Parallel Programming and Optimization with Intel Xeon Phi Coprocessors

Parallelism and Optimizations

Vadim Karpusenko, PhD

Colfax Workshop

Chapters 3-4 – September 20, 2013
§3. Parallel scalability with Intel Xeon Phi coprocessors

Vectorization
SIMD Operations: Concept and History

SIMD - Single Instruction Multiple Data

Scalar Loop

```plaintext
for (i = 0; i < n; i++)
```

SIMD Loop

```plaintext
for (i = 0; i < n; i += 4)
    A[i:(i+4)] = A[i:(i+4)] + B[i:(i+4)];
```

Each SIMD addition operator acts on 4 numbers at a time.
## MMX, SSE, AVX and IMCI Instruction Sets

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Year and Intel Processor</th>
<th>Vector registers</th>
<th>Packed Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>1997, Pentium</td>
<td>64-bit</td>
<td>8-, 16- and 32-bit integers</td>
</tr>
<tr>
<td>SSE</td>
<td>1999, Pentium III</td>
<td>128-bit</td>
<td>32-bit single precision FP</td>
</tr>
<tr>
<td>SSE2</td>
<td>2001, Pentium 4</td>
<td>128-bit</td>
<td>8 to 64-bit integers; SP &amp; DP FP</td>
</tr>
<tr>
<td>SSE3–SSE4.2</td>
<td>2004 – 2009</td>
<td>128-bit</td>
<td>(additional instructions)</td>
</tr>
<tr>
<td>AVX</td>
<td>2011, Sandy Bridge</td>
<td>256-bit</td>
<td>single and double precision FP</td>
</tr>
<tr>
<td>AVX2</td>
<td>2013, (future) Haswell</td>
<td>256-bit</td>
<td>integers, additional instructions</td>
</tr>
<tr>
<td>IMCI</td>
<td>2012, Knights Corner</td>
<td>512-bit</td>
<td>32- and 64-bit integers; single &amp; double precision FP</td>
</tr>
</tbody>
</table>
Features of the IMCI Instruction Set

Intel IMCI is the instruction set supported by Intel Xeon Phi copr.

- **512-bit wide registers**
  - can pack up to eight 64-bit elements (long int, double)
  - up to sixteen 32-bit elements (int, float)

- **Arithmetic Instructions**
  - Addition, subtraction and multiplication
  - Fused Multiply-Add instruction (FMA)
  - Division and reciprocal calculation;
  - Error function, inverse error function;
  - Exponential functions (natural, base 2 and base 10) and the power function.
  - Logarithms (natural, base 2 and base 10).
  - Square root, inverse square root, hypothenuse value and cubic root;
  - Trigonometric functions (sin, cos, tan, sinh, cosh, tanh, asin, acos, atan);
  - Rounding functions
Features of the IMCI Instruction Set

- Initialization, Load and Store, Gather and Scatter
- Comparison
- Conversion and type cast
- Bitwise instructions: NOT, AND, OR, XOR, XAND
- Reduction and minimum/maximum instructions
- Vector mask instructions
- Scalar instructions
- Swizzle and permute
Explicit Vectorization: Intrinsic Functions and Class Libraries

Allows fine control over vectorization, but inconsistent with cross-platform probability.
Compiler Intrinsics

SSE2 Intrinsics

```
for (int i=0; i<n; i+=4) {
    __m128 Avec=_mm_load_ps(A+i);
    __m128 Bvec=_mm_load_ps(B+i);
    Avec=_mm_add_ps(Avec, Bvec);
    _mm_store_ps(A+i, Ave);
}
```

IMCI Intrinsics

```
for (int i=0; i<n; i+=16) {
    __m512 Avec=_mm512_load_ps(A+i);
    __m512 Bvec=_mm512_load_ps(B+i);
    Avec=_mm512_add_ps(Avec, Bvec);
    _mm512_store_ps(A+i, Ave);
}
```

- The arrays float A[n] and float B[n] are aligned on a 16-byte (SSE2) and 64-byte (IMCI) boundary
- n is a multiple of 4 for SSE and a multiple of 16 for IMCI
- Variables Avec and Bvec are
  \[ 128 = 4 \times \text{sizeof(float)} \] bits in size for SSE2 and
  \[ 512 = 16 \times \text{sizeof(float)} \] bits for the Intel Xeon Phi architecture
Class Libraries

SSE2 Intel C++ Vector class library

```cpp
for (int i=0; i<n; i+=4) {
    F32vec4 *Avec=(F32vec4*)(A+i);
    F32vec4 *Bvec=(F32vec4*)(B+i);
    *Avec = *Avec + *Bvec;
}
```

IMCI Intel C++ Vector class library

```cpp
for (int i=0; i<n; i+=16) {
    F32vec16 *Avec=(F32vec16*)(A+i);
    F32vec16 *Bvec=(F32vec16*)(B+i);
    *Avec = *Avec + *Bvec;
}
```

- Addition of two arrays using the Intel C++ vector class libraries
- Codes assume that the arrays `float A[n]` and `float B[n]` are aligned on a 16- & 64-byte boundary
- `n` is a multiple of 4 for SSE2 and a multiple of 16 for the Intel Xeon Phi architecture
Class Libraries

SSE2 Intel C++ Vector class library

```c
for (int i=0; i<n; i+=4) {
    F32vec4 *Avec=(F32vec4*)(A+i);
    F32vec4 *Bvec=(F32vec4*)(B+i);
    *Avec = *Avec + *Bvec;
}
```

IMCI Intel C++ Vector class library

```c
for (int i=0; i<n; i+=16) {
    F32vec16 *Avec=(F32vec16*)(A+i);
    F32vec16 *Bvec=(F32vec16*)(B+i);
    *Avec = *Avec + *Bvec;
}
```

- Addition of two arrays using the Intel C++ vector class libraries
- Codes assume that the arrays `float A[n]` and `float B[n]` are aligned on a 16- & 64-byte boundary
- `n` is a multiple of 4 for SSE2 and a multiple of 16 for the Intel Xeon Phi architecture

... is there any other way?
#include <stdio.h>

int main()
{
    const int n=8;
    int i;
    int A[n] __attribute__((aligned(64)));
    int B[n] __attribute__((aligned(64)));

    // Initialization
    for (i=0; i<n; i++)
        A[i]=B[i]=i;

    // This loop will be auto-vectorized
    for (i=0; i<n; i++)
        A[i]+=B[i];

    // Output
    for (i=0; i<n; i++)
        printf("%2d %2d %2d\n", i, A[i], B[i]);
}
Automatic Vectorization of Loops on MIC architecture

Compilation and runtime output of the code for Intel Xeon Phi execution

```
user@host% icpc autovec.c -vec-report3 -mmic
autotest.c(10): (col. 3) remark: LOOP WAS VECTORIZED.
autotest.c(14): (col. 3) remark: LOOP WAS VECTORIZED.
autotest.c(18): (col. 3) remark: loop was not vectorized:
    existence of vector dependence.
user@host% micnativeloadex a.out
  0  0  0
  1  2  1
  2  4  2
  3  6  3
  4  8  4
  5 10  5
  6 12  6
  7 14  7
```
Automatic Vectorization of Loops

Limitations:

- Only for-loops can be auto-vectorized. Number of iterations must be known at a runtime and/or compilation time.
- Memory access in the loop must have a regular pattern, ideally with unit stride.
- Non-standard loops that cannot be automatically vectorized:
  - loops with irregular memory access pattern
  - calculations with vector dependence
  - while-loops, for-loops with undetermined number of iterations
  - outer loops (unless #pragma simd overrides this restriction)
  - loops with complex branches (i.e., if-conditions)
  - anything else that cannot be, or is very difficult to vectorize.
Multi-threading: OpenMP, Intel Cilk Plus
Parallelism in Shared Memory: OpenMP and Intel Cilk Plus

- **Intel Cilk Plus**
  - Good performance “out of the box”
  - Little freedom for fine-tuning
  - Programmer should focus on exposing the parallelism
  - Low-level optimization (thread creation, work distribution and data sharing) is performed by the Cilk Plus library
  - Novel framework

- **OpenMP**
  - Easy to use for simple algorithms
  - For complex parallelism, may require more tuning to perform well
  - Allows more control over synchronization, work scheduling and distribution
  - Well-established framework
Program Structure in OpenMP

```c
main() {
  // Begin serial execution.
  ...
  // Only the initial thread executes
  #pragma omp parallel
  { // Begin a parallel construct and form
    #pragma omp sections
    { // a team.
      #pragma omp section
      { // One unit of work.
        #pragma omp section
        { } // Another unit of work.
      } // Wait until both units of work complete.
      ...
      // This code is executed by each team member.
      #pragma omp for
      for(...) { // Begin a work-sharing Construct
        // Each iteration chunk is unit of work.
        ...
        // Work is distributed among the team members.
      } // End of work-sharing construct.
  }
}
```

Vadim Karpusenko (Colfax International)

About OpenMP and Intel Cilk Plus

Chapters 3-4 – September 20, 2013
Program Structure in OpenMP

```c
#pragma omp critical
    // Begin a critical section.
    {...}
    // Only one thread executes at a time.
#pragma omp task
    // Execute in another thread without blocking
    {...}
    // This code is executed by each team member.
#pragma omp barrier
    // Wait for all team members to arrive.
    ...
    // This code is executed by each team member.
    }
    // End of Parallel Construct
    // Disband team and continue serial execution.
    ...
    // Possibly more parallel constructs.
}
    // End serial execution.
```

1. Code outside `#pragma omp parallel` is serial, i.e., executed by only one thread
2. Code directly inside `#pragma omp parallel` is executed by each thread
3. Code inside work-sharing construct `#pragma omp for` is distributed across the threads in the team
Program Structure with Intel Cilk Plus

- Only three keywords in the Cilk Plus standard:
  - _Cilk_for
  - _Cilk_spawn
  - _Cilk_sync

- Programming for coprocessors may also require keywords
  - _Cilk_shared
  - _Cilk_offload

- Language extensions
  - array notation
  - hyperobjects
  - elemental function
  - #pragma simd

- Guarantees that serialized code will produce the same results as parallel code
- Integrates vectorization and thread-parallelism
```c
#include <omp.h>
#include <stdio.h>

int main(){
    const int nt=omp_get_max_threads();
    printf("OpenMP with %d threads\n", nt);

    #pragma omp parallel
    printf("Hello World from thread %d\n", omp_get_thread_num());
}
```
“Hello World” OpenMP Programs

```bash
user@host$ export OMP_NUM_THREADS=5
user@host$ icpc -openmp hello_omp.cc
user@host$ ./a.out
OpenMP with 5 threads
Hello World from thread 0
Hello World from thread 3
Hello World from thread 1
Hello World from thread 2
Hello World from thread 4

user@host$ icpc -openmp-stubs hello_omp.cc
hello_omp.cc(8): warning #161: unrecognized #pragma
    #pragma omp parallel
~

user@host$ ./a.out
OpenMP with 1 threads
Hello World from thread 0
```
```c
double f = 1.0;
while (f < 1.0e40) f *= 2.0; // Extra workload: gets parallelized
printf("Hello Again from worker %d (%f)\n", __cilkrts_get_worker_number(), f);
```
user@host$ export CILK_NWORKERS=5
user@host$ icpc hello_cilk.cc
user@host$ ./a.out
Cilk Plus with 5 workers.
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0 (10889035741470030830827987437816582766592.000000)
Hello World from worker 0 (10889035741470030830827987437816582766592.000000)
Hello World from worker 1 (10889035741470030830827987437816582766592.000000)
Hello World from worker 3 (10889035741470030830827987437816582766592.000000)
Hello World from worker 0 (10889035741470030830827987437816582766592.000000)
user@host$
user@host$ export CILK_NWORKERS=5
user@host$ icpc -cilk-serialize hello_cilk.cc
user@host$ ./a.out
Cilk Plus with 5 workers.
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello World from worker 0
Hello Again from worker 0 (10889035741470030830827987437816582766592.000000)
Hello Again from worker 0 (10889035741470030830827987437816582766592.000000)
Hello Again from worker 0 (10889035741470030830827987437816582766592.000000)
Hello Again from worker 0 (10889035741470030830827987437816582766592.000000)
Hello Again from worker 0 (10889035741470030830827987437816582766592.000000)
user@host$
Loop-Centric Parallelism: For-Loops in OpenMP and Intel Cilk Plus

Figure: Parallelizing a for-loop with OpenMP and Intel Cilk Plus.
The OpenMP library will distribute the iterations of the loop following the 
#pragma omp parallel for across threads.

```c
#pragma omp parallel for
for (int i=0; i<n; i++) {
    printf("Iteration %d is processed by thread %d\n", i, omp_get_thread_num());
    // ... iterations will be distributed across available threads...
}
```
#pragma omp parallel
{
    // Code placed here will be executed by all threads.
    // Stack variables declared here will be private to each thread.
    int private_number=0;
    #pragma omp for schedule(dynamic, 4)
    for (int i=0; i<n; i++) {
        // ... iterations will be distributed across available threads...
    }
    // ... code placed here will be executed by all threads
}
Loop-Centric Parallelism: For-Loops in Intel Cilk Plus

Intel Cilk Plus distributes the iterations between the available workers

```cilk
_Cilk_for (int i=0; i<n; i++) {
    // ... iterations will be distributed across available threads...
    printf("Iteration %d is processed by worker %d\n",
            i, __cilkrts_get_worker_number());
}
```

Controlling grain size in Intel Cilk Plus

```cilk
#pragma cilk grainsize = 4
_Cilk_for (int i = 0; i < N; i++) {
    // ...
}
```

Intel Cilk Plus uses the “work stealing” scheduling mode.
Fork-Join Model of Parallel Execution

Figure: Fork-join model of shared-memory parallel execution.
```c
#include <omp.h>
#include <stdio.h>

int main() {
    const int n = 1000;
    int total = 0;
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        // Race condition
        total = total + i;
    }
    printf("total=%d (must be %d)\n", total, ((n-1)*n)/2);
}
```

```
user@host% icpc -o omp-race omp-race.cc -openmp
user@host% ./omp-race
total=208112 (must be 499500)
```
```c
#include <omp.h>
#include <stdio.h>

int main() {
    const int n = 1000;
    int total = 0;

    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        #pragma omp critical
        {
            // Only one thread at a time can execute this section
            total = total + i;
        }
    }
}
```

```bash
user@host% icpc -o omp-critical omp-critical.cc -openmp
user@host% ./omp-race
total=499500 (must be 499500)
```
This parallel fragment of code has predictable behavior, because the race condition was eliminated with *an atomic operation*:

```c
#pragma omp parallel for
   for (int i = 0; i < n; i++) {
    // Lightweight synchronization
    #pragma omp atomic
    sum += i;
   }
```
Synchronization: Avoiding Unpredictable Program Behavior

**Read** : operations in the form \( v = x \)

**Write** : operations in the form \( x = v \)

**Update** : operations in the form \( x++, x--, --x, ++x, x \text{ binop} = \text{expr} \)

and \( x = x \text{ binop expr} \)

**Capture** : operations in the form \( v = x++, v = x-, v = -x, v = +x, v = x \text{ binop expr} \)

- Here \( x \) and \( v \) are scalar variables
- \( \text{binop} \) is one of \( +, *, -, -, /, \&, \^, |, \ll, \gg \).
- No “trickery” is allowed for atomic operations:
  - no operator overload,
  - no non-scalar types,
  - no complex expressions.
Reduction: Avoiding Synchronization

```c
#include <omp.h>
#include <stdio.h>

int main() {
    const int n = 1000;
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < n; i++) {
        sum = sum + i;
    }
    printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
}
```

user@host% icpc -o omp-reduction omp-reduction.cc -openmp
user@host% ./omp-reduction
sum=499500 (must be 499500)
Implementation of Reduction using Private Variables

```c
#include <omp.h>
#include <stdio.h>

int main() {
    const int n = 1000;
    int sum = 0;
    #pragma omp parallel
    {
        int sum_th = 0;
        #pragma omp for
        for (int i = 0; i < n; i++)
            sum_th = sum_th + i;
        #pragma omp atomic
        sum += sum_th;
    }
    printf("sum=%d (must be %d)\n", sum, ((n-1)*n)/2);
}
```
The most commonly used framework for distributed memory HPC calculations is the Message Passing Interface (MPI).

Pure MPI: one single-threaded MPI process per core (e.g., 32 MPI processes on hosts, 240 on coprocessors).
Compiling and Running MPI applications

1. Compile and link with the MPI wrapper of the compiler:
   - mpiicc for C,
   - mpiicpc for C++,
   - mpiifort for Fortran 77 and Fortran 95.

2. Set up MPI environment variables \( \text{and I_MPI_MIC=1} \)

3. NFS-share or copy the MPI library and the application executable to the coprocessors

4. Launch with the tool mpirun
   - Colon-separated list of executables and hosts (argument -host \( \text{hostname} \)),
   - Alternatively, use the machine file to list hosts
   - Coprocessors have hostnames defined in /etc/hosts
Structure of MPI Applications

```c
#include "mpi.h"

int main(int argc, char** argv) {
    int ret = MPI_Init(&argc,&argv); // Set up MPI environment
    if (ret != MPI_SUCCESS) {
        MyErrorLogger("...");
        MPI_Abort(MPI_COMM_WORLD, ret);
    }
    int worldSize, myRank, myNameLength;
    char myName[MPI_MAX_PROCESSOR_NAME];
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Get_processor_name(myName, &myNameLength);
    // ... Perform work, exchange messages with MPI_Send, MPI_Recv, etc. ...
    // Terminate MPI environment
    MPI_Finalize();
}
```
Point to Point Communication

```c
if (rank == receiver) {
    char incomingMsg[messageLength];
    MPI_Recv (&incomingMsg, messageLength, MPI_CHAR, sender,
               tag, MPI_COMM_WORLD, &stat);
    printf ("Received message with tag %d: '%s'\n", tag, incomingMsg);
}
else if (rank == sender) {
    char outgoingMsg[messageLength];
    strcpy(outgoingMsg, "/Jenny");
    MPI_Send(&outgoingMsg, messageLength, MPI_CHAR, receiver, tag, MPI_COMM_WORLD);
}
```
Collective Communication: Broadcast

```c
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype,
                int root, MPI_Comm comm );
```
Collective Communication: Scatter

```c
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf,
                int recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm);
```
Collective Communication: Gather

```
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```
Collective Communication: Reduction

```c
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);
```

Available reducers: max/min, minloc/maxloc, sum, product, AND, OR, XOR (logical or bitwise).
Section 3 Overview: Expressing Parallelism

1. Data parallelism (vectorization)
   - Automatic vectorization by the compiler: portable and convenient
   - For-loops and array notation can be vectorized
   - Compiler hints (#pragma simd, #pragma ivdep, etc.) to assist the compiler

2. Shared-memory parallelism with OpenMP and Intel Cilk Plus
   - Parallel threads access common memory for reading and writing
   - Parallel loops: #pragma omp parallel for and _Cilk_for — automatic work distribution
   - In OpenMP: private and shared variables; synchronization, reduction.

3. Distributed-memory parallelism with MPI
   - MPI processes do not share memory, but can send information to each other
   - All MPI processes execute the same code; role is determined by its rank
   - Point-to-point and collective communication patterns
§4. Optimization Techniques

Scalar Optimizations
Optimization Level

The default optimization level `-O2`
- optimization for speed
- automatic vectorization
- inlining
- constant propagation
- dead-code elimination
- loop unrolling

Optimization level `-O3`
- enables more aggressive optimization
- loop fusion
- block-unroll-and-jam
- if-statement collapse

```
user@host% icc -o mycode -O3 source.c
```

```
#pragma intel optimization_level 3
void my_function() {
    //...
}
```
Using the `const` Qualifier

```c
#include <stdio.h>
int main() {
    const int N = 1 << 28;
    double w = 0.5;
    double T = (double)N;
    double s = 0.0;
    for (int i = 0; i < N; i++)
        s += w * (double)i / T;
    printf("%e\n", s);
}
```

```
#include <stdio.h>
int main() {
    const int N = 1 << 28;
    const double w = 0.5;
    const double T = (double)N;
    double s = 0.0;
    for (int i = 0; i < N; i++)
        s += w * (double)i / T;
    printf("%e\n", s);
}
```

```
user@host% icpc noconst.cc
user@host% time ./a.out
6.710886e+07
real 0m0.461s
user 0m0.460s
sys 0m0.001s
```

```
user@host% icpc const.cc
user@host% time ./a.out
6.710886e+07
real 0m0.097s
user 0m0.094s
sys 0m0.003s
```
Array Reference by Index instead of Pointer Arithmetics

With **Pointer arithmetics**, the code is 5x slower than with reference to array elements by index.
Common Subexpression Elimination

- The value of \( \sin_A \) can be calculated once and re-used \( m \) times in the \( j \)-loop
- In some cases, at -O2 compiler eliminates common subexpressions automatically
Ternary if-operator Trap

- Ternary if operator ( ? : ) is a short-hand for if ... else
- Example: the min() function as a pre-processor expression

```
#define min(a, b) ((a) < (b) ? (a) : (b))
const float c = min(my_function(x), my_function(y));
```

- Problem: line 2 calls my_function() 3 times
- Optimization:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
const float result_a = my_function(x);
const float result_b = my_function(y);
const float c = min(result_a, result_b);
```
Consistency of Precision: Constants

1. Operations on type `float` is faster than operations on type `double`. Avoid type conversions and define single-precision literal constants with suffix `-f`.

   ```
   const double twoPi = 6.283185307179586;
   const float phase = 0.3f; // single precision
   ```

2. Use 32-bit `int` values including 64-bit `long` where possible, including array indices. Avoid type conversions and define 64-bit literal constants with suffix `-L` or `UL`.

   ```
   const long N2 = 1000000*1000000; // Overflow error
   const long N3 = 1000000L*1000000L; // Correct
   ```
Consistency of Precision: Functions

1. `math.h` contains fast single precision versions of arithmetic functions ending with suffix `-f`

   ```c
   double sin(double x);
   float sinf(float x);
   ```

2. `math.h` contains fast base 2 exponential and logarithmic functions:

   ```c
   double exp(double x); // Double precision, natural base
   float expf(float x); // Single precision, natural base
   double exp2(double x); // Double precision, base 2
   float exp2f(float x); // Single precision, base 2
   ```
#include <stdio.h>
#include <math.h>

int main() {
    const int N = 1000000;
    const int P = 10;
    double A[N];
    const double startValue = 1.0;
    A[:] = startValue;
    for (int i = 0; i < P; i++)
        #pragma simd
        for (int r = 0; r < N; r++)
            A[r] = exp(-A[r]);

    printf("Result=%.17e\n", A[0]);
}

Vadim Karpusenko (Colfax International)
Precision Control for Transcendental Functions

```
user@host% icpc -o precision-1 -mmic \
  -fimf-precision=low precision.cc
user@host% scp precision-1 mic0:~/
% precision-1  100%  11KB  11.3KB/s
user@host% ssh mic0 time ./precision-1
Result=5.68428695201873779e-01
real 0m 0.08s
user 0m 0.06s
sys 0m 0.02s
user@host%
```

```
user@host% icpc -o precision-2 -mmic \
  -fimf-precision=high precision.cc
user@host% scp precision-2 mic0:~/
% precision-2  100%  19KB  19.4KB/s
user@host% ssh mic0 time ./precision-2
Result=5.68428725029060722e-01
real 0m 0.14s
user 0m 0.12s
sys 0m 0.02s
user@host%
```
Library Functions for Standard Tasks

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    const size_t N = 1<<29L;
    const size_t F = sizeof(float);
    float* A = (float*)malloc(N*F);
    srand(0); // Initialize RNG
    for (int i = 0; i < N; i++) {
        A[i] = (float)rand() / (float)RAND_MAX;
    }
    printf("Generated %ld random numbers\nA[0]=%e\n", N, A[0]);
    free(A);
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <mkl_vsl.h>

int main() {
    const size_t N = 1<<29L;
    const size_t F = sizeof(float);
    float* A = (float*)malloc(N*F);
    VSLStreamStatePtr rnStream;
    vslNewStream( &rnStream, //Init RNG
                  VSL_BRNG_MT19937, 1 );
    vsRngUniform(VSL_RNG_METHOD_UNIFORM_STD, rnStream, N, A, 0.0f, 1.0f);
    printf("Generated %ld random numbers\nA[0]=%e\n", N, A[0]);
    free(A);
}
```
Library Functions for Standard Tasks

```bash
user@host% icpc -mmic -o rand % rand.cc
user@host% # Run on coprocessor
user@host% # and benchmark
user@host% time micnativeloadex % rand
Generated 536870912 random numbers
A[0]=8.401877e-01

real 0m56.591s
user 0m0.002s
sys 0m0.011s
```

```bash
user@host% icpc -mkl -mmic -o % rand-mkl rand-mkl.cc
user@host% export SINK_LD_LIBRARY_PATH=% /opt/intel/composerxe/mkl/lib/mic:
% /opt/intel/composerxe/lib/mic
user@host% time micnativeloadex rand-mkl
Generated 536870912 random numbers
A[0]=1.343642e-01

real 0m7.951s
user 0m0.053s
sys 0m0.168s
```

On Intel Xeon Phi coprocessor, *random number generation* with Intel MKL is 7x faster than with the C standard Library.
Data Parallelism: Vectorization the Right and Wrong Way
Unit-Stride Access and Spatial Locality of Reference

- optimization advice for automatic vectorization
- unit-stride access
- elimination of real and assumed vector dependence
- data alignment
- supplementary compiler functions
- vectorization pragmas
- vectorization report

The rule of thumb for achieving unit-stride access

Use structures of arrays (SoA) instead of arrays of structures (AoS)
Unit-Stride Access and Spatial Locality of Reference

\[
\Phi(\tilde{R}_j) = -\sum_{i=1}^{m} \frac{q_i}{|\tilde{r}_i - \tilde{R}_j|},
\]

(1)

\[
|\tilde{r}_i - \tilde{R}| = \sqrt{(r_{i,x} - R_x)^2 + (r_{i,y} - R_y)^2 + (r_{i,z} - R_z)^2}.
\]

(2)
Elegant, but Inefficient Solution

```
struct Charge {
    // Elegant, but ineffective data layout
    float x, y, z, q;
} chg[m];       // Coordinates and value of this charge

for (int i=0; i<m; i++) {
    // This loop will be auto-vectorized
    // Non-unit stride: (&chg[i+1].x - &chg[i].x) != sizeof(float)
    const float dx=chg[i].x - Rx;
    const float dy=chg[i].y - Ry;
    const float dz=chg[i].z - Rz;
    phi -= chg[i].q / sqrtf(dx*dx+dy*dy+dz*dz);       // Coulomb’s law
}
```
Arrays of Structures versus Structures of Arrays of Structures

Array of Structures (AoS)

```c
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)

```c
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
};
```
Optimized Solution with Unit-Stride Access

```c
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);
}
```
Electric potential calculation with Coulomb’s law

**Electric potential calculation**

- **Non-unit stride (array of structures)**: 0.90 s (Host system), 0.73 s (Intel Xeon Phi Coprocessor)
- **Unit-stride (structure of arrays)**: 0.51 s (Host system), 0.37 s (Intel Xeon Phi Coprocessor)
- **Unit-stride with relaxed precision**: 0.51 s (Host system), 0.22 s (Intel Xeon Phi Coprocessor)
// Compiler may be able to simultaneously parallelize and auto-vectorize it
#pragma omp parallel for
#pragma simd
for (int i = 0; i < n; i++) {
    // ... do work
}

// The strip-mining technique separates parallelization from vectorization
const int STRIP=80;
#pragma omp parallel for
for (int ii = 0; ii < n; ii += STRIP)
#pragma simd
    for (int i = ii; i < ii + STRIP; i++) {
        // ... do work
    }
Loop Collapse

// In the example below, m iterations are distributed across threads
#pragma omp parallel for
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) { /* ... do work */ }

// In the example below, m*n iterations are distributed across threads
#pragma omp parallel for collapse(2)
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) { /* do work */ }

// The example below demonstrates explicit loop collapse.
// A total of m*n iterations are distributed across threads
#pragma omp parallel for
for (int c = 0; c < m*n; c++) {
    const int i = c / n;  const int j = c % n;
    /* ... do work */
}
Row-Wise Reduction of a Short, Wide Matrix

\[ S_i = \sum_{j=0}^{n} M_{ij}, \quad i = 0 \ldots m. \]  

- \( m \) is small, smaller than the number of threads in the system
- \( n \) is large, large enough so that the matrix does not fit into cache

```c
void sum_unoptimized(const int m, const int n, long* M, long* s){
    #pragma omp parallel for
    for (int i=0; i<m; i++) {
        long sum=0;
        #pragma simd
        #pragma vector aligned
        for (int j=0; j<n; j++)
            sum+=M[i*n+j];
        s[i]=sum; }
}
```
Row-Wise Reduction of a Short, Wide Matrix
void sum_stripmine(const int m, const int n, long* M, long* s) {
    const int STRIP = 800;
    assert(n % STRIP == 0);
    s[0:m] = 0;
    #pragma omp parallel
    {
        long sum[m]; sum[0:m] = 0;
        #pragma omp for collapse(2) schedule(guided)
        for (int i = 0; i < m; i++)
            for (int jj = 0; jj < n; jj += STRIP)
                #pragma simd
                #pragma vector aligned
                for (int j = jj; j < jj + STRIP; j++)
                    sum[i] += M[i*n + j];
        #pragma omp atomic
        for (int i = 0; i < m; i++)  // Reduction
            s[i] += sum[i];
    }
}
Strip-Mining and Loop Collapse
Row-Wise Reduction of a Short, Wide Matrix

Parallel row-wise matrix reduction

- Host system
- Intel Xeon Phi Coprocessor

<table>
<thead>
<tr>
<th>Method</th>
<th>Performance, GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>47.5</td>
</tr>
<tr>
<td>Parallel inner loop</td>
<td>38.6</td>
</tr>
<tr>
<td>Collapse nested loops</td>
<td>28.3</td>
</tr>
<tr>
<td>Strip-mine and collapse</td>
<td>53.7</td>
</tr>
<tr>
<td>Intel Xeon Phi Coprocessor</td>
<td>131.6</td>
</tr>
</tbody>
</table>
Loop Tiling (Blocking)

// Plain nested loops
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        compute(a[i], b[j]); // Memory access is unit-stride in j

// Tiled nested loops
for (int ii = 0; ii < m; ii += TILE)
    for (int j = 0; j < n; j++)
        for (int i = ii; i < ii + TILE; i++) //Re-use data for each j with several i
            compute(a[i], b[j]); // Memory access is unit-stride in j

// Doubly tiled nested loops
for (int ii = 0; ii < m; ii += TILE)
    for (int jj = 0; jj < n; jj += TILE)
        for (int i = ii; i < ii + TILE; i++) //Re-use data for each j with several i
            for (int j = jj; j < jj + TILE; j++)
                compute(a[i], b[j]); // Memory access is unit-stride in j
Tiling a Parallel For-Loop (Matrix Transposition)

```c
_Cilk_for (int ii = 0; ii < n; ii += TILE) {
    const int iMax = (n < ii+TILE ? n : ii+TILE);
    for (int jj = 0; jj <= ii; jj += TILE) {
        for (int i = ii; i < iMax; i++) {
            const int jMax = (i < jj+TILE ? i : jj+TILE);
            #pragma loop count avg(TILE)
            #pragma simd
            for (int j = jj; j<jMax; j++) {
                const int c = A[i*n + j];
                A[i*n + j] = A[j*n + i];
                A[j*n + i] = c;
            }
        }
    }
}
```

- loop tiling is 5x-6x faster
Tiling a Parallel For-Loop (Matrix Transposition)

Parallel, in-place square matrix transposition

- **Host system**
  - Unoptimized ("cache-ignorant"): 545 ms
  - Loop tiling ("cache-aware"): 112 ms
  - Recursion ("cache-oblivious"): 76 ms

- **Intel Xeon Phi Coprocessor**
  - Unoptimized ("cache-ignorant"): 478 ms
  - Loop tiling ("cache-aware"): 95 ms
  - Recursion ("cache-oblivious"): 71 ms

Time, ms (lower is better)
Tiling a Parallel For-Loop (VTune analysis)
The Monte Carlo Method of Computing the Number \( \pi \)

\[ A_{\text{quarter circle}} = \frac{1}{4} \pi R^2 \]

\[ A_{\text{square}} = L^2. \]

\[ \langle N_{\text{quarter circle}} \rangle = \frac{A_{\text{quarter circle}}}{A_{\text{square}}} N. \]

\[ 4 \frac{\langle N_{\text{quarter circle}} \rangle}{N} = 4 \frac{\pi R^2}{4L^2} = \pi. \]

\[ \pi \approx 4 \frac{N_{\text{quarter circle}}}{N}. \]

\[ \pi = 3.141592653589793\ldots \]
The Monte Carlo Method of Computing the Number $\pi$

```c
#include <mkl_vsl.h>

const long BLOCK_SIZE = 4096;

// Random number generator from MKL
VSLStreamStatePtr stream;
vs1NewStream( &stream, VSL_BRNG_MT19937, seed );

for (long j = 0; j < nBlocks; j++) {
    vsRngUniform( 0, stream, BLOCK_SIZE*2, r, 0.0f, 1.0f );
    for (i = 0; i < BLOCK_SIZE; i++) {
        const float x = r[i];
        const float y = r[i+BLOCK_SIZE];
        if (x*x + y*y < 1.0f) dUnderCurve++;
    }
}

const double pi = (double)dUnderCurve / (double)iter * 4.0
```
The Monte Carlo Method of Computing the Number $\pi$

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

const double blocksPerProc = (double)nBlocks / (double)nRanks;
const long myFirstBlock = (long)(blocksPerProc*rank);
const long myLastBlock = (long)(blocksPerProc*(rank+1));

RunMonteCarlo(myFirstBlock, myLastBlock, stream, dUC);
// Compute $\pi$
MPI_Reduce(&dUC, &UnderCurveSum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank==0)
    const double pi = (double)UnderCurveSum / (double)iter * 4.0 ;

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
```
The Monte Carlo Method of Computing the Number $\pi$

Host, coprocessor, heterogeneous

<table>
<thead>
<tr>
<th>Command</th>
<th>Time, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpirun -np 32 -host localhost ./pi_mpi</td>
<td>0.84</td>
</tr>
<tr>
<td>mpirun -np 240 -host mic0 ~/pi_mpi</td>
<td>0.44</td>
</tr>
<tr>
<td>mpirun -np 32 -host localhost ./pi_mpi : -np 240 -host mic0 ~/pi_mpi</td>
<td>0.36</td>
</tr>
</tbody>
</table>

- Coprocessor is 1.9x faster than the host system
- $T_{host} \approx 0.84$ seconds, $T_{Phi} \approx 0.44$ seconds
- Expect $T_{both} = 1/(1/0.84 + 1/0.44) \approx 0.29$ seconds
- $T_{measured} \approx 0.36$ seconds, which is 25% worse than expected. Why?
Using Intel Trace Analyzer and Collector
Load Balancing with Static Scheduling

\[ \alpha = \frac{b_{\text{host}}}{b_{\text{MIC}}}, \]

\[ B_{\text{total}} = b_{\text{host}} P_{\text{host}} + b_{\text{MIC}} P_{\text{MIC}}, \quad (4) \]

\[ b_{\text{host}} = \frac{B_{\text{total}}}{\alpha P_{\text{host}} + P_{\text{MIC}}}, \quad (5) \]

\[ b_{\text{MIC}} = \frac{\alpha B_{\text{total}}}{\alpha P_{\text{host}} + P_{\text{MIC}}}. \quad (6) \]
Load Balancing with Static Scheduling

Effect of load balancing between host and coprocessor in the Monte Carlo calculation of $\pi$

- **Run time**
- **Load imbalance on Host**
- **Load imbalance on Coprocessor**

Baseline (no load balancing)

Theoretical best

Parameter $\alpha$

Time, s (lower is better)

Chapters 3-4 – September 20, 2013
Load Balancing with Static Scheduling

Load balance: execution times

- Xeon only (32 processes)
- Xeon Phi only (240 processes)
- Xeon + Xeon Phi, $\alpha = 1.0$
- Xeon + Xeon Phi, $\alpha = 3.4$

Time, s (lower is better): 0.839, 0.449, 0.366, 0.283
Summary

1. Scalar optimizations:
   - Maintain consistency of precision (single/double) in constants and functions
   - Implement strength reduction and common subexpression elimination

2. Automatic vectorization optimizations:
   - Structures of arrays instead of arrays of structures (use unit-stride access)
   - Produce and read the vectorization report (-vec-report3) for diagnostics

3. Shared-memory optimizations:
   - Avoid synchronization. Instead, use reduction where possible
   - Expand parallel iteration space to utilize all threads
   - Data locality. Cache-oblivious algorithms

4. Distributed Memory calculations:
   - Heterogeneous computation environment requires appropriate load balancing.
Thank you!

http://research.colfaxinternational.com/
<vadim@colfax-intl.com>