

Parallel Programming. Multicore processors TODAY, many-core co-processors READY.



Solutions for parallel programming work best on Intel® Architecture. Programming for multicore processors today enables scaling forward to many-core co-processors.

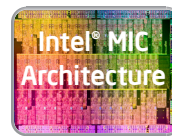
Multicore processors



The versatility of Intel® Architecture has both increased and remained accessible as the number of hardware threads and cores has risen over the past decade. New tools and programming models only make it better.

Intel® Xeon® processors are designed for intelligent performance and smart energy efficiency.

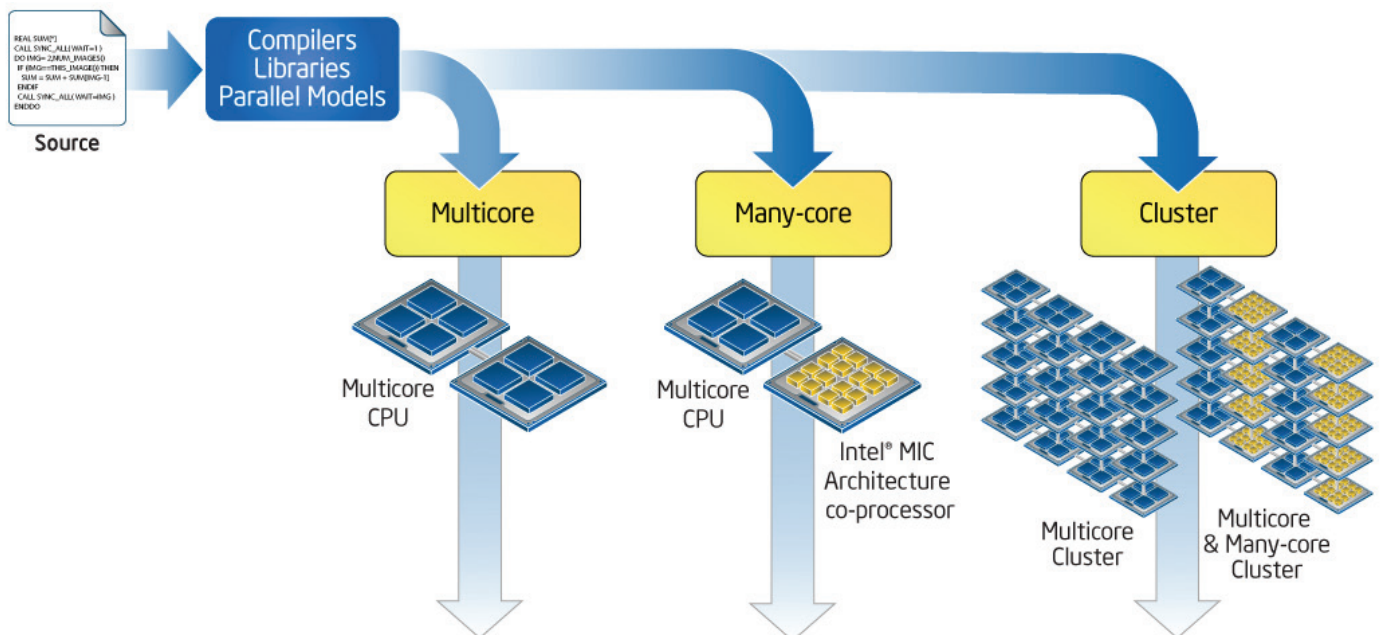
Many-core co-processors



Co-processors based on Intel® Many Integrated Core (Intel® MIC) Architecture have high core counts, each with multiple hardware threads and wider vector units.

These can be ideal for highly parallel applications. The familiarity of established programming models and tools for Intel Architecture processors makes parallel programming widely accessible.

One source base, tuned to many targets



Solutions exist today. Programming and optimizing for multicore is the best path to being many-core ready.

Code your way

Innumerable programming languages, models, and tools support Intel architecture. Use them with Intel multicore processors and many-core co-processors.

Here are some code samples implementing vector, task, and cluster parallelism. All are available for multicore and many-core. Mix and match to suit your needs.

Summing vector elements in C using OpenMP - openmp.org

```
#pragma omp parallel for reduction(+: s)
for (int i = 0; i < n; i++) {
    s += x[i];
}
```

Per element multiply in C++ using Intel® Array Building Blocks - intel.com/go/arbb

```
void products( const arbb::dense<arbb::f32>& a,
               const arbb::dense<arbb::f32>& b,
               arbb::dense<arbb::f32>& c) {
    c = a * b;
}
```

Dot product in Fortran using OpenMP - openmp.org

```
!$omp parallel do reduction ( + : adotb )
  do j = 1, n
    adotb = adotb + a(j) * b(j)
  end do
!$omp end parallel do
```

MPI code in C for clusters - intel.com/go/mapi

```
for (d=1; d<ntasks; d++) {
    rows = (d <= extra) ? avrow+1 : avrow;
    printf(" sending %d rows to task %d\n", rows, dest);
    MPI_Send(&offset, 1, MPI_INT, d, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, d, mtype, MPI_COMM_WORLD);
    MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, d, mtype, MPI_COMM_WORLD);
    MPI_Send(&b, NCA*NCB, MPI_DOUBLE, d, mtype, MPI_COMM_WORLD);
    offset = offset + rows;
}
```

Matrix Multiply in Fortran using Intel® Math Kernel Library - intel.com/software/products

```
call DGEMM(transa,transb,m,n,k,alpha,a,lda,b,ldb,beta,c,ldc)
```

Sum in Fortran, using co-array feature - intel.com/software/products

```
REAL SUM[*]
CALL SYNC_ALL( WAIT=1 )
DO IMG= 2,NUM_IMAGES()
  IF (IMG==THIS_IMAGE()) THEN
    SUM = SUM + SUM[IMG-1]
  ENDIF
CALL SYNC_ALL( WAIT=IMG )
ENDDO
```

Parallel function invocation in C using Intel® Cilk™ Plus - cilk.org

```
cilk_for (int i=0; i<n; ++i) {
    Foo(a[i]);
}
```

Parallel function invocation in C++ using Intel® Threading Building Blocks - threadingbuildingblocks.org

```
parallel_for (0, n,
    [=](int i) { Foo(a[i]); }
);
```

Per element multiply in C using OpenCL - intel.com/go/opencl

```
kernel void
dotprod( global const float *a,
         global const float *b,
         global float *c) {
    int myid = get_global_id(0);
    c[myid] = a[myid] * b[myid];
}
```

For more information go to: intel.com/software/products

