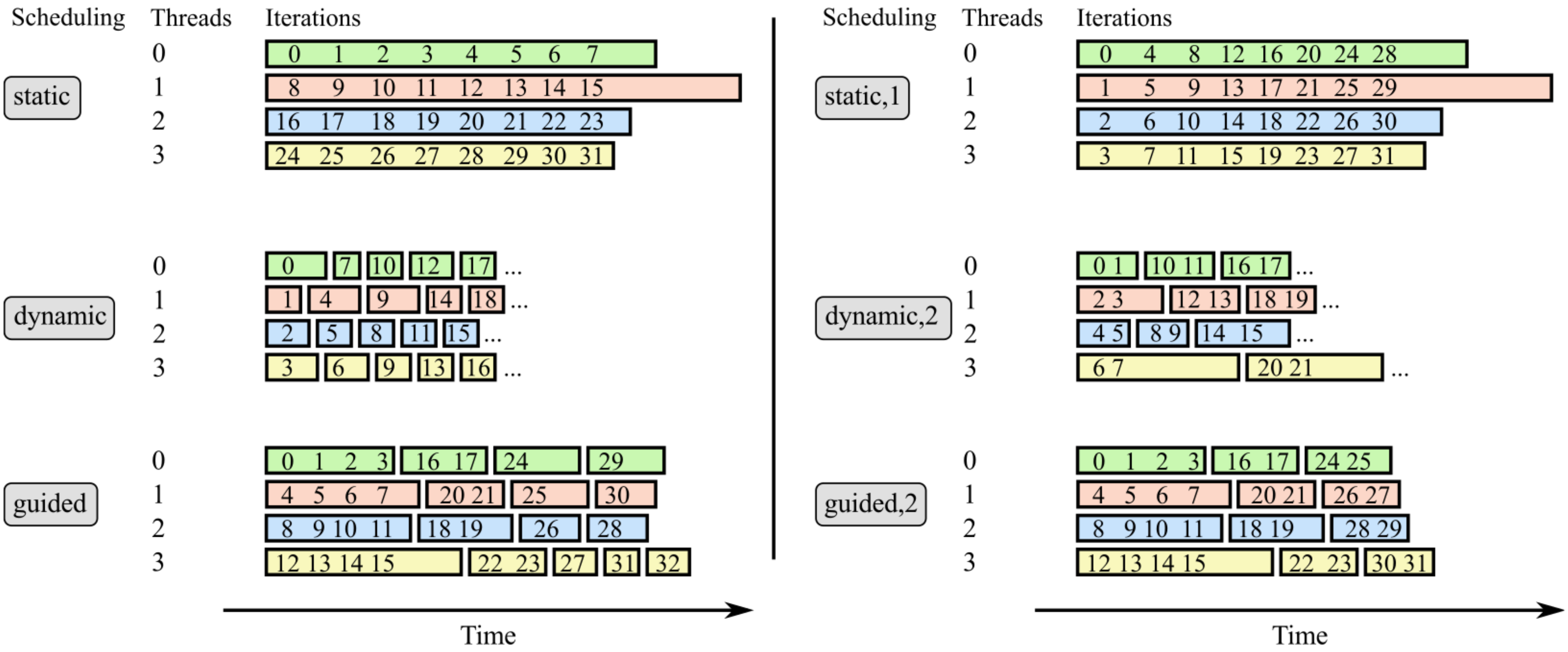
- Default scheduling determined by implementation
- Static
    - Chunk = Num_iterations / Num_threads
    - Statically assigned at the beginning of loop
    - Load imbalance may be an issue if iterations have different amounts of work
    - Low overhead
- Dynamic
    - Assignment of threads determined at runtime (round robin)
    - Each thread gets more work after completing current work
    - Load balance is possible
    - Introduces extra overhead

Worksharing: Loop Scheduling

Type	Chunks?	Chunk Size	# Chunks	Overhead	Description
`static`	no	N/P	P	Lowest	Simple static
`static`	yes	C	N/C	Low	Interleaved
`dynamic`	no	N/P	P	Medium	Simple dynamic
`dynamic`	yes	C	N/C	High	Dynamic, good loadbalance
`guided`	N/A	$\leq N/P$	$\leq N/C$	Highest	Dynamic optimized
`runtime`	Varies	Varies	Varies	Varies	Set by environment variable. E.g. OMP_SCHEDULE="dynamic,4 "
`auto`	Compiler	Compiler	Compiler		Set by compiler

Note: N = size of loop, P = number of threads, C = chunk size

Worksharing: Loop Scheduling



Which Loops Are Parallelizable?

Parallelizable

- Number of iterations known upon entry, and does not change
- Each iteration independent of all others
- No data dependence

Not Parallelizable

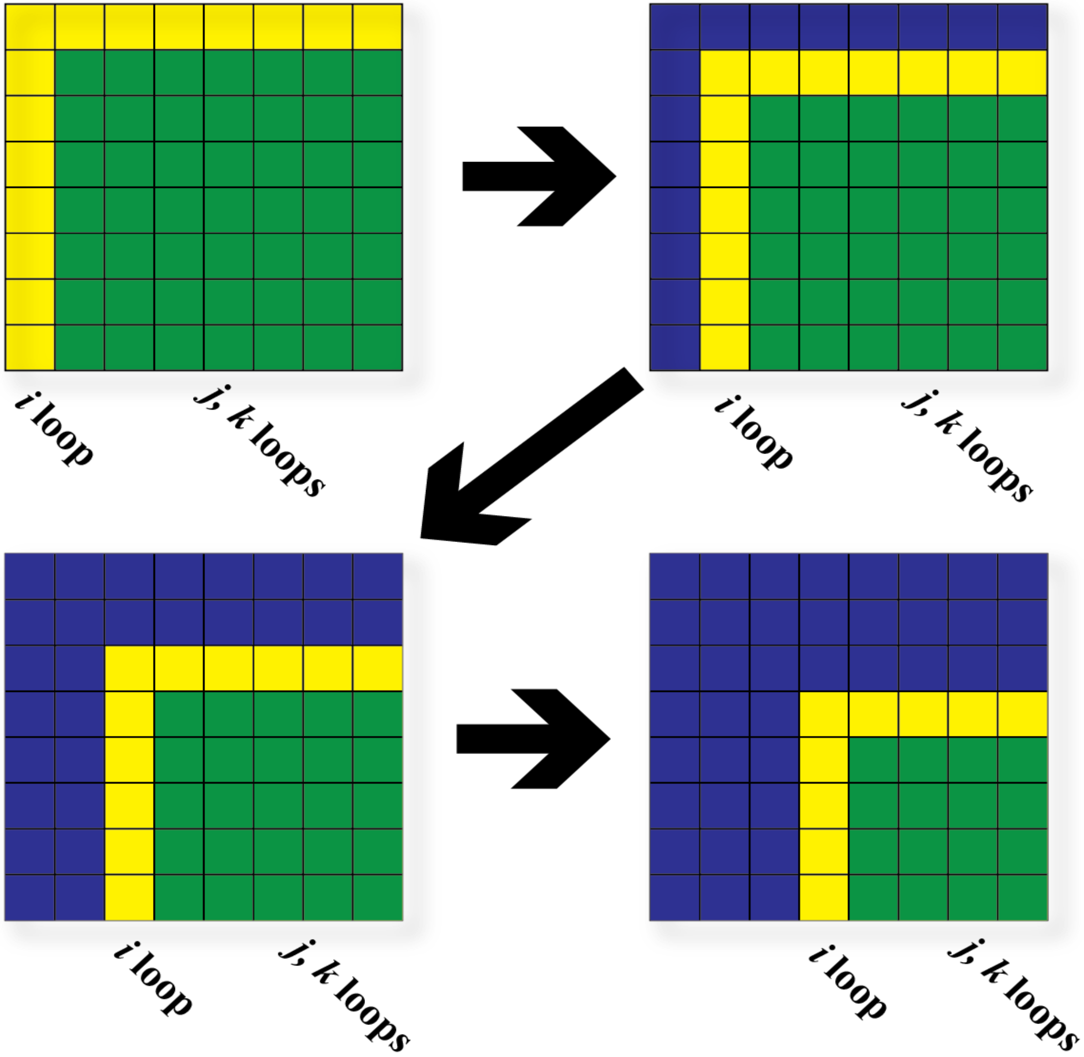
- Conditional loops (many while loops)
- Iterator loops (e.g., iterating over std:: list<…> in C++)
- Iterations dependent upon each other
- Data dependence

Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):
   x = A\b                          */


for (int i = 0; i < N-1; i++){
   for (int j = i; j < N; j++){
     double ratio = A[j][i]/A[i][i];
     for (int k = i; k < N; k++){
       A[j][k] -= (ratio*A[i][k]);
       b[j] -= (ratio*b[i]);
     }
   }
 }
```

Example: Parallelizable?



🟨 **Pivot row/ column**

🟩 **Updated entries**

🟦 **Unused entries**

Example: Parallelizable?

Outermost Loop (**i**):

- **N-1 iterations**
- Iterations depend upon each other (values computed at step **i-1** used in step **i**)

Inner loop (**j**):

- **N-i iterations (constant for given i)**
- Iterations can be performed in any order

Innermost loop (**k**):

- **N-i iterations (constant for given i)**
- Iterations can be performed in any order

Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):
  x = A\b                          */

for (int i = 0; i < N-1; i++) {
#pragma omp parallel for shared(N,A,b,i) private(j,k)
for (int j = i; j < N; j++) {
     double ratio = A[j][i]/A[i][i];
     for (int k = i; k < N; k++) {
       A[j][k] -= (ratio*A[i][k]);
       b[j] -= (ratio*b[i]);
    }
  }
}
```

Note: can combine **`parallel`** and **`for`** into single **`pragma`** line

Loop: Simple Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000
int 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 */
  return 0;
}
```

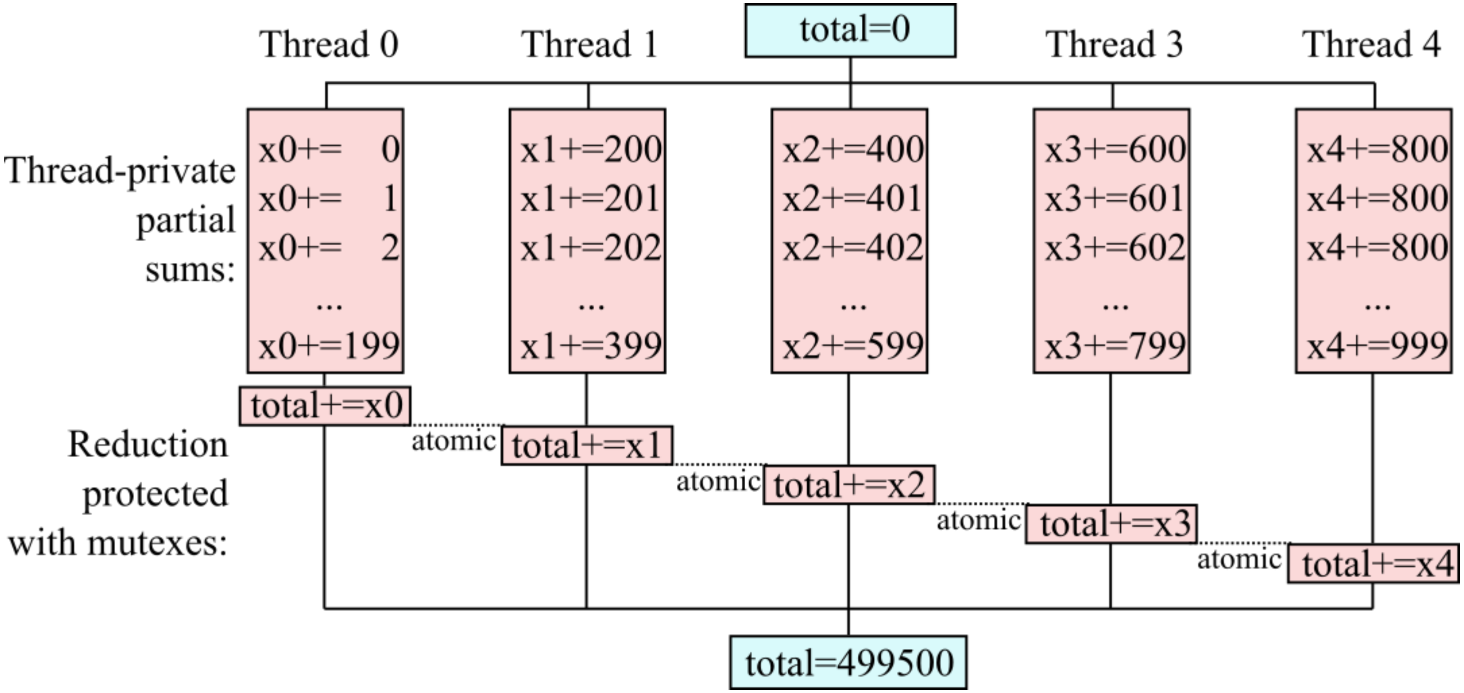
Worksharing: Reduction

Reduces list of variables into one, using operator (e.g., max, sum, product, etc.)

Syntax:
#pragma omp reduction(*op* : *list*)

!$omp reduction(*op* : *list*)
where ***op*** can be of the following:

op: +, -, *, &, ^, |, &&, or ||



total=0

Thread 0

Thread 1

Thread 3

Thread 4

Thread-private partial sums:

x0+= 0
x0+= 1
x0+= 2
...
x0+=199

x1+=200
x1+=201
x1+=202
...
x1+=399

x2+=400
x2+=401
x2+=402
...
x2+=599

x3+=600
x3+=601
x3+=602
...
x3+=799

x4+=800
x4+=800
x4+=800
...
x4+=999

Reduction protected with mutexes:

total+=x0

atomic total+=x1

atomic total+=x2

atomic total+=x3

atomic total+=x4

total=499500

Exercise 4
(use OpenMP – second attempt)

Can we do better than exercise 3 version?

Edit pi-omp-v2. c in vi editor

Hints:

- Possibly applying **`reduction`** on variable `sum` ??

Runtime Library Routines and Environment Variables

OpenMP Runtime Library Routines

```
void omp_set_num_threads(int num_threads)
```

Sets number of threads used in next parallel region

Must be called from serial portion of code

```
int omp_get_num_threads()
```

Returns number of threads currently in team executing parallel region from which it is called

```
int omp_get_thread_num()
```

Returns rank of thread

```
0 ≤ omp_get_thread_num() < omp_get_num_threads()
```

OpenMP Environment Variables

Set environment variables to control execution of parallel code

OMP_SCHEDULE

- Determines how iterations of loops are scheduled
- E.g., setenv OMP_SCHEDULE ”dynamic, 4”

OMP_NUM_THREADS

- Sets maximum number of threads
- E.g., setenv OMP_NUM_THREADS 4

Using OpenMP

Conditional Compilation

- Can write single source code for use with or without OpenMP
- Pragmas are ignored
- What about OpenMP runtime library routines?
- **_OPENMP** macro is defined if OpenMP available: can use this to conditionally include **omp.h** header file, else redefine runtime library routines

Conditional Compilation:
Preprocessing

```
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif
…
int me = omp_get_thread_num();
…
```

Enabling OpenMP Directives

- Most standard compilers support OpenMP directives
- Enable using compiler flags

Compiler	Intel	PGI	GNU	Cray
Flag	`-openmp`	`-mp`	`-fopenmp`	`-h omp`

Note: For Cray compiler, OpenMP is enabled by default.
Disable OpenMP with -h noomp

Running Programs with OpenMP Directives

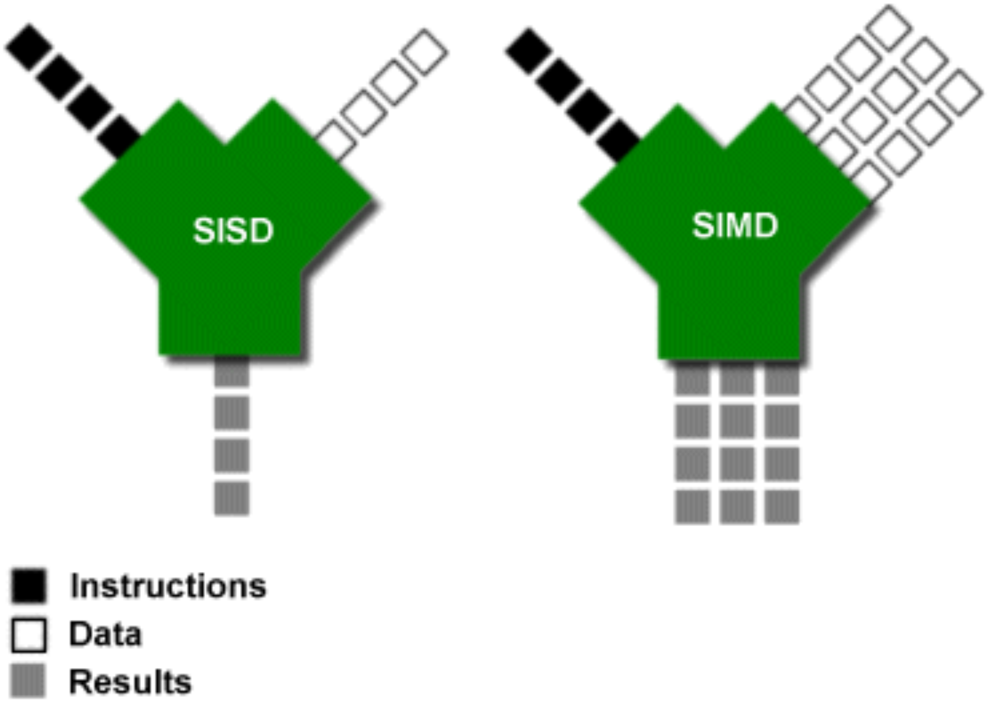
Set OpenMP environment variables in batch scripts (e.g., include definition of OMP_NUM_THREADS in script)

Example: to run a code with 8 MPI processes and 3 threads/MPI process:

```
export OMP_NUM_THREADS=3
srun -n 8 ./myprog
```

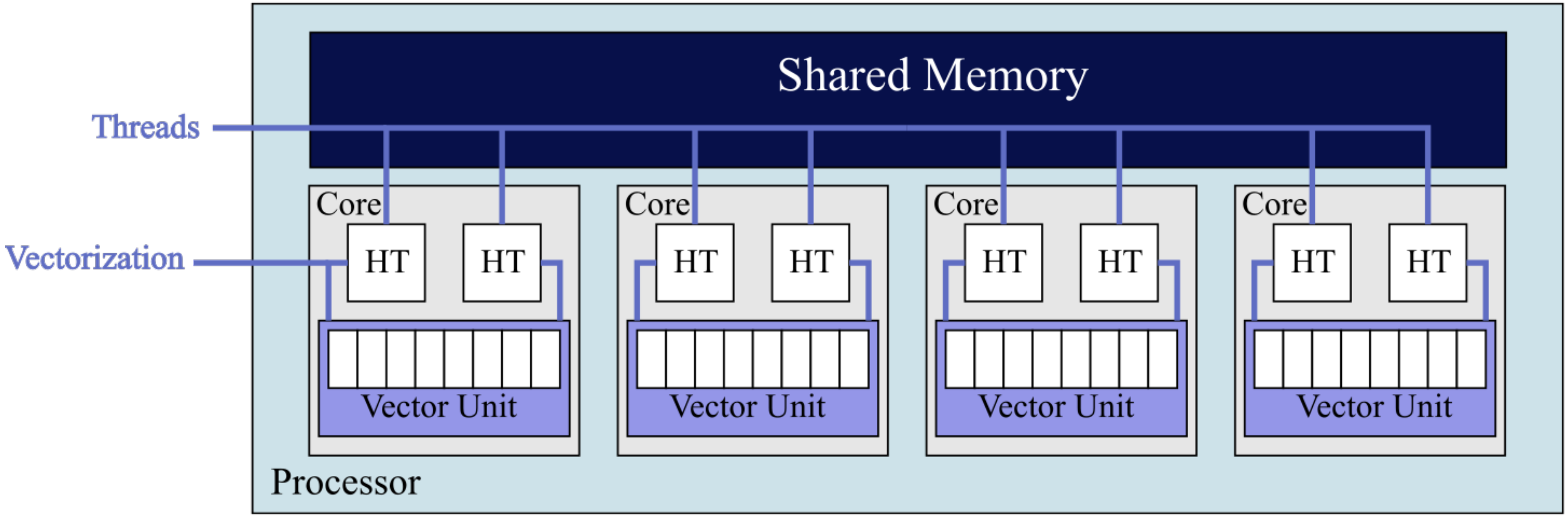
Single Instruction Multiple Data

SIMD



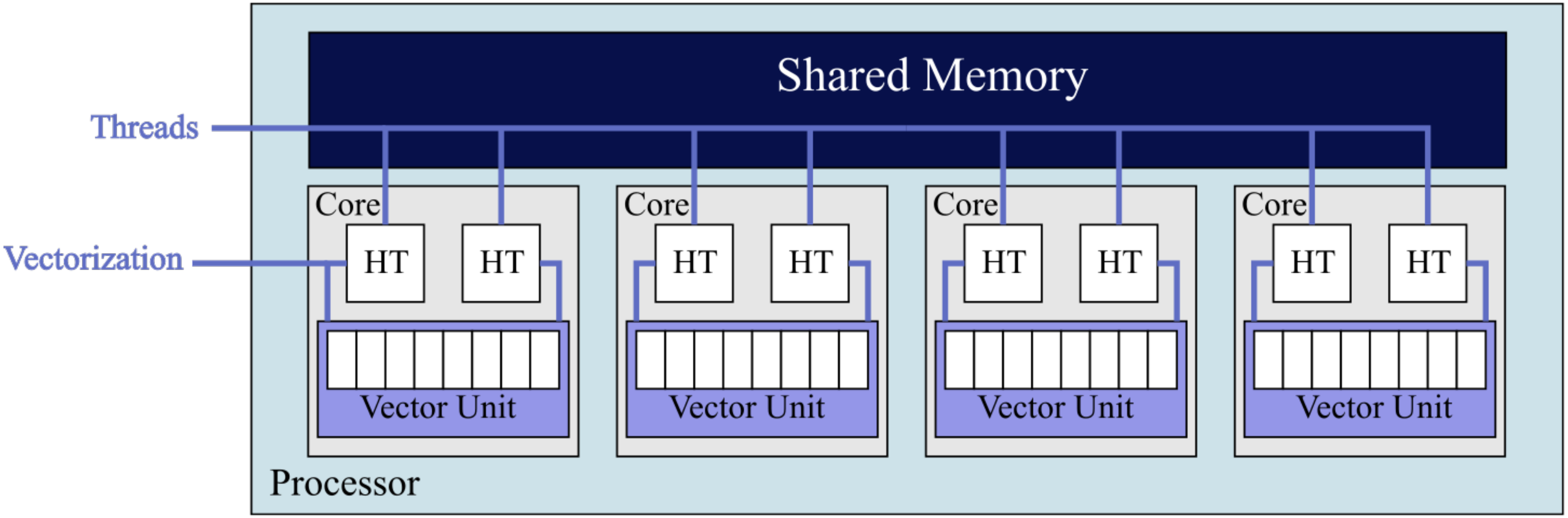
Node level parallelism

- The vector length increasing
- Threads implement – Multiple Instructions Multiple Data (MIMD)
- Vector Units implements – Single Instruction Multiple Data (SIMD)



Node level parallelism

- The catch
- Developer has to write good code – performance not free
  - Allocate memory as you access it – improve data locality – improve cache layout

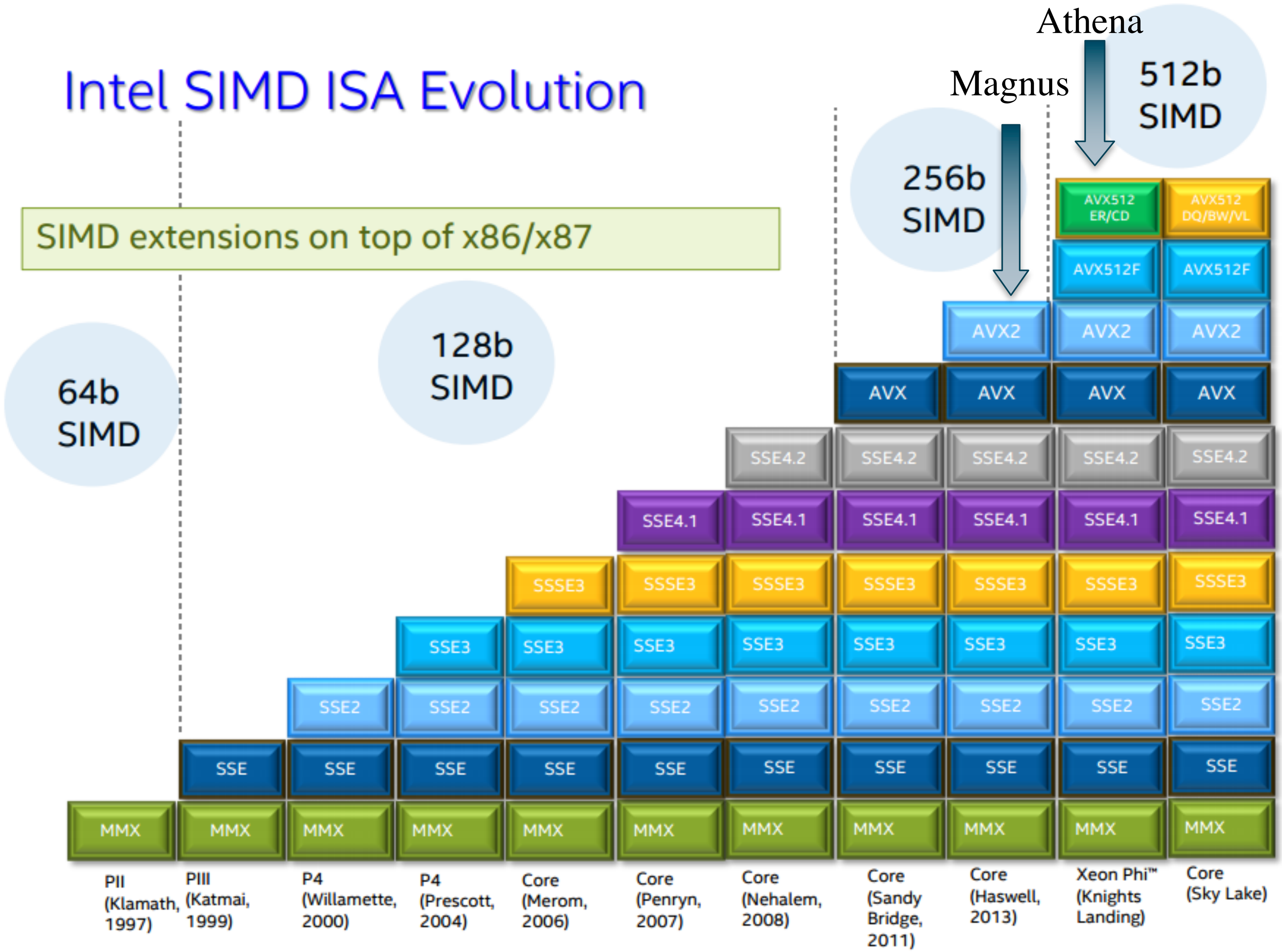


ADACS

PUTING SERVICES

Intel SIMD ISA Evolution

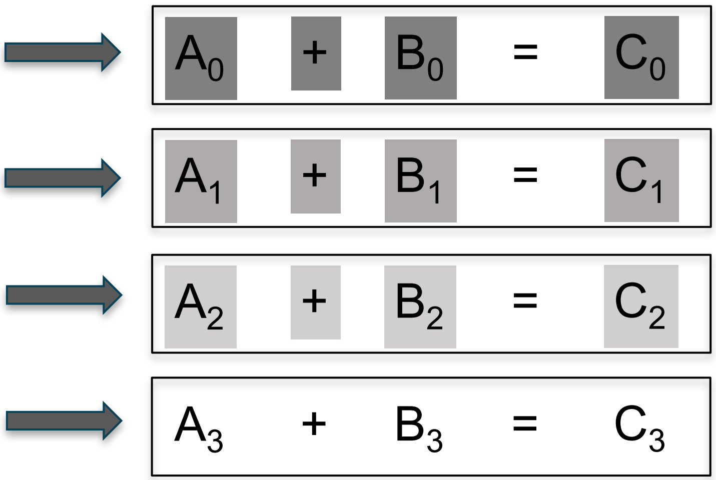
SIMD extensions on top of x86/x87



PII (Klamath, 1997)	PIII (Katmai, 1999)	P4 (Willamette, 2000)	P4 (Prescott, 2004)	Core (Merom, 2006)	Core (Penryn, 2007)	Core (Nehalem, 2008)	Core (Sandy Bridge, 2011)	Core (Haswell, 2013)	Xeon Phi™ (Knights Landing)	Core (Sky Lake)
									AVX512 ER/CD	AVX512 DQ/BW/VL
									AVX512F	AVX512F
								AVX2	AVX2	AVX2
							AVX	AVX	AVX	AVX
						SSE4.2	SSE4.2	SSE4.2	SSE4.2	SSE4.2
					SSE4.1	SSE4.1	SSE4.1	SSE4.1	SSE4.1	SSE4.1
				SSSE3	SSSE3	SSSE3	SSSE3	SSSE3	SSSE3	SSSE3
			SSE3	SSE3	SSE3	SSE3	SSE3	SSE3	SSE3	SSE3
		SSE2	SSE2	SSE2	SSE2	SSE2	SSE2	SSE2	SSE2	SSE2
	SSE	SSE	SSE	SSE	SSE	SSE	SSE	SSE	SSE	SSE
MMX	MMX	MMX	MMX	MMX	MMX	MMX	MMX	MMX	MMX	MMX

Scalar vs Vector

Solving $A[i] + B[i] = C[i]$

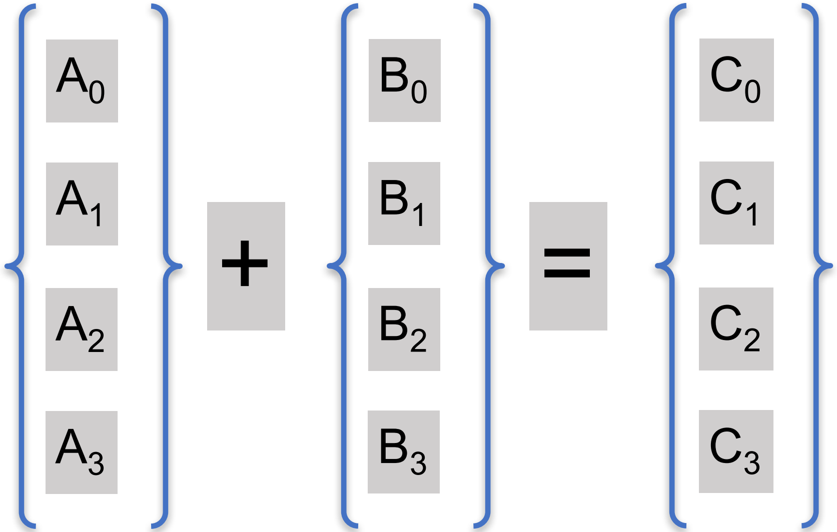


$A_0 + B_0 = C_0$

$A_1 + B_1 = C_1$

$A_2 + B_2 = C_2$

$A_3 + B_3 = C_3$



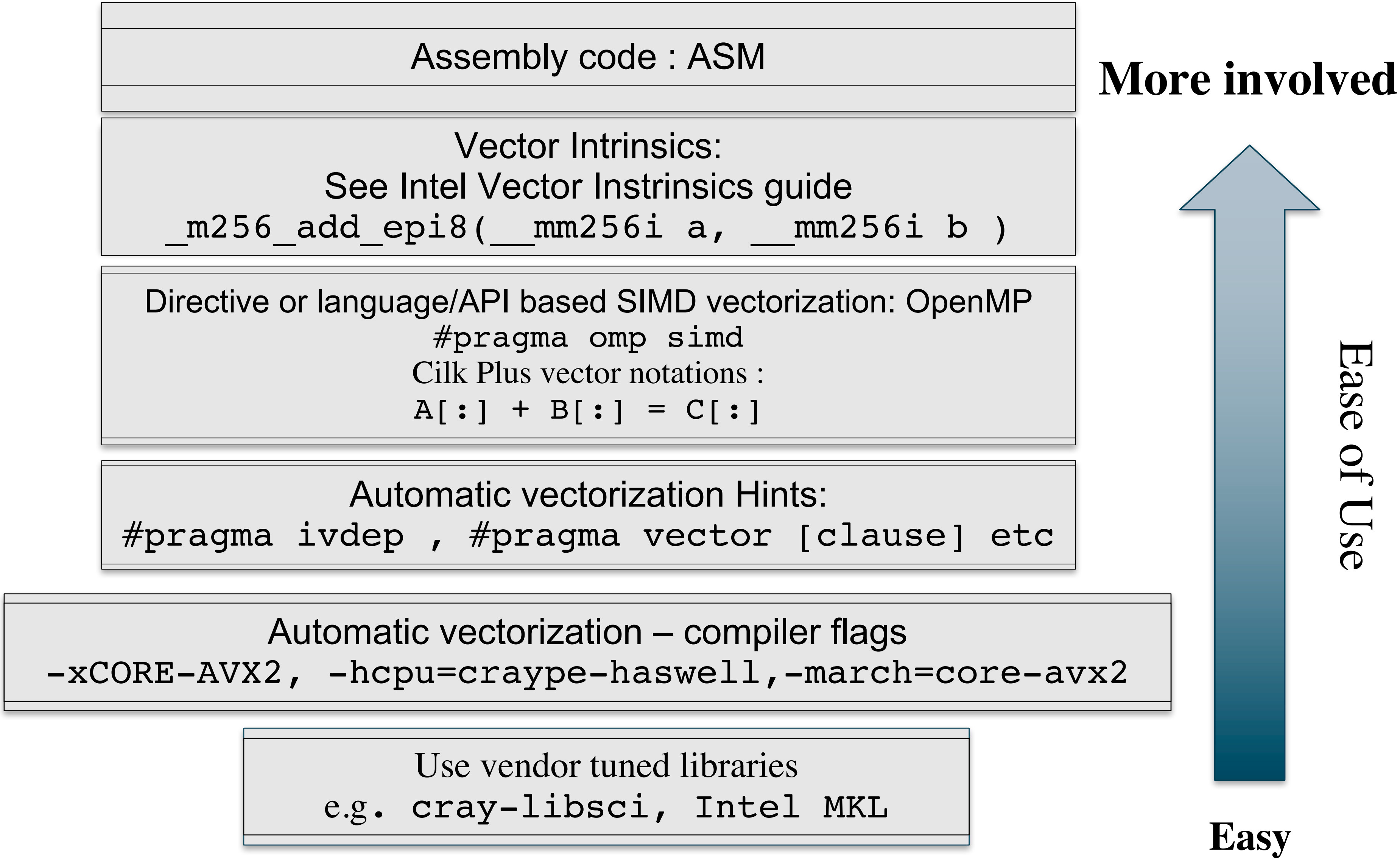
$\{A_0, A_1, A_2, A_3\} + \{B_0, B_1, B_2, B_3\} = \{C_0, C_1, C_2, C_3\}$

```
for (i=0; i<n; i++)
   C[i] = A[i] + B[i]
```

Supposing vector length = 4

```
for (i=0; i<n; i+=4)
   C[i] = A[i] + B[i]
```

Approaches to vectorization



More involved

Assembly code : ASM

Vector Intrinsics:
See Intel Vector Instrinsics guide
`_m256_add_epi8(__mm256i a, __mm256i b )`

Directive or language/API based SIMD vectorization: OpenMP
`#pragma omp simd`
Cilk Plus vector notations :
`A[:] + B[:] = C[:]`

Automatic vectorization Hints:
`#pragma ivdep , #pragma vector [clause] etc`

Automatic vectorization – compiler flags
`-xCORE-AVX2, -hcpu=craype-haswell,-march=core-avx2`

Use vendor tuned libraries
e.g. `cray-libsci, Intel MKL`

Easy

Utilizing SIMD with OpenMP

Used to “enforce vectorization of loops”,which includes:

- Loops with SIMD-enabled functions
- Second innermost loops
- Failed vectorization due to compiler decision
- Where guidance is required (vector length, reduction, etc.)

SeeOpenMP reference for syntax; similar to Intel’s #pragma simd

SIMD loop construct

- Vectorize a loop nest (usually second inner most loops)
  - Cut loop into chunks that fit a SIMD vector register
  - No parallelization of the loop body
- Syntax :

```
#pragma omp simd [clause[[,] clause],...]
for-loops
```

SIMD loop construct :Clauses

`safelen (length)` : Maximum number of iterations that can run concurrently without

- breaking a dependence
- in practice, maximum vector length

`simdlen (length)` : Specify preferred length of SIMD registers used

- Must be less or equal to safelen if also present

`linear (list[:linear-step])` : The variable’s value is in relationship with the iteration number

- `xi = xorig + i * linear-step`

SIMD loop construct :Clauses

`aligned (list[:alignment])` : Specifies that the list items have a given alignment

- Default is alignment for the architecture

`collapse (n)` : Apply SIMD to next n nested loops. If nocollapse in present than only the immediate loop is vectorized.

Example : (IVOP)

Inner vectorize Outer Parallelize

```
const int N=128, T=4;
float A[N*N], B[N*N], C[T*T];
for (int jj = 0; jj < N; jj+=T) // Tile in j
  for (int ii = 0; ii < N; ii+=T) // and tile in i
#pragma omp simd // Vectorize outer loop
    for (int k = 0; k < N; ++k)  // long loop, vectorize it
          for (int i = 0; i < T; i++)  // Loop between ii and
ii+T. Instead of a loop between jj and jj+T, unrolling that
loop:
          {
          C[0*T + i] += A[(jj+0)*N + k]*B[(ii+i)*N + k];
          C[1*T + i] += A[(jj+1)*N + k]*B[(ii+i)*N + k];
          C[2*T + i] += A[(jj+2)*N + k]*B[(ii+i)*N + k];
          C[3*T + i] += A[(jj+3)*N + k]*B[(ii+i)*N + k];
          }
```

Example: SIMD reduction

```
float result(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp simd
    reduction(+:sum) for (int k=0; k<n;
k++) sum += a[k] * b[k];
    return sum;
}
```

Example: SIMD Enabled functions

```
// Compiler will produce 3 versions:
#pragma omp declare simd
float my_simple_add(float x1, float x2)
{
     return x1 + x2;
}


// May be in a separate file
#pragma omp simd
for (int i = 0; i < N, ++i)
{
     output[i] = my_simple_add(inputa[i], inputb[i]);
}
```

Acknowledgement

Some useful figures and examples are taken from the following sources

- Coflax Research
- Intel
- OpenMP.org

Questions ?