Our Vision: Making models survive future architectures

Single Source – portable across systems and into the future

CPU

SSE 4

AVX

ManyCore Processors

CPU w/Accelerator

LRBni

Hybrid Processors
Today’s Parallelism

• There are lots of programming options from Intel:
  – Old recommendations (e.g. OpenMP)
  – Newer recommendations (e.g. TBB, Ct)
  – New acquisitions (i.e. Cilk, RapidMind)
  – Competitive offerings (e.g. OpenCL, CUDA)

• An initiative (or a few initiatives) to clarify the programming options for our customers
#include <stdio.h>
#include <pthread.h>
const int num_threads = 4;

void* thread_func(void* arg) {
    do_work();
    return NULL;
}

int main() {
    pthread_t threads[num_threads];
    for (int i = 0; i < num_threads; i++)
        pthread_create(&threads[i], NULL, thread_func, NULL);

    for (int j = 0; j < num_threads; j++)
        pthread_join(threads[j], NULL);
}
OpenMP* Threads: Implicit

Create Threads

Split loop iterations among threads

Make local variables for each thread

```c
#pragma omp parallel for private(pixelX,pixelY) for (pixelX = 0; pixelX < imageWidth; pixelX++) {
    for (pixelY = 0; pixelY < imageHeight; pixelY++) {
        newImage[pixelX,pixelY] = ProcessPixel(pixelX, pixelY, image);
    }
}
```
**General MPI Program Structure**

```c
#include "mpi.h"

int main(argc,argv){
    rc = MPI_Init(&argc,&argv);
    if (rc != MPI_SUCCESS) {
        printf ("Error starting MPI program. \n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("Number of tasks= %d My rank= %d\n",ntasks,rank);
    /****** do some work ******/
    MPI_Finalize();
}
```
What about the other approaches?
What about Cilk++?

- Cilk++ offers a compiler keyword alternative to TBB
- Every Cilk program preserves the **serial semantic**
- Cilk provides **performance guarantees** since it is based on theoretically efficient scheduler
- There are three additional keywords: *cilk*, *spawn* and *sync*
- Cilk uses parallel stacks (cactus stacks) in contrast to linear stacks (TBB)
- Beta starts in ~ Q2 2010
Cilk - The main principle

• The programmer should be responsible for *exposing* the parallelism:
  – identifying elements that can safely be executed in parallel

• It should then be left to the run-time environment, particularly the scheduler, to decide *during execution* how to actually divide the work between processors.
  – It is because these responsibilities are separated that a Cilk program can run without rewriting on any number of processors, including one.
Fibonacci – recall the functionality

\[ F_n = \begin{cases} 
0 & \text{if } n = 0; \\
1 & \text{if } n = 1; \\
F_{n-1} + F_{n-2} & \text{if } n > 1. 
\end{cases} \]

|   | \( F_0 \) | \( F_1 \) | \( F_2 \) | \( F_3 \) | \( F_4 \) | \( F_5 \) | \( F_6 \) | \( F_7 \) | \( F_8 \) | \( F_9 \) | \( F_{10} \) | \( F_{11} \) | \( F_{12} \) | \( F_{13} \) | \( F_{14} \) | \( F_{15} \) | \( F_{16} \) | \( F_{17} \) | \( F_{18} \) |
|   | 0  | 1  | 1  | 2  | 3  | 5  | 8  | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | 610 | 987 | 1597 | 2584 |
nth Fibonacci number – C

```c
int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = fib(n-1);
        y = fib(n-2);
        return (x+y);
    }
}
```
Fibonacci – Cilk code

cilk int fib (int n) {
    if (n<2) return (n);
    else {
        int x,y;
        x = spawn fib(n-1);
        y = spawn fib(n-2);
        sync;
        return (x+y);
    }
}
Identifies a function as a Cilk procedure, capable of being spawned in parallel.

The named child Cilk procedure can execute in parallel with the parent caller.

Control cannot pass this point until all spawned children have returned. The sync done implicitly on every return.
Why Do We Need One More?

- So many cores, so hard to harvest parallelism

- Parallel programming is too hard for the masses of developers → need to emphasize ease of use
- Data races are too easy to create, way too hard to debug → preclude races through safety
- Target-specific specialization is required to get performance, but makes code very difficult to maintain and port → raise to a natural level of abstraction and enable forward scaling across architectures and ISAs
- Uniform model for harvesting SIMD and thread-level parallelism in either scalar kernels or array syntax → compiler + runtime
- ValArray too limited → universal set of operators across (even irregular) data structures
What about Threading Building Blocks?

• It’s a C++ library and integrates nicely with the STL
• TBB provides advanced C++ abstraction concepts
• Particularly suited when parallelism is hidden in generic C++ structures like containers & iterators

• You specify task patterns instead of threads
• A task scheduler does the mapping to the threads
• Targets threading for performance (not usability)
Intel® Threading Building Blocks

Generic Parallel Algorithms
- parallel_for
- parallel_reduce
- parallel_do (new)
- pipeline
- parallel_sort
- parallel_scan

Concurrent Containers
- concurrent_hash_map
- concurrent_queue
- concurrent_vector

Task scheduler

Synchronization Primitives
- atomic operations
- scoped locks

Miscellaneous
- tick_count
- tbb_thread

Memory Allocation
- tbb_allocator (new), cache_aligned_allocator, scalable_allocator
OpenCL is a portable intermediate low-level language layer for a wide variety of different Hardware (FPA, GPU, Cell, CPU, ...)

- Intel is part of the OpenCL consortium
- Intel provided input for the specification
- Plans to have some OpenCL support out next year
- OpenCL is already available on Snow Leopard

My take on OpenCL

- Data parallel meets Task parallel on a very low abstraction level
- A good target language for API developers like Ct/RapidMind
- Programmers should use higher levels of abstraction such as Ct/RapidMind, TBB, Cilk++ or OpenMP if at all possible
Performance versus effort

- Theoretical speedup limited by number of CPU's per cluster
- Theoretical speedup limited by number of Core's per CPU
- Theoretical speedup limited by Core

- Code Restructuring
- Serial optimization
- OpenMP
- TBB
- Cilk
- MPI
- Cluster

Development effort
BUT - Today’s On-going Realities

- Programming parallel apps is \(~ \textbf{100x less}\) productive than sequential
  - Non-deterministic programming errors (race conditions, ...)
  - Performance tuning is extremely complicated
- Strong interest by ISVs for a parallel programming model which is:
  - **Abstract**: Avoid dealing with OS and HW details
  - **Simple**: Deterministic, eliminate threading problems
  - **Fast & Scalable**: Achievable through simpler programming API
  - **Portable**: Desire the flexibility to target various HW platforms (CPU, LRB, GPU, Cell and a Mix)
Where does Ct fit?

“Ct Technology: a new perspective on data-parallel programming”

What’s new and different?

- Why do we need one more programming model?
- Where does Ct fit within Intel other tools?
- Where does Ct fit within the rest of the industry?
Forward Scaling with Ct

- Compile once, generating optimized, native IA code
- Dynamically reoptimize for:
  - More cores
  - More cache
  - More bandwidth
  - More instruction set enhancements

Ct forward scales software with Moore’s law for Throughput and Visual Computing
Design Constraints

Target language: C++
• C++ will continue to be the dominant languages for high performance for the next 5+ years

...and we mean **standard** C++!

• Custom syntactic extensions face huge barriers to adoption
• It is possible to design a desirable semantics through an API-like interface with some Macro magic
Design Constraints

Target language: C++
• C++ will continue to be the dominant languages for high performance for the next 5+ years

...and we mean **standard** C++!
• Custom syntactic extensions face huge barriers to adoption
• It is possible to design a desirable semantics through an API-like interface with some Macro magic
Ct’s Parallel Collections

Regular Vecs

- Vec
- Vec2D
- Vec3D
- Vec<Tuple<...>>

Irregular Vecs

- VecIndexed
- VecSparse, Vec2DSparse, VecND

& growing...
The basic type in Ct is a *polymorphic collection* called a Vec

- Vecs are managed by the Ct runtime
- Vecs are single-assignment vectors
- Vecs are flat, multidimensional, sparse, or nested
- Vec values are exclusively created & manipulated through Ct API

Declared Vecs are simply references to immutable values

```c
Vec<DoubleVec> DoubleVec; // DoubleVec can refer to any vector of doubles
...
DoubleVec = Src1 + Src2;
DoubleVec = Src3 * Src4;
```

Assigning a value to `DoubleVec` doesn’t modify the value representing the result of the add, it simply refers to a new value.
Segregated C/C++ and Ct Spaces

C/C++ space  

Ct space

Ct is garbage collected; practically, a small number of Vecs are uncollected by compiler (ref counting)
float s[N], x[N], r[N], v[N], t[N];
float result[N];

__ct{
    Vec<F32> S(s, N), X(x, N), R(r, N), V(v, N), T(t, N);
    Vec<F32> d1 = S / ln(X);
    d1 += (R + V * V * 0.5f) * T;
    d1 /= sqrt(T);
    Vec<F32> d2 = d1 - sqrt(T);
    Vec<F32> tmp = X * exp(R * T) *
        (1.0f - CND(d2)) + (-S) * (1.0f - CND(d1));
    copyOut(tmp, result, N * Sizeof(float));
}__endCt

Red color shows the differences between “normal” and function Ct code

float s[N], x[N], r[N], v[N], t[N];
float result[N];

__ct{
    Vec<F32> S(s, N), X(x, N), R(r, N), V(v, N), T(t, N);
    tmp = call(BlackScholes)(S, X, R, V, T);
    tmo.copyOut(result, N * Sizeof(float));
}__endCt
ROI Crossover Graph
Key Features and Benefits - Productivity

- **Integrates with existing IDEs, tools, and compilers**: no new compiler needed

- **Incremental**: allows selective and targeted modification of existing code bases

- **Generalized data parallel model**: widely applicable to many types of computations

- **Safe by default**: deterministic semantics avoid race conditions and deadlock by construction

- **Easy to learn**: serially consistent semantics and simple interface leverage existing skills
Key Features and Benefits - Portability

- **High-level**: avoids dependencies on particular hardware mechanisms or architectures

- **ISA extension independent**: common binary can exploit different ISA extensions transparently

- **Hardware independent**: Allows choice of deployment hardware today: including scaling to many cores

- **Scaling**: Allows migration and forward-scaling: will support AVX, Larrabee and beyond
Language Vehicle for General Purpose Parallel Programming

- Ct Api
  - Nested Data Parallelism
  - Deterministic Task Parallelism

- Deterministic parallel programming
- Fine grained concurrency and synch
- Dynamic (JIT) compilation
- High-performance memory management
- Forward-scaling binaries for SSEx, LRBNi
- Parallel application library development
- Performance tools for Future Architectures

Software & Services Group, Developer Products Division
Copyright © 2009, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
What Does the Product Based on Intel Ct Technology Look Like?

• Core API
  – Flexible, forward scaling data parallelism in C++

• Application Libraries
  – Linear Algebra, FFT, Random Number Generation
    – Powered by Intel® Math Kernel Library (Intel® MKL)!

• Samples
  – Medical Imaging, Financial Analytics, Seismic Processing, and more

• Initial release on Windows, followed by Linux
  – IA-32 and Intel® 64
  – Works with Intel® C/C++ Compiler, Microsoft* Visual C++*, and GCC*
  – Works with Intel® VTune™ Analyzer
When do we recommend what?

• TBB can be considered the lead or default option for hands-on parallel programmers
  - General solution - addresses the most use cases, supports system access
  - Widely available - open source and compiler independent

• Compiler extensions (e.g. Cilk) target minimal syntax changes
  - Provides the easiest way to introduce parallelism into an app
  - OpenMP involves less invasive code changes for very regular, singly-nested loops
    - When a compiler is preferred vs. a library
    - Unique in supporting FORTRAN
Summary

• Ct enables you to write simple parallel algorithms in standard C++
• Ct can get you performance on Intel Architecture today
• Ct apps scale to future architectures: Larrabee, SandyBridge (+AVX), and beyond
• Ct will intermix with other parallel programming models and tools

Ct extends the many choices that Intel provides for Parallel Computing
Any questions?