OPENMP — Shared Memory Parallelism (threads)

Compilation & Execution

```
g++ -O2 -fopenmp prog.cpp -o prog
export OMP_NUM_THREADS=4
./prog
```

Parallel Region + Thread Info

```
#include "omp.h"

#pragma omp parallel num_threads(N)
{
  int tid  = omp_get_thread_num();  // 0..N-1
  int nthr = omp_get_num_threads(); // N
  // each thread executes this block
} // implicit barrier (all threads sync)
```

omp for — Split Loop Iterations

```
#pragma omp parallel
{
#pragma omp for
  for (int i = 0; i < N; i++) {
    A[i] = i;  // no conflict: each i unique
  } // implicit barrier
}
```

Equivalent manual split:

```
int thId  = omp_get_thread_num();
int numTh = omp_get_num_threads();
int chunk = N / numTh;
int debut = thId * chunk;
int fin = (thId==numTh-1) ? N : debut+chunk;
for (int i = debut; i < fin; i++)
  A[i] = i;
```

Reduction — Avoid Write Conflicts

```
double somme = 0.0;
#pragma omp parallel
{
#pragma omp for reduction(+:somme)
  for (int i = 0; i < N; i++)
    somme += A[i];
  // private copy per thread, merged with +
}
```

Operators: `+ - * & | ^ && || min max`

Manual equivalent (local var + atomic):

```
double piLocal = 0.0;
```

```
for (int i = debut; i < fin; i++)
  piLocal += ...;
#pragma omp atomic
pi += piLocal; // 1 atomic instead of N
```

Sections — Independent Tasks

```
#pragma omp parallel num_threads(3)
{
#pragma omp sections
  {
#pragma omp section
    { deux   = calcul2(); }  // thread A
#pragma omp section
    { trois  = calcul3(); }  // thread B
#pragma omp section
    { quatre = calcul4(); }  // thread C
  } // implicit barrier
}
neuf = deux + trois + quatre;
```

atomic vs critical

```
#pragma omp atomic      // single mem op, HW
k = k + 1;

#pragma omp critical    // any block, mutex
{ /* complex update */ }
```

Pattern: Parallel Mergesort

```
#pragma omp parallel num_threads(4)
{
#pragma omp sections // Phase 1: sort quarters
  {
#pragma omp section
    { sort(A.begin(),       A.begin()+N/4); }
#pragma omp section
    { sort(A.begin()+N/4,   A.begin()+N/2); }
#pragma omp section
    { sort(A.begin()+N/2,   A.begin()+3*N/4);}
#pragma omp section
    { sort(A.begin()+3*N/4, A.begin()+N); }
  } // barrier
#pragma omp sections // Phase 2: merge pairs
  {
#pragma omp section
    { merge(&t[0],   &A[0],   N/4,&A[N/4],   N/4);}
#pragma omp section
    { merge(&t[N/2],&A[N/2],N/4,&A[3*N/4],N/4);}
  } // barrier
}
merge(&A[0], &t[0], N/2, &t[N/2], N/2);
```

done by Yehor Korotenko

MPI — Distributed Memory Parallelism (processes)

Compilation & Execution

```
mpic++ -O2 prog.cpp -o prog
mpirun -np 4 ./prog
```

Basics: Init, Rank, Size, Finalize

```
#include "mpi.h"

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("Process %d/%d\n", rank, size);

  MPI_Finalize();
  return 0;
}
```

Send & Recv (blocking, point-to-point)

```
MPI_Send(&data, count, MPI_TYPE, dest,
         tag, MPI_COMM_WORLD);

MPI_Recv(&data, count, MPI_TYPE, source,
         tag, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

Types: `MPI_INT` `MPI_DOUBLE` `MPI_FLOAT` `MPI_CHAR`

count = size of array, **dest**, **source** = rank (id) to who we send or receive from, **tag** = don't know, set to 0

Pattern: Ping-Pong (pairs 0↔1, 2↔3 ...)

```
int msg;
if (rank%2==0 && rank+1<size) {
  msg = rank;
  MPI_Send(&msg,1,MPI_INT,rank+1,0,
          MPI_COMM_WORLD);
  MPI_Recv(&msg,1,MPI_INT,rank+1,0,
          MPI_COMM_WORLD,MPI_STATUS_IGNORE);
\} else if (rank%2==1) {
  MPI_Recv(&msg,1,MPI_INT,rank-1,0,
          MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  msg += 10 * rank;
  MPI_Send(&msg,1,MPI_INT,rank-1,0,
          MPI_COMM_WORLD);
\}
```

Pattern: Split Work + Gather (calcul-pi)

```
int begin = (N/size) * rank;
int end = (rank==size-1) ? N : (N/size)*(rank+1);

double piLocal = 0.0;
for (int i = begin; i < end; i++)
  piLocal += s * (f(i*s)+f((i+1)*s)) / 2;

if (rank == 0) {
  pi = piLocal;
  for (int i = 1; i < size; i++) {
    MPI_Recv(&piLocal, 1, MPI_DOUBLE, i, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    pi += piLocal;
  }
  // broadcast result back
  for (int i = 1; i < size; i++)
    MPI_Send(&pi, 1, MPI_DOUBLE, i, 0,
             MPI_COMM_WORLD);
} else {
  MPI_Send(&piLocal, 1, MPI_DOUBLE, 0, 0,
           MPI_COMM_WORLD);
  MPI_Recv(&pi, 1, MPI_DOUBLE, 0, 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Pattern: Tree Merge (mergesort distribué)

```
// Each process: sorted local array A, size N
int count = N;
for (int r = size; r > 1; r = r/2) {
  if (rank < r/2) {
    // receiver: get partner's data & merge
    int partner = rank + r/2;
    vector<double> recv(count);
    vector<double> merged(2*count);
    MPI_Recv(&recv[0], count, MPI_DOUBLE,
      partner, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
    merge(&merged[0], &A[0], count,
        &recv[0], count);
    count *= 2;
    swap(A, merged);
  } else if (rank>=r/2 && rank<r) {
    // sender: send & exit loop
    int partner = rank - r/2;
    MPI_Send(&A[0], count, MPI_DOUBLE,
      partner, 0, MPI_COMM_WORLD);
    break;
  }
}
// rank 0 has final sorted array (N*size)
```

Pattern: Calcul Neuf (3 ind. tasks)

```
if (rank == 0) {
  deux = calcul2();
  MPI_Recv(&trois,1,MPI_INT,1,0,...);
  MPI_Recv(&quatre,1,MPI_INT,2,0,...);
  neuf = deux + trois + quatre;
  MPI_Send(&neuf,1,MPI_INT,1,0,...);
  MPI_Send(&neuf,1,MPI_INT,2,0,...);
} else if (rank == 1) {
  trois = calcul3();
  MPI_Send(&trois,1,MPI_INT,0,0,...);
  MPI_Recv(&neuf,1,MPI_INT,0,0,...);
} else if (rank == 2) {
  quatre = calcul4();
  MPI_Send(&quatre,1,MPI_INT,0,0,...);
  MPI_Recv(&neuf,1,MPI_INT,0,0,...);
}
```

done by Yehor Korotenko

AVX / SIMD — Data-Level Parallelism (within one core)

Compilation

```
g++ -O1 -mavx2 -mfma prog.cpp -o prog
// -O1 (not -O2) to control vectorization
```

Types, Load, Store, Arithmetic

```
#include "immintrin.h"

__m256  v;  // 8 x float  (256-bit)
__m256d vd; // 4 x double (256-bit)
__m256i vi; // 8 x int32  (256-bit)

// Load 8 floats from memory
__m256 vx = _mm256_loadu_ps(&x[i]);
__m256 vy = _mm256_loadu_ps(&y[i]);

// Arithmetic (8 ops in 1 instruction)
__m256 vz;
vz = _mm256_add_ps(vx, vy); // +
vz = _mm256_sub_ps(vx, vy); // -
vz = _mm256_mul_ps(vx, vy); // *
vz = _mm256_div_ps(vx, vy); // /

// FMA: fused multiply-add (-mfma)
vz = _mm256_fmadd_ps(vx, vy, vz);
// vz = vz + vx * vy

// Store 8 floats to memory
_mm256_storeu_ps(&z[i], vz);

// Zero register
vz = _mm256_setzero_ps();
```

Pattern: Loop Vectorization (stride 8)

```
// z[i] = x[i] + y[i] for N floats
int i;
for (i = 0; i < N - 7; i += 8) {
  __m256 vx = _mm256_loadu_ps(&x[i]);
  __m256 vy = _mm256_loadu_ps(&y[i]);
  __m256 vz = _mm256_add_ps(vx, vy);
  _mm256_storeu_ps(&z[i], vz);
}
// remainder (tail loop)
for (; i < N; i++)
  z[i] = x[i] + y[i];
```

Pattern: Dot Product (horizontal reduction)

```
float dotAVXFMA(float *A, float *B, int N)
{
  __m256 res = _mm256_setzero_ps();
  for (int i = 0; i < N; i += 8) {
    __m256 a = _mm256_loadu_ps(A + i);
    __m256 b = _mm256_loadu_ps(B + i);
    res = _mm256_fmadd_ps(a, b, res);
  }
  // horizontal sum
  float tmp[8];
  _mm256_storeu_ps(tmp, res);
  return tmp[0]+tmp[1]+tmp[2]+tmp[3]
        +tmp[4]+tmp[5]+tmp[6]+tmp[7];
}
```

Optimization: Loop Unrolling (factor 4)

Multiple accumulators to hide FMA latency:

```
__m256 r1=_mm256_setzero_ps();
__m256 r2=_mm256_setzero_ps();
__m256 r3=_mm256_setzero_ps();
__m256 r4=_mm256_setzero_ps();
for (int i = 0; i < N; i += 32) {
  r1 = _mm256_fmadd_ps(
    _mm256_loadu_ps(A+i),
    _mm256_loadu_ps(B+i),    r1);
  r2 = _mm256_fmadd_ps(
    _mm256_loadu_ps(A+i+8),
    _mm256_loadu_ps(B+i+8),  r2);
  r3 = _mm256_fmadd_ps(
    _mm256_loadu_ps(A+i+16),
    _mm256_loadu_ps(B+i+16), r3);
  r4 = _mm256_fmadd_ps(
    _mm256_loadu_ps(A+i+24),
    _mm256_loadu_ps(B+i+24), r4);
}
r1 = _mm256_add_ps(
  _mm256_add_ps(r1,r2),
  _mm256_add_ps(r3,r4));
// then horizontal sum of r1
```

Copy with AVX + Unrolling

```
int i;
for (i = 0; i < dim-31; i += 32) {
  __m256 a=_mm256_loadu_ps(src+i);
  __m256 b=_mm256_loadu_ps(src+i+8);
  __m256 c=_mm256_loadu_ps(src+i+16);
  __m256 d=_mm256_loadu_ps(src+i+24);
  _mm256_storeu_ps(dst+i,    a);
  _mm256_storeu_ps(dst+i+8,  b);
  _mm256_storeu_ps(dst+i+16, c);
  _mm256_storeu_ps(dst+i+24, d);
}
for (; i < dim; i++) dst[i] = src[i];
```

Quick Reference Table

	OpenMP	**MPI**	**AVX**
Model	threads	processes	SIMD regs
Memory	shared	distributed	registers
Scope	1 node	cluster	1 core
Compile	`-fopenmp`	`mpic++`	`-mavx2 -mfma`
Split	`omp for`	manual range	stride 8
Sync	barrier/atomic	Send/Recv	horiz. sum
Merge	reduction	gather to P0	add_ps
Speed	×cores	×nodes	×8 float

AVX parallelizes within a core, OpenMP parallelizes across cores on one machine, and MPI parallelizes across machines.

done by Yehor Korotenko