# Outline

<table>
<thead>
<tr>
<th>Topic</th>
<th>Minutes</th>
<th>Slides</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>Code Monkey Mechanics</td>
<td>75</td>
<td>40</td>
</tr>
<tr>
<td>Writing Scalable Programs</td>
<td>60</td>
<td>47</td>
</tr>
<tr>
<td>Writing Correct Programs</td>
<td>40</td>
<td>24</td>
</tr>
<tr>
<td>Wrap Up</td>
<td>20</td>
<td>11</td>
</tr>
</tbody>
</table>
Related Book

To be published this summer.

*Structured Parallel Programming*

• By Michael McCool
• Arch Robison
• James Reinders

Uses Cilk Plus and TBB as primary frameworks for examples.

Appendices concisely summarize Cilk Plus and TBB.
Motivation

Hardware Trends

Coordination Issues

Composability
Cores and Vectors are Growing

Need to **multithread** and **vectorize** to get full benefit from the hardware.
Coordination Issues

Load balancing
Locality
Races and determinism
Deadlock
Load Balancing and Locality
**Races**

**Thread 1**

extern float sum;
for( int i=0; i<n/2; ++i )
    sum += a[i];

**Thread 2**

extern float sum;
for( int i=n/2; i<n; ++i )
    sum += a[i];
Race-Free ≠ Deterministic

Parallel programs

Deterministic

Race free

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = 1;</code></td>
<td><code>x = 1;</code></td>
</tr>
<tr>
<td><code>m.lock();</code></td>
<td><code>m.lock();</code></td>
</tr>
<tr>
<td><code>x = 1;</code></td>
<td><code>x = 2;</code></td>
</tr>
<tr>
<td><code>m.unlock();</code></td>
<td><code>m.unlock();</code></td>
</tr>
</tbody>
</table>
Deadlock

Thread 1
a.lock();
b.lock();
++A;
--B;
b.unlock();
a.unlock();

Thread 2
b.lock();
a.lock();
--B;
++A;
a.unlock();
b.unlock();
Serial-Parallel Composition
Parallel-Parallel Composition (Nesting)
Compose Threads *and* Vector Parallelism

**Message Driven**

MPI, tbb::flow

**Fork-Join**

OpenMP, TBB, or **Cilk**

**SIMD**

Array Notation or `#pragma SIMD`

*Other brands and names are the property of their respective owners.*
# Philosophy of Cilk Plus

## Division of Responsibilities

<table>
<thead>
<tr>
<th>Programmer</th>
<th>Cilk Plus</th>
</tr>
</thead>
</table>
| Specify what *can* run in parallel. | Make parallelism easy to express.  
| | Enable clean composition. |
| Provide much more *potential* parallelism than system can use. | Throttle *actual* parallelism.  
| | • Make unused parallelism cheap.  
| | • Balance load. |
| Express SIMD opportunities. | Make SIMD easy to express.  
| | Generate SIMD code. |
| Avoid races. | Synchronize strands of execution. |
| Minimize use of locks. | Provide hyperobjects. |
| Promote locality via cache-oblivious style. | Depth-first serial execution. |
Code Monkey Mechanics

Thread Parallelism

Vector Parallelism

Example

Summary of Common Parallel Patterns
Summary of Cilk™ Plus

Thread Parallelism
- cilk_spawn
- cilk_sync
- cilk_for

Reducer
- reducer
- reducer_op{add, and, or, xor}
- reducer_{min, max}{_index}
- reducer_list
- reducer_ostream
- reducer_string

Vector Parallelism
- array notation
- #pragma simd
- elemental functions

Can borrow atomic operations, scalable memory allocator, concurrent containers from Intel®TBB.
Note on Cilk™ Plus Keywords

Include `<cilk/cilk.h>` to get nice spellings

```c
#include <cilk/cilk.h>

int main() {
    cilk_for( int i=0; i<10; ++i ) {
        cilk_spawn f();
        g();
        cilk_sync;
    }
}
```

In `<cilk/cilk.h>`

- `#define cilk_spawn _Cilk_spawn`
- `#define cilk_sync _Cilk_sync`
- `#define cilk_for _Cilk_for`
cilk_for

Grants permission to run iterations in parallel.

```c
void saxpy( float a, float x[], float y[], size_t n ) {
    cilk_for( size_t i=0; i<n; ++i )
    y[i] += a*x[i];
}
```
Syntax for cilk_for

Has restrictions so that iteration space can be computed before executing loop.

```
cilk_for( type index = expr; condition; incr )
body;
```

Must be integral type or random access iterator.

`limit` and `stride` might be evaluated only once.

Iterations must be okay to execute in parallel.

Restrictions:

- `index` relop `limit` and `stride` might be evaluated only once.
- `index += stride` or `index -= stride`
- `++index` or `--index`
Controlling grainsize

By default, cilk_for tiles the iteration space.
• Thread executes entire tile
• Avoids excessively fine-grained synchronization

For severely unbalanced iterations, this might be suboptimal.
• Use **pragma cilk grainsize** to specify size of a tile

```c
#pragma cilk grainsize = 1
cilk_for( int i=0; i<n; ++i )
    a[i] = f(b[i]);
```
cilk_spawn/cilk_sync

spawn = asynchronous function call

Optional assignment

\[
x = \text{cilk\_spawn} \ f(*p++);
\]
\[
y = g(*p--);
\]
\[
cilk\_sync;
\]
\[
z = x+y;
\]
cilk_sync Has Function Scope

Scope of `cilk_sync` is entire function.

There is implicit `cilk_sync` at end of a function.

```c
void bar() {
    for( int i=0; i<3; ++i ) {
        cilk_spawn f(i);
        if( i&1 ) cilk_sync;
    }
    // implicit cilk_sync
}
```

**Serial call/return property:**
All Cilk Plus parallelism created by a function completes before it returns.
Style Issue

// Bad Style
cilk_spawn f();
cilk_spawn g();
// nop
cilk_sync;

// Preferred style
cilk_spawn f();
g();
// nop
cilk_sync;

Wasted fork
Spawning a Statement in Cilk Plus

Just spawn a lambda expression.

cilk_spawn [&]{
    for( int i=0; i<n; ++i )
        a[i] = 0;
}();

... 
cilk_sync;

Do not forget the ().
Reducers in Cilk Plus

Enable lock-free use of shared reduction locations

• Work for any associative reduction operation
• Does not have to be commutative
• Reducer does not have to be local variable, or even be a variable!

```cpp
silk::reducer_opadd<float> sum = 0;
...
    silk_for(size_t i=1; i<n; ++i)
        sum += f(i);
...
    = sum.get_value();
```

Updates local view of `sum`.

Get global view of `sum`.

Not lexically bound to a particular loop.
Summary of Predefined Reducers

add

bitwise and, or, xor

min, max, index of max, index of min

append to list, prepend to list

append to string

output to ostream

holder
Global Reducer with Non-Commutative Operation

```cpp
#include <iostream>
#include <cilk/reducer_ostream.h>

cilk::reducer_ostream Out(std::cerr);

void Iota(int i, int j) {
    if (j>i+1) {
        int m = i+(j-i)/2;
        cilk_spawn Iota(i,m);
        Iota(m,j);
    } else if (j>i)
        Out << i << std::endl;
    // implicit cilk_sync
}

int main() {
    Iota(0,1000);
}
```

Even with parallel execution, final output is identical to output from serial execution.
Serial Elision

Cilk keywords can be trivially eliminated:

```c
#define cilk_spawn
#define cilk_sync
#define cilk_for for
```

Resulting program is called the **serial elision**

- It is a valid serial C/C++ program!

Likewise, the serial elision is always a valid implementation of a Cilk program:

- Means a Cilk program can always run on a single thread.
- Fundamental requirement for avoiding oversubscription.
Races

Race

- Two unordered memory references and at least one is a write.

Cilk program is deterministic if:

- It has no races
- It uses no locks
- Reducer operations are associative

Deterministic Cilk program has same effect as its serial elision.

Floating-point + and * are almost associative.

Will talk about automatic race detection later.
Array Notation

Grants permission to vectorize

```c
// Set y[i] ← y[i] + a · x[i] for i ∈ [0..n)
void saxpy(float a, float x[], float y[], size_t n) {
    y[0:n] += a*x[0:n];
}
```
Array Section Notation

$base[first:length:stride]$  
- **Pointer or array**
- **First index**
- **Optional stride**
- **Number of elements (different than F90!)**

**Rules for $section_1$ op $section_2$**
- Elementwise application of $op$
- Also works for $func(section_1, section_2)$
- Sections must be the same length
- Scalar arguments implicitly extended
More Examples

• Rank 2 Example – Update m×n tile with corner [i][j].

\[ Vx[i:m][j:n] += a \times (U[i:m][j+1:n] - U[i:m][j:n]); \]

• Function call

\[ \text{theta}[0:n] = \text{atan2}(y[0:n], 1.0); \]

• Gather/scatter

\[ w[0:n] = x[i[0:n]]; \]
\[ y[i[0:n]] = z[0:n]; \]
Improvement on Fortran 90

Compiler does not generate temporary arrays.
• Would cause unpredictable space demands and performance issues.
• Want abstraction with minimal penalty.
• Partial overlap of left and right sides is undefined.

Exact overlap still allowed for updates.
• Just like for structures in C/C++.

x[0:n] = 2*x[1:n]; // Undefined – partial overlap*
x[0:n] = 2*x[0:n]; // Okay – exact overlap
x[0:n:2] = 2*x[1:n:2]; // Okay – interleaved

*unless n≤1.
Reductions

A __sec_reduce_op reduces a section to a scalar.

Examples:

- Dot product
  
  ```c
  sum = __sec_reduce_add(a[0:n]*b[0:n]);
  ```

- Index of minimum element of d[s], d[s+1], ... d[s+n-1]
  
  ```c
  int i = s + __sec_reduce_min_ind(d[s:n]);
  ```
Masking

Using section as control expression of “if” acts like a F90 where statement.

```c
if( a[0:n] < b[0:n] )
    c[0:n] += 1;
else
    d[0:n] -= 1;
```

Be aware that execution time is sum of “then” and “else”.

*Other brands and names are the property of their respective owners.*
__sec_implicit_index

Returns equivalent loop index (not array index)

Example:

```
a[s:m][t:n] = __sec_implicit_index(0)--__sec_implicit_index(1);
```

Serial equivalent:

```
for( int i=0; i<m; ++i )  // 0th index
  for( int j=0; j<n; ++j )  // 1st index
    x[s+i][t+j] = i-j;
```
Another way to specify vectorization

- Ignorable by compilers that do not understand it.
- Similar in style to OpenMP “#pragma parallel for”

```c
void saxpy( float a, float x[], float y[], size_t n ) {
    #pragma simd
    for( size_t i=0; i<n; ++i )
        y[i] += a*x[i];
}
```
Clauses for Trickier Cases

**reduction** clause for reduction variables

**linear** clause for induction variables

**private, firstprivate, lastprivate** à la OpenMP

```c
float altsum( float *x, size_t n ) {
    float s = 0;
    #pragma simd reduction(+:s), linear(x:2)
    for( size_t i=0; i<n; ++i ) {
        s += *x++;
        s -= *x++;
    }
    return s;
}
```

x has step of 2 per iteration.
Loop Body Can Have Structured Control Flow

- if-else
- for
- do-while
- switch
- while

```c
void foo( float c[16], float a[16][16], float b[16] ) {
    #pragma simd
    for( int k=0; k<16; ++k )
        for( int i=0; i<16; ++i )
            if( b[i]>0 )
                c[k] += a[i][k]*b[i];
}
```

Be aware that cost model is different than serial code
- Both paths of if-else execute, with masking.

No guarantee the compiler will vectorize the code.
- Keep control flow as simple as possible.
- Intel compiler reports #pragma simd loops that it cannot vectorize.
Elemental Functions

Enables vectorization of separately compiled scalar callee.

In file with definition.

```c
__declspec(vector)
float add(float x, float y) {
    return x + y;
}
```

In file with call site.

```c
__declspec(vector)
float add(float x, float y);

void saxpy( float a, float x[], float y[], size_t n ) {
    #pragma simd
    for( size_t i=0; i<n; ++i )
        y[i] = add(y[i], a*x[i]);
}
```
Uniform and Linear Arguments

```c
__declspec(vector(linear(b),uniform(c)))
double foo(double* a, double* b, double* c) {
    return *a + *b + *c;
}

__declspec(vector(linear(b),uniform(c)))
float foo(double* a, double* b, double* c);

void bar() {
    #pragma simd
    for( int i=0; i<n; ++i )
        D[i] = foo( &A[P[i]], &B[i], &C );
}
```
Advice on Which to Use

#pragma simd is usually preferable unless you are an APL refugee.

• No need to introduce array temporaries.
• No need for optimizer to recognize fusion opportunities.

Exceptions

• Array notation can be effective and more readable alternative to __m128, __m256, etc. when section length matches vector register size.
• For single statement, array notation is often more convenient.
• Sometimes fusion makes code slower.
Final Comment on Array Notation and \#pragma simd

No magic – just does tedious bookkeeping.

Use “structure of array” (SoA) instead of “array of structure” (AoS) to get SIMD benefit.

Read parts of vectorizer report from the compiler that apply to time-intensive parts of your code.

Know target machine’s vector capabilities.

Tip: use “icc –xHost” to exploit instruction set of host.
## Polynomial Multiplication

Example: $c = a \cdot b$

<table>
<thead>
<tr>
<th>x^2 + 2x + 3</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>x^2 + 4x + 5</td>
<td>a</td>
</tr>
<tr>
<td>5x^2 + 10x + 15</td>
<td></td>
</tr>
<tr>
<td>4x^3 + 8x^2 + 12x</td>
<td></td>
</tr>
<tr>
<td>x^4 + 2x^3 + 3x^2</td>
<td></td>
</tr>
<tr>
<td>x^4 + 6x^3 + 16x^2 + 22x + 15</td>
<td>c</td>
</tr>
</tbody>
</table>
Storage Scheme for Coefficients

\[ b[2] \quad b[1] \quad b[0] \quad b \]
\[ a[2] \quad a[1] \quad a[0] \quad a \]
\[ c[4] \quad c[3] \quad c[2] \quad c[1] \quad c[0] \quad c \]
Vector Parallelism
with Cilk Plus Array Notation

void simple_mul( T c[], const T a[], const T b[], size_t n ) {
    c[0:2*n-1] = 0;
    for (size_t i=0; i<n; ++i)
        c[i:n] += a[i]*b[0:n];
}

vector initialization

vector addition

More concise than serial version

Highly parallel: \( T_1/T_\infty = n^2/n = \Theta(n) \)

What’s not to like about it?
## Too Much Work!

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grade school</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>Karatsuba</td>
<td>$\Theta(n^{1.5})$</td>
</tr>
<tr>
<td>FFT method</td>
<td>$\Theta(n \lg n)$</td>
</tr>
</tbody>
</table>

However, the FFT approach has high constant factor. For $n$ about 32-1024, Karatsuba is a good choice.
Karatsuba Trick: Divide and Conquer

Suppose polynomials a and b have degree n

- let $K = x^{\lfloor n/2 \rfloor}$

  $$a = a_1 K + a_0$$
  $$b = b_1 K + b_0$$

Compute:

  $$t_0 = a_0 \cdot b_0$$
  $$t_1 = (a_0 + a_1) \cdot (b_0 + b_1)$$
  $$t_2 = a_1 \cdot b_1$$

Then

  $$a \cdot b \equiv t_2 K^2 + (t_1 - t_0 - t_2) K + t_0$$

- Partition coefficients.
- 3 half-sized multiplications. Do these recursively.
- Sum products, shifted by multiples of K.
void karatsuba( T c[], const T a[], const T b[], size_t n ) {
    if( n<=CutOff ) {
        simple_mul( c, a, b, n );
    } else {
        size_t m = n/2;
        karatsuba( c, a, b, m );       // t₀ = a₀ × b₀
        karatsuba( c+2*m, a+m, b+m, n-m );  // t₂ = a₁ × b₁
        temp_space<T> s(4*(n-m));
        T *a_=s.data(),  *b_=a_+(n-m),  *t=b_+(n-m);
        a_[0:m] = a[0:m]+a[m:m];        // a₀ = (a₀+a₁)
        b_[0:m] = b[0:m]+b[m:m];        // b₀ = (b₀+b₁)
        karatsuba( t, a_, b_, n-m );     // t₁ = (a₀+a₁)×(b₀+b₁)
        t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1];  // t = t₁−t₀−t₂
        c[2*m-1] = 0;
        c[m:2*m-1] += t[0:2*m-1];       // c = t₂K²+(t₁−t₀−t₂)K+t₀
    }
}
Sub-Products Can Be Computed in Parallel

\[ t_0 = a_0 \times b_0 \]
\[ t_2 = a_1 \times b_1 \]
\[ t_1 = (a_0 + a_1)(b_0 + b_1) \]

\[ a \cdot b \equiv t_2 K^2 + (t_1 - t_0 - t_2)K + t_0 \]
Multithreaded Karatsuba in Cilk Plus

```c
void karatsuba( T c[], const T a[], const T b[], size_t n ) {
    if( n<=CutOff ) {
        simple_mul( c, a, b, n );
    } else {
        size_t m = n/2;
        cilk_spawn karatsuba( c, a, b, m );  // t₀ = a₀×b₀
        cilk_spawn karatsuba( c+2*m, a+m, b+m, n-m );  // t₂ = a₁×b₁
        temp_space<T> s(4*(n-m));
        T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);
        a_[0:m] = a[0:m]+a[m:m];  // a₀ = (a₀+a₁)
        b_[0:m] = b[0:m]+b[m:m];  // b₀ = (b₀+b₁)
        karatsuba( t, a_, b_, n-m );  // t₁ = (a₀+a₁)×(b₀+b₁)
        cilk_sync;
        t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1];  // t = t₁-t₀-t₂
        c[2*m-1] = 0;
        c[m:2*m-1] += t[0:2*m-1];  // c = t₂K²+(t₁-t₀-t₂)K+t₀
    }
}
```

Only change is insertion of Cilk Plus keywords.
Parallelism is Recursive
Structured Programming with Patterns

Patterns are “best practices” for solving specific problems.

Patterns can be used to organize your code, leading to algorithms that are more scalable and maintainable.

A pattern supports a particular “algorithmic structure” with an efficient implementation.
Map Pattern

\textbf{Thread parallelism}

\begin{verbatim}
cilk_for( int i=0; i<n; ++i )
a[i] = f(b[i]);
\end{verbatim}

\textbf{Vector parallelism}

\begin{verbatim}
a[0:n] = f(b[i:n]);
\end{verbatim}

\begin{verbatim}
#pragma pragma simd
for( int i=0; i<n; ++i )
a[i] = f(b[i]);
\end{verbatim}
Reduction Pattern

cilk::reducer_opadd<float>
sum = 0;
cilk_for ( int i=0; i<n; ++i )
   sum += a[i];
... = sum.get_value();

Thread parallelism

float sum = __sec_reduce_add(a[i:n]);

#pragma simd reduction(+:sum)
float sum=0;
for( int i=0; i<n; ++i )
   sum += a[i];

Vector parallelism
Fork-Join Pattern

Thread parallelism

cilk_spawn a();
cilk_spawn b();
c();
cilk_sync();

Fake Fork-Join for Vector parallelism

if( x[0:n] > u )
  x[0:n] = u-x[0:n];

#pragma simd
for( int i=0; i<n; ++i )
  if( x[i] > u )
    x[i] = u-x[i];

Effective Cilk Plus: Writing Scalable Programs

Work-span model of complexity

Load balancing

Amortizing scheduling overhead

Hazards of locks

Hyperobjects revisited

Correctness tools survey
DAG Model of Computation

Program is a directed acyclic graph (DAG) of tasks

The hardware consists of workers

Scheduling is greedy

• No worker idles while there is a task available.
## Departures from Greedy Scheduling

<table>
<thead>
<tr>
<th>Departure</th>
<th>Alternative</th>
</tr>
</thead>
</table>
| Contended mutex.  
• Blocked worker could be doing another task | User wait-free atomic operation or a hyperobject. |
| One linear stack per worker  
• Caller blocked until callee completes | Intel® Cilk™ Plus has cactus stack. |
Work-Span Model

$T_p = \text{time to run with } P \text{ workers}$

$T_1 = work$
- time for serial execution
- sum of all work

$T_\infty = span$
- time for critical path
Work-Span Example

\[ T_1 = \text{work} = 7 \]
\[ T_\infty = \text{span} = 5 \]

Graph showing nodes and connections.
Burdened Span

Includes extra cost for synchronization

Often dominated by cache line transfers.
Lower Time Bound on Greedy Scheduling

(Implicit upper bound on speedup)

Work-Span Limit

$max(T_1/P, T_{\infty}) \leq T_P$
Upper Time Bound on Greedy Scheduling

(Imply lower bound on speedup)

Brent’s Lemma

\[ T_P \leq \frac{(T_1 - T_\infty)}{P} + T_\infty \]
Applying Brent’s Lemma to 2 Processors

\[ T_1 = 7 \]

\[ T_\infty = 5 \]

\[ T_2 \leq \frac{(T_1 - T_\infty)}{P} + T_\infty \]
\[ \leq \frac{(7 - 5)}{2} + 5 \]
\[ \leq 6 \]
Amdahl Was An Optimist

Amdahl’s Law

\[ \frac{T_{\text{serial}} + T_{\text{parallel}}}{P} \leq T_P \]

Graph showing speedup versus P with different bounds:
- Amdahl's Law
- Work-Span Bound
- Brent's Lemma

Graph structure indicating dependencies or operations between tasks.
**Estimating Running Time**

Scalability requires that $T_\infty$ be dominated by $T_1$.

$$T_P \approx \frac{T_1}{P} + T_\infty \quad \text{if } T_\infty \ll T_1$$

Increasing work hurts parallel execution proportionately.

The span impacts scalability, even for finite $P$. 
Parallel Slack

Parallel slack = \( \frac{T_1}{T_\infty} / P \)

Sufficient parallelism implies linear speedup.

\[ T_p \approx \frac{T_1}{T_P} \quad \text{if} \quad \frac{T_1}{T_\infty} \gg P \]

Linear speedup

Parallel slack is large
Master Method

If equations have this form:
\[ T(N) = aT(N/b) + cN^d \]
\[ T(1) = e \]

Then solution is:
\[ T(N) = \Theta(N^{\log_b a}) \quad \text{if} \quad \log_b a > d \]
\[ T(N) = \Theta(N^d \log N) \quad \text{if} \quad \log_b a = d \]
\[ T(N) = \Theta(N^d) \quad \text{if} \quad \log_b a < d \]
Work-Span Analysis for Karatsuba Routine

Let N be number of coefficients

**Equations for work**

\[ T_1(N) = 3T_1(N/2) + cN \]

\[ T_1(1) = \Theta(1) \]

**Equations for span**

\[ T_\infty(N) = T_\infty(N/2) + cN \]

\[ T_\infty(1) = \Theta(1) \]

Equations almost identical, except for one coefficient.

**Solutions**

\[ T_1(N) = \Theta(N^{\log_2 3}) \]

\[ T_\infty(N) = \Theta(N) \]

**speedup**

\[ \frac{T_1(N)}{T_\infty(N)} = \Theta(N^{0.58...}) \]
Space Analysis for Karatsuba Routine

Let N be number of coefficients

Equations for serial space

\[ S_1(N) = S_1(N/2) + cN \]
\[ S_1(1) = \Theta(1) \]

Equations for parallel space?

\[ S_\infty(N) \leq 3S_\infty(N/2) + cN \]
\[ S_\infty(1) = \Theta(1) \]

Solutions

\[ S_1(N) = \Theta(N) \]
\[ S_\infty(N) = O(N^{\log_2 3}) \]

But what about the space \( S_p \)?
Cilk Plus Bound for Karatsuba Routine

Cilk Plus guarantees \( S_p \leq P \cdot S_1 + P \cdot K \)

\[
S_p(N) = P \cdot O(N) + P \cdot K \\
= O(P \cdot N)
\]

For small \( P \), this is a big improvement on the bound \( S_\infty(N) = \Theta(N^{\log_2 3}) \).
Load Balancing by Work-stealing

Each processor has a deque of spawned tasks.

When each processor has work to do, a spawn is roughly the cost of about 25 function calls.
Load Balancing by Work-stealing

P

spawn
spawn
spawn
spawn
spawn

P

spawn
spawn
spawn
spawn
spawn

Return!

Spawn!

P

spawn
spawn
spawn
spawn
spawn

P

spawn
spawn
spawn
spawn
spawn

P

spawn
spawn
spawn
spawn
spawn

spawn
spawn
spawn
spawn
spawn

spawn
spawn
spawn
spawn
spawn

spawn
spawn
spawn
spawn
spawn

spawn
spawn
spawn
spawn
spawn

Copyright© 2012, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Load Balancing by Work-stealing

Return!
Work-stealing task scheduler

spawn
spawn
spawn
spawn
spawn

spawn
spawn
spawn
spawn
spawn

spawn
spawn
spawn
spawn

Steal!
With sufficient parallelism, the steals are rare, and we get \textit{linear speedup} (ignoring memory effects).
Parallelizing Loops

```c
// cilk_for
int i = 0; i < 8; ++i)
    do_work(i);
```

...recursive divide-and-conquer...
Divide and Conquer

Divide and conquer results in fewer steals and more parallelism.
Controlling grainsize (cilk_for)

By default, cilk_for tiles the iteration space.
• Thread executes entire tile
• Avoids excessively fine-grained synchronization

For severely unbalanced iterations, this might be suboptimal.
• Use `pragma cilk grainsize` to specify size of a tile

```
#pragma cilk grainsize = 1
cilk_for( int i=0; i<n; ++i )
a[i] = f(b[i]);
```
Grainsize for Recursion

Switch from parallel recursion to serial algorithm when problem is small.

Parallel Quicksort Example:
• Parallel recursion for >1000 keys
• Serial recursion for \( \leq 1000 \) keys
• Insertion sort for \( \leq 7 \) keys
Serial Merge

Inherently serial algorithm.
• Each iteration depends on prior iteration.

Merge of sequences \([xs,xe)\) and \([ys,ye)\).

```cpp
void serial_merge( const T* xs, const T* xe, const T* ys, const T* ye, T* zs ) {
    while( xs!=xe && ys!=ye ) {
        bool which = *ys < *xs;
        *zs++ = std::move(which ? *ys++ : *xs++);
    }
    std::move( xs, xe, zs );
    std::move( ys, ye, zs );
}
```
Recursive Approach To Merge

Choose middle key $m$ of longer sequence

Binary search for $m$ in shorter sequence.

Recursively merge.

Recursively merge.
The Code

```c
void parallel_merge( const T* xs, const T* xe, const T* ys, const T* ye, T* zs ) {
    const size_t MERGE_CUT_OFF = 2000;
    if( xe-xs + ye-ys <= MERGE_CUT_OFF ) {
        serial_merge(xs,xe,ys,ye,zs);
    } else {
        const T *xm, *ym;
        if( xe-xs < ye-ys ) {
            ym = ys+(ye-ys)/2;
            xm = std::upper_bound(xs,xe,*ym);
        } else {
            xm = xs+(xe-xs)/2;
            ym = std::lower_bound(ys,ye,*xm);
        }
        T* zm = zs + (xm-xs) + (ym-ys);
        cilk_spawn parallel_merge( xs, xm, ys, ym, zs );
        /*nospawn*/parallel_merge( xm, xe, ym, ye, zm );
        // implicit cilk_sync
    }
}
```

**Base case does serial merge.**

**Parallel recursion**
Vectorization Issues

Use #pragma simd or array notation

Read the vectorizer report

Keep control-flow as simple as possible

Natural alignment is usually fastest

Contiguous accesses are fastest

Prefer gather to scatter

Long vector operations have poor locality

Too many streams can cause TLB thrashing
Rough Guide to “Access Poker”

Card Ranks
- Aligned contiguous
- Unaligned contiguous
- Strided
- Gather/scatter

Card Suites
- Read
- Write

more traffic

best card = aligned read
worst card = scatter
SoA Versus AoS

Array of Structure (AoS)

```c
struct state {
    float pos, vel, acc;
};
state s[n];
```

Structure of Array (SoA)

```c
struct state_array {
    float *pos, *vel, *acc;
    int n;
};
state_array s;
```

<table>
<thead>
<tr>
<th>pos0</th>
<th>vel0</th>
<th>acc0</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos1</td>
<td>vel1</td>
<td>acc1</td>
</tr>
<tr>
<td>pos2</td>
<td>vel2</td>
<td>acc2</td>
</tr>
<tr>
<td>pos3</td>
<td>vel3</td>
<td>acc3</td>
</tr>
</tbody>
</table>

stride = 3

<table>
<thead>
<tr>
<th>pos0</th>
<th>pos1</th>
<th>pos2</th>
<th>pos3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>vel0</td>
<td>vel1</td>
<td>vel2</td>
<td>vel3</td>
<td>...</td>
</tr>
<tr>
<td>acc0</td>
<td>acc1</td>
<td>acc2</td>
<td>acc3</td>
<td>...</td>
</tr>
</tbody>
</table>
SoA Versus AoS Example

**Array of Structure (AoS)**

```c
struct state {
    float pos, vel, acc;
};

state* alloc( int n ) {
    return new state[n];
}

void update( state s[], int n ) {
    #pragma simd
    for( int i=0; i<n; ++i ) {
        s[i].vel += s[i].acc;
        s[i].pos += s[i].vel;
    }
}
```

**Structure of Array (SoA)**

```c
struct state_array {
    float *pos, *vel, *acc;
    int n;
};

void alloc( state_array& s, int n ) {
    s.pos = new float[n];
    s.vel = new float[n];
    s.acc = new float[n];
    s.n = n;
}

void update( state_array& s ) {
    #pragma simd
    for( int i=0; i<s.n; ++i ) {
        s.vel[i] += s.acc[i];
        s.pos[i] += s.vel[i];
    }
}
```
Changing Scatter to Gather

Sometimes there is a choice when designing an algorithm.

**scatter**

```c
#pragma simd
for( int i=0; i<n; ++i )
a[perm[i]] = b[i];
```

**gather**

```c
#pragma simd
for( int i=0; i<n; ++i )
a[i] = b[inverse_perm[i]];
```

Replace with inverse permutation.
Vectorizer Report

By default, compiler reports #pragma simd loops that it cannot vectorize

• -vect-report options give more control over report.
• Note: “vectorized” not always faster than scalar.

```
... ...
7 #pragma simd
8   for( int i=0; i<n; ++i )
9       a[i] *= 2;
10 #pragma simd
11   for( int i=0; i<n; ++i )
12       b[i] *= 2;
... ...
```

float a[n];
**long double** b[n];

“Long double” processed by scalar x87 floating-point unit.

```
$ icc -xHost -vec-report5 foo.cpp
foo.cpp(12): (col. 8) remark: loop was not vectorized: unsupported data type.
foo.cpp(11): (col. 5) warning #13379: loop was not vectorized with "simd"
```
Cache Oblivious Algorithms

Modern processor ≠ PRAM

• Cache miss can be hundreds of cycles
• Page lookup also involves a cache called the Translation Look-aside Buffer (TLB)

Two solutions

• Cache aware: write algorithm for specific cache parameters
  - Example: blocked matrix multiplication where blocks fit in L1 cache.
• Cache oblivious – “cache paranoid”
  - Recursively block for all possible sizes/levels of cache.
Example: Transpose Big Matrix

Key step in 2D FFT

2D FFT

transpose
cilk_for( size_t j=0; j<m; ++j ) {
    apply FFT to row j
    adjust phase of row j
}
transpose
cilk_for( size_t j=0; j<m; ++j ) {
    apply FFT to row j

transpose
### Serial Transpose-Swap

**template<typename T>**
```cpp
void SerialTransposeSwap(size_t m, size_t n, T* a, T* b, size_t s) {
    if (a==b)
        for (size_t i=0; i<m; ++i)
            for (size_t j=0; j<i; ++j)
                std::swap(a[i*s+j], b[j*s+i]);
    else
        for (size_t i=0; i<m; ++i)
            for (size_t j=0; j<n; ++j)
                std::swap(a[i*s+j], b[j*s+i]);
}
```
## Transpose 4096 × 4096 Matrix of Float

The diagram illustrates the transpose of a 4096 × 4096 matrix of float values, showing how the matrix is laid out across multiple pages. Each page touches 4 pages horizontally and 4096 pages vertically, as indicated by the arrows and columns labeled 'page'.

<table>
<thead>
<tr>
<th>page</th>
<th>page</th>
<th>page</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>page</td>
<td>page</td>
<td>page</td>
</tr>
<tr>
<td>page</td>
<td>page</td>
<td>page</td>
<td>page</td>
</tr>
<tr>
<td>page</td>
<td>page</td>
<td>page</td>
<td>page</td>
</tr>
<tr>
<td>page</td>
<td>page</td>
<td>page</td>
<td>page</td>
</tr>
</tbody>
</table>

The diagram visualizes the memory layout and access patterns of a large matrix, which is critical for optimizing performance in applications that involve such data structures.
Blocked Solution

Transpose 128x128 submatrices of float
template< typename T>
void ParallelTransposeSwap(size_t m, size_t n, T* a, T* b, size_t s) {
    const size_t TRANSPOSE_CUT = 32;
    if (m*n<=TRANSPOSE_CUT)
        SerialTransposeSwap(m, n, a, b, s);
    else {
        size_t m2 = m/2;
        size_t n2 = n/2;
        cilk_spawn ParallelTransposeSwap ( m2,   n2, a       , b       , s);
        cilk_spawn ParallelTransposeSwap (m-m2, n2, a+m2*s, b+m2  , s);
        if (a!=b)
            cilk_spawn ParallelTransposeSwap(m2, n-n2, a+n2,   b+n2*s, s);
        ParallelTransposeSwap( m-m2, n-n2, a+m2*s+n2, b+n2*s+m2, s);
    }
    // implicit cilk_sync
}
OpenMP Tactics to Unlearn
(Thanks to James Cownie for List)

1. Creating one work item per thread.
2. Anything involving \texttt{omp\_get\_thread\_num()}.
3. Fear of nested parallelism.
Problem with One Work Item Per Thread

Destroys composability

- No way to know if running as child or sibling of other parallel work.

Hurts load balancing.

- Gives scheduler no parallel slack.

Advice: Choose grain size based on amortizing scheduling overhead, not balancing load.
Problem with Using Thread Ids

Thus thread id can change in surprising ways.

• Id after spawn can be *different* than before spawn.
• Id after sync can be *different* than before spawns.

**Advice**: Use hyperobjects (reducers and holders).

```cpp
#include <cilk/cilk_api.h>

std::vector<int> A;

void bar() {
    int j = __cilkrts_get_worker_number()
    A[j]++;
}

int main() {
    A.resize (__cilkrts_get_nworkers());
    int i = __cilkrts_get_worker_number();
    cilk_spawn f();
    A[i]++;
    cilk_sync;
}
```

Race, because i==j!
Weak Guarantee on Thread Ids

A thread *not* created by the Cilk runtime always returns on the thread that called it.

Makes it safe to call black box written in Cilk from code expecting callee to preserve thread id.

```c
#include <cilk/cilk_api.h>

void bar( int d ) {
    if( d>20 ) return;
    int i = __cilkrts_get_worker_number();
    cilk_spawn bar(d+1);
    bar(d+1);
    cilk_sync;
    int k = __cilkrts_get_worker_number();
    assert( i==k || d>0 );
}

int main() {
    bar(0);
}
```

\( i==k \) when caller is not a Cilk thread.
Embrace Nested Parallelism

Cilk was designed for nested parallelism.

Unused nested parallelism is inexpensive.

• Execution is serial when all threads are busy.
Common Pitfall

Serial loop around parallel loop
• Time-stepping stencils
• Naïvely written benchmarks

```c
for( int t=t0; t<t1; ++t )   // Serial loop
  #omp parallel for
  for( int i=0; i<n; ++i )
    a[i] = a[i-1]+a[i]+a[i+1])*(1/3.0f);
```

Consider using “cache-oblivious stencil”.
Performance Tools

Intel® Cilk™ View
• Automatic work-span analysis for Cilk™ Plus

Intel® Amplifier
• General threading analysis
• Good for spotting hardware-related bottlenecks
Sample Cilk View Output

Uses *burdened span* that estimates scheduling costs.
Effective Cilk Plus: Writing Correct Programs

Race conditions and serial elision revisited

Strict fork-join nature of Cilk

Mutual exclusion

Hazards of locks

Hyperobjects revisited

Correctness tool survey
**Strands**

**strand** = sequence of instructions with no intervening *spawn* or *sync*.

- DAG model interpretation: a vertex with at most one incoming and one outgoing edge.

```c
// Four strands
x = cilk_spawn f(*p++);
y = g(*p--);
cilk_sync;
z = x+y;
```

```c
// Four strands
x = f(tmp);
tmp = *p++;
y = g(*p--);
z = x+y;
```
Strands are partially ordered. Any strands $s_1$ and $s_2$ must be related in one of the following ways:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1 = s_2$</td>
<td>$s_1$ and $s_2$ are the same strand</td>
</tr>
<tr>
<td>$s_1 &lt; s_2$</td>
<td>$s_1$ happens before $s_2$</td>
</tr>
<tr>
<td>$s_2 &lt; s_1$</td>
<td>$s_2$ happens before $s_1$</td>
</tr>
<tr>
<td>$s_1 \parallel s_2$</td>
<td>$s_1$ and $s_2$ are logically parallel</td>
</tr>
</tbody>
</table>
**Definition of a Data Race**

**Data Race** happens if two logically parallel strands:

- Access the same shared location.
- At least one of the accesses is a write.
- The two strands hold no locks in common.
Serial Elision Revisited

A Cilk Plus program is deterministic if:

• It has no data races
• If uses no locks
• It is oblivious to numerical values of memory addresses

A deterministic Cilk Plus program behaves the same as its serial elision.

```c
#define cilk_spawn
#define cilk_sync
#define cilk_for for
```
Reasoning About Races in Cilk Plus (1)

\[ \text{readset}(x) = \text{set of locations read by a call } x \]

\[ \text{writeset}(x) = \text{set of locations written by a call } x \]

To analyze whether Bar() has a race:

- Find \text{readset} and \text{writeset} of B()
- Find \text{readset} and \text{writeset} of C()
- Apply definition of a race.

void Bar() {
    A();
    cilk_spawn B();
    C();
    cilk_sync;
    D();
    // implicit cilk_sync
}

Accesses of A() and D() are irrelevant to whether Bar itself creates a race.
Reasoning About Races in Cilk Plus (2)

Implicit sync enables hierarchical reasoning.

- No dangling parallelism!

To analyze Qux, just need to know readset/writeset of Foo and Bar, not whether they are internally parallel.

```c
void Qux() {
    cilk_spawn Foo();
    Bar();
    // implicit cilk_sync
}

void Bar() {
    A();
    cilk_spawn B();
    C();
    cilk_sync;
    D();
    // implicit cilk_sync
}

void Foo() {
    ...
}
```
Pros And Cons of Strict Fork-Join Model

Analogous to pros and cons of stack-allocated variables (versus heap-allocated memory)

• Easier to reason about
• More efficient

However, sometimes less structure is worth the costs

• See NABBIT
• TBB-style pipelines in Cilk Plus are possible via “consumer reducer”
Hazard of Locks: Deadlock

Problem
• Each thread waits on lock held by the other thread.

Solutions
• Hold only one lock at a time
• Acquire locks in consistent order
  • Stratify into mutex levels
  • Sort mutexes by address
• Use lock-free techniques

Advice: Never hold a lock while calling someone else’s code!
Hazard of Locks: Priority Inversion

Problem
• High priority task waits for lock
• Medium priority task gets attention
• Low priority task holds lock

Solutions
• Priority inheritance
• Priority ceiling
• Lock-free techniques
Hazards of Locks: Lock Holder Preemption

Problem

- Holder of spin lock is interrupted
- Other threads pointlessly spin

Solutions

- Spin/yield/block
- Minimize oversubscription
- Use lock-free techniques
Hyperobjects Revisited

A hyperobject is an object with multiple views.

• No two concurrent strands see the same view.
  – No locking required.
• Multiple views are merged via an associative operation.
• Final result is same as for serial execution.

```cpp
cilk::reducer_opadd<float> sum = 0;
...
cilk_for( size_t i=1; i<n; ++i )
    sum += f(i);
... = sum.get_value();
```
How A Hyperobject Works

```c
#include <cilk_plus>

float sum;

cilk::reducer_opadd<float> sum;

void f(int m) {
    sum += m;
}

float g() {
    cilk_spawn f(1);
    f(2);
    cilk_sync;
    return sum.get_value();
}
```

When no steal happens

New view!

Merge views

Global variable

- `sum += 1`
- `sum = 0`
- `sum += 2`
- `sum += sum`
- `sum += 1`
- `sum += 2`
When Do Two Strands See the Same View?

Two strands $s_1$ and $s_2$ see the same view if both:
• $s_1$ and $s_2$ are not parallel
• There is no strand $s_3$ that is parallel to one but not the other.

$$(s_1 \parallel s_2) \land (\forall s_3 \in S : s_1 \parallel s_3 \leftrightarrow s_2 \parallel s_3)$$

Two strands labeled “a” see the same view. Likewise for the two strands labeled “b”.

Copyright© 2012, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Tricky Example

#include <cilk/reducer_opadd.h>

float A[N], B[N][N];
cilk::reducer_opadd<float> Tmp;

void SumRows() {
    // strand r
    cilk_for(int i=0; i<N; ++i) {
        Tmp.set_value(0);  // strand s_i
        cilk_for(int j=0; j<N; ++j)
            Tmp += B[i][j];  // strand t_{i,j}
        A[i] = Tmp.get_value();  // strand u_i
    }
    return;  // strand v
}
Quiz

The C operators +, *, &, |, ^, &&, || are associative for integral types.

There’s one more associative operator in C.

What is it?
Holder is a reducer with trivial reduction operation.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Implementation of $x = \text{reduce}(x, y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>keep_indeterminant</td>
<td>(no-op)</td>
</tr>
<tr>
<td>keep_last</td>
<td>(one of variants below)</td>
</tr>
<tr>
<td>keep_last_copy</td>
<td>$x = y;$</td>
</tr>
<tr>
<td>keep_last_swap</td>
<td>$\text{std::swap}(x, y)$</td>
</tr>
<tr>
<td>keep_last_move</td>
<td>$x = \text{std::move}(y)$</td>
</tr>
</tbody>
</table>
Example: Dependence that is Costly to Break

serial code

```c
void bar() {
    T state = ab_initio(0);
    for( int i=0; i<n; ++i ) {
        foo(state);
        state = next(state);
    }
}
```

parallel code

```c
void bar() {
    T state = ab_initio(0);
    cilk_for( int i=0; i<n; ++i ) {
        foo(state);
        state = next(state);
    }
}
```

Loop-carried dependence.

Eliminates loop-carried dependence.

But what if $ab\_initio$ is much more expensive than $next$?
Solution with “Keep Last” Holder

```c
#include <cilk/holder.h>

struct X {
    bool initialized;
    T state;
    X() : initialized(false) {}
};

void bar() {
    cilk::holder<X, cilk::holder_keep_last> h;
    cilk_for( int i=0; i<n; ++i ) {
        X& x = h.view();
        if( !x.initialized ) {
            x.state = ab_initio(i);
            x.initialized = true;
        }
        foo(x.state);
        x.state = next(x.state);
    }
}
```

Lazily initialize new view

Similar to serial code
Pop Quiz

Parallelism inside foo might create new views

The code assumes the assertion shown below.

Why is it always true?

```cpp
void bar() {
    cilk::holder<X,cilk::holder_keep_last> h;
    cilk_for( int i=0; i<n; ++i ) {
        X& x = h.view();
        if( !x.initialized ) {
            x.state = ab_initio(i);
            x.initialized = true;
        }
        foo(x.state);
        assert(&x==&h.view());
        x.state = next(x.state);
    }
}
```
Theory of Automatic Race Detection

“Happens Before” approach focuses on partial ordering of accesses.

Data Race happens if two logically parallel strands:

• Access the same shared location.
• At least one of the accesses is a write.
• The two strands hold no locks in common.

“Lockset” approach focuses on locking. (useless for lock-free programming)

Both approaches find races in a program execution, even if races did not cause problem.
Two Race Detectors for Cilk Plus

Intel® Cilk Screen
• “Happens before” on strands + “Lock set”
• Theoretically efficient implementation that strict fork-join nature of Cilk

Intel® Parallel Inspector
• “Happens before” on threads + “Lock set”
• Also detects potential deadlock
• Also has memory checker
• GUI integrates into Visual Studio

Both based on “Pin” dynamic instrumentation technology.
http://www.pintool.org/
Cilk Screen Example

```c
void f() {
    int x[10];
    cilk_for( int i=0; i<10; ++i )
        x[i] = pseudo_random();
}
```

```c
int pseudo_random() {
    static int state = 1;
    return state = a*state+b;
}
```

$ icc -g randomfill.cpp$
$ cilkscreen a.out$
CilkScreen Race Detector V2.0.0, Build 2516

Race condition on location 0x600b84
write access at 0x40062b: (/tmp/randomfill.cpp:7, pseudo_random+0x19)
read access at 0x40061a: (/tmp/randomfill.cpp:7, pseudo_random+0x8)
called by 0x2b2156f08b07: (__$U0+0xc7)
called by 0x2b2156f08848: (cilk_for_recursive<unsigned int, void (*)(void*, unsigned int, unsigned int)>+0x128)
called by 0x2b2156f086b8: (__$U1+0xb8)
called by 0x2b2156f082c5: (cilk_for_root<unsigned int, void (*)(void*, unsigned int, unsigned int)>+0x135)
called by 0x2b2156f0818a: (__cilkrts_cilk_for_32+0xa)
Wrap UP

Reminder of Key Points
Compose Threads and Vector Parallelism

Message Driven
MPI, tbb::flow

Fork-Join
OpenMP, TBB, or Cilk

Intel® Cilk™ Plus

SIMD
Array Notation or #pragma SIMD
## Philosophy of Cilk Plus

<table>
<thead>
<tr>
<th>Division of Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programmer</strong></td>
</tr>
<tr>
<td>Specify what <em>can</em> run in parallel.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Provide much more <em>potential</em> parallelism than system can use.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Express SIMD opportunities.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Avoid races.</td>
</tr>
<tr>
<td>Minimize use of locks.</td>
</tr>
<tr>
<td>Promote locality via cache-oblivious style.</td>
</tr>
</tbody>
</table>
Map Pattern

**Thread parallelism**

```cilk
cilk_for( int i=0; i<n; ++i )
a[i] = f(b[i]);
```

**Vector parallelism**

```cilk
a[0:n] = f(b[i:n]);
```

```cilk
#pragma simd
for( int i=0; i<n; ++i )
a[i] = f(b[i]);
```
Reduction Pattern

cilk::reducer_opadd<float>
sum = 0;
cilk_for( int i=0; i<n; ++i )
    sum += a[i];
... = sum.get_value();

Thread parallelism

float sum = __sec_reduce_add(a[i:n]);

#pragma simd reduction(+:sum)
float sum=0;
for( int i=0; i<n; ++i )
    sum += a[i];

Vector parallelism
Fork-Join Pattern

Thread parallelism

```cilk
void cilk_spawn()
{
    c();
}
```

Fake Fork-Join for Vector parallelism

```cilk
#pragma simd
for (int i=0; i<n; ++i)
{
    if (x[i] > u)
    {
        x[i] = u - x[i];
    }
}
```
Minimize Span; Maximize Slack

Span = $T_\infty = \text{shortest possible running time}$

Speedup $\leq T_1/T_\infty$

Slack = $(T_1/T_\infty)/P$

$T_1/T_\infty$ is a measure of available parallelism.

Slack enables system to balance load.
Grain Size Considerations

Make grain size easily modifiable.
- You will want to tune it.

Set grain size large enough to amortize scheduling overheads.
- Avoid one grain per processor.
- Cilk Plus ≠ OpenMP

Recursive parallelism
- Might need cutoff where recursion switches from parallel to serial.
Access Patterns

- Aligned contiguous
- Unaligned contiguous
- Strided
- Gather/scatter
tend to be slower
Hardware and Compiler Realities

Cache line boundaries can be important.
• Cache line = quantum of information transfer
• False sharing can be trouble

Instruction sets and compilers can have sweet spots
• Read vectorization report
Tools

Race Detectors
• Intel® Cilk Screen
• Intel® Parallel Inspector

Parallel Profilers
• Intel® Cilk™ View
• Intel® Amplifier
URLs

Cilk Plus home page
• http://cilkplus.org

Cilk Plus Forum

Cilk Plus Specifications

Intel® Cilk™ Plus Software Development Kit
  - Cilk Screen Race Detector
  - Cilk View Scalability Analyzer

GCC 4.7 Branch
• http://gcc.gnu.org/svn/gcc/branches/cilkplus/

Intel ® Parallel Inspector
Dankeschön
Legal Disclaimer & Optimization Notice

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.

Software and workloads used in performance tests may have been optimized for performance only on
Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using
specific computer systems, components, software, operations and functions. Any change to any of
those factors may cause the results to vary. You should consult other information and performance
tests to assist you in fully evaluating your contemplated purchases, including the performance of that
product when combined with other products.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk
are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers 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 SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for
Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information
regarding the specific instruction sets covered by this notice.

Notice revision #20110804