

# Linear algebra on multi-core and heterogeneous architectures

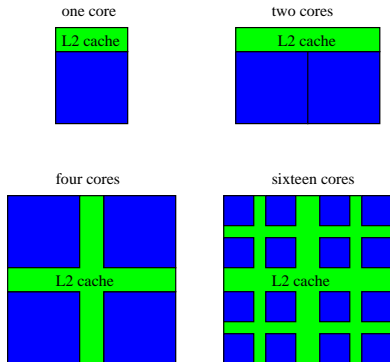
## Multi-core processors

Web page:

<http://www.math.ethz.ch/~kressner/gpucomp.php>

Daniel Kressner

# Multi-core CPUs



- ▶ Cores are independent CPUs, sharing a single coherent cache (L2 or L3). (Memory bandwidth for cache significantly higher than for main memory.)
- ▶ In theory:  $\text{Gflops}(\text{proc}) = \#\text{cores} \times \text{Gflops}(\text{core})$
- ▶ In practice: ?

# Contents

- ▶ Programming multi-core CPUs
  - ▶ PThreads
  - ▶ OpenMP
- ▶ Multi-threaded BLAS
- ▶ Case study: QR factorization.

# PThreads

**Thread** = fork of a program into several concurrently running tasks.

**PThreads** = POSIX threads, standard for managing threads.

PThreads contain support for

- ▶ creating parallelism
- ▶ synchronizing
- ▶ no explicit support for communication, because shared memory is implicit

Characteristics:

- ▶ tight control of shared memory parallelism
- ▶ painful to program

**PThreads** commonly used for low-level programming (e.g., implementation of BLAS).

**OpenMP** commonly used for higher-level programming

# Forking and joining PThreads

```
pthread_create(&thread_id, &thread_attribute,  
              &thread_fun, &fun_arg)
```

`thread_id` thread id or handle

`thread_attribute` optional attributes (default by passing NULL)

`thread_fun` function to be run

`fun_arg` argument to be passed to `thread_fun` when it starts

```
pthread_join (&threadid, &status)
```

`pthread_create` creates a new thread;

`pthread_join` waits for a thread to terminate.

# Hello World with PThreads

Compile with `gcc -lpthread ...`

```
#include <stdio.h>
#include <pthread.h>
void* SayHello(void *foo) {
    printf( "Hello, world!\n" );
    return NULL;
}
int main() {
    pthread_t threads[4];
    int tn;
    for(tn=0; tn<4; tn++) {
        pthread_create(&threads[tn], NULL, SayHello, NULL);
    }
    for(tn=0; tn<4 ; tn++) {
        pthread_join(threads[tn], NULL);
    }
    return 0;
}
```

## Returns

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

# Hello World with OpenMP

Compile with `gcc -fopenmp ...`

```
#include <stdio.h>
#include <omp.h>
int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        printf( "Hello, World!\n" );
    }
    return 0;
}
```

Also returns

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

`#pragma omp parallel` specifies a parallel region of the code;  
4 threads are spawned to execute code independently.

# A scalar product with OpenMP

Scalar product  $y^T x$  with  $x = [1, \dots, 1]$ ,  $y = [1, 2, \dots, 1000]$ .

```
#pragma omp parallel for
for (j = 0; j < k; j++) {
    sum = sum + x[j] * y[j];
}
```

`omp_set_num_threads(1)`  $\rightsquigarrow$  returns 499500.

`omp_set_num_threads(4)`  $\rightsquigarrow$  returns **417578** (first run).

`omp_set_num_threads(4)`  $\rightsquigarrow$  returns **449950** (second run).

`omp_set_num_threads(4)`  $\rightsquigarrow$  returns **439703** (third run).

What happened?

## Race condition

```
#pragma omp parallel for
for (j = 0; j < k; j++) {
    sum += x[j] * y[j];
}
```

- ▶ Loop counter is *private* to each thread but all other variables (in particular `sum`) are shared by all threads.
- ▶ Variable `sum` is read and updated in an asynchronous way; timing of instruction affects output  $\rightsquigarrow$  **race condition**.

Possible way out: define critical regions of code to prevent modification of `sum` by several threads simultaneously.

```
#pragma omp parallel for
for (j = 0; j < k; j++) {
    #pragma omp critical
    { sum += x[j] * y[j]; }
}
```

Takes about 4.5 seconds (independent of `#threads`) for  $k = 10^8$ .

**Large overhead; much too slow!**

# Reduction variables

More practical way out:

```
#pragma omp parallel for reduction(+ : sum)
for (j = 0; j < k; j++) {
    sum += x[j] * y[j];
}
```

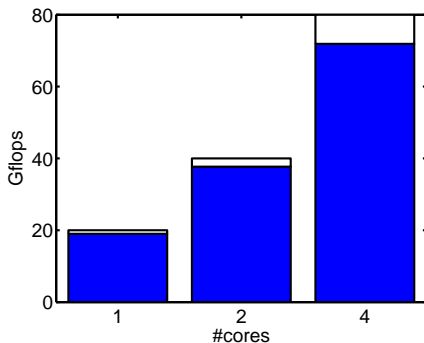
- ▶ **reduction** specifies variable private to each thread and subject to a reduction operation (+) at the end of the parallel region.
- ▶ Correct result; OK performance.
- ▶ OpenMP does not specify how a reduction is implemented; possible implementation: split up for loop into  $p$  loops (one loop/thread) and sum up the output from each loop in a critical region.

For more on OpenMP, see [www.openmp.org/](http://www.openmp.org/).

# Harvesting Multi-Cores with BLAS

Simplest way of benefiting in linear algebra algorithms from multi-cores: [Link to multi-threaded BLAS!](#) (MKL, ATLAS, Goto).

Multiplication of two  $4000 \times 4000$  real SP matrices:

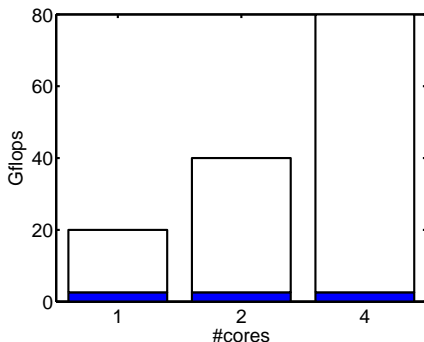


(1 core = 19, 2 cores = 38, 4 cores = 72)

All experiments on 2.5 GHz Intel Core2Quad with 20 SP-Gflops/core;  
Intel MKL 10.1.

## Need level 3 BLAS, not level 2 BLAS

Multiplication of  $4000 \times 4000$  with vector real SP matrices:



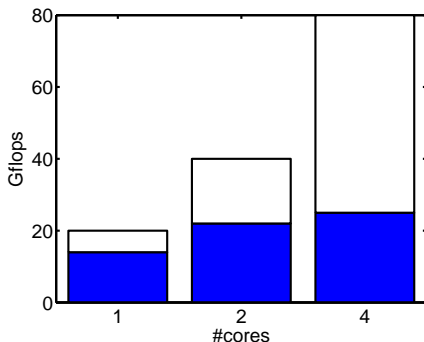
(1 core = 2.6, 2 cores = 2.6, 4 cores = 2.6)

- ▶ Memory bandwidth (main memory  $\rightarrow$  level 2 cache) limits performance. Memory bandwidth remains constant w.r.t. #cores.
- ▶ Level 2 BLAS is performed on a single core.

## Case Study: QR factorization

To compute QR factorization  $A = QR$ , simply link SGEQRF (algorithm based on compact WY representations) with multi-threaded BLAS. High portion of level 3 BLAS  $\leadsto$  good performance expected?

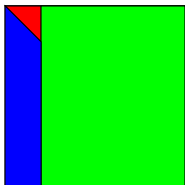
QR factorization of  $4000 \times 4000$  real SP matrix:



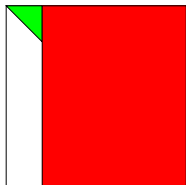
(1 core = 14, 2 cores = 22, 4 cores = 25)

What happened?

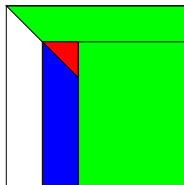
Panel reduction (BLAS 2)



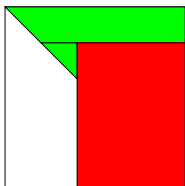
Update (BLAS 3)



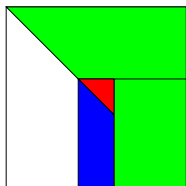
Panel reduction (BLAS 2)



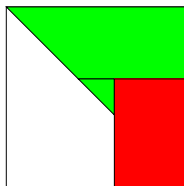
Update (BLAS 3)



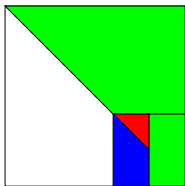
Panel reduction (BLAS 2)



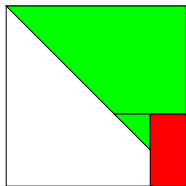
Update (BLAS 3)



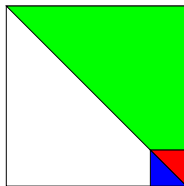
Panel reduction (BLAS 2)



Update (BLAS 3)



Panel reduction (BLAS 2)

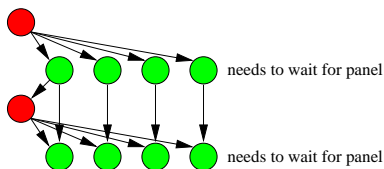


# Profiling QR

Level 2 panel factorizations and level 3 updates.



Picture source: Jack Dongarra. Corresponds to the LU factorization, which has a similar work flow.



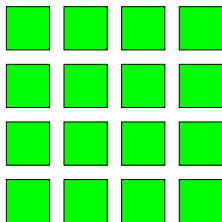
- ▶ Update cannot start until panel factorization is completed.
- ▶ Inherit algorithmic limitation; requires redesign.

## Tiling – Revival of an old idea

- ▶ Problem is memory bandwidth limitations.
- ▶ Decades ago: algorithms dealing with out-of-core memory (= hard disk) issues.

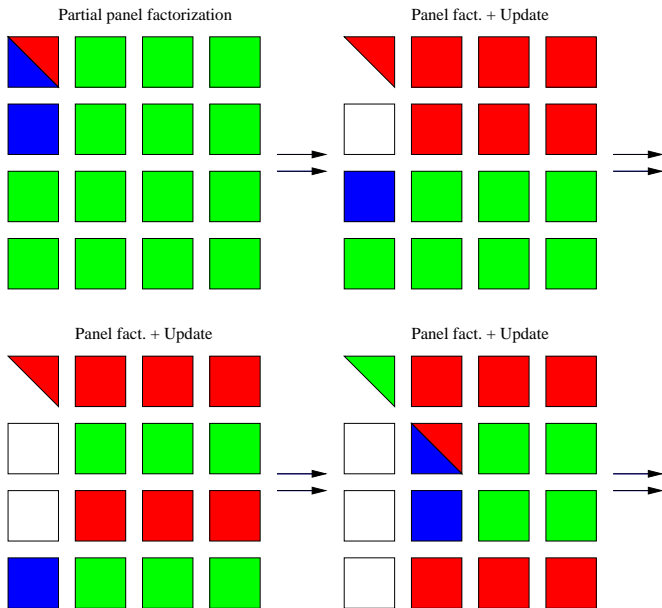
Basic idea:

- ▶ Partition matrix into tiles:



- ▶ Factorize much smaller “panels”.

# Tiling – Revival of an old idea





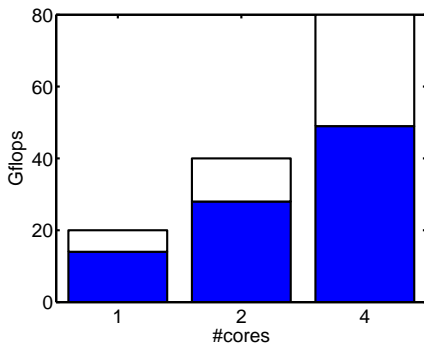
# Traversing the DAG of dependencies

- ▶ Can only execute top nodes (with no fathers).
- ▶ After computation, remove nodes and outgoing edges.
- ▶ Always sufficiently many independent tasks, except for very beginning and very end.
- ▶ Tile size should fit with L1 cache.
- ▶ **Static**: Task scheduling hardwired. Little compt. overhead, but painful to program.
- ▶ **Dynamic**: Task scheduling determined dynamically during execution of program. ( $\exists$  assisting software packages)
- ▶ Old vs. new profile:



# QR factorization with tiling

Static pipeline implementation



(1 core = 14, 2 cores = 28, 4 cores = 49)

# Dynamic scheduling with SMPs

**SMP Superscalar framework** by Barcelona SC center.

- ▶ Download from [www.bsc.es/smpsuperscalar](http://www.bsc.es/smpsuperscalar)
- ▶ Assumption: Application composed of coarse grained functions/tasks.
- ▶ User needs to identify inputs/outputs of each task:

```
#pragma css task input(<input parameters>)  
                inout(<inout parameters>)  
                output(<output parameters>)
```

(some tricks might be needed to, e.g., declare access to lower/upper triangular parts of a matrix)

- ▶ SMPs pragmas are preprocessed and transformed into C code.
- ▶ SMPs creates DAG automatically.

# QR factorization with SMPs

```
#pragma css task inout(RV1[NB][NB]) output(T[NB][NB])
void sgeqrt(real *RV1, real *T);

#pragma css task inout(R[NB][NB], V2[NB][NB]) output(T[NB][NB])
void stsqrt(real *R, real *V2, real *T);

#pragma css task input(V1[NB][NB], T[NB][NB]) inout(C1[NB][NB])
void slarfb(real *V1, real *T, real *C1);

#pragma css task input(V2[NB][NB], T[NB][NB]) inout(C1[NB][NB], C2[NB][NB])
void sssrfb(real *V2, real *T, real *C1, real *C2);

#pragma css start
for (k = 0; k < TILES; k++) {
    sgeqrt(A[k][k], T[k][k]);

    for (m = k+1; m < TILES; m++)
        stsqrt(A[k][k], A[m][k], T[m][k]);

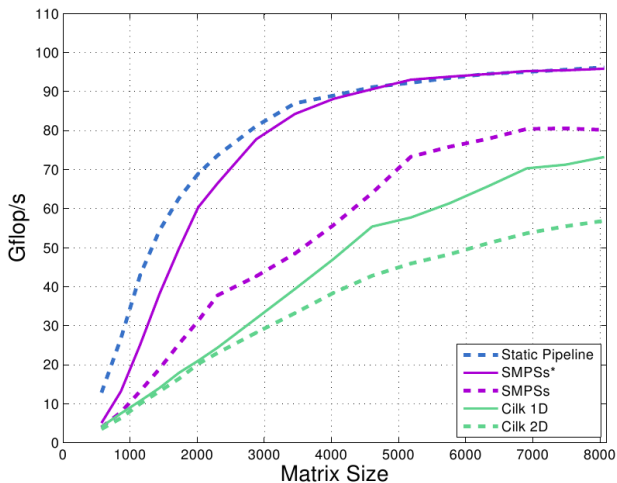
    for (n = k+1; n < TILES; n++) {
        slarfb(A[k][k], T[k][k], A[k][n]);

        for (m = k+1; m < TILES; m++)
            sssrfb(A[m][k], T[m][k], A[k][n], A[m][n]);
    }
}
#pragma css finish
```

# QR factorization with tiling – more performance results

Source: Kurzak, Ltaief, Dongarra, Badia

On 16 cores Intel Tigerton (peak performance 147 Gflops):



# Summary

- ▶ Linking to multi-threaded BLAS good first start but has obvious limitations.
- ▶ There are good tools (like SMPSSs) available with which the use of PThreads or OpenMP can be avoided.
- ▶ Further reading:  
Several books on pthreads and OpenMP, but none specific to linear algebra.

Presentation draws heavily on work by Dongarra and his group, see in particular: J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia.

Scheduling Linear Algebra Operations on Multicore Processors, 2009.

LAPACK Working Note 213, see

<http://www.netlib.org/lapack/lawns/downloads/>.