Cilk Plus

GETTING STARTED

Overview

- Fundamentals of Cilk Plus
- Hyperobjects
- Compiler Support
- Case Study

Fundamentals of Cilk Plus

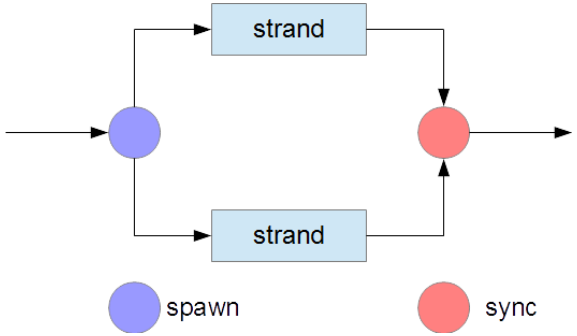
- Terminology
- Execution Model
- Language Extensions
- Spawn
- Synchronize
- Loop Parallelism
- Array Notation
- Loop Vectorization

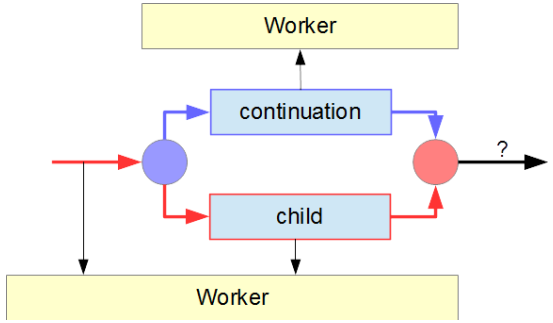
Fundamentals - Terminology

Terms

- task = sub-problem that can be solved independently
- strand = section of a program without any parallel control structures
- parallel control point = the start and end points of a strand
- worker = operating system thread for executing a task

A Cilk Plus program is a graph of strands connecting parallel control points





A strand maps to a worker and the child executes on the same worker as the caller

Fundamentals - Execution Model

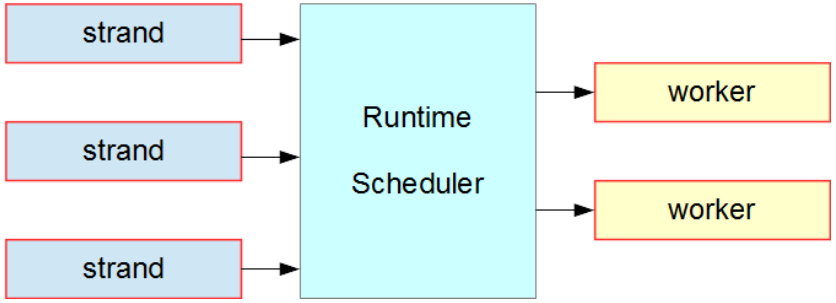
Work-stealing algorithm: minimizes the number of times that work moves from one processor to another

Runtime scheduler maps strands to workers dynamically

Runtime allocates workers using native OS facilities

A strand never migrates between workers

Runtime scheduler may yield workers to other programs



The model lets us control the granularity implemented in iterations

- Grain size too large – logical parallelism decreases
- Grain size too small – spawn overhead increases
- Compiler uses a default formula that supposedly works well in most circumstances

Fundamentals - C, C++ Extensions

Header Files

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

Task Parallelism

- `cilk_spawn` - fork
- `cilk_sync` - join
- `cilk_for` - iterate

Data Parallelism

- `#pragma simd`
- array notation
- SIMD-enabled functions

Fundamentals - Task Parallelism - Spawn

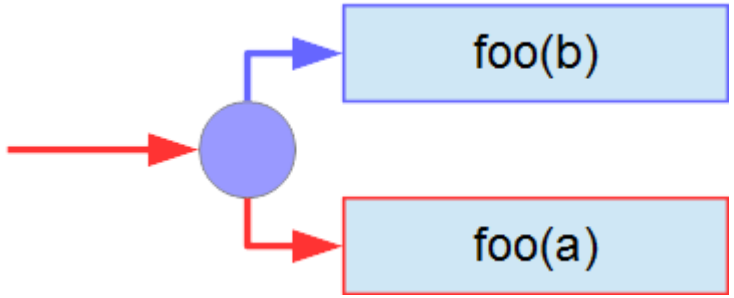
Parallel execution is optional

Programmer grants permission to execute a function in parallel with the code that follows

```
variable = cilk_spawn function call
```

Example

```
x = cilk_spawn foo(a); // child
foo(b); // continuation
```



Notes:

- Child = function that is spawned
- Continuation = code that follows the spawn

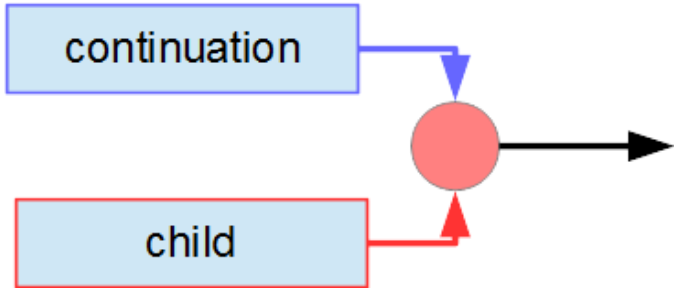
Fundamentals - Task Parallelism - Sync

Execution may not continue until all spawn requests in the block have completed

```
cilk_sync
```

Example

```
cilk_spawn foo(a);
foo(b);
cilk_sync;
```



Notes:

- does not affect parallel strands in other functions
- implicit `cilk_sync` at the end of a block

Fundamentals - Task Parallelism - Iterate

```
cilk_for (initialization; condition; increment) statement block
```

Example

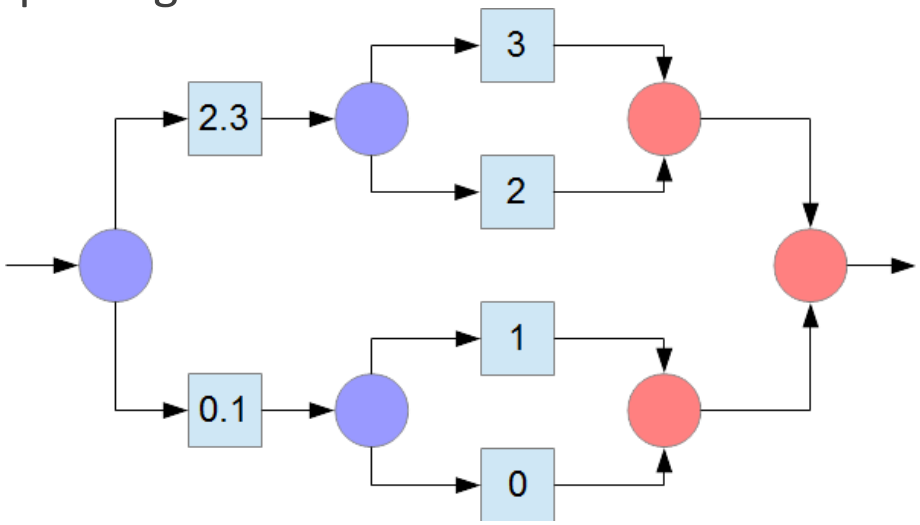
```
cilk_for (int i = 0; i < limit; i++) {
    a[i] = b[i] + c[i];
}
```

Constraints (unlike `for`):

- control cannot be transferred into or out of the body (`return, break, goto`)
- initialize single variable only – control variable
- control variable may not be modified within the statement block
- increment is one of `++`, `--`, `+=step`, `-=step`
- condition compares control variable to `limit`
- `step` and `limit` are not affected by the statement block

Fundamentals - cilk_for Implementation

Divide and conquer algorithm



Notes:

- work is well-balanced
- `cilk_sync` waits only for completion of the children spawned within the same iteration

Fundamentals - Data Parallelism – Array Notation

```
var_name[first:length:stride], var_name[first:length], var_name[:]
```

Examples

```
a[0:n] = b[0:n] + c[0:n]
```

```
y[0:8] = pow(x[0:8], 3.5)
```

```
std::swap(x[0:8], y[0:8])
```

Constraints:

- first, length, stride must be integers
- array size must be known for [:]
- avoid partial overlaps

Note:

- compilers implement array notation using vectorization

Fundamentals - Data Parallelism – Loop Vectorization

Vectorization is optional not mandatory

Programmer grants permission to parallelize an iteration using vectorization

```
#pragma simd
```

Example

```
#pragma simd
for (int i = 0; i < 100; i++)
    a[i] = 2 * a [i + 1];
```

Notes:

- can be used on any iteration for which `cilk_for` is legal, but not vice versa
- not restricted to inner loops

Fundamentals - Data Parallelism – SIMD-Enabled Functions

A SIMD-Enabled function describes in scalar syntax an algorithm that operates on one element

Examples

```
__attribute__((vector)) double add(double a, double b) { return a + b; } // Linux OSX
__delcspec((vector))    double add(double a, double b) { return a + b; } // Windows
a[:] = add(b[:], c[:]);
cilk_for (int i = 0; i < limit; i++) { a[i] = add(b[i], c[i]); }
```

Constraints:

- `goto` disallowed
- `switch` with more than 16 `case` disallowed
- expressions with array notation disallowed
- `cilk_spawn` disallowed

Hyperobjects

Hyperobject = a linguistic construct that allows strands to coordinate the updating of a shared variable.

A hyperobject provides a different view of the shared variable for each strand

Implementations

- Reducers
- Reducing Array Sections

Hyperobjects - Reducers

A reducer is a hyperobject designed for reductions on associative operations

Example

```
#include <cilk/cilk.h>            // for cilk_for
#include <cilk/reducer_opadd.h>   // for addition or subtraction (+=, -=, ++, --)
cilk::reducer_opadd<double> sum;  // reducer hyperobject
cilk_for (int i = 0; i < limit; i++) {
    sum += a[i] * b[i];
}                                 // implicit sync
double total = sum.get_value();   // terminal value
```

Hyperobjects - Reducing Array Sections

```
__sec_reduce_add(array notation)
__sec_reduce_mul(array notation)
__sec_reduce_max(array notation)
__sec_reduce_min(array notation)
```

Examples

```
double total   = __sec_reduce_add(a[0:n])
double product = __sec_reduce_mul(a[0:n])
double maximum = __sec_reduce_max(a[0:n])
double minimum = __sec_reduce_min(a[0:n])
```

Compiler Support

Serial Elision

Runtime Controls

Compilers

- Intel Composer XE
- GCC
- Windows

Compiler Support - Serial Elision

Add Header File

- `#include <cilk/cilk.h>`
- `#include <cilk/cilk_stub.h>`

Compiler Support - Runtime Controls

Header File

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

Environment Variable

- `export CILK_NWORKERS=8`

Runtime Calls

- `__cilk_set_param("nworkers", 8)`
- `__cilk_get_worker_number()`
- `__cilk_get_total_workers()`

Compiler Support - Compilers

Intel Parallel Studio XE 2013

- Composer XE – C, C++, Fortran Compiler
- Advisor XE – Prototyping Tool
- Inspector XE – Memory and Threading Debugger
- VTune Amplifier XE – Advanced Threading and Performance Profiler

GCC Compiler Collection

- Release 4.9.0 – April 22 2014
- Partial implementation – cilk_spawn and cilk_sync
- Open source
- Cilkview – parallelism checker
- Cilkscreen – race condition checker

Compiler Support – Intel Composer XE

Auto-Parallelization

- `-parallel` or `/Qparallel`
- Exploits parallel architecture of SMP systems
- Partitions data for threaded code generation
- Reports parallelization and default vectorization

Auto-Vectorization

- `-O2` or `/O2`
- `-vec-report1` or `/Qvec-report1`

Compiler Support - GCC

Flags

- `O0` – no optimization, suppress vectorization
- `O2` – optimize for speed
- `fcilkplus` – include cilk plus
- `include cilk/cilk_stub.h` – serialize
- `cilk-serialize` – serialize

Command Line

- `g++ -O2 -o example -fcilkplus example.cpp`

Compiler Support - Windows

Flags

- `/O0` – no optimization, suppress vectorization
- `/O2` – optimize for speed
- `/FI cilk/cilk_stub.h` - serialize
- `/Qcilk-serialize` - serialize

Command Line

- `icl /O2 example.cpp`

Case Study – Tiled Algorithm

Matrix Transpose Vladimirov, A. (2013). Cache Traffic Optimization with Cilk Plus and OpenMP

```
template <typename T>
void transpose(T* const a, int n) {
    const int TILE = 16;
    cilk_for(int ii = 0; ii < n; ii += TILE) {
        const int imax = (n < ii + TILE ? n : ii + TILE);
        for (int jj = 0; jj <= ii; jj += TILE) {
            for (int i = ii; i < imax; i++) {
                const int jmax = (i < jj + TILE ? i : jj + TILE);
                #pragma loop count avg(TILE)
                #pragma simd
                for (int j = jj; j < jmax; j++) {
                    const T temp = a[i * n + j];
                    a[i * n + j] = a[j * n + i];
                    a[j * n + i] = temp;
                }
            }
        }
    }
}
```

```
template <typename T>
void transpose(T* const a, int n) {
    const int TILE = 16;
    #pragma omp parallel for schedule(static)
    for (int ii = 0; ii < n; ii += TILE) {
        const int imax = (n < ii + TILE ? n : ii + TILE);
        for (int jj = 0; jj <= ii; jj += TILE) {
            for (int i = ii; i < imax; i++) {
                const int jmax = (i < jj + TILE ? i : jj + TILE);
                #pragma loop_count avg(TILE)
                #pragma simd
                for (int j = jj; j < jmax; j++) {
                    const T temp = a[i * n + j];
                    a[i * n + j] = a[j * n + i];
                    a[j * n + i] = temp;
                }
            }
        }
    }
}
```

Case Study – Recursive Algorithm

Matrix Transpose Vladimirov, A. (2013). Cache Traffic Optimization with Cilk Plus and OpenMP

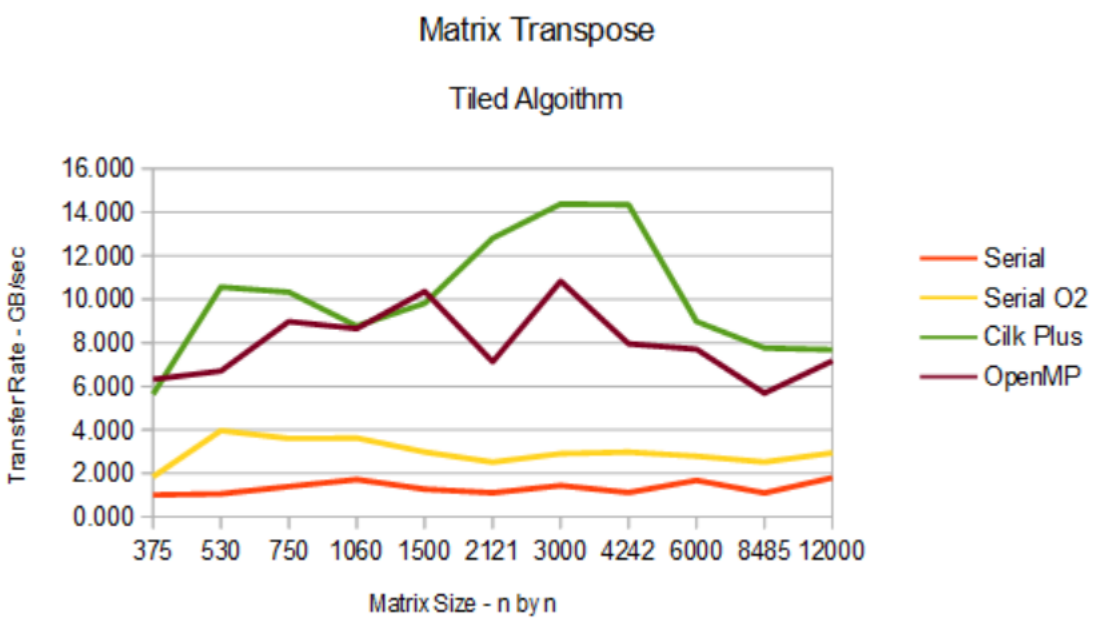
```
template <typename T>
void transpose_cache_oblivious(int is, int ie, int js, int je, T* const a, int n) {
    const int RT = 32;  // recursion threshold
    if (((ie - is) <= RT) && ((je- js) <= RT)) {
        for (int i = is; i < ie; i++) {
            int jm = (je < i ? je : i);
            #pragma simd
            #pragma loop_count avg(RT)
            for (int j = js; j < jm; j++) {
                const T temp = a[i * n + j];
                a[i * n + j] = a[j * n + i];
                a[j * n + i] = temp;
            }
        }
        return;
    }

    // recursive fork-join
    if ((je - js) > (ie - is)) {
        int jsplit = js + (je - js) / 2;
        if (jsplit % (64 / sizeof(T))) // split at 64-byte aligned boundary
            jsplit -= jsplit % (64 / sizeof(T));
        cilk_spawn transpose_cache_oblivious(is, ie, js, jsplit, a, n);
        transpose_cache_oblivious(is, ie, jsplit, je, a, n);
    } else {
        int isplit = is + (ie - is) / 2;
        const int jm = (je < isplit ? je : isplit);
        if (isplit % (64 / sizeof(T)))
            isplit -= isplit % (64 / sizeof(T));
        cilk_spawn transpose_cache_oblivious(is, isplit, js, jm, a, n);
        transpose_cache_oblivious(isplit, ie, js, je, a, n);
    }
}
```

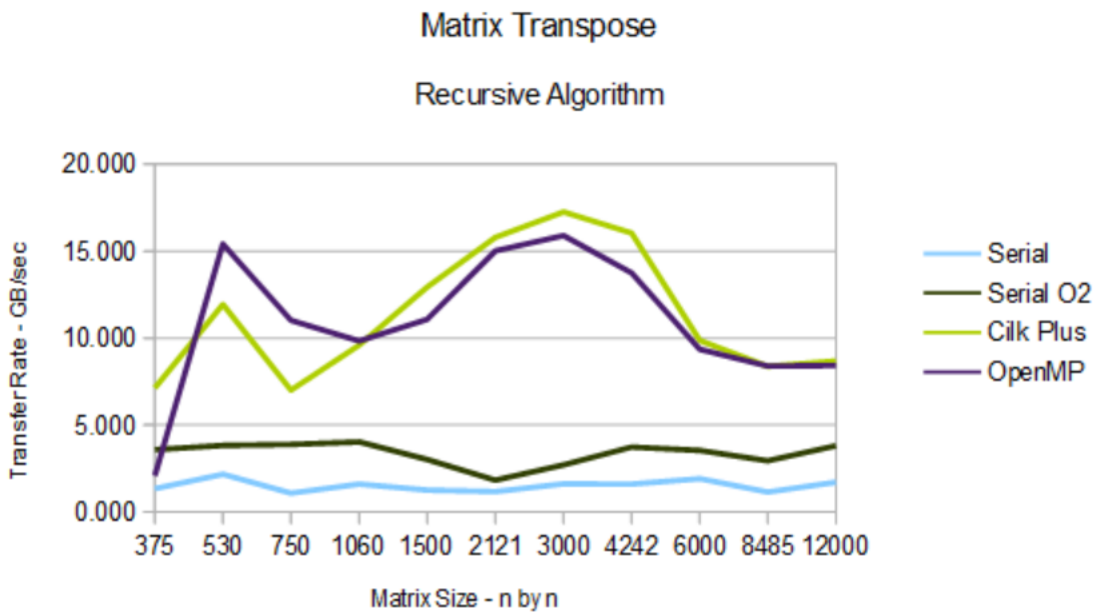
```
template <typename T>
void transpose_cache_oblivious(int is, int ie, int js, int je, T* const a, int n) {
    const int RT = 32;  // recursion threshold
    if (((ie - is) <= RT) && ((je- js) <= RT)) {
        for (int i = is; i < ie; i++) {
            int jm = (je < i ? je : i);
            #pragma simd
            #pragma loop_count avg(RT)
            for (int j = js; j < jm; j++) {
                const T temp = a[i * n + j];
                a[i * n + j] = a[j * n + i];
                a[j * n + i] = temp;
            }
        }
        return;
    }

    // recursive fork-join
    if ((je - js) > (ie - is)) {
        int jsplit = js + (je - js) / 2;
        if (jsplit % (64 / sizeof(T))) // split at 64-byte aligned boundary
            jsplit -= jsplit % (64 / sizeof(T));
        #pragma omp task firstprivate(is, ie, js, n, a, jsplit)
        {
            transpose_cache_oblivious(is, ie, js, jsplit, a, n);
        }
        transpose_cache_oblivious(is, ie, jsplit, je, a, n);
    } else {
        int isplit = is + (ie - is) / 2;
        const int jm = (je < isplit ? je : isplit);
        if (isplit % (64 / sizeof(T)))
            isplit -= isplit % (64 / sizeof(T));
        #pragma omp task firstprivate(is, ie, js, n, a, isplit)
        {
            transpose_cache_oblivious(is, isplit, js, jm, a, n);
        }
        transpose_cache_oblivious(isplit, ie, js, je, a, n);
    }
    #pragma omp taskwait
}
```

Case Study

Matrix Transpose Vladimirov, A. (2013). Cache Traffic Optimization with Cilk Plus and OpenMP

Intel I7 -4960HQ 2.60GHz – Win64

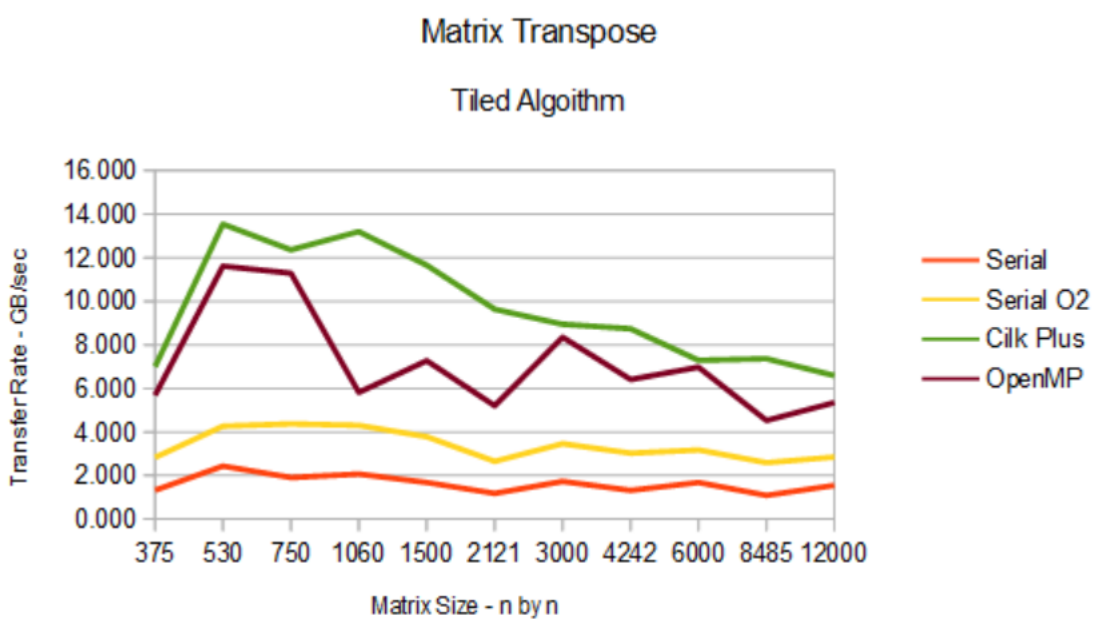


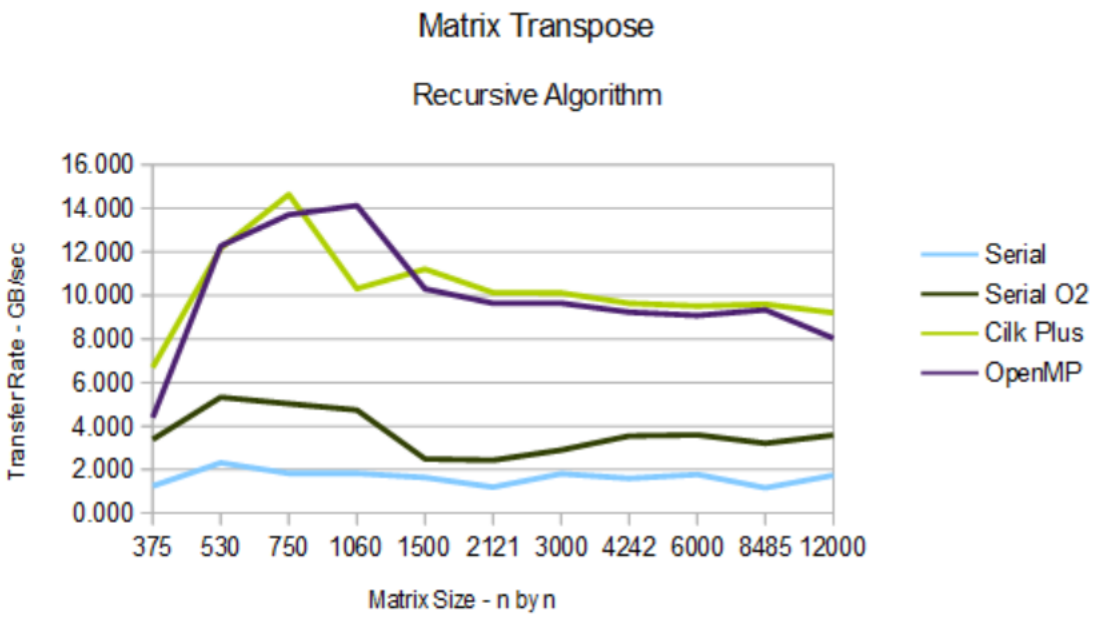


Case Study

Matrix Transpose Vladimirov, A. (2013). Cache Traffic Optimization with Cilk Plus and OpenMP

Intel Xeon E5-1620 3.60 GHz – Win32

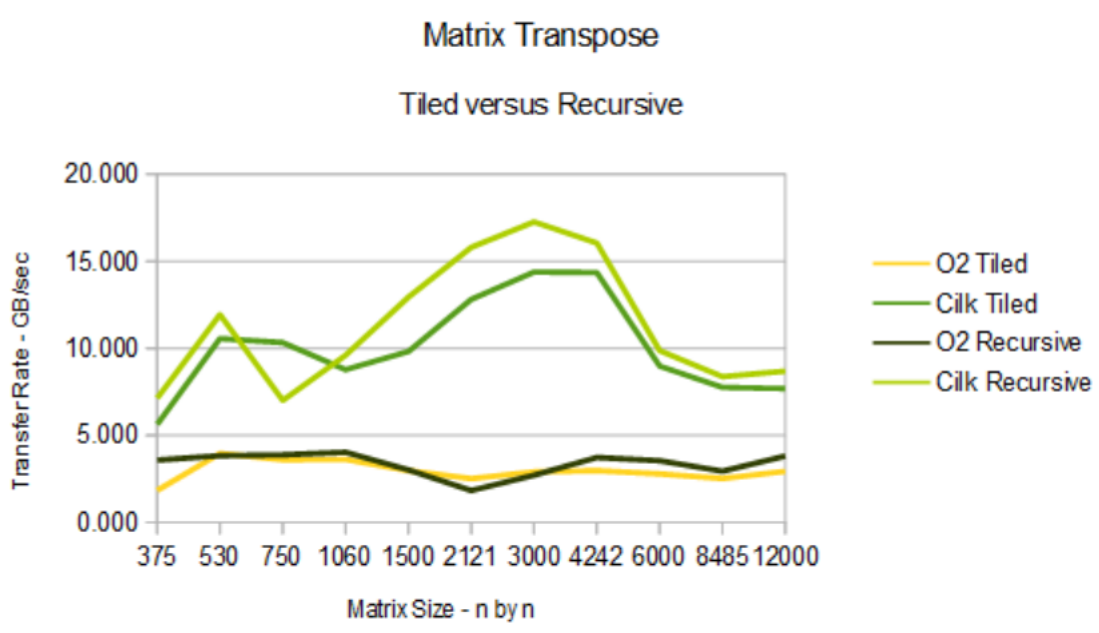




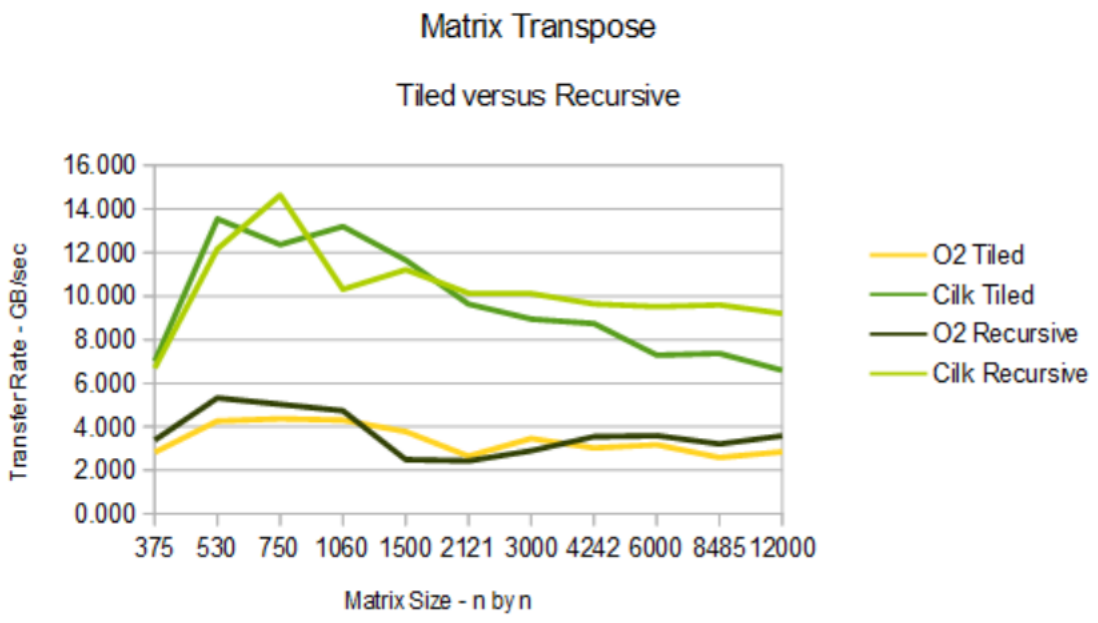
Case Study

Matrix Transpose Vladimirov, A. (2013). Cache Traffic Optimization with Cilk Plus and OpenMP

Intel i7-4960HQ 2.60 GHz – Win64



Intel Xeon E5-1620 3.60 GHz – Win32



References

Intel (2014). User and Reference Guide Intel C++ Compiler 14.0 – Cilk Plus

Intel (2013). User and Reference Manual Intel C++ Compiler 13.1 - Cilk Plus

Intel (2014). Comparing Cilk Plus FAQ

Vladimirov, A. (2013). Cache Traffic Optimization with Cilk Plus and OpenMP

Jarp, S. etal. (2012). Comparison of Software Technologies for Vectorization and Parallelization