An Introduction to Parallel Programming: OpenMP, SSE/AVX, and MPI

Will Cunningham

Department of Physics
Network Science Institute
Northeastern University

April 27, 2016
1. **OpenMP**
   - Background
   - For Loops with OpenMP
   - Pairwise Operations with OpenMP

2. **SSE/AVX**
   - Background
   - For Loops with SSE/AVX
   - Combining SSE/AVX with OpenMP

3. **MPI**
   - Background
   - For Loops with MPI and OpenMP
   - Parallelization with OpenMP, SSE/AVX, and MPI

4. **Conclusions**
   - Choosing the Right Tool
   - External Resources
   - References
What is OpenMP?

OpenMP is a work-sharing construct which uses two or more cores on a computer. If a processor supports hyperthreading, two OpenMP threads may execute on a single core.
What is OpenMP?

OpenMP is a work-sharing construct which uses two or more cores on a computer. If a processor supports hyperthreading, two OpenMP threads may execute on a single core.

Compile with the flags `-fopenmp` and `-lgomp` and include the header `omp.h` in your source code.

ALWAYS BENCHMARK YOUR CODE.
For Loops with OpenMP

Vector Addition

Suppose we want to add two vectors: \( \vec{z} = \vec{x} + \vec{y} \)
For Loops with OpenMP

Vector Addition

Suppose we want to add two vectors: \( \vec{z} = \vec{x} + \vec{y} \)

```c
long N = 1000000000;
for (long i = 0; i < N; i++)
    z[i] = x[i] + y[i];
```

Elapsed Time: 3.272219 sec
Efficiency: 0.305603 GFLOPS
Suppose we want to add two vectors: \( \mathbf{\vec{z}} = \mathbf{\vec{x}} + \mathbf{\vec{y}} \)

```c
long N = 1000000000;
for (long i = 0; i < N; i++)
    z[i] = x[i] + y[i];
```

Elapsed Time: 3.272219 sec
Efficiency: 0.305603 GFLOPS

We can easily parallelize this with OpenMP:

```c
long N = 1000000000;
#pragma omp parallel for
for (long i = 0; i < N; i++)
    z[i] = x[i] + y[i];
```

Elapsed Time: 0.838140 sec
Efficiency: 1.193118 GFLOPS
Threads Used: 48
For Loops with OpenMP

Amdahl’s Law

![Amdahl’s Law Graph](image-url)
What if we want to use a counter?
Let $x$ and $y$ contain random numbers between 0.0 and 0.5:

```c
long N = 1000000000;
long counter = 0;
#pragma omp parallel for reduction (+ : counter)
for (long i = 0; i < N; i++)
    if (x[i] + y[i] < 0.5)
        counter++;
```
This is a 24x speedup!
What if we want to use a counter?
Let \( x \) and \( y \) contain random numbers between 0.0 and 0.5:

```c
long N = 1000000000;
long counter = 0;
#pragma omp parallel for reduction (+ : counter)
for (long i = 0; i < N; i++)
    if (x[i] + y[i] < 0.5)
        counter++;
```

This is a 24x speedup!
What if we want to use a counter?
Let $x$ and $y$ contain random numbers between 0.0 and 0.5:

```
long N = 1000000000;
long counter = 0;
#pragma omp parallel for reduction (+ : counter)
for (long i = 0; i < N; i++)
    if (x[i] + y[i] < 0.5)
        counter++;
```

Elapsed Time: 8.686166 sec
Efficiency: 0.115126 GFLOPS

Elapsed Time: 0.355168 sec
Efficiency: 2.815569 GFLOPS
Threads Used: 48

This is a 24x speedup!
Generating Random Numbers

Most random number generators are NOT thread-safe!
Use different seeds for each thread to fix this:
Generating Random Numbers

Most random number generators are NOT thread-safe!
Use different seeds for each thread to fix this:

```cpp
long seed = 9325723957;
#pragma omp parallel
{
    boost::mt19937 eng(seed ^ omp_get_thread_num());
    boost::uniform_real<double> udist(0.0, 1.0);
    boost::variate_generator<boost::mt19937&, boost::uniform_real<double> > rng(
        eng, udist);

    #pragma omp for
    for (long i = 0; i < N; i++) {
        x[i] = rng();
        y[i] = rng();
    }
}
```

This is a 21x speedup!
Most random number generators are NOT thread-safe! Use different seeds for each thread to fix this:

```cpp
long seed = 9325723957;
#pragma omp parallel
{
    boost::mt19937 eng(seed ^ omp_get_thread_num());
    boost::uniform_real<double> udist(0.0, 1.0);
    boost::variate_generator<boost::mt19937&, boost::uniform_real<double> > rng(
        eng, udist);

    #pragma omp for
    for (long i = 0; i < N; i++) {
        x[i] = rng();
        y[i] = rng();
    }
}
```

Elapsed Time: 105.882217 sec
Efficiency: 0.009444 GFLOPS

Elapsed Time: 5.035584 sec
Efficiency: 0.19857 GFLOPS
Threads Used: 48

This is a 21x speedup!
Generating an Adjacency Matrix

How are nested for loops parallelized?
One way is to parallelize the outer or inner loop:
Generating an Adjacency Matrix

How are nested for loops parallelized?
One way is to parallelize the outer or inner loop:

```c
long N = 10000;
#pragma omp for
for (long i = 0; i < N; i++)
    for (long j = i + 1; j < N; j++)
        if (rng() < 0.5)
            adj[i*N+j] = 1.0;
```

This is not the best way to parallelize - the inner loop is still serialized.
Matrix Index Mapping

We want a relationship between 1-D indexing and 2-D indexing. To do this we can map indices to regain a rectangular matrix:

Figure: By mapping region B in the lower-right quadrant to the upper-left quadrant of the matrix we have a rectangular matrix!
We assume the matrix has an even number of elements!
We assume the matrix has an **even** number of elements!

```c
long npairs = N * (N - 1) >> 1;
#pragma omp for
for (long k = 0; k < npairs; k++) {
    int i = (int)(k / (N - 1));
    int j = (int)(k % (N - 1) + 1);
    int do_map = i >= j;

    if (j < N >> 1) {
        i += do_map * ((((N >> 1) - i) << 1) - 1);
        j += do_map * (((N >> 1) - j) << 1);
    }

    if (rng() < 0.5)
    adj[i*N+j] = 1.0;
}
```
Pairwise Operations with OpenMP

Mapping Formula

We assume the matrix has an **even** number of elements!

```c
long npairs = N * (N - 1) >> 1;
#pragma omp for
for (long k = 0; k < npairs; k++) {
    int i = (int)(k / (N - 1));
    int j = (int)(k % (N - 1) + 1);
    int do_map = i >= j;

    if (j < N >> 1) {
        i += do_map * ((((N >> 1) - i) << 1) - 1);
        j += do_map * (((N >> 1) - j) << 1);
    }

    if (rng() < 0.5)
        adj[i*N+j] = 1.0;
}
```

Never go by theory - always benchmark your code!
1 OpenMP
   - Background
   - For Loops with OpenMP
   - Pairwise Operations with OpenMP

2 SSE/AVX
   - Background
   - For Loops with SSE/AVX
   - Combining SSE/AVX with OpenMP

3 MPI
   - Background
   - For Loops with MPI and OpenMP
   - Parallelization with OpenMP, SSE/AVX, and MPI

4 Conclusions
   - Choosing the Right Tool
   - External Resources
   - References
What is SSE/AVX?

The **Streaming SIMD Extensions (SSE)** and **Advanced Vector Extensions (AVX)** are extensions to the x86 instruction set which permit use of the XMM and YMM registers.

<table>
<thead>
<tr>
<th>General-Purpose Registers (GPRs)</th>
<th>64-Bit Media and Floating-Point Registers</th>
<th>128-Bit Media Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>MMX0/FPR0</td>
<td>XMM0</td>
</tr>
<tr>
<td>RBX</td>
<td>MMX1/FPR1</td>
<td>XMM1</td>
</tr>
<tr>
<td>RCX</td>
<td>MMX2/FPR2</td>
<td>XMM2</td>
</tr>
<tr>
<td>RDX</td>
<td>MMX3/FPR3</td>
<td>XMM3</td>
</tr>
<tr>
<td>RBP</td>
<td>MMX4/FPR4</td>
<td>XMM4</td>
</tr>
<tr>
<td>RSI</td>
<td>MMX5/FPR5</td>
<td>XMM5</td>
</tr>
<tr>
<td>RDI</td>
<td>MMX6/FPR6</td>
<td>XMM6</td>
</tr>
<tr>
<td>RSP</td>
<td>MMX7/FPR7</td>
<td>XMM7</td>
</tr>
<tr>
<td>R8</td>
<td>EFLAGS</td>
<td>RFLAGS</td>
</tr>
<tr>
<td>R9</td>
<td>63</td>
<td>63</td>
</tr>
<tr>
<td>R10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R11</td>
<td>Flags Register</td>
<td>RFLAGS</td>
</tr>
<tr>
<td>R12</td>
<td>Instruction Pointer</td>
<td>128</td>
</tr>
<tr>
<td>R13</td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>R14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>R15</td>
<td>EIP</td>
<td>XMM8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>XMM15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Legacy x86 registers, supported in all modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register extensions, supported in 64-bit mode</td>
</tr>
</tbody>
</table>

Application-programming registers also include the 128-bit media control-and-status register and the x87 tag-word, control-word, and status-word registers.
Can I Use SSE/AVX?

Different processors support different subsets of these instructions. Use the command

```
cat /proc/cpuinfo | grep -m 1 flags
```

to see what your processor supports. Compilation can be tricky. Use the header `x86intrin.h` if you’re not sure which one to use. E.g.: For a Haswell processor which supports everything up to AVX2, compile with

```
-mavx2 -march=core-avx2 -mtune=core-avx2
```
Suppose we want to find the inner product

\[ z = \vec{x} \cdot \vec{y} \]
Suppose we want to find the inner product

\[ z = \vec{x} \cdot \vec{y} \]

Start with a vector multiplication:

```c
for (long i = 0; i < N; i++)
    z[i] = x[i] * y[i];
```
Suppose we want to find the inner product

\[ z = \mathbf{x} \cdot \mathbf{y} \]

Start with a vector multiplication:

```c
for (long i = 0; i < N; i++)
    z[i] = x[i] * y[i];
```

We can use SIMD here:

```c
__m256d data0, data1;
for (long i = 0; i < N; i += 4) {
    data0 = _mm256_loadu_pd(&x[i]);
    data1 = _mm256_loadu_pd(&y[i]);
    data0 = _mm256_mul_pd(data0, data1);
    _mm256_storeu_pd(&z[i], data0);
}
```
Suppose we want to find the inner product

\[ z = \vec{x} \cdot \vec{y} \]

Start with a vector multiplication:

```c
for (long i = 0; i < N; i++)
    z[i] = x[i] * y[i];
```

We can use SIMD here:

```c
__m256d data0, data1;
for (long i = 0; i < N; i += 4) {
    data0 = _mm256_loadu_pd(&x[i]);
    data1 = _mm256_loadu_pd(&y[i]);
    data0 = _mm256_mul_pd(data0, data1);
    _mm256_storeu_pd(&z[i], data0);
}
```

Elapsed Time: 3.389569 sec
Efficiency: 0.295023 GFLOPS

Elapsed Time: 2.767988 sec
Efficiency: 0.361273 GFLOPS
We also want to perform the reduction (sum):
We also want to perform the reduction (sum):

```c
__m256d sum = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
for (long i = 0; i < N; i += 4) {
    data0 = _mm256_loadu_pd(&x[i]);
    data1 = _mm256_loadu_pd(&y[i]);
    data0 = _mm256_mul_pd(data0, data1);
    sum = _mm256_add_pd(sum, data0);
}
_mm256_storeu_pd(&z[0], sum);
double total = z[0] + z[1] + z[2] + z[3];
```
We also want to perform the reduction (sum):

```c
__m256d sum = __m256d_mm256_set_pd(0.0, 0.0, 0.0, 0.0);
for (long i = 0; i < N; i += 4) {
    data0 = __m256d_mm256_loadu_pd(&x[i]);
    data1 = __m256d_mm256_loadu_pd(&y[i]);
    data0 = __m256d_mm256_mul_pd(data0, data1);
    sum = __m256d_mm256_add_pd(sum, data0);
}
_mm256d_mm256_storeu_pd(&z[0], sum);
double total = z[0] + z[1] + z[2] + z[3];
```

How did including an extra step (reduction) reduce the overall time? Not all operations are created equally.
Putting It All Together

Finally, we can parallelize this loop with OpenMP:

```c
__m256d sum = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
#pragma omp parallel for
for (long i = 0; i < N; i += 4) {
    __m256d data0 = _mm256_loadu_pd(&x[i]);
    __m256d data1 = _mm256_loadu_pd(&y[i]);
    data0 = _mm256_mul_pd(data0, data1);
    #pragma omp critical
    sum = _mm256_add_pd(sum, data0);
}
_mm256_storeu_pd(&z[0], sum);
double total = z[0] + z[1] + z[2] + z[3];
```
Finally, we can parallelize this loop with OpenMP:

```c
__m256d sum = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
#pragma omp parallel for
for (long i = 0; i < N; i += 4) {
    __m256d data0 = _mm256_loadu_pd(&x[i]);
    __m256d data1 = _mm256_loadu_pd(&y[i]);
    data0 = _mm256_mul_pd(data0, data1);
    #pragma omp critical
    sum = _mm256_add_pd(sum, data0);
}
_mm256_storeu_pd(&z[0], sum);
double total = z[0] + z[1] + z[2] + z[3];
```
Finally, we can parallelize this loop with OpenMP:

```
__m256d sum = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
#pragma omp parallel for
for (long i = 0; i < N; i += 4) {
    __m256d data0 = _mm256_loadu_pd(&x[i]);
    __m256d data1 = _mm256_loadu_pd(&y[i]);
    data0 = _mm256_mul_pd(data0, data1);
    #pragma omp critical
    sum = _mm256_add_pd(sum, data0);
}
_mm256_storeu_pd(&z[0], sum);
double total = z[0] + z[1] + z[2] + z[3];
```

The small difference from OpenMP can be explained by Amdahl’s Law - it indicates the code is close to the theoretical maximum speedup.
1. **OpenMP**
   - Background
   - For Loops with OpenMP
   - Pairwise Operations with OpenMP

2. **SSE/AVX**
   - Background
   - For Loops with SSE/AVX
   - Combining SSE/AVX with OpenMP

3. **MPI**
   - Background
   - For Loops with MPI and OpenMP
   - Parallelization with OpenMP, SSE/AVX, and MPI

4. **Conclusions**
   - Choosing the Right Tool
   - External Resources
   - References
What is MPI?

The **Message Passing Interface** is a communication protocol commonly used in HPC which allows two or more computers to share information.

It is common to use MPI and OpenMP together. In this case, however, you need to think about **load balancing**.
Typically you won’t use MPI outside a cluster.

- Include the header `mpi.h`
- Compile with `mpicc` instead of `gcc` or `g++`
- Make sure you know which flavor of MPI you are using
- When using with a cluster, know rules for LSF vs SLURM
How to Use MPI

Typically you won’t use MPI outside a cluster.

- Include the header `mpi.h`
- Compile with `mpicc` instead of gcc or g++
- Make sure you know which flavor of MPI you are using
- When using with a cluster, know rules for LSF vs SLURM

```
mpirun -np 4 -prot -tcp -hostfile hosts ./mybinary -myflag flags
```
Basics of MPI

There are several parts to an MPI program:
Basics of MPI

There are several parts to an MPI program:

```c
int nthreads, rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nthreads);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Bulk of the code here

MPI_Finalize();
exit 0;
```
Basics of MPI

There are several parts to an MPI program:

```c
int nthreads, rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nthreads);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Bulk of the code here

MPI_Finalize();
exit 0;
```

The variable `nthreads` indicates how many computers are participating, while the variable `rank` indicates a computer’s index (starting at 0).
Sometimes you want to send data from one computer to all others:

```c
int *data = (int*)malloc(sizeof(int) * N);
MPI_Bcast(data, N, MPI_INT, 0, MPI_COMM_WORLD);
```
Sometimes you want to send data from one computer to all others:

```c
int *data = (int*)malloc(sizeof(int) * N);
MPI_Bcast(data, N, MPI_INT, 0, MPI_COMM_WORLD);
```

If you want to break up data across computers:

```c
int *r_data = (int*)malloc(sizeof(int) * N / 4);
MPI_Scatter(data, N / 4, MPI_INT, r_data, N / 4, MPI_INT, 0, MPI_COMM_WORLD);
```
Common MPI Operations

Sometimes you want to send data from one computer to all others:

```c
int *data = (int*)malloc(sizeof(int) * N);
MPI_Bcast(data, N, MPI_INT, 0, MPI_COMM_WORLD);
```

If you want to break up data across computers:

```c
int *r_data = (int*)malloc(sizeof(int) * N / 4);
MPI_Scatter(data, N / 4, MPI_INT, r_data, N / 4, MPI_INT, 0, MPI_COMM_WORLD);
```

If you want to gather results from a computation:

```c
MPI_Gather(data, N / 4, MPI_INT, r_data, N / 4, MPI_INT, 0, MPI_COMM_WORLD);
```
For Loops with MPI and OpenMP

Common MPI Operations

Sometimes you want to send data from one computer to all others:

```c
int *data = (int*)malloc(sizeof(int) * N);
MPI_Bcast(data, N, MPI_INT, 0, MPI_COMM_WORLD);
```

If you want to break up data across computers:

```c
int *r_data = (int*)malloc(sizeof(int) * N / 4);
MPI_Scatter(data, N / 4, MPI_INT, r_data, N / 4, MPI_INT, 0, MPI_COMM_WORLD);
```

If you want to gather results from a computation:

```c
MPI_Gather(data, N / 4, MPI_INT, r_data, N / 4, MPI_INT, 0, MPI_COMM_WORLD);
```

Make sure you synchronize computers before and after important operations:

```c
MPI_Barrier(MPI_COMM_WORLD);
```
Let’s return to our original example:

```c
long N = 1000000000;
for (long i = 0; i < N; i++)
    z[i] = x[i] + y[i];
```
Let's return to our original example:

```c
long N = 1000000000;
for (long i = 0; i < N; i++)
    z[i] = x[i] + y[i];
```

This is split up in the following way:

```c
long mpi_chunk = N / nthreads;
long start = mpi_chunk * rank;
long finish = start + mpi_chunk;
for (long i = start; i < finish; i++)
    z[i] = x[i] + y[i];
```
Let’s return to our original example:

```c
long N = 1000000000;
for (long i = 0; i < N; i++)
    z[i] = x[i] + y[i];
```

This is split up in the following way:

```c
long mpi_chunk = N / nthreads;
long start = mpi_chunk * rank;
long finish = start + mpi_chunk;
for (long i = start; i < finish; i++)
    z[i] = x[i] + y[i];
```

And OpenMP can easily be added to this:

```c
long mpi_chunk = N / nthreads;
long start = mpi_chunk * rank;
long finish = start + mpi_chunk;
#pragma omp parallel for
for (long i = start; i < finish; i++)
    z[i] = x[i] + y[i];
```
Finding the Average Degree

Let’s find the average degree of an undirected, unweighted graph using parallelization:

We will suppose the graph is dense to avoid arguments about sparse algorithms.
Finding the Degree Vector

First we will find the degree of each node (reduction):
Finding the Degree Vector

First we will find the degree of each node (reduction):

```c
long mpi_chunk = N / nthreads;
long start = mpi_chunk * rank;
long finish = start + mpi_chunk;
__m256d *degree = (__m256d*)malloc(sizeof(__m256d) * mpi_chunk);
for (int i = 0; i < mpi_chunk; i++)
    degree[i] = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);
for (int i = 0; i < mpi_chunk; i++) {
    #pragma omp parallel for reduction (+ : sum)
    for (long j = start + i * N; j < start + (i + 1) * N; j += 4) {
        __m256d data0 = _mm256_loadu_pd(&adj[(start+i)*N+j]);
        #pragma omp critical
        degree[start+i] = _mm256_add_pd(degree[start+i], data0);
    }
    degree[start+i] = _mm256_hadd_pd(degree[start+i], degree[start+i]);
    degree[start+i] = _mm256_hadd_pd(degree[start+i], degree[start+i]);
}
```
Finding the Degree Vector

First we will find the degree of each node (reduction):

```c
long mpi_chunk = N / nthreads;
long start = mpi_chunk * rank;
long finish = start + mpi_chunk;
__m256d *degree = (__m256d*)malloc(sizeof(__m256d) * mpi_chunk);
for (int i = 0; i < mpi_chunk; i++)
    degree[i] = _mm256_set_pd(0.0, 0.0, 0.0, 0.0);

for (int i = 0; i < mpi_chunk; i++) {
    #pragma omp parallel for reduction (+ : sum)
    for (long j = start + i * N; j < start + (i + 1) * N; j += 4) {
        __m256d data0 = _mm256_loadu_pd(&adj[(start+i)*N+j]);

        #pragma omp critical
        degree[start+i] = _mm256_add_pd(degree[start+i], data0);
    }

    degree[start+i] = _mm256_hadd_pd(degree[start+i], degree[start+i]);
    degree[start+i] = _mm256_hadd_pd(degree[start+i], degree[start+i]);
}
```

The degree vector is now stored in a SIMD array, broken up across all the computers.
Averaging the Degree Vector

First we want to convert the SIMD array to a normal array.

```c
double *deg = (double*)malloc(sizeof(double) * mpi_chunk);
for (int i = start; i < finish; i++) {
    __m128d lower = _mm256_extractf128_pd(degree[i], 0);
    _mm_storel_pd(&deg[i], lower);
}
```

Next we want to perform another parallel reduction:

```c
__m256d local_sum = _mm256d_set_pd(0.0, 0.0, 0.0, 0.0);
double loc_sum;
#pragma omp parallel for
for (int i = start; i < finish; i++) {
    __m256d data0 = _mm256_loadu_pd(&deg[i]);
#pragma omp critical
    local_sum = _mm256_add_pd(local_sum, data0);
}
local_sum = _mm256_hadd_pd(local_sum, local_sum);
local_sum = _mm256_hadd_pd(local_sum, local_sum);
__m128d lower = _mm256_extractf128_pd(local_sum, 0);
_mm_storel_pd(&loc_sum, local_sum);
```

We now have one sum on each computer.
Averaging the Degree Vector

First we want to convert the SIMD array to a normal array.

```c
double *deg = (double*)malloc(sizeof(double) * mpi_chunk);
for (int i = start; i < finish; i++) {
    __m128d lower = _mm256_extractf128_pd(degree[i], 0);
    _mm_storel_pd(&deg[i], lower);
}
```

Next we want to perform another parallel reduction:

```c
__m256d local_sum = _mm256d_set_pd(0.0, 0.0, 0.0, 0.0);
double loc_sum;
#pragma omp parallel for
for (int i = start; i < finish; i++) {
    __m256d data0 = _mm256_loadu_pd(&deg[i]);

    #pragma omp critical
    local_sum = _mm256_add_pd(local_sum, data0);
}
local_sum = _mm256_hadd_pd(local_sum, local_sum);
local_sum = _mm256_hadd_pd(local_sum, local_sum);
__m128d lower = _mm256_extractf128_pd(local_sum, 0);
_mm_storel_pd(&loc_sum, local_sum);
```
Averaging the Degree Vector

First we want to convert the SIMD array to a normal array.

double *deg = (double*)malloc(sizeof(double) * mpi_chunk);
for (int i = start; i < finish; i++) {
    __m128d lower = _mm256_extractf128_pd(degree[i], 0);
    _mm_storel_pd(&deg[i], lower);
}

Next we want to perform another parallel reduction:

__m256d local_sum = _mm256d_set_pd(0.0, 0.0, 0.0, 0.0);
double loc_sum;
#pragma omp parallel for
for (int i = start; i < finish; i++) {
    __m256d data0 = _mm256_loadu_pd(&deg[i]);

    #pragma omp critical
    local_sum = _mm256_add_pd(local_sum, data0);
}
local_sum = _mm256_hadd_pd(local_sum, local_sum);
local_sum = _mm256_hadd_pd(local_sum, local_sum);
__m128d lower = _mm256_extractf128_pd(local_sum, 0);
_mm_storel_pd(&loc_sum, local_sum);

We now have one sum on each computer.
Averaging with MPI

Simply reduce the sums across all computers:

```c
MPI_Barrier(MPI_COMM_WORLD);
double glob_sum;
MPI_Reduce(&loc_sum, &glob_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0) {
    double k_avg = global_sum / N;
    printf("Average Degree: \%f\n", k_avg);
}
```
Averaging with MPI

Simply reduce the sums across all computers:

```c
MPI_Barrier(MPI_COMM_WORLD);
double glob_sum;
MPI_Reduce(&loc_sum, &glob_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
    double k_avg = global_sum / N;
    printf("Average Degree: %f\n", k_avg);
}
```

Typically, you would add each of these features one-by-one and benchmark to see whether there is a good speed-up. It is not always obvious which tool to use, so you may end up guessing and checking a lot.
1. OpenMP
   - Background
   - For Loops with OpenMP
   - Pairwise Operations with OpenMP

2. SSE/AVX
   - Background
   - For Loops with SSE/AVX
   - Combining SSE/AVX with OpenMP

3. MPI
   - Background
   - For Loops with MPI and OpenMP
   - Parallelization with OpenMP, SSE/AVX, and MPI

4. Conclusions
   - Choosing the Right Tool
   - External Resources
   - References
Choosing the Right Tool

Where Do I Start?

Usually you start with OpenMP since it’s the easiest to write.

- Try to convert all (nested) loops to single for loops
- Longer tasks provide a better speedup - OpenMP is not good for short loops
- Play around with clauses like `schedule`, `reduction`, `atomic`, `critical`, and `if`

SSE/AVX Whenever you are doing numerical operations with vectors or matrices.

Start without OpenMP first

Make sure you are using the correct data types

Use only HALF the maximum number of OpenMP threads - no hyperthreading

MPI comes last, and can be more difficult to write.

Use when you need more memory

Make sure the problem is load balanced
Choosing the Right Tool

Where Do I Start?

Usually you start with OpenMP since it’s the easiest to write.

- Try to convert all (nested) loops to single for loops
- Longer tasks provide a better speedup - OpenMP is not good for short loops
- Play around with clauses like `schedule`, `reduction`, `atomic`, `critical`, and `if`

Use SSE/AVX whenever you are doing numerical operations with vectors or matrices.

- Start without OpenMP first
- Make sure you are using the correct data types
- Use only HALF the maximum number of OpenMP threads - no hyperthreading
Choosing the Right Tool

Where Do I Start?

Usually you start with OpenMP since it’s the easiest to write.
- Try to convert all (nested) loops to single for loops
- Longer tasks provide a better speedup - OpenMP is not good for short loops
- Play around with clauses like `schedule`, `reduction`, `atomic`, `critical`, and `if`

Use SSE/AVX whenever you are doing **numerical operations** with vectors or matrices.
- Start without OpenMP first
- Make sure you are using the correct data types
- Use only HALF the maximum number of OpenMP threads - no hyperthreading

MPI comes last, and can be more difficult to write.
- Use when you need more memory
- Make sure the problem is **load balanced**
Useful References

**OpenMP Tutorial:**

**Intel Intrinsics Guide:**

**MPI Tutorial:**
http://mpitutorial.com/tutorials/

**How to Use Discovery Cluster:**
http://nuweb12.neu.edu/rc/?page_id=18
Image References:

2. https://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg
3. http://www.viva64.com/media/images/content/a/amd64_em64t/image2.png

Special thanks to Nilay Roy.
Questions