DAY 6: OPTIMIZATION ON PARALLEL INTEL® ARCHITECTURES

Lecture day 6

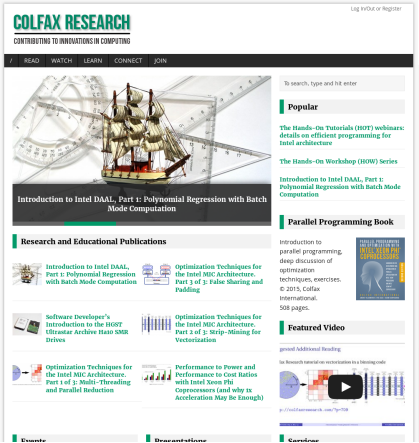
Ryo Asai
Colfax International — *colfaxresearch.com*

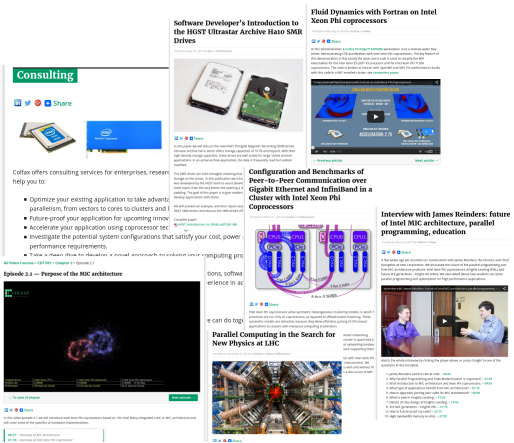
April 2017

DISCLAIMER

While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.

COLFAX RESEARCH

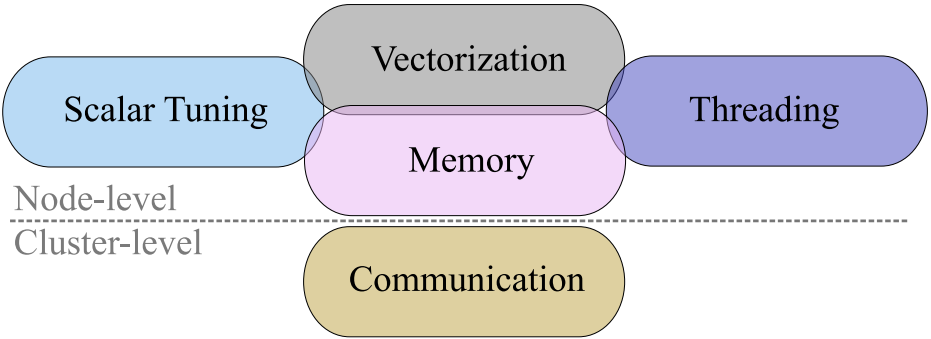




http://colfaxresearch.com/

Code Modernization

Optimizing software to better utilize features available in modern computer architectures.

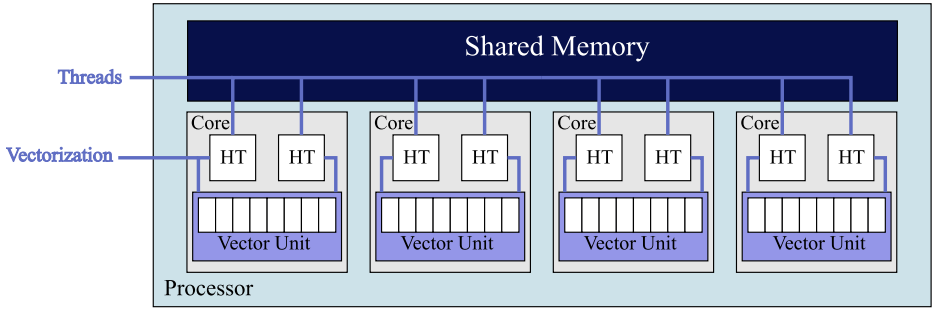


§1. QUICK REVIEW

PARALLELISM

Task Parallelism – multiple instructions multiple data elements (MIMD)

Data Parallelism – single instruction multiple data elements (SIMD)



Unbounded growth opportunity, but **not automatic**

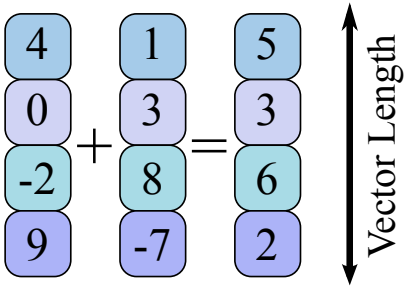
SHORT VECTOR SUPPORT

Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

Scalar Instructions

$$4 + 1 = 5$$
$$0 + 3 = 3$$
$$-2 + 8 = 6$$
$$9 + -7 = 2$$

Vector Instructions



NOW WHAT?

I have a vectorized and multi-threaded code!

Some people stop here. But even if your application is multi-threaded and vectorized, it may not be optimal. Optimization could unlock more performance for your application.

Example areas for consideration:

- Multi-threading
  - Do my threads have enough work?
  - Are my threads independent?
  - Is work distributed properly?
- Vectorization
  - Is my data organized well for vectorization?
  - Do I have regular loop patterns?

§2. OPTIMIZATION EXAMPLES

EXAMPLE: NUMERICAL INTEGRATION

$$I(a,b) = \int_0^a \frac{1}{\sqrt{x}} \mathrm{d}x$$

Rectangle method:

$$\begin{aligned} \Delta x &= \frac{a}{n}, \\ x_i &= (i+1)\Delta x, \\ I(a,b) &= \sum_{i=0}^{n-1} \frac{1}{\sqrt{x_i}} \Delta x + O(\Delta x). \end{aligned}$$

```
1  float Integrate(const float a,
2                  const int N) {
3    const float dx = a/float(n);
4    float S = 0.0f;
5    for (int i = 0; i < n; i++) {
6      const float xi = dx*float(i+1);
7      S += 1.0f/sqrtf(xi) * dx;
8    }
9    return S;
10 }
```

HOW NOT TO PARALLELIZE

Incorrect: Race condition on `integral`

```
    const double dx = (x_upper_bound - x_lower_bound)/nSteps;
    double integral = 0.0;
#pragma omp parallel for
    for(int i = 0; i < nSteps; i++) {
      const double x = x_lower_bound + dx*(double(i) + 0.5);
      integral += 1.0/sqrt(x) * dx;
    }
```

Non-optimal: Too much synchronization. Slower than serial implementation.

```
#pragma omp parallel for
    for(int i = 0; i < nSteps; i++) {
      const double x = x_lower_bound + dx*(double(i) + 0.5);
#pragma omp atomic
      integral += 1.0/sqrt(x) * dx;
    }
```

REDUCTION

Manual reduction implementation

```
  double integral = 0.0;
#pragma omp parallel
  {
    double integral_th=0.0;
#pragma omp for
    for(int i = 0; i < nSteps; i++) {
      const double x = x_lower_bound + dx*(double(i) + 0.5);
      integral_th += 1.0/sqrt(x) * dx;
    }
#pragma omp atomic
    integral += integral_th;
  }
```

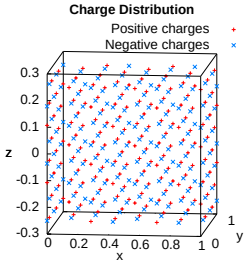
OpenMP reduction. Array support from 4.5.

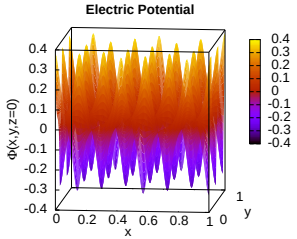
```
#pragma omp parallel for reduction(+:integral)
```

EXAMPLE: COULOMB'S LAW APPLICATION

$$\Phi\left(\vec{R}_j\right) = -\sum_{i=1}^{m} \frac{q_i}{\left|\vec{r}_i - \vec{R}_j\right|}, \tag{1}$$

$$\left|\vec{r}_i - \vec{R}\right| = \sqrt{\left(r_{i,x} - R_x\right)^2 + \left(r_{i,y} - R_y\right)^2 + \left(r_{i,z} - R_z\right)^2}. \tag{2}$$





INITIAL IMPLEMENTATION

```
struct Charge {
    float x, y, z, q;
};
```

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++) {
      const float Rx = (float) i, Ry = (float) j, Rz = (float) k;
      float phi_p = 0.0;
      for (int l=0; l<m; l++)  {
        const float dx=chg[l].x - Rx;
        const float dy=chg[l].y - Ry;
        const float dz=chg[l].z - Rz;
        phi_p -= chg[l].q / (dx*dx+dy*dy+dz*dz); // Coulomb's law
      }
      phi[i*n*n+j*n+k] = phi_p;
    }
```

ADDING THREAD PARALLELISM

Bad: parallelizing the inner loop increases overhead of OpenMP

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

Better: if n is small, not enough work to occupy all threads.

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

Good: collapse loops means threads have more iterations to work with.

```
#pragma omp parallel for  collapse(2)
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++) {
```

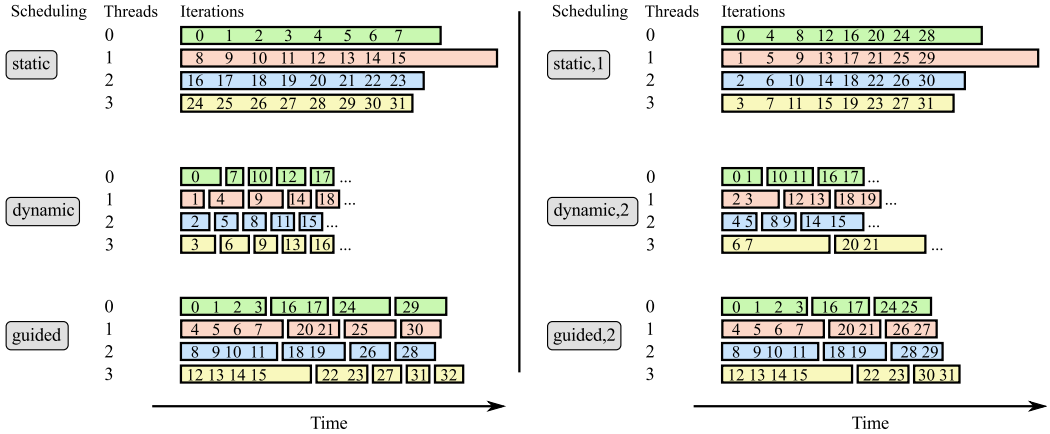
LOOP COLLAPSE: PRINCIPLE

Idea: combine iterations spaces of the inner loop and the outer loop.

```
1 #pragma omp parallel for collapse(2)
2   for (int i = 0; i < m; i++)
3     for (int j = 0; j < n; j++) {
4       // ...
5       // ...
6     }
```

```
1 #pragma omp parallel for
2   for (int c = 0; c < m*n; c++)  {
3     i = c / n;
4     j = c % n;
5     // ...
6   }
```

LOOP SCHEDULING MODES IN OPENMP



CONTROL OF SCHEDULING MODES

To set scheduling for a particular loop in code (example):

```
1 #pragma omp parallel for schedule(dynamic,4)
2   // ...
```

To set scheduling for the entire application at run time (example):

```
1 #pragma omp parallel for schedule(runtime)
2   // ...
```

```
vega@lyra% export OMP_SCHEDULE=dynamic,4
vega@lyra% ./run-my-app
```

ARRAYS OF STRUCTURES VERSUS STRUCTURES OF ARRAYS

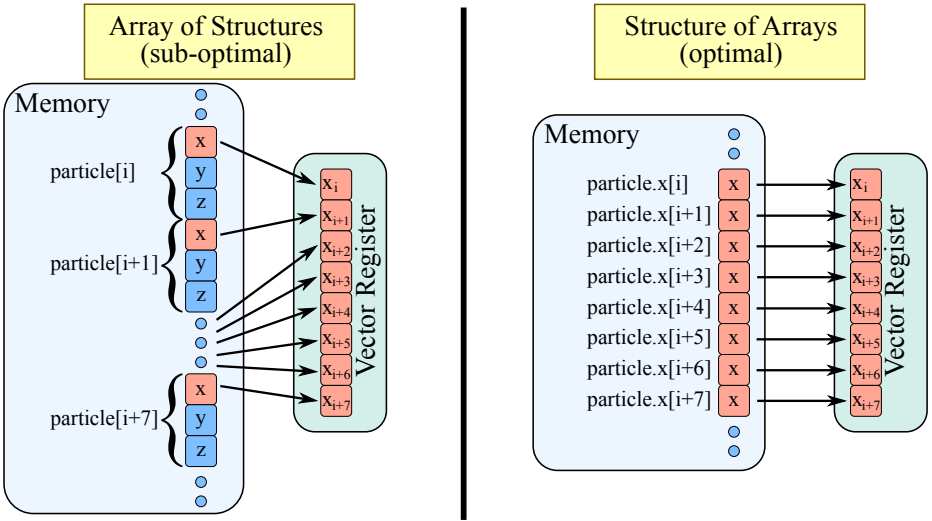
Array of Structures (AoS)

```
struct Charge { // Elegant, but ineffective data layout
  float x, y, z, q; // Coordinates and value of this charge
};
// The following line declares a set of m point charges:
Charge chg[m];
```

Structure of Arrays (SoA)

```
struct Charge_Distribution {
  // Data layout permits effective vectorization of Coulomb's law application
  const int m; // Number of charges
  float * x; // Array of x-coordinates of charges
  float * y; // ...y-coordinates...
  float * z; // ...etc.
  float * q; // These arrays are allocated in the constructor
};
```

ARRAYS OF STRUCTURES VERSUS STRUCTURES OF ARRAYS



OPTIMIZED SOLUTION: STRUCTURE OF ARRAYS, UNIT-STRIDE ACCESS

```
struct Charge_Distribution {
  // Data layout permits effective vectorization of Coulomb's law application
  const int m; // Number of charges
  float *x, *y, *z, *q; // Arrays of x-, y- and z-coordinates of charges
};
```

```
// This version vectorizes better thanks to unit-stride data access
for (int i=0; i<chg.m; i++)  {
  // Unit stride: (&chg.x[i+1] - &chg.x[i]) == sizeof(float)
  const float dx=chg.x[i] - Rx;
  const float dy=chg.y[i] - Ry;
  const float dz=chg.z[i] - Rz;
  phi -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
}
```

§3. ADDITIONAL TOPIC: UNROLL AND JAM

LOOP TILING (UNROLL-AND-JAM/REGISTER BLOCKING)

```
1  for (int i = 0; i < m; i++)  // Original code:
2    for (int j = 0; j < n; j++)
3      compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
1  // Step 1: strip-mine outer loop
2  for (int ii = 0; ii < m; ii += TILE)
3    for (int i = ii; i < ii + TILE; i++)
4      for (int j = 0; j < n; j++)
5        compute(a[i], b[j]); // Same order of operation as original
```

```
1  // Step 2: permute and vectorize outer loop
2  for (int ii = 0; ii < m; ii += TILE)
3#pragma simd
4    for (int j = 0; j < n; j++)
5      for (int i = ii; i < ii + TILE; i++)
6        compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times
```

UNROLL AND JAM

```
1   for (int k = 0; k < n; k+=4) {
2     const float Rx =  (float) i,    Ry =  (float) j,    Rz1 = (float) k,
3                 Rz2 = (float) k+1,  Rz3 = (float) k+2,  Rz4 = (float) k+3;
4     float phi_p1 = 0.0, phi_p2 = 0.0, phi_p3 = 0.0, phi_p4 = 0.0;
5 #pragma omp simd reduction(+: phi_p1, phi_p2, phi_p3, phi_p4)
6     for (int l=0; l<m; l++)  {
7       const float dx =chg.x[l] - Rx,   dy=chg.y[l] - Ry;
8                   dz1=chg.z[l] - Rz1,  dz2=chg.z[l] - Rz2,
9                   dz3=chg.z[l] - Rz3,  dz4=chg.z[l] - Rz4;
10      phi_p1 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz1*dz1);
11      phi_p2 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz2*dz2);
12      phi_p3 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz3*dz3);
13      phi_p4 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz4*dz4);
14    }
15    phi[i*n*n+j*n+k]   = phi_p1;   phi[i*n*n+j*n+k+1] = phi_p2;
16    phi[i*n*n+j*n+k+2] = phi_p3;   phi[i*n*n+j*n+k+3] = phi_p4;
17  } // Remainders to follow (dealing with case k%4 != 0) ...
```

§4. MEMORY CONSIDERATIONS

FLOPS VS BANDWIDTH

HOW CHEAP ARE FLOPS?

Theoretical estimates, Intel Xeon E5-2697 V3 processor

$$\text{Performance} = 28\ \text{cores} \times 2.7\ \text{GHz} \times (256/64)\ \text{vec.lanes} \times 2\ \text{FMA} \times 2\ \text{FPU} \approx 1.2\ \text{TFLOP/s}$$

$$\text{Required Data Rate} = 1.2\ \text{TFLOP/s} \times 8\ \text{bytes} \approx 10\ \text{TB/s}$$

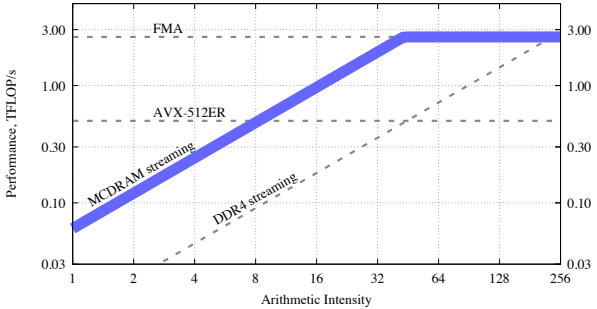
$$\text{Memory Bandwidth} = \eta \times 2 \times 59.7 \approx 0.1\ \text{TB/s}$$

$$\text{Ratio} = 10/0.1 \approx 100\ \text{(FLOPs)/(Memory Access)}$$

If the Arithmetic Intensity is...

- > 100 (FLOPs)/(Memory Access) — Compute Bound Application
- < 100 (FLOPs)/(Memory Access) — Bandwidth Bound Application

ARITHMETIC INTENSITY AND ROOFLINE MODEL



More on roofline model: Williams et al.

ON COMPUTATIONAL COMPLEXITY OF ALGORITHMS

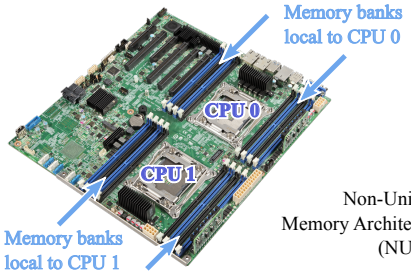
Type	Properties	Examples
$O(N)$	Each data element is used a fixed number of times. Memory-bound unless the number of times is large.	Array scaling, image brightness adjustment, vector dot-product.
$O(N^{\alpha})$	Each element is used $N^{\alpha-1}$ times. A lot of data reuse for $\alpha > 1$. Good implementation can be compute-bound, poor one – memory-bound.	Matrix-matrix multiplication: $O\left(N^{3/2}\right)$ (N = amount of data in matrix), direct N-body calculation: $O\left(N^2\right)$
$O(N \log N)$	Each element is used $\log N$ times. For small problems – memory-bound, for very large problems transitions to compute-bound	Fast Fourier transform, merge sort
$O(\log N)$	Always memory-bound.	Binary search

N = data size

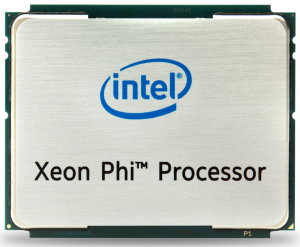
NUMA ARCHITECTURE

NUMA ARCHITECTURES

NUMA = Non-Uniform Memory Access. Cores have fast access to local memory, slow access to remote memory.



Non-Uniform
Memory Architecture
(NUMA)

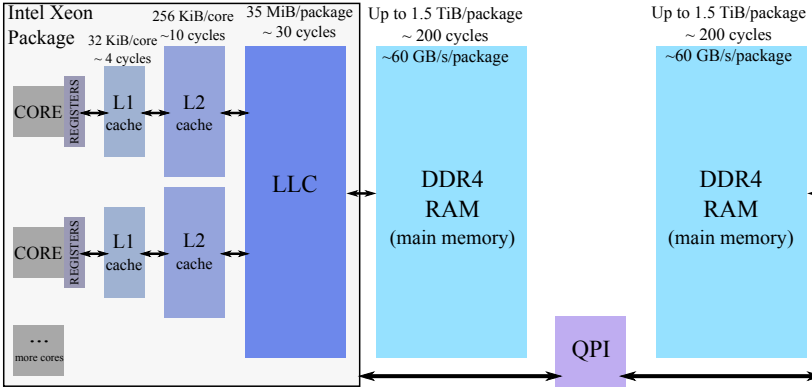


Examples:

- Multi-socket Intel Xeon processors
- Second generation Intel Xeon Phi in sub-NUMA clustering mode

INTEL XEON CPU: MEMORY ORGANIZATION

- Hierarchical cache structure
- Two-way processors have NUMA architecture



WORKING WITH NUMA ARCHITECTURES

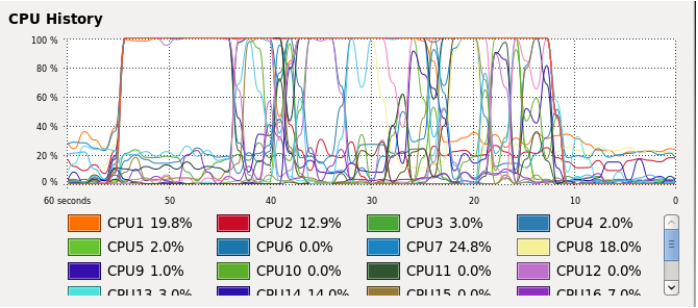
BINDING TO NUMA NODES WITH `numactl`

- `libnuma` – a Linux library for fine-grained control over NUMA policy
- `numactl` – a tool for global NUMA policy control

```
vega@lyra% numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 12 13 14 15 16 17
node 0 size: 65457 MB
node 0 free: 24426 MB
node 1 cpus: 6 7 8 9 10 11 18 19 20 21 22 23
node 1 size: 65536 MB
node 1 free: 53725 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
vega@lyra% numactl --membind=<nodes> --cpunodebind=<nodes> ./myApplication
```

WHAT IS THREAD AFFINITY

- OpenMP threads may migrate between cores
- Forbid migration — improve locality — increase performance
- Affinity patterns “scatter” and “compact” may improve cache sharing, relieve thread contention



OMP_PROC_BIND AND OMP_PLACES VARIABLES

Control the binding pattern, including nested parallelism:

```
OMP_PROC_BIND=type[,type[,...]]
```

Here `type=true`, `false`, `spread`, `close` or `master`.
Comma separates settings for different levels of nesting (`OMP_NESTED` must be enabled).

Control the granularity of binding:

```
OMP_PLACES=<threads|cores|sockets|(explicit)>
```

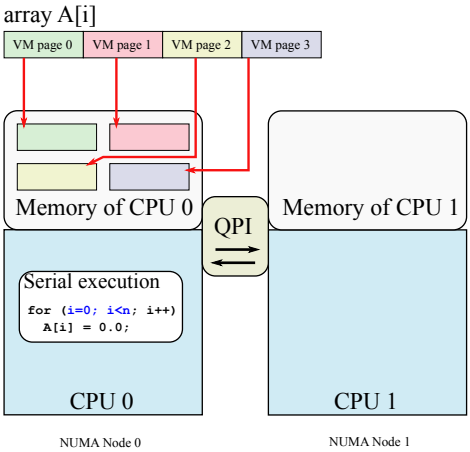
ALLOCATION ON FIRST TOUCH

- Memory allocation occurs not during `_mm_malloc()`, but upon the first write to the buffer (“first touch”)
- Default NUMA allocation policy is “on first touch”
- For better performance in NUMA systems, initialize data with the same parallel pattern as during data usage

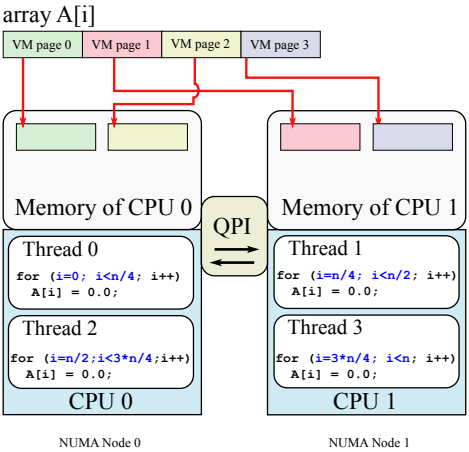
```
1  float* A = (float*)_mm_malloc(n*m*sizeof(float), 64);
2
3  // Initializing from parallel region for better performance
4  #pragma omp parallel for
5  for (int i = 0; i < n; i++)
6    for (int j = 0; j < m; j++)
7      A[i*m + j] = 0.0f;
```

FIRST-TOUCH ALLOCATION POLICY

Poor First-Touch Allocation



Good First-Touch Allocation



IMPACT OF FIRST-TOUCH ALLOCATION