C Language Constructs for Parallel Programming

Robert Geva

Cilk Plus

Parallel tasks	• Easy to learn: 3 keywords • Tasks, not threads • Load balancing
Hyper Objects	• Mitigate data races on non-local variables
Array notations	• Data-parallel array operations • Targets SIMD
Elemental Functions	• Data-parallel function mapping
SIMD Loops	• Vectorization annotation for loops • Single threaded vector parallelism

cilk_spawn and cilk_sync Keywords

```
int tree_walk(node *nodep)
{
    int a = 0, b = 0;
    if (nodep->left)
        a = _Cilk_spawn tree_walk(nodep->left);
    if (nodep->right)
        b = _Cilk_spawn tree_walk(nodep->right);
    int c = f(nodep->value);
    _Cilk_sync;
    return a + b + c;
}
```

Asynchronous recursive call to tree_wak

Call to f() can run in parallel with recursive tree walks

Implicit sync at the end of every function keeps code well structured

“Serialization” of Tree-walk Example

```
int tree_walk(node *n)
{
    int a = 0, b = 0;
    if (n->left)
        a = _Cilk_spawn tree_walk(n->left);
    if (n->right)
        b = _Cilk_spawn tree_walk(n->right);
    int c = f(n->value);
    _Cilk_sync;
    return a + b + c;
}
```

Example of keyword vs. pragma

```
X = f1(a,b) + _Cilk_spawn f2(c,d);
```

```
X = _Cilk_spawn f1(a,b) + f2(c,d);
```

- The above is currently disallow in Cilk Plus
  - But this is not a necessary restriction
  - Can be allowed
- The pragmas are separate from the C expression
- Hard to point out an exact point within a sub expression

Optimization Notice

cilk_for Loop

Loop invariant.

```
cilk_for (int i = start; i < finish; i += stride)
    { /* Body of loop uses i */ }
f();
```

Iterations can execute in parallel.

All iterations complete before f() execute

The loops has to be a countable loop

Multiple linear increment allowed

Reducer Hyperobjects

- "Traditional" reduction on a parallel for loop:

```
long a[sz];
reducer_opadd<int> sum = 0;
cilk_for (int i = 0; i < sz; ++i)
    sum += a[i];
```

Parallel accesses each get their own "view"

- Generalized reduction for any code executing in parallel:

```
reducer_opadd<int> sum = 0;
void sum_tree(node* nodep) {
  if (nodep->left) cilk_spawn sum_tree(nodep->left);
  if (nodep->right) cilk_spawn sum_tree(nodep->right);
  sum  += nodep->value;
}
```

Array Notation Example

- Serial Example

```
float dot_product(unsigned int sz,
                  float A[], float B[]) {
    float dp=0.0f;
    for (int i=0; i<size; i++)
         dp += A[i] * B[i];
    return dp;
}
```

- Array Notation Version

```
float dot_product(unsigned int sz,
                  float A[], float B[]) {
    return __sec_reduce_add(A[0:sz] * B[0:sz]);
}
```

Intrinsic reduction

Array Section

Element-wise multiplication

Rank and Shape

- An array section doesn't have a new kind of type
  - the type of an array section is exactly that of the analogous subscript expression.
  - Additionally, an array section has rank and shape.
- A section implicitly iterates over some elements of an array.
  - Rank is the number of levels of loop nesting (i.e. dimensions) in the iteration space.
  - Shape is a (mathematical) vector of lengths. (The rank is the same as the length of the shape vector.)

Rank and Shape (continued)

- The rank of an expression is determined statically. In general the shape of a section is determined dynamically.

Expression	Rank	Shape
`a[0]`	0	
`a[0:n]`	1	n
`a[0][i:10]`	1	10
`a[i:n][j:m]`	2	n×m

Shapes have to match

- If array size is not known, both lower-bound and length must be specified
- Section ranks and lengths (“shapes”) must match.
  - Scalars are OK.

```
a[0:5] = b[0:6]; // No. Size mismatch.
a[0:5][0:4] = b[0:5]; // No. Rank mismatch.
a[0:5] = b[0:5][0:5]; // No. No 2D->1D
a[0:4] = 5; // OK. 4 elements of A filled w/ 5.
a[0:4] = b[i]; // OK. Fill with scalar b[i].
a[10][0:4] = b[1:4]; // OK. Both are 1D sections.
b[i] = a[0:4]; // No. 1D → 0 D
```

Array Notations → Vector Operations

- Selection of array elements
  - "vector" refers to a 1D array. Current implementation is does not allow [:] to be overloaded, e.g., for std::vector.

```
A[:]        // All of vector A
B[2:6]      // Elements 2 to 7 of vector B
C[:][5]     // Column 5 of matrix C
D[0:3:2]    // Elements 0,2,4 of vector D
```

- Masked vector operations

```
if (a[:] > b[:]) {          // Create a (logical) bit-mask, M
    c[:] = d[:] * e[:];     // For elements where M contains 1
} else {
    c[:] = d[:] * 2;        // For elements where M contains 0
}
```

Array x scalar operation

Vector Loop: Order of Evaluation

```
simd_for (int n = 0; n < N; ++n) {
    a[n] += b[n];
    c[n] += d[n];
}
```

```
for (int n = 0; n < N; n+=2) {
    t1 = a[n]; t2 = a[n+1]; // a[n+1] can be written
                            // before c[n] and d[n] are read
    t5 = b[n]; t6 = b[n+1];
    t1 += t5; t2 += t6;
    a[n] = t1; a[n+1] = t2;
    t3 = c[n]; t4 = c[n+1]; // c[n+1] can only be accessed
                            // after a[n]
    t5 = d[n]; t6 = d[n+1];
    t3 += t5; t4 += t6
    c[n] = t3; d[n] = t4;
}
```

Uniform vs. Private: Illustration

```
double b = get_position();
simd_for (int i = 0; i < N; ++i) {
    double t;
    t = y[i] * cos(z[i]);
    a[i] = t / b;
}
```

- b is uniform, t is private
  - The proposal is mapping the concepts of a uniform and a private variables onto existing syntax
- Assignments to b inside the loop shall result in uniform values, otherwise the behavior is undefined.

Elemental Functions - Example

- Defining an elemental function:

```
double option_price_call_black_scholes(
    double S, double K, double r,
   double sigma, double time) _Simd
{
    double time_sqrt = sqrt(time);
    double d1 = (log(S/K)+r*time)/(sigma*time_sqrt) +
        0.5*sigma*time_sqrt;
    double d2 = d1-(sigma*time_sqrt);
    return S*N(d1) - K*exp(-r*time)*N(d2);
}
```

Illustration

```
void
vec_add ( float *r, float *op1, float *op2, int i)
    simd (chunk (N))
    simd (uniform (r,op1, op2) , linear (i), chunk(N))
{
    r[i] = op1[i] + op2[i];
}
```

Two vector versions and one scalar

```
ssimd_for (int i = 0; i<N; ++i) {
    vec_add(a,b,c,i);
}
```

Call matches the version with the uniforms

```
simd_for (int i = 0; i<N; ++i) {
    vec_add(a[x1[[i]],b[x2[[i]],c[x3[[i]],i);
}
```

Call matches the version w/o the uniforms intel®

Software

Optimization Notice

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Cilk, Core Inside, FlashFile, i960, InstantIP, Intel, the Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

http://intel.com/software/products

Joint proposal between Cilk Plus and OpenMP

- A minimal language
- The language does not mandate a scheduling technique
- The language allows / does not disallow dynamic load balancing
- Serial semantics and serial equivalence
- Well integrated into the C language