Trends in Parallel and Heterogeneous Programming

Robert Geva
Parallel Programming Language Architect
robert.geva@intel.com
Main Topics

- Parallelism at multiple levels
- Implicit parallelism
- Heterogeneous programming
- Composeable task parallelism
Intel Architecture Spans The Compute Continuum

Multi core everywhere
SIMD Instructions Compute Multiple Operations per Instruction

256b Intel® Advanced Vector Extensions (Intel® AVX)

Intel® microarchitecture codename Sandy Bridge
256-bit Multiply + 256-bit ADD + 256-bit Load per clock...
Double your FLOPs with great energy-efficiency
Intel® Core™ i7-980X Processor Die Map
32nm Westmere High-k + Metal Gate Transistors

Transistor count: 1.17B Die size: 248mm²

** 12MB of cache is shared across all 6 cores
Programmer Personalities

Different programmers want different levels of control over how their program executes.
OpenMP is good for monolithic applications:

A SW architect needs to break the application work into chunks, determine which thread does what work, and worry about making the threads do equal amount of work.

This model provides good performance when it works, but some applications are too complex for a single person to design with a global view.

OpenMP is hard to use when the application is composed of libraries, or of independently developed modules.
Work-stealing Task Scheduler

Each processor has a work queue of spawned tasks

When a worker hits a spawn, it posts a work item it its own work dequeue, not on another core’s

Each core has it’s own dequeue

Adapted from copyrighted originals developed by Charles E. Leiserson of MIT.
Work-stealing Task Scheduler

Each processor has a work queue of spawned tasks

Each processor places spawned work items on its own dequeue

Multiple Spawns at the same time with no synchronization

Adapted from copyrighted originals developed by Charles E. Leiserson of MIT.
Work-stealing Task Scheduler

Each processor has a work queue of spawned tasks

Upon completion of a work item, the processors pops a work item from its own dequeue

Compiler support reduces the cost of tasking

Adapted from copyrighted originals developed by Charles E. Leiserson of MIT.
Work-stealing Task Scheduler

Each processor has a work queue of spawned tasks

When each processor has work to do, a spawn is roughly the cost of a function call.

Compiler support reduces the cost of tasking

Adapted from copyrighted originals developed by Charles E. Leiserson of MIT.
**Work-stealing Task Scheduler**

Each processor has a work queue of spawned tasks

When a processor has no work, it steals from another processor.

*Work-stealing delivers load balancing*

Adapted from copyrighted originals developed by Charles E. Leiserson of MIT.
Work-stealing Task Scheduler

Each processor has a work queue of spawned tasks

With sufficient parallelism, the steals are rare, and we get *linear speedup* (ignoring memory effects)
Work Stealing Delivers Load Balancing

1. Each worker has its own work queue
2. Workers spawn work items on their own queues
3. Upon completion of work, they pop from their own queues
4. When a worker has no work, it steals from another worker
5. No syntax for the application to interfere with the dynamic scheduler
6. Scheduling works independent of program structure, across components, libraries, plus ins
Parallelism at All levels: A Three Layer Cake

- **Message Driven**
  - MPI, tbb::flow

- **Fork-Join**
  - OpenMP, TBB, or Cilk

- **SIMD**
  - Array Notations, Elemental functions

Need to be cache efficient
Elemental Functions

Use a function to describe the operation on a single element

Invoke the function in a data parallel context

The compiler generates a vector version of the function:
- Takes a vector of arguments instead of a single one
- Operates on all of them
- Vectorized the operation across them
- Each “vector lane” performs one operations of the function
- Can yield a vector of results as fast as a single result

```c
__declspec(vector) double option_price_call_black_scholes(  
double S,       // spot (underlying) price  
double K,       // strike (exercise) price,  
double r,       // interest rate  
double sigma,   // volatility  
double time)    // time to maturity
{
    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);  
}
```

// invoke calculations for call-options
Cilk_for (int i=0; i<NUM_OPTIONS; i++) {
    call[i] = option_price_call_black_scholes(S[i], K[i], r, sigma, time[i]);
}
Parallelizing Matrix Multiplication

Parallelize the loop – not cache efficient

```cilk
C+= A*B

cilk_for (int i=0; i<n; i++) {
    cilk_for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i*n+j] += A[i*n+k] * B[k*n+j];
        }
    }
}
```
Matrix multiplication using divide and conquer

The identity leads to a recursive implementation that subdivides the matrix into quadrants.
Cilk spawn is a natural way to parallelize recursion.

Quadrants are reused → cache efficiency.
But recursion is harder to write.
Main Topics

- Parallelism at multiple levels
- Implicit parallelism
- Heterogeneous programming
- Composeable task parallelism
Abstractions: TBB Pipeline

Special support for a design pattern: useful when some operations can be done in parallel, others are required to be serial

Example: read input text, square the numbers, write output text

```c
void RunPipeline( int ntoken, FILE* input_file, FILE* output_file ) {
    tbb::parallel_pipeline(
        ntoken,
        tbb::make_filter<void,TextSlice*>(
            tbb::filter::serial_in_order, MyInputFunc(input_file))
        &
        tbb::make_filter<TextSlice*,TextSlice*>(
            tbb::filter::parallel, MyTransformFunc())
        &
        tbb::make_filter<TextSlice*,void>(
            tbb::filter::serial_in_order,
            MyOutputFunc(output_file)));}
```

Provide a Specification → Get a Parallel Program
MIT Pochoir: Language for stencils

```plaintext
stencil {
    dimension: size of the grid
    shape: which neighbors are accessed
    array: the grid
    kernel: function
    boundary: function
    initialization: function
}
```

- Pochoir is a C++ based language for stencils
- The programmer writes the kernel, the data structure, the access pattern
- The Pochoir compiler generates parallel, recursive, cache oblivious optimal code
UC Berkeley SEJITS

- Selective embedded just in time specializers
- The programmer writes in Python (efficiency language)
- Uses classes provided by the SEJITS system
- Simple code, don’t worry about parallelism and performance
- A specializer is designed to work on a specific patterns
- UBC observed 50+ parallel patterns used in applications
- The system
  - reads the efficiency language (Python) program
  - generates code in lower, performance language.
  - These are C/C++ using Cilk or OMP, or CUDA.

Write simple code → get a high performance parallel code
Main Topics

- Parallelism at multiple levels
- Implicit parallelism
- Heterogeneous programming
- Composeable task parallelism
Data parallel heterogeneity: GPU and MIC

**Similarities**
- All HW dedicated to compute
- Fast GDDR memory
- Two distinct physical memories, connected via PCIe
- Many execution units

**Differences**
- Hide memory latency via cache vs. task switching
- Programming via C/FTN extensions vs. CUDA, OCL
- SW threading / tasking vs. threads in HW
- Offload anything vs. offload kernels

GPU: offload kernels. MIC: Offload anything
Heterogeneous Programming with Intel MIC

Programming MIC is the same as programming a CPU
Heterogeneous parallelism thru kernels or similar mechanisms
Improvements in HW drive improvements in ease of programming
Summary

Parallelism at multiple levels
Implicit parallelism
Heterogeneous programming
Composeable task parallelism

Integration improving
Building critical mass
Evolving Quickly
Multiple products

Developers
Rock your code.
Intel® Parallel Building Blocks
Tools to optimize app performance for the latest platform features

<table>
<thead>
<tr>
<th><strong>Intel® Cilk Plus</strong></th>
<th><strong>Intel® Threading Building Blocks</strong></th>
<th><strong>Intel® Array Building Blocks</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Language extensions to simplify data, task, and vector parallelism</td>
<td>Widely used C++ template library for data and task parallelism</td>
<td>Sophisticated C++ library for data and vector parallelism</td>
</tr>
</tbody>
</table>

*Mix and Match to Optimize Your Applications' Performance*

Compatible with Microsoft* Visual Studio* and GCC
Supports multiple operating systems and platforms

*The best solution for parallel programming*
Optimization Notice

Intel® Parallel Composer 2011 includes compiler 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® Parallel Composer 2011 are reserved for Intel microprocessors. For a detailed description of these compiler options, including the instruction sets they implicate, please refer to "Intel® Parallel Composer 2011 Documentation > Intel® C++ Compiler 12.0 User and Reference Guides > Compiler Options." Many library routines that are part of Intel® Parallel Composer 2011 are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® Parallel Composer 2011 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.

While the paragraph above describes the basic optimization approach for Intel® Parallel Composer 2011, with respect to Intel's compilers and associated libraries as a whole, Intel® Parallel Composer 2011 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.

Intel recommends that you evaluate other compilers to determine which best meet your requirements.