C Language Constructs for Parallel Programming

Robert Geva
Today’s objective

- Present a proposal for addition of language constructs for parallel programming to C
- Get feedback:
  - Is there an interest in adding parallel programming to C?
  - Possible next steps
Parallel Programming Required for Current HW

Multiple cores

Tasks

SIMD instructions

Vectors

Array Notation

Vector loops
Why Parallelism?

• Virtually all computers today contain multiple cores and vector instruction sets,
  – Even mobile devices are rapidly catching up.

• Many-core architectures such as Intel’s MIC and modern GPUs are being tapped for computation.

• It is more power efficient to use multiple compute elements than to increase the clock rate of a single element.

• These developments will continue/accelerate
Why Add Parallelism Constructs to C

• Parallel programming is **Hard!**
• Without standard support, parallel programming often falls back on error-prone, ad-hoc protocols.
• Programming directly with threads often leads to undesirable non-determinism

• Treads and locks are not composable: Combining components introduces errors (e.g., deadlocks) or performance problems (e.g., oversubscription).

• C is behind other languages: OpenMP, OpenCL etc

Multicore and vector parallelism technologies have matured. It is time that we give C programmers access to them.
Parallelism versus Concurrency

**Parallel computing**
A form of computing in which computations are broken into many pieces that are executed simultaneously.

**Concurrent computing**
A form of computing in which computations are designed as collections of interacting processes.
Characteristics of the Proposal

1. Standardize existing practices
   – Codify what users are actually doing

2. Based on existing implementations
   – Intel compiler, GCC, similar concepts in other languages,
     many years of Cilk research

3. A composable tasking model

4. Parallelism is not mandatory, can be turned off,
   with serial equivalence

5. Vector programming
### Cilk Plus

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallel tasks</strong></td>
<td>• Easy to learn: 3 keywords&lt;br&gt;&lt;br&gt;• Tasks, not threads&lt;br&gt;&lt;br&gt;• Load balancing</td>
</tr>
<tr>
<td><strong>Hyper Objects</strong></td>
<td>• Mitigate data races on non-local variables</td>
</tr>
<tr>
<td><strong>Array notations</strong></td>
<td>• Data-parallel array operations&lt;br&gt;&lt;br&gt;• Targets SIMD, GPU</td>
</tr>
<tr>
<td><strong>Elemental Functions</strong></td>
<td>• Data-parallel function mapping</td>
</tr>
<tr>
<td><strong>SIMD Loops</strong></td>
<td>• Vectorization annotation for loops&lt;br&gt;&lt;br&gt;• Single threaded vector parallelism</td>
</tr>
</tbody>
</table>
#include <cilk/cilk.h>
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;
}
Spawning is not Thread Creation

• Spawns and syncs describe the parallel structure of the code.
  – Code is *processor oblivious*: the number of cores is not specified.
  – Expressed parallelism usually exceeds actual parallelism

• A *cilk_spawn* gives the runtime *permission* to continue in parallel.
  – No new threads are created
  – Low cost (5x to 10x cost of a function call)

• A *cilk_sync* is a local synchronization point
  – No global barrier is implied
  – Threads do not stall on a sync.
"Serialization" of Tree-walk Example

```c
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;
}
```
Why Work Stealing?

- A work-stealing scheduler can be shown mathematically to be within a factor of 2 of optimal, for a program with sufficient parallelism.
  - In practice, it is usually very close to optimal.
  - Gracefully handles control-flow and data divergence.
  - Used by most modern parallel programming systems

- Intel® Cilk™ Plus implements lazy task creation
  - Scheduler performs parent stealing, not child stealing
  - Serial semantics, even when using futures or the like.
  - Deterministic memory use

- Any C++ parallel extension should support (though not necessarily require) a work stealing scheduler that uses lazy task creation.
cilk_for Loop

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

The loops has to be a countable loop
Multiple linear increments allowed

• A high-quality implementation will use dynamic load-balancing for unbalanced iterations.
• Iterations are independent -- compiler can apply data-parallel optimizations such as vectorization.

Iterations can execute in parallel.
All iterations complete before f() execute
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];

• 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;
  }
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, GPU

Elemental Functions
- Data-parallel function mapping

SIMD Loops
- Vectorization annotation for loops
- Single threaded vector parallelism
## Significance of vectorization - RTM stencil

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk</td>
<td>65.64</td>
<td>33.18</td>
<td>16.83</td>
<td>9.13</td>
<td>13.17</td>
<td>5.04</td>
<td>5.76</td>
</tr>
<tr>
<td>Cilk+vec</td>
<td>12.96</td>
<td>6.4</td>
<td>3.38</td>
<td>2.06</td>
<td>2.23</td>
<td>1.56</td>
<td>1.73</td>
</tr>
<tr>
<td>OpenCL</td>
<td>17.72</td>
<td>9.5</td>
<td>4.73</td>
<td>2.51</td>
<td>2.84</td>
<td>1.65</td>
<td>1.89</td>
</tr>
<tr>
<td>TBB</td>
<td>74.66</td>
<td>32.93</td>
<td>16.91</td>
<td>8.88</td>
<td>12.42</td>
<td>6.26</td>
<td>6.29</td>
</tr>
<tr>
<td>TBB+vec</td>
<td>17.49</td>
<td>8.64</td>
<td>4.38</td>
<td>2.29</td>
<td>2.78</td>
<td>1.81</td>
<td>2.09</td>
</tr>
</tbody>
</table>

- In both Cilk+vec and TBB+vec, significant speed up over tasking alone, at all thread counts
- Without vectorization, OpenCL (SPMD model) wins over C/C++
And now with pictures

![Diagram showing performance comparison between Cilk, Cilk+Cean, OpenCL, TBB, and TBB+SIMD using vect/cilk and vect/tbb.]
**Significance of vectorization – Track Fitting**

<table>
<thead>
<tr>
<th>nthreads</th>
<th>cilk</th>
<th>cilk_simd</th>
<th>opencl</th>
<th>tbb</th>
<th>tbb_simd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>47.27</td>
<td>24.94</td>
<td>16.96</td>
<td>43.04</td>
<td>22.43</td>
</tr>
<tr>
<td>2</td>
<td>24.02</td>
<td>12.79</td>
<td>8.74</td>
<td>20.9</td>
<td>11.49</td>
</tr>
<tr>
<td>4</td>
<td>12.38</td>
<td>6.63</td>
<td>4.8</td>
<td>10.7</td>
<td>5.77</td>
</tr>
<tr>
<td>8</td>
<td>6.85</td>
<td>3.47</td>
<td>2.85</td>
<td>5.45</td>
<td>2.94</td>
</tr>
<tr>
<td>16</td>
<td>6.17</td>
<td>3.21</td>
<td>2.61</td>
<td>5.2</td>
<td>2.71</td>
</tr>
<tr>
<td>32</td>
<td>2.48</td>
<td>1.41</td>
<td>1.66</td>
<td>2.02</td>
<td>1.16</td>
</tr>
<tr>
<td>64</td>
<td>2.08</td>
<td>1.19</td>
<td>1.56</td>
<td>1.55</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Vector level parallelism provides significant improvement over thread level parallelism.
Array Notations

• Concise data-parallel notation encourages effective exploitation of vectors

• The [:] operator delineates an *array section*:
  
  \[
  \text{array-expression}[ \text{lower-bound} : \text{length} : \text{stride} ]
  \]

• Each argument to [:] may be omitted:
  – Default *lower-bound* is 0
  – Default *length* is the length of the array (if known)
  – Default *stride* is 1 (second colon may be omitted)

• Array sections can be used with unary and binary operators for element-by-element computation:
  
  \[
  a[10:\text{count}] = b[0:\text{count}] + c[0:\text{count}:2];
  \]

• Intrinsic functions operate on entire array sections
Array Notation Example

• Serial Example

```c
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

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rank</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>a[0:n]</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>a[0][i:10]</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>a[i:n][j:m]</td>
<td>2</td>
<td>n×m</td>
</tr>
</tbody>
</table>
Array Notations → Vector Operations

• Selection of array elements
  - “vector” refers to a 1D array. Current implementation 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[:]) {
    c[:] = d[:] * e[:];
    // For elements where M contains 1
  } else {
    c[:] = d[:] * 2;
    // For elements where M contains 0
  }

  Array x scalar operation
Elemental Functions

- A general construct to express data parallelism:
  - Write a function to describe the operation on a single element
  - Invoke the function across a parallel data structure (arrays)
  - Implementation: A high-quality compiler vectorizes across consecutive invocations of the function

- Polymorphic: a vectorizing compiler may create both array and scalar versions of the function.

- Function parameters can be varying, uniform, linear
  - Allows mapping to the most efficient load/store available.
  - Allows optimization of address computations.

- Authoring the function is independent of its invocation
  - The function can invoked on scalars, within serial for or cilk_for loops, using array notation, etc.
Elemental Functions - Example

• Defining an elemental function:
  ```c
  __declspec (vector) double option_price_call_black_scholes(
    double S, double K, double r, double sigma, double time)
  {
    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);
  }
  ```

• Invoking the elemental function:
  ```c
  // The following loop can also use cilk_for
  for (int i=0; i<NUM_OPTIONS; i++)
    call[i] = option_price_call_black_scholes(S[i], K[i], r,
                                             sigma, time[i]);
  ```
Vector loops

- Loop annotation informs the compiler that vectorized loop will have same semantics as serial loop:
  
  ```
  void f(float *a, const float *b, const int *e, int n)
  {
    simd_for (int i = 0; i < n; ++i)
      a[i] = 2 * b[e[i]];
  }
  ```

- The loop has to be countable
- Multiple linear increments allowed
- Semantics: relaxed order of evaluation to allow vectorization
  - But vectorization is not mandatory

Potential aliasing and loop-carried dependencies would thwart auto-vectorization
Vector Loops vs. Parallel Loops

• Both are countable
• Parallel loops
  – are multi threaded
  – Iterations can execute in any order
  – Admit synchronization (e.g. critical sections)
  – No data dependence
• Vector Loops
  – Are single threaded
  – Allow forward data dependence
  – No synchronization
• Prevalent use case: manage parallelism at the outer level, vectorize at the inner level
  – in a deep loop hierarchy
  – Divide and conquer algorithms
Countable Loops

some_for ( init ; compare ; increment-list ) statement

• Init: no restrictions
• Compare: must be present
  – One operand has to be a variable
• Increment-list: at least one increment
  – Increment the variable used in the compare
  – All increments are linear.
• Body: no break, return.
Cilk_for

- A countable loop $\rightarrow$ efficient scheduling
- Parallelism is allowed, not mandatory
- Same scheduler as cilk_spawn, therefore
  - Same space efficiency guarantees
  - Same serial equivalence guarantees
  - Well defined serial elision
  - Reductions works when operations combine both cilk_for and cilk_spawn
  - The body of the loop is a task block, impact the scope of a cilk_sync

- Synchronization (e.g. critical sections) is expected and allowed, and
  - Loops w/o synchronization can also be vectorized
  - When in doubt, the loop cannot be vectorized, even partially.
  - Other compiler loop optimizations apply
Example: Mandelbrot in Cilk

```
int mandel(complex c, int max_count) {
    int count = 0; complex z = 0;
    for (int i = 0; i < max_count; ++i) {
        if (abs(z) >= 2.0) break;
        z = z*z + c; count++;
    }
    return count;
}

cilk_for (int i = 0; i < max_row; i++){
    for (int j = 0; j < max_col; j++) {
        p[i][j] = mandel( complex(scale(i), scale(j)), depth);
    }
}
```
Divide and Conquer Parallelism

Split range...

.. recursively...

...until \( \leq \) grainsize.

cilk_for recursively divides a loop into tasks
Vector loops

• We are not inventing vector execution.
• We are just adding language to express it
• Vector execution is well understood, and customers have clear expectations regarding what they can do.

```
simd_for <chunk=N> (init ; compare; increment-list) statement
```
Vector Loops - Expectations

- Any loops in a loop hierarchy can be a vector loop
  - E.g. there can be a loop inside the vector loop
  - But not a parallel loop inside a vector loop
  - The vector loop participates in the compiler’s loop hierarchy optimization (blocking, splitting etc)

- The loop is countable, but trip count can be any
  - not specific to size of HW vector registers
  - The compiler is responsible for peeling (alignment) and remainder

- Functions can be called from the vector loop, execute efficiently
  - E.g. sin(), exp()

- Data alignment is not necessarily known in the lexical scope of the vector loop

- Can mix scalar and vector operations on the same data

- Some (forward) data dependence patterns are expected

- Therefore: single threaded execution expected
  - Both for semantics and performance model

- Results should be the same as if the loop was not vectorized
  - Some programmers do deviate on this expectation.

This proposal attempts to capture existing expectations, not to invent something completely new.
```c
simd_for (i=0; i<n; i++) {
    S1;
    S2;
    S3;
    S4;
}
```

- Parallel execution
  - No colored dependences allowed
- Vector execution
  - Red dependence not allowed (backward)
  - Green dependence allowed (forward)
  - Refinement with explicit chunk size
    - Red dependence allowed if dependence distance is >= chunk
Stencils

• For a given point, a *stencil* is a fixed subset of nearby neighbors.

• A *stencil code* updates every point in an d-dimensional spatial grid at time $t$ as a function of nearby grid points at times $t-1$, $t-2$, ..., $t-k$.

• Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.
Looping Implementation

A nested loop implementation is straightforward:

\[
\text{for } (t = 1; t \leq T, ++t) \{ \\
\quad \text{for } (i0 = 0, i0 < n0, ++i0) \{ \\
\quad \quad \text{for } (i1 = 0, i1 < n1, ++i1) \{ \\
\quad \quad \quad \text{for } (i2 = 0, i2 < n2, ++i2) \{ \\
\quad \quad \quad \quad \langle \langle \text{update } A[t\%k,i0,i1,i2] \text{ according to stencil } \rangle \rangle \\
\quad \quad \} \} \} \}
\]

Conventional Optimization: Loop Tiling
Issue: Looping is memory intensive, especially for parallel implementations, and it uses caches poorly. Assuming data-set size $N$, cache-block size $B$, cache size $M < N$, the number of cache misses is $\Theta(N/B)$.
Cache-Oblivious Algorithms

Divide-and-conquer cache-oblivious techniques, based on *trapezoidal decompositions* are known to be effective. DnC is a recursive algorithm that cuts the grid. The recursion is parallelized. The base case is the original loop. It should also be vectorized. It cannot be a parallel loop.
No 1:1 correspondence between source code and vector code

int A[1000]; double B[1000];
void foo(int n){
    int i;
    simd_for (i=0; i<n; i++){
        B[i] += ABS(A[i]);
    }
}

- SSE2
vpabsd  xmm0, [A+r9+rax*4]
vcvtdq2pd ymm1, xmm0
vaddpd  ymm2, ymm1, [B+r9+rax*8]
vmovupd  [B+r9+rax*8], ymm2
add     rax, 4
cmp     rax, rcx
jb      .B1.4

AVX
vpabsd  xmm0, [A+r9+rax*4]
pxor    xmm0, xmm0
pcmpgtd xmm0, xmm1
pxor    xmm1, xmm0
psubd   xmm1, xmm0
cvtdq2pd xmm2, xmm1
vaddpd  xmm2, [B+r9+rax*8]
movaps  [B+r9+rax*8], xmm2
add     rax, 2
cmp     rax, rcx
jb      .B1.4

SSSE3
movq    xmm0, [A+r9+rax*4]
pabsd   xmm1, xmm0
cvtdq2pd xmm2, xmm1
vaddpd  xmm2, [B+r9+rax*8]
movaps  [B+r9+rax*8], xmm2
add     rax, 2
cmp     rax, rcx
jb      .B1.4
An outer loop can also be a vector loop. This one has a while loop inside. It means that each “vector lane” executes the inner while loop.
Cilk™ Plus Implementation Experience

• Current features available in Intel compiler
  – For CPU, Many integrated cores (MIC), and integrated GPU
  – Run-time library is open source

• Partial implementation in Gnu compiler – ongoing

• At least three approaches have been used successfully for the work-stealing cactus stack
  – Heap-based (Cilk 5 from MIT, Cilk++ from Cilk Arts)
  – Multiple stacks (Intel® Cilk™ Plus)
  – Per-core memory-mapped stacks (Cilk M from MIT)

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>

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

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,
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.

Copyright © 2011. Intel Corporation.

http://intel.com/software/products