

Parallelizing for Performance with OpenMP

Stanford University, ME344 — Summer 2018

Andrey Vladimirov, Colfax International

Today

- Review example: numerical integration
 - Parallel loops
 - Mutexes
 - Reduction
- Advanced example: LU decomposition
 - Parallel dependencies
 - Tasks
 - Semaphores

Example 1: Numerical Integration

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

where

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

Serial Implementation

```
float IntegrateMyFunction(int const n, float const a, float const b) {  
    float const dx = (b-a)/float(n);           // Integration interval  
    float I = 0.0f;                            // Running sum of the integral  
  
    for (int i = 0; i < n; i++) {                // Loop through the integration range  
        float const x = a + dx*(float(i) + 0.5f); // Midpoint of the integration interval  
        float const f = 1.0f/sqrtn(x);           // Function value at the midpoint  
        I += f;                                // Incrementing the running sum  
    }  
    I *= dx; // Scale according to the integration interval  
  
    return I;  
}
```

Incorrect Parallelization

```
float IntegrateMyFunction(int const n, float const a, float const b) {  
  
    float const dx = (b-a)/float(n);                      // Integration interval  
    float I = 0.0f;                                         // Running sum of the integral  
  
#pragma omp parallel for  
for (int i = 0; i < n; i++) {                            // Loop through the integration range  
    float const x = a + dx*(float(i) + 0.5f);           // Midpoint of the integration interval  
    float const f = 1.0f/sqrdf(x);                        // Function value at the midpoint  
    I += f;                                              // Incrementing the running sum  
}  
I *= dx; // Scale according to the integration interval  
  
    return I;  
}
```

Critical Section: Correct but Very Slow

```
float IntegrateMyFunction(int const n, float const a, float const b) {  
  
    float const dx = (b-a)/float(n);           // Integration interval  
    float I = 0.0f;                            // Running sum of the integral  
  
#pragma omp parallel for  
for (int i = 0; i < n; i++) {                // Loop through the integration range  
    float const x = a + dx*(float(i) + 0.5f); // Midpoint of the integration interval  
    float const f = 1.0f/sqrdf(x);            // Function value at the midpoint  
#pragma omp critical  
{  
    I += f;                                // Incrementing the running sum  
}  
}  
I *= dx; // Scale according to the integration interval  
  
return I;  
}
```

Atomic Mutex: Correct but Slow

```
float IntegrateMyFunction(int const n, float const a, float const b) {  
  
    float const dx = (b-a)/float(n);           // Integration interval  
    float I = 0.0f;                           // Running sum of the integral  
  
#pragma omp parallel for  
for (int i = 0; i < n; i++) {                // Loop through the integration range  
    float const x = a + dx*(float(i) + 0.5f); // Midpoint of the integration interval  
    float const f = 1.0f/sqrdf(x);           // Function value at the midpoint  
#pragma omp atomic  
    I += f;                                // Incrementing the running sum  
}  
I *= dx; // Scale according to the integration interval  
  
return I;  
}
```

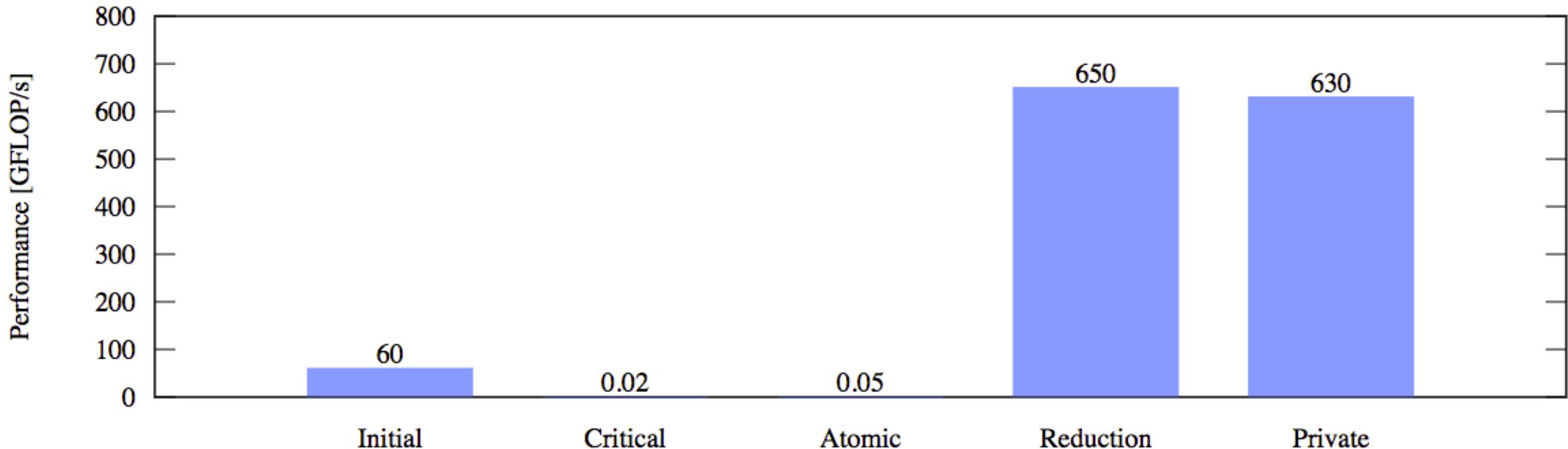
Reduction Clause: Correct and Fast

```
float IntegrateMyFunction(int const n, float const a, float const b) {  
  
    float const dx = (b-a)/float(n);           // Integration interval  
    float I = 0.0f;                            // Running sum of the integral  
  
#pragma omp parallel for reduction(+: I)  
for (int i = 0; i < n; i++) {                // Loop through the integration range  
    float const x = a + dx*(float(i) + 0.5f); // Midpoint of the integration interval  
    float const f = 1.0f/sqrdf(x);            // Function value at the midpoint  
    I += f;                                  // Incrementing the running sum  
}  
I *= dx; // Scale according to the integration interval  
  
    return I;  
}
```

Private Container: Correct and Fast

```
float IntegrateMyFunction(int const n, float const a, float const b) {  
  
    float const dx = (b-a)/float(n);                      // Integration interval  
    float I = 0.0f;                                         // Running sum of the integral  
  
#pragma omp parallel  
{  
    float I_partial = 0.0f;                                // Private to the current thread  
    #pragma omp for  
    for (int i = 0; i < n; i++) {                          // Loop through the integration range  
        float const x = a + dx*(float(i) + 0.5f);          // Midpoint of the integration interval  
        float const f = 1.0f/sqrdf(x);                      // Function value at the midpoint  
        I_partial += f;                                     // Incrementing the running sum  
    }  
    #pragma omp atomic  
    I += I_partial;                                       // Reduction into a shared counter  
  
}  
I *= dx; // Scale according to the integration interval
```

Performance

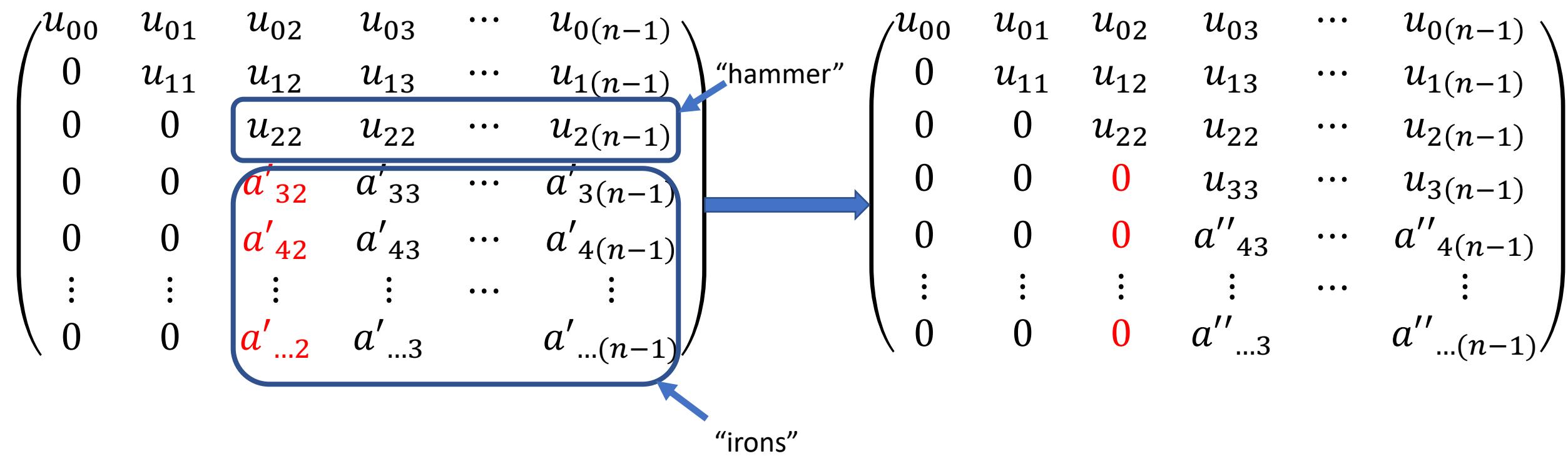


Example 2: LU Decomposition

$$A = \begin{matrix} L & \times & U \end{matrix}$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = LU = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} \begin{pmatrix} 1 & u_{12} & \cdots & u_{1n} \\ 0 & 1 & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

Mnemonic: Hammers and Irons



Serial Implementation (ijk)

```
void LU_decomp(int const n, int const lda, double* const A) {  
    int i, j, k;  
  
    for (i = 1; i < n; i++) {  
        double * const Ai = A + i*lda;      // Pointer to row i  
        for (k = 0; k < i; k++) {          // For all "hammer" rows  
            double * const Ak = A + k*lda;  // Pointer to row k  
            Ai[k] /= Ak[k];                // Compute the scaling factor (and the element of L)  
            for (j = k + 1; j < n; j++)     // Hit row "iron" row i with "hammer" row k  
                Ai[j] -= Ai[k]*Ak[j];  
        }  
    }  
}
```

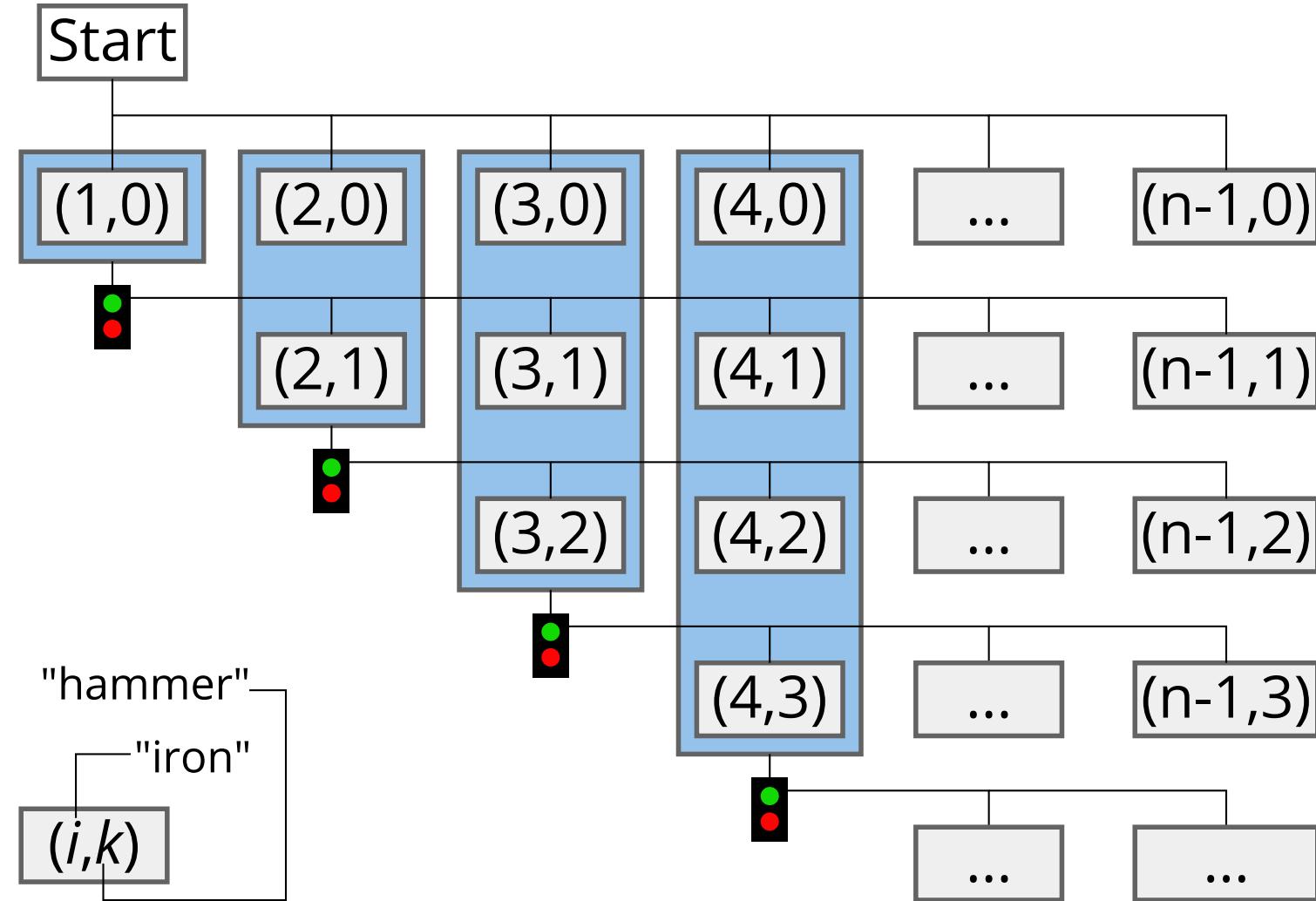
Permuted Implementation (kij)

```
void LU_decomp(int const n, int const lda, double* const A) {  
    int i, j, k;  
  
    for (k = 0; k < n; k++) {          // For all "hammer" rows  
        double * const Ak = A + k*lda; // Pointer to row k  
        for (i = k + 1; i < n; i++) {  
            double * const Ai = A + i*lda; // Pointer to row i  
            Ai[k] /= Ak[k];           // Compute the scaling factor (and the element of L)  
            for (j = k + 1; j < n; j++) // Hit row "iron" row i with "hammer" row k  
                Ai[j] -= Ai[k]*Ak[j];  
        }  
    }  
}
```

Parallel Implementation in kij

```
void LU_decomp(int const n, int const lda, double* const A) {  
    int i, j, k;  
  
    for (k = 0; k < n; k++) {          // For all "hammer" rows  
        double * const Ak = A + k*lda; // Pointer to row k  
        #pragma omp parallel for private(i)  
        for (i = k + 1; i < n; i++) {  
            double * const Ai = A + i*lda; // Pointer to row i  
            Ai[k] /= Ak[k];           // Compute the scaling factor (and the element of L)  
            for (j = k + 1; j < n; j++) // Hit row "iron" row i with "hammer" row k  
                Ai[j] -= Ai[k]*Ak[j];  
        }  
    }  
}
```

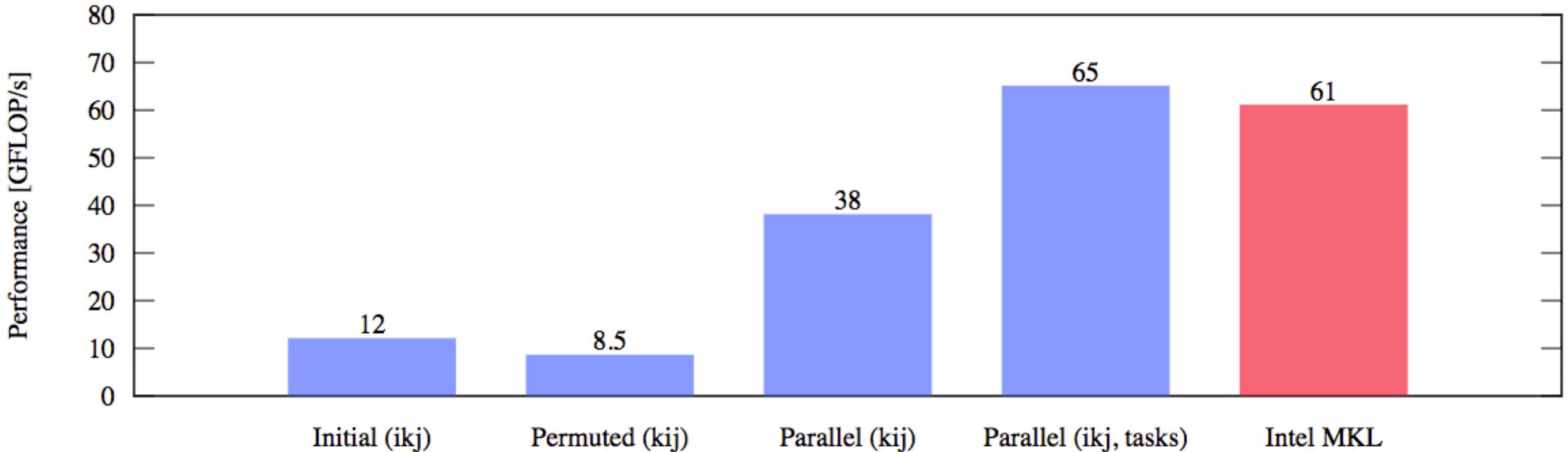
Task Dependencies



Parallel Implementation in ikj

```
#pragma omp parallel private(i, j, k)      // Start up all threads
{
    #pragma omp master                      // Restrict execution to one thread
    { for (i = 1; i < n; i++) {
        #pragma omp task firstprivate(i)      // For all "iron" rows
        { double * const Ai = A + i*llda;    // Spawn async task for each "iron"
            for (k = 0; k < i; k++) {
                double * const Ak = A + k*llda; // Pointer to row i
                while (!row_is_ready[k]) {      // For all "hammer" rows
                    #pragma omp taskyield       // Pointer to row k
                    // Wait for "hammer" number k to become ready
                    // Yield to other tasks until we can proceed
                }
                Ai[k] /= Ak[k];               // Compute the scaling factor (and the element of L)
                #pragma omp simd
                for (j = k + 1; j < n; j++)
                    Ai[j] -= Ai[k]*Ak[j];
            }
            row_is_ready[i] = 1;             // Hit row "iron" row i with "hammer" row k
            #pragma omp flush
        }
    }
}
```

Performance



Efficient Parallelism With OpenMP

- Expose parallelism of code as far out in the loop hierarchy as possible
- Avoid excessive synchronization through reduction
- Set thread affinity
- Use highly-optimized libraries for common math

Today's code:

<https://github.com/ColfaxResearch/2018-me344>

Next Week

- Using Intel Parallel Studio XE tools for performance tuning.
- Access materials at
<https://mc2learning.com/account/>
- Voucher code: (shown in class)

Learn more: HowSeries.com

About Colfax: ColfaxResearch.com