Parallel $\mathcal{H}$-Matrices

Algorithms and Arithmetic

Ronald Kriemann

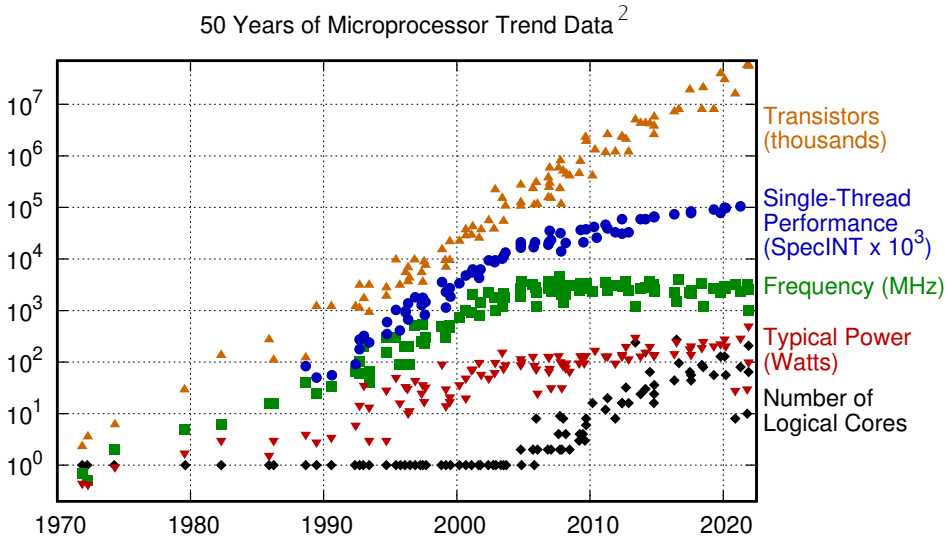
Winterschool on $\mathcal{H}$-Matrices

2024

MAX PLANCK INSTITUTE
FOR MATHEMATICS IN THE SCIENCES

Introduction

The Free Lunch is over![1]

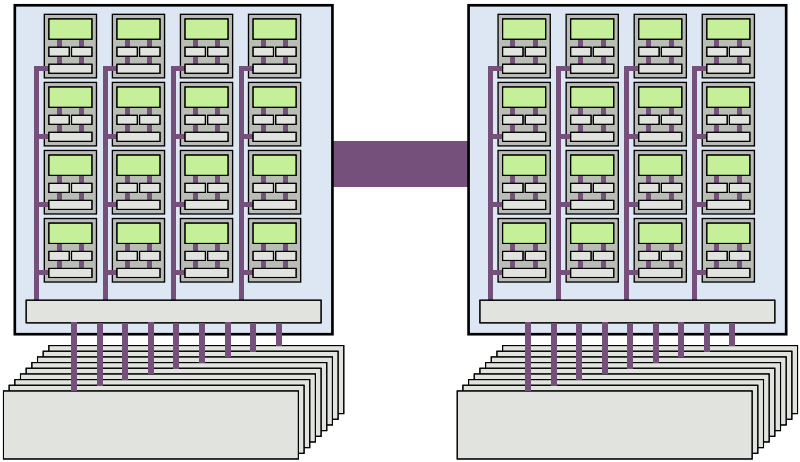


[1] H. Sutter: "*The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*", Dr. Dobb's Journal, 30(3), March 2005.

[2] K. Rupp: "*Microprocessor Trend Data*", `https://github.com/karlrupp/microprocessor-trend-data`

Parallel Systems

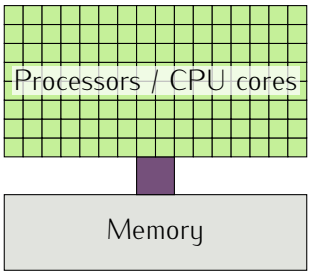
A typical compute server looks like this:



Each processor has several CPU cores and multiple levels of memory.

Parallel Systems

For programming, this is simplified to a *shared memory* system based on *symmetric multiprocessing*:

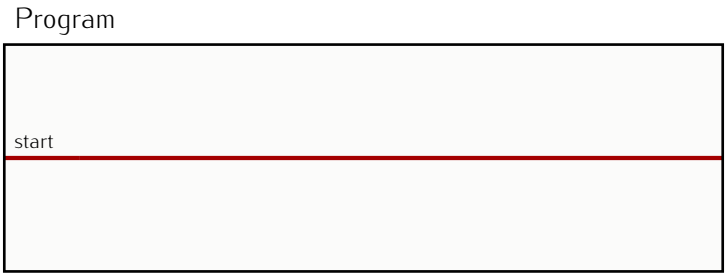


- All memory is *shared* by all processors.
- All processors/CPU cores are considered *equal*.

Threads

To enable shared memory parallelism, a program needs to use *threads*, i.e., parallel execution paths.

When a program starts, a single thread exists.

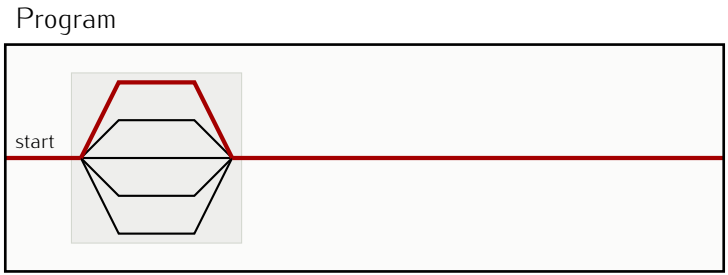


Threads

To enable shared memory parallelism, a program needs to use *threads*, i.e., parallel execution paths.

When a program starts, a single thread exists.

One may then spawn new threads, which run in parallel with the existing, *master* thread.



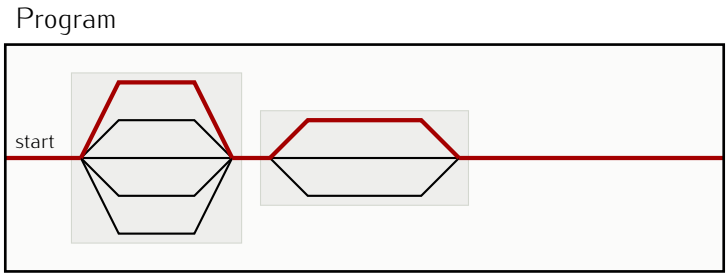
When all threads finish execution, a *synchronisation* takes place, i.e. the master thread only proceed when all other threads have finished.

Threads

To enable shared memory parallelism, a program needs to use *threads*, i.e., parallel execution paths.

When a program starts, a single thread exists.

One may then spawn new threads, which run in parallel with the existing, *master* thread.



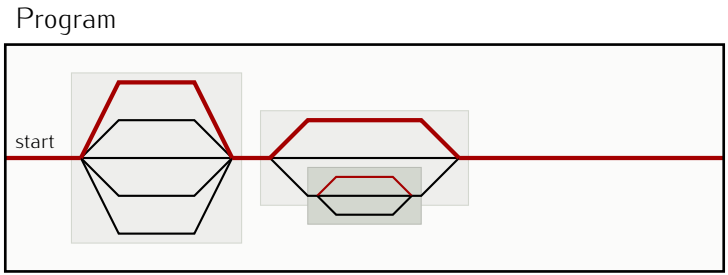
When all threads finish execution, a *synchronisation* takes place, i.e. the master thread only proceed when all other threads have finished.

Threads

To enable shared memory parallelism, a program needs to use *threads*, i.e., parallel execution paths.

When a program starts, a single thread exists.

One may then spawn new threads, which run in parallel with the existing, *master* thread.



When all threads finish execution, a *synchronisation* takes place, i.e. the master thread only proceed when all other threads have finished.

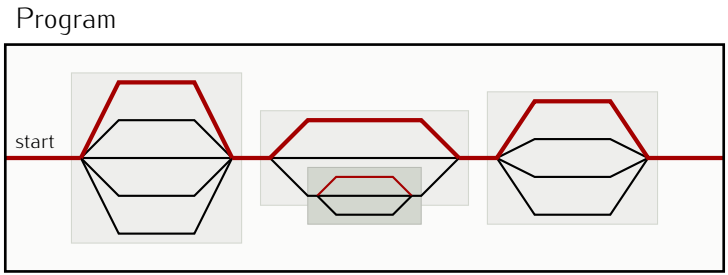
This may also be done in a *nested* way: any thread may create new threads.

Threads

To enable shared memory parallelism, a program needs to use *threads*, i.e., parallel execution paths.

When a program starts, a single thread exists.

One may then spawn new threads, which run in parallel with the existing, *master* thread.



When all threads finish execution, a *synchronisation* takes place, i.e. the master thread only proceed when all other threads have finished.

This may also be done in a *nested* way: any thread may create new threads.

Tasks and DAGs

Task

A task is considered an *atomic* computational unit, i.e. is not divided any further.

Different tasks of the same computation may be of different size.

The goal is to define as many *independent* tasks as possible to compute as much as possible in parallel.

Tasks and DAGs

Task

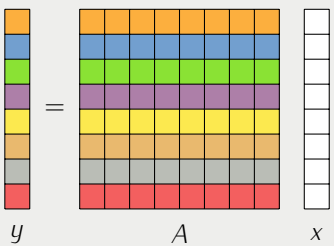
A task is considered an *atomic* computational unit, i.e. is not divided any further.

Different tasks of the same computation may be of different size.

The goal is to define as many *independent* tasks as possible to compute as much as possible in parallel.

Example: Dense Matrix-Vector Multiplication $y = Ax$

The computation may be split into tasks for each dot-product $y_i := \sum_j a_{ij} x_j$:



All of these n tasks are independent and may be computed in parallel.

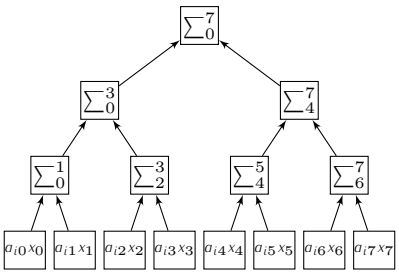
Tasks and DAGs

One could divide the dot-products into smaller tasks, i.e. *one task per* $a_{ij}x_j$, yielding n^2 independent tasks.

Tasks and DAGs

One could divide the dot-products into smaller tasks, i.e. *one task per* $a_{ij}x_j$, yielding n^2 independent tasks.

All products $a_{ij}x_j$ then need to be summed up!

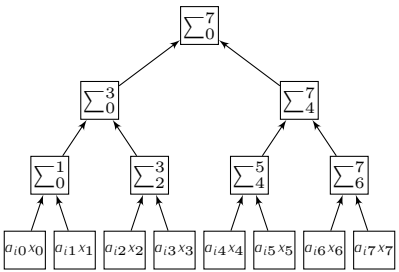


Some tasks of the computation now *depend* on the results of other tasks.

Tasks and DAGs

One could divide the dot-products into smaller tasks, i.e. *one task per* $a_{ij}x_j$, yielding n^2 independent tasks.

All products $a_{ij}x_j$ then need to be summed up!



Some tasks of the computation now *depend* on the results of other tasks.

The dependencies form edges between tasks/nodes in a *directed acyclic graph* (DAG), the *task-dependency graph*.

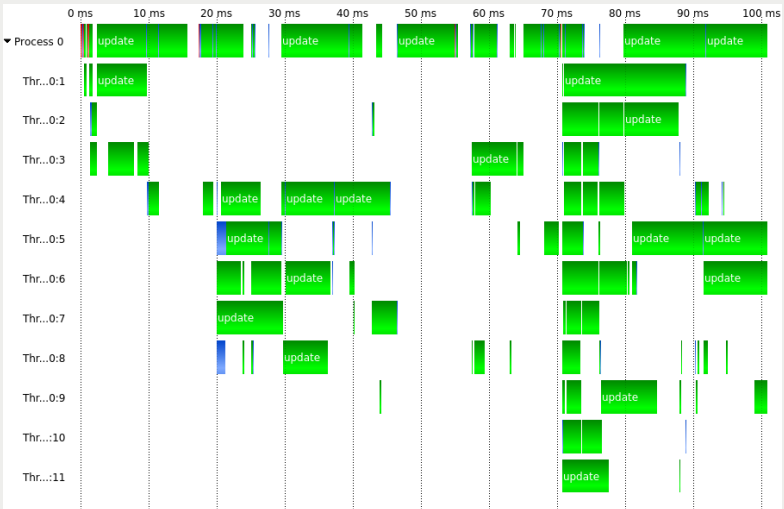
Task Scheduling

Scheduling is the assignment of tasks to processors for the execution of the task.

Task Scheduling

Scheduling is the assignment of tasks to processors for the execution of the task.

Example: $\mathcal{H}$-LU Factorization

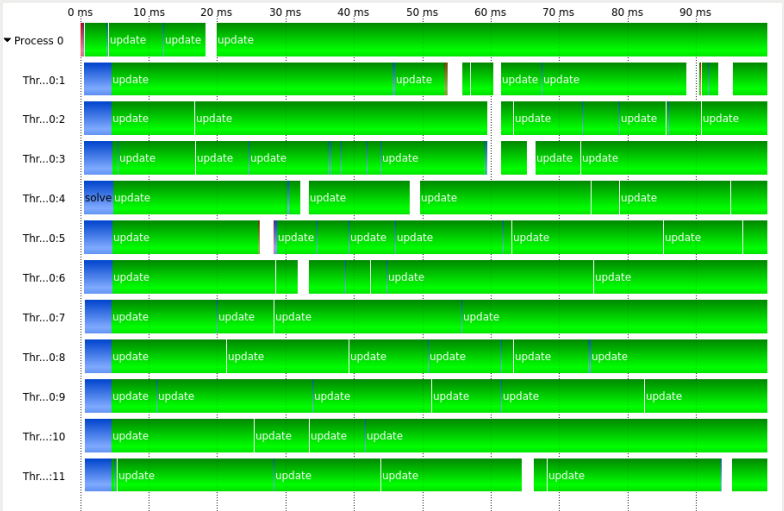


Traditional Parallelization

Task Scheduling

Scheduling is the assignment of tasks to processors for the execution of the task.

Example: $\mathcal{H}$-LU Factorization



DAG-based Approach

Task Scheduling

Task scheduling is (normally) performed *automatically* by the *runtime system* enabling task based programming.

Available runtime systems:

- OpenMP
- StarPU
- PaRSEC
- TBB
- HPX
- C++-TaskFlow
- ...

Task Scheduling

Task scheduling is (normally) performed *automatically* by the *runtime system* enabling task based programming.

Available runtime systems:

- **OpenMP**
- StarPU
- PaRSEC
- TBB
- HPX
- C++-TaskFlow
- ...

OpenMP

OpenMP

OpenMP started as a joined initiative of the major hardware and software vendors to provide compiler support for parallel programs.

1997: OpenMP 1.0 for Fortran
1998: OpenMP 1.0 for C/C++
2000: OpenMP 2.0 for Fortran with fixes and clarifications
2002: OpenMP 2.0 for C/C++
2005: OpenMP 2.5 combining Fortran and C/C++
2008: OpenMP 3.0 introduces *tasks*
2013: OpenMP 4.0 support for vector units and special targets
2015: OpenMP 4.5 taskloops
2018: OpenMP 5.0 taskloop reductions, C++ range loops

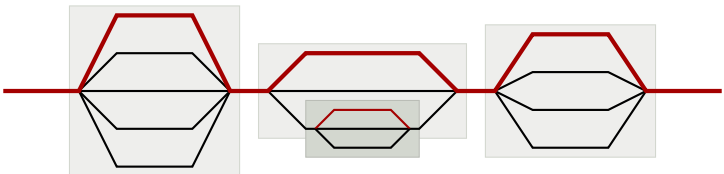
Beside the compiler directives (pragmas), OpenMP also contains a set of data types and functions, imported via the header file `omp.h`.

All major C/C++ and Fortran compilers support OpenMP.

OpenMP

The underlying model of OpenMP is the *fork–join model* which is based on threads:

> *A master thread creates a team of worker threads which run in parallel together with the master thread until all worker threads finish.*



The number of threads in a team may differ in different parallel sections of the program.

OpenMP

OpenMP extends the underlying programming language by *compiler directives*.

Each OpenMP directive starts with the pragma

```
#pragma omp <directive> new-line
```

The directive is applied to the immediately following source code block, i.e.

```
void f () {
  #pragma omp <directive>
  {
    ...
  }

  #pragma omp <directive>
  ...
}
```

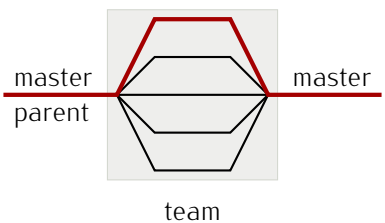
After the source code block, the worker threads of the thread team will stop and only the master thread will proceed.

A new team of threads is created by the directive

```
#pragma omp parallel
```

The thread encountering the `parallel` directive is called the *master* or *parent* thread of the created team.

```
void f () {
  ...                       // executed by master thread
  #pragma omp parallel
  {
    ...                     // executed by team
  }
  ...                       // executed by master thread
}
```



Example: Hello World

```
#include <stdio.h>
int main () {

  printf( "Hello, world!\n" );  // executed by master thread
}
```

Example: Hello World

```
#include <stdio.h>
int main () {
  #pragma omp parallel
  printf( "Hello, world!\n" );  // executed by all threads
}
```

Example: Hello World

```
#include <stdio.h>
int main () {
  #pragma omp parallel
  printf( "Hello, world!\n" );  // executed by all threads
}
```

To enable OpenMP during compilation, a special compiler flag has to be provided:

```
> gcc -fopenmp -o hello -c hello.c
```

Example: Hello World

```
#include <stdio.h>
int main () {
  #pragma omp parallel
  printf( "Hello, world!\n" );  // executed by all threads
}
```

To enable OpenMP during compilation, a special compiler flag has to be provided:

```
> gcc -fopenmp -o hello -c hello.c
```

Possible program outputs on a 4-Core CPU may be:

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

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

The garbled output is due to a *critical section* where all CPUs compete for the same output.

Remark

A typical programming mistake is to forget the `parallel` *directive:*

```
int main () {
  #pragma omp             // no "parallel"
  {
    ...
  }
}
```

In this case, no new threads are created and the code will be executed by the master thread alone, i.e. sequential execution.

Loops

A standard loop in a program may be parallelized with the directive

```
#pragma omp for
```

Example: Vector addition $y := \alpha x + y$

```
void axpy ( int n, double alpha, double * x, double * y ) {


  for ( int i = 0; i < n; ++i )
    y[i] += alpha * x[i];
}
```

Loops

A standard loop in a program may be parallelized with the directive

```
#pragma omp for
```

Example: Vector addition $y := \alpha x + y$

```
void axpy ( int n, double alpha, double * x, double * y ) {
  #pragma omp parallel
  #pragma omp for
  for ( int i = 0; i < n; ++i )
    y[i] += alpha * x[i];
}
```

Loops

A standard loop in a program may be parallelized with the directive

```
#pragma omp for
```

Example: Vector addition $y := \alpha x + y$

```
void axpy ( int n, double alpha, double * x, double * y ) {
  #pragma omp parallel
  #pragma omp for
  for ( int i = 0; i < n; ++i )
    y[i] += alpha * x[i];
}
```

The `parallel` and the `for` directive may be joined:

```
void axpy ( int n, double alpha, double * x, double * y ) {
  #pragma omp parallel for
  for ( int i = 0; i < n; ++i )
    y[i] += alpha * x[i];
}
```

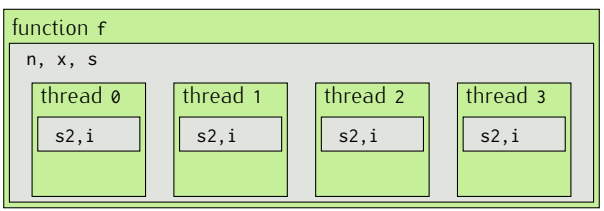
Shared and Private Data

Data defined in the surrounding block of a parallel code block is *shared* by all threads. Data defined within a parallel code block is *private* to a specific thread.

At line 4, the variables `n`, `x` and `s` are already defined. In the block 5–13, these variables are then shared by all threads.

The variables `s2` and `i` are defined *within* the parallel code block and are therefore, private to each thread.

```
1  void f ( int n, double * x ) {
2    double  s = 0.0;
3
4    #pragma omp parallel
5    {
6      double s2 = 0;
7
8      #pragma omp for
9      for ( int i = 0; i < n; ++i )
10       s2 += x[i];
11
12     s = s+s2;
13   }
14 }
```



Thread Dependencies

Any change to a shared variable will affect all threads of the thread team, and hence, defines a *critical section*.

```
1  void f ( int n, double * x ) {
2    double  s = 0.0;
3
4    #pragma omp parallel
5    {
6      double s2 = 0;
7
8      #pragma omp for
9      for ( int i = 0; i < n; ++i ) // n : read-only
10       s2 += x[i];                 // x : read-only
11
12
13     s = s+s2;                     // s : read-write
14   }
15 }
```

At line 11 the shared variable `s` is updated, forming a critical section.

Thread Dependencies

Any change to a shared variable will affect all threads of the thread team, and hence, defines a *critical section*.

```
1  void f ( int n, double * x ) {
2    double  s = 0.0;
3
4    #pragma omp parallel
5    {
6      double s2 = 0;
7
8      #pragma omp for
9      for ( int i = 0; i < n; ++i ) // n : read-only
10       s2 += x[i];                 // x : read-only
11
12     #pragma omp critical
13     s = s+s2;                     // s : read-write
14   }
15 }
```

At line 11 the shared variable `s` is updated, forming a critical section.

With the `critical` directive, a critical section is defined and ensured that only a single thread is in the section at any time. However, this then applies to *all* threads.

Thread Dependencies

A better approach are *mutexes*, which are standard variables and handled as such.

```
#include <omp.h>                // include OpenMP functions/types

void f ( int n, double * x ) {
  double      s = 0.0;
  omp_lock_t  mtx;              // define mutex variable

  omp_init_lock( & mtx );       // initialize mutex

  #pragma omp parallel          // mtx is a shared variable
  {
    double s2 = 0;

    #pragma omp for
    for ( int i = 0; i < n; ++i )
      s2 += x[i];

    omp_set_lock( & mtx )       // lock mutex
    s = s+s2;
    omp_unset_lock( & mtx )     // unlock mutex
  }

  omp_destroy_lock( & mtx );    // destroy mutex
}
```

Task based Computations

Since v3.0, OpenMP supports explicit task creation without thread binding:

```
#pragma omp task
```

Each task consists of

- *code* to execute and
- a *data environment*.

The code is defined by the code block:

```
#pragma omp task
{
  ...           // code executed in task
}
```

All code reachable by code in the block defines the *task region*.

Task based Computations

Since v3.0, OpenMP supports explicit task creation without thread binding:

```
#pragma omp task
```

Each task consists of

- *code* to execute and
- a *data environment*.

The code is defined by the code block:

```
#pragma omp task
{
  ...           // code executed in task
}
```

All code reachable by code in the block defines the *task region*.

Important:

- The construction time of a task and the execution time of a task may differ.
- The constructing thread of a task and the executing thread of a task may differ.

Task based Computations

All threads, which encounter a `task` directive, will generate a new task:

```
#pragma omp parallel
{
  for ( int i = 0; i < 4; ++i ) {
    #pragma omp task
    {
      ...
} } }
```

Each of the p threads will execute the `parallel` code, generating $4 \cdot p$ tasks!

Task based Computations

All threads, which encounter a `task` directive, will generate a new task:

```
#pragma omp parallel
{
  for ( int i = 0; i < 4; ++i ) {
    #pragma omp task
    {
      ...
} } }
```

Each of the p threads will execute the `parallel` code, generating $4 \cdot p$ tasks!

If tasks should be generated only *once* for an algorithm, the `single` or `master` directive may be used:

```
#pragma omp parallel
{
  #pragma omp single
  for ( int i = 0; i < 4; ++i ) {
    #pragma omp task
    {
      ...
} } }
```

Now, 4 tasks are generated but *all* team threads will execute the tasks.

Data Environment

The *data environment* of a task is the set of all variables which are in the scope of the task region.

shared

Shared variables will refer to the memory address at task construction. This address must be *valid* until the task has finished execution:

```
void f () {
  double  x = produce_x();    // lifetime restricted to f, not the task
  #pragma omp task shared(x)
  { consume_x( x ); }         // error: x may no longer exist
}
```

Data Environment

The *data environment* of a task is the set of all variables which are in the scope of the task region.

shared

Shared variables will refer to the memory address at task construction. This address must be *valid* until the task has finished execution:

```
void f () {
  double  x = produce_x();    // lifetime restricted to f, not the task
  #pragma omp task shared(x)
  { consume_x( x ); }         // error: x may no longer exist
}
```

firstprivate

Variable is defined with value at task construction. It is allocated and packaged together with the task code.

```
void f () {
  double  x = produce_x();
  #pragma omp task firstprivate(x)
  { consume_x( x ); }         // ok: copy of x is packaged with task
}
```

Data Environment

private

Private variables will be uninitialised and hence, only allocated when the task is scheduled for execution.

By default, variables referenced in the task code block with no explicit data sharing rules are shared.

```
void f () {
  double  x1 = 1.0;
  double  x2 = 2.0;

  #pragma omp task firstprivate(x2)
  {
    double  x3 = 4.0;   // private due to scope

    // x1 : shared        ( shared by all implicit tasks )
    // x2 : private       ( due to “firstprivate(x2)” )
    // x3 : private
  }
}
```

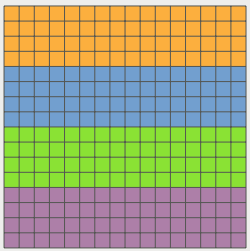
Task Loops

Loops can be automatically partitioned into tasks by the *taskloop* directive:

```
#pragma omp taskloop
```

Example: Matrix-Multiplication

```
void matmul ( int n, double * A, double * B, double * C ) {
  #pragma omp parallel
  #pragma omp single
  #pragma omp taskloop
  for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
      for ( int l = 0; l < n; ++l )
        C[i+j*n] += A[i+l*n]*B[l+j*n];
} } }
```



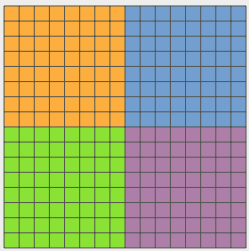
Task Loops

Loops can be automatically partitioned into tasks by the *taskloop* directive:

```
#pragma omp taskloop
```

Example: Matrix-Multiplication

```
void matmul ( int n, double * A, double * B, double * C ) {
  #pragma omp parallel
  #pragma omp single
  #pragma omp taskloop collapse(2)
  for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
      for ( int l = 0; l < n; ++l )
        C[i+j*n] += A[i+l*n]*B[l+j*n];
} } }
```



With the `collapse(l)` option, loop decomposition is applied to l nested loops.

Task Dependencies

A task encountering a `task` directive becomes the *parent* task to the newly created *child* task.

Code in the code block of a child task is *not* part of the task region of the parent task:

```
#pragma omp task
{
  ...                  // part of parent task

  #pragma omp task
  {
    ...                // part of child task, not of parent task
  }

  ...                  // part of parent task
}
```

After creating the child task, the parent may immediatly proceed with the execution of it's task region.

Task Dependencies

Tasks may be part of a *task group*.

```
#pragma omp taskgroup
```

When finishing the task group, the parent tasks blocks and waits for all child tasks to finish.

```
double dot ( int n, double * x, double * y ) {
  double  d1, d2;



    #pragma omp task
    d1 = dot( n/2, x, y );

    #pragma omp task
    d2 = dot( n/2, x + n/2, y + n/2 );


  return d1+d2;
}
```

Task Dependencies

Tasks may be part of a *task group*.

```
#pragma omp taskgroup
```

When finishing the task group, the parent tasks blocks and waits for all child tasks to finish.

```
double dot ( int n, double * x, double * y ) {
  double  d1, d2;

  #pragma omp taskgroup
  {
    #pragma omp task
    d1 = dot( n/2, x, y );

    #pragma omp task
    d2 = dot( n/2, x + n/2, y + n/2 );
  }                                    // wait for child tasks

  return d1+d2;
}
```

Task Dependencies

Full example for task based computation:

```
double dot ( int n, double * x, double * y ) {
  if ( n >= 32 ) {        // defines task granularity
    double  d1, d2;       // parallel recursion

    #pragma omp taskgroup
    {
      #pragma omp task
      d1 = dot( n/2, x, y );
      #pragma omp task
      d2 = dot( n/2, x + n/2, y + n/2 );
    }                     // wait for child tasks/sub results
    return d1+d2;
  } else {
    double  d = 0;        // sequential execution
    for ( int i = 0; i < n; ++i ) d += x[i] * y[i];
    return d;
  }
}

int main () {
  int n = ...; double * x = ...; double * y = ...;
  double d = 0;

  #pragma omp parallel    //  start threads
  #pragma omp single      //  ensure only single thread generates tasks
  #pragma omp task        //  top-level task
  d = dot( n, x, y );
}
```

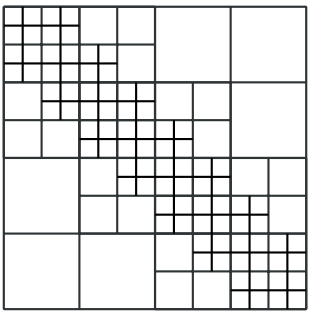
$\mathcal{H}$-Matrix Construction

$\mathcal{H}$-Matrix Construction

Let I be an index set, $T(I)$ a $\mathcal{H}$-tree over I and $T = T(I \times I)$ a $\mathcal{H}_\times$-tree over $T(I)$ with $\mathcal{L}(T)$ being the set of leaves of T.

The basic algorithm for $\mathcal{H}$-matrix construction is

```
function BUILD((τ, σ))
    if (τ, σ) ∈ L(T) then
        if (τ, σ) is admissible then
            build low-rank matrix;
        else
            build dense matrix;
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            build((τ', σ'));
```

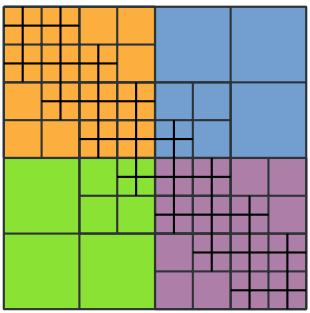


$\mathcal{H}$-Matrix Construction

Let I be an index set, $T(I)$ a $\mathcal{H}$-tree over I and $T = T(I \times I)$ a $\mathcal{H}_\times$-tree over $T(I)$ with $\mathcal{L}(T)$ being the set of leaves of T.

The basic algorithm for $\mathcal{H}$-matrix construction is

```
function BUILD((τ, σ))
    if (τ, σ) ∈ L(T) then
        if (τ, σ) is admissible then
            build low-rank matrix;
        else
            build dense matrix;
    else
        #pragma omp for
        for all (τ', σ') ∈ S((τ, σ)) do
            build((τ', σ'));
```



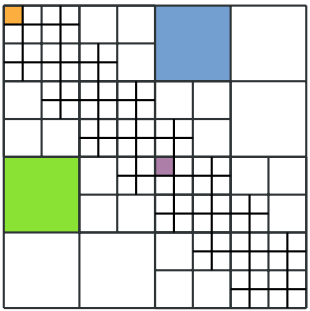
With classical OpenMP loop based parallelization would be used.

However, this may lead to the problems with load balancing.

$\mathcal{H}$-Matrix Construction

Using a task based approach, the algorithm looks as

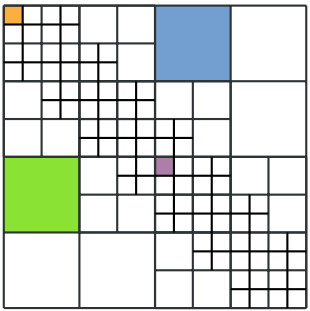
```
function BUILD((τ, σ))
    if (τ, σ) ∈ L(T) then
        #pragma omp task
            if (τ, σ) is admissible then
                build low-rank matrix;
            else
                build dense matrix;
    else
        for all (τ′, σ′) ∈ S((τ, σ)) do
            build((τ′, σ′));
```



$\mathcal{H}$-Matrix Construction

Using a task based approach, the algorithm looks as

```
function BUILD((τ, σ))
    if (τ, σ) ∈ L(T) then
        if (τ, σ) is admissible then
            build low-rank matrix;
        else
            build dense matrix;
    else
        for all (τ′, σ′) ∈ S((τ, σ)) do
            #pragma omp task
                build((τ′, σ′));
```

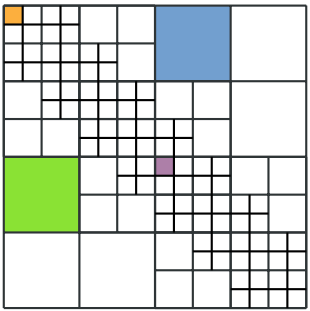


The construction of *each* node in T defines a new task.

$\mathcal{H}$-Matrix Construction

Using a task based approach, the algorithm looks as

```
function BUILD((τ, σ))
    if (τ, σ) ∈ L(T) then
        if (τ, σ) is admissible then
            build low-rank matrix;
        else
            build dense matrix;
    else
        #pragma omp taskgroup          // optional
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
            build((τ', σ'));
```

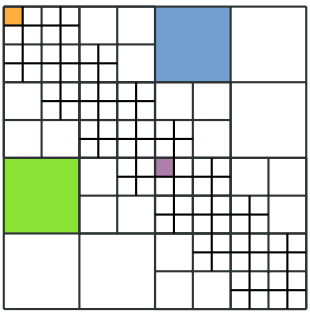


The construction of *each* node in T defines a new task.

$\mathcal{H}$-Matrix Construction

Using a task based approach, the algorithm looks as

```
function BUILD((τ, σ))
    if (τ, σ) ∈ L(T) then
        if (τ, σ) is admissible then
            build low-rank matrix;
        else
            build dense matrix;
    else
        #pragma omp taskgroup          // optional
            for all (τ', σ') ∈ S((τ, σ)) do
                #pragma omp task
                    build((τ', σ'));
```



The construction of *each* node in T defines a new task.

In $\mathcal{H}$-matrix construction:

- there are much more tasks than processors ($\#\mathcal{L}(T) \gg p$) and
- each leaf block can be built *independently*.

$\mathcal{H}$-Matrix Construction

hmatrix.c

```
phmatrix
build_from_block_hmatrix ( pcblock b, uint k ) {
  phmatrix  h = NULL;

  if (b->son) {
    int rsons = b->rsons;
    int csons = b->csons;
    h = new_super_hmatrix( b->rc, b->cc, rsons, csons );



    for (int j = 0; j < csons; j++) {
      for (int i = 0; i < rsons; i++) {
        pcblock  b1 = b->son[i+j*rsons];
        phmatrix h1 = build_from_block_hmatrix( b1, k );
        ref_hmatrix(h->son + i + j * rsons, h1);
    } }
  } else if (b->a > 0) {
    h = new_rk_hmatrix(b->rc, b->cc, k);
  } else {
    h = new_full_hmatrix(b->rc, b->cc);
  }

  update_hmatrix(h);
  return h;
}
```

example_hmatrix_bem3d.c

```
V = build_from_block_hmatrix(broot,m*m*m);
```

$\mathcal{H}$-Matrix Construction

hmatrix.c

```
phmatrix
build_from_block_hmatrix ( pcblock b, uint k ) {
  phmatrix  h = NULL;

  if (b->son) {
    int rsons = b->rsons;
    int csons = b->csons;
    h = new_super_hmatrix( b->rc, b->cc, rsons, csons );

    #pragma omp taskgroup
    #pragma omp taskloop collapse(2)
    for (int j = 0; j < csons; j++) {
      for (int i = 0; i < rsons; i++) {
        pcblock  b1 = b->son[i+j*rsons];
        phmatrix h1 = build_from_block_hmatrix( b1, k );
        ref_hmatrix(h->son + i + j * rsons, h1);
    } }
  } else if (b->a > 0) {
    h = new_rk_hmatrix(b->rc, b->cc, k);
  } else {
    h = new_full_hmatrix(b->rc, b->cc);
  }

  update_hmatrix(h);
  return h;
}
```

example_hmatrix_bem3d.c

```
#pragma omp parallel
#pragma omp single
#pragma omp task   // optional
V = build_from_block_hmatrix(broot,m*m*m);
```

$\mathcal{H}$-Matrix Construction

block.c

```
void
iterate_rowlist ( ... ) {
  ...
  #pragma omp taskgroup
  #pragma omp taskloop
  for (i = 0; i < rc->sons; i++)
    iterate_rowlist( rc->son[i], rnames[i], pb1[i],
                     (pardepth > 0 ? pardepth - 1 : 0),
                     pre, post, data);
  ...
 }
```

example_hmatrix_bem3d.c

```
#pragma omp parallel
#pragma omp single
assemble_bem3d_hmatrix(bem_slp, broot, V);
```

Numerical Results

Results for Laplace-SLP, $n = 131.072$

Intel i7-13700		
# Cores	Runtime	Speedup
1	67.0s	

Numerical Results

Results for Laplace-SLP, $n = 131.072$

Intel i7-13700		
# Cores	Runtime	Speedup
1	67.0s	
2	34.8s	1.93x

Numerical Results

Results for Laplace-SLP, $n = 131.072$

Intel i7-13700		
# Cores	Runtime	Speedup
1	67.0s	
2	34.8s	1.93x
4	17.0s	3.94x

Numerical Results

Results for Laplace-SLP, $n = 131.072$

Intel i7-13700		
# Cores	Runtime	Speedup
1	67.0s	
2	34.8s	1.93x
4	17.0s	3.94x
8	8.4s	7.98x

Numerical Results

Results for Laplace-SLP, $n = 131.072$

Intel i7-13700		
# Cores	Runtime	Speedup
1	67.0s	
2	34.8s	1.93x
4	17.0s	3.94x
8	8.4s	7.98x
16	6.1s	10.98x

Numerical Results

Results for Laplace-SLP, $n = 131.072$

Intel i7-13700

# Cores	Runtime	Speedup
1	67.0s	
2	34.8s	1.93x
4	17.0s	3.94x
8	8.4s	7.98x
16	6.1s	10.98x
24	5.8s	11.55x

Numerical Results

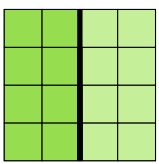
Results for Laplace-SLP, $n = 131.072$

Intel i7-13700

# Cores	Runtime	Speedup
1	67.0s	
2	34.8s	1.93x
4	17.0s	3.94x
8	8.4s	7.98x
16	6.1s	10.98x
24	5.8s	11.55x

Intel i7-13700

- 8 performance cores with max. 5.1 GHz,
- 8 efficiency cores with max. 4.1 GHz,



Numerical Results

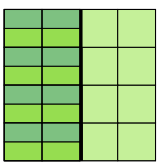
Results for Laplace-SLP, $n = 131.072$

Intel i7-13700

# Cores	Runtime	Speedup
1	67.0s	
2	34.8s	1.93x
4	17.0s	3.94x
8	8.4s	7.98x
16	6.1s	10.98x
24	5.8s	11.55x

Intel i7-13700

- 8 performance cores with max. 5.1 GHz and two hyperthreads,
- 8 efficiency cores with max. 4.1 GHz,



$\mathcal{H}$-Matrix-Vector Multiplication

$\mathcal{H}$-Matrix-Vector Multiplication

The standard $\mathcal{H}$-matrix-vector multiplication looks very similar to the construction algorithm:

function MATVEC($M_{\tau,\sigma}, x, y$)
  if $(\tau, \sigma) \in \mathcal{L}(T)$ **then**
    $y|_\tau := y|_\tau + M_{\tau,\sigma} x|_\sigma$
  else
    for all $(\tau', \sigma') \in \mathcal{S}((\tau, \sigma))$ **do**
      matvec($M_{\tau',\sigma'}, x, y$);

$\mathcal{H}$-Matrix-Vector Multiplication

The standard $\mathcal{H}$-matrix-vector multiplication looks very similar to the construction algorithm:

```
function MATVEC(M_{τ,σ}, x, y)
    if (τ, σ) ∈ L(T) then
        y|_τ := y|_τ + M_{τ,σ} x|_σ
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y);
```

Again, each sub block multiplication can be defined as a new task.

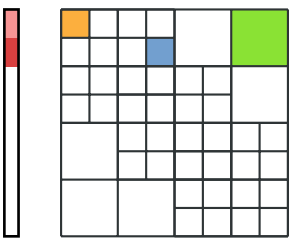
$\mathcal{H}$-Matrix-Vector Multiplication

The standard $\mathcal{H}$-matrix-vector multiplication looks very similar to the construction algorithm:

```
function MATVEC(M_{τ,σ}, x, y)
    if (τ, σ) ∈ L(T) then
        y|_τ := y|_τ + M_{τ,σ} x|_σ                    // critical section
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y);
```

Again, each sub block multiplication can be defined as a new task.

However, the update of y_τ forms a critical section as more than one task may update the same vector.



Solution 1

Add a mutex to guard the critical section.

	Intel i7-13700	
	Runtime	Speedup
sequential:	9.12s	

```
function MATVEC(M_{τ,σ}, x, y, mtx)
    if (τ, σ) ∈ L(T) then
        omp_set_lock( mtx );
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
        omp_unset_lock( mtx );
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y, mtx);
```

Solution 1

Add a mutex to guard the critical section.

	Intel i7-13700	
	Runtime	Speedup
sequential:	9.12s	
parallel:	59.60s	0.15x

```
function MATVEC(M_{τ,σ}, x, y, mtx)
    if (τ, σ) ∈ L(T) then
        omp_set_lock( mtx );
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
        omp_unset_lock( mtx );
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y, mtx);
```

$\mathcal{H}$-Matrix-Vector Multiplication

Solution 1

Add a mutex to guard the critical section.

	Intel i7-13700	
	Runtime	**Speedup**
sequential:	9.12s	
parallel:	59.60s	0.15x

```
function MATVEC(M_{τ,σ}, x, y, mtx)
    if (τ, σ) ∈ L(T) then
        omp_set_lock( mtx );
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
        omp_unset_lock( mtx );
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y, mtx);
```

Solution 2

Split local computation and update of y.

	i7-13700
sequential:	9.12s
parallel:	

```
function MATVEC(M_{τ,σ}, x, y, mtx)
    if (τ, σ) ∈ L(T) then
        t := M_{τ,σ} x|_σ;      // local matvec
        omp_set_lock( mtx );
        y|_τ := y|_τ + t;       // update
        omp_unset_lock( mtx );
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y, mtx);
```

$\mathcal{H}$-Matrix-Vector Multiplication

Solution 1

Add a mutex to guard the critical section.

	Intel i7-13700	
	Runtime	**Speedup**
sequential:	9.12s	
parallel:	59.60s	0.15x

```
function MATVEC(M_{τ,σ}, x, y, mtx)
    if (τ, σ) ∈ L(T) then
        omp_set_lock( mtx );
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
        omp_unset_lock( mtx );
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y, mtx);
```

Solution 2

Split local computation and update of y.

	i7-13700
sequential:	9.12s
parallel:	3.41s / 2.7x

```
function MATVEC(M_{τ,σ}, x, y, mtx)
    if (τ, σ) ∈ L(T) then
        t := M_{τ,σ} x|_σ;      // local matvec
        omp_set_lock( mtx );
        y|_τ := y|_τ + t;       // update
        omp_unset_lock( mtx );
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y, mtx);
```

$\mathcal{H}$-Matrix-Vector Multiplication

Solution 1

Add a mutex to guard the critical section.

	Intel i7-13700	
	Runtime	**Speedup**
sequential:	9.12s	
parallel:	59.60s	0.15x

```
function MATVEC(M_{τ,σ}, x, y, mtx)
    if (τ, σ) ∈ L(T) then
        omp_set_lock( mtx );
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
        omp_unset_lock( mtx );
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y, mtx);
```

Solution 2

Split local computation and update of y.

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:	3.41s / 2.7x	5.90s / 2.5x

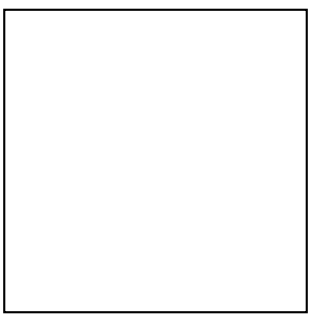
```
function MATVEC(M_{τ,σ}, x, y, mtx)
    if (τ, σ) ∈ L(T) then
        t := M_{τ,σ} x|_σ;        // local matvec
        omp_set_lock( mtx );
        y|_τ := y|_τ + t;         // update
        omp_unset_lock( mtx );
    else
        for all (τ', σ') ∈ S((τ, σ)) do
            #pragma omp task
                matvec(M_{τ',σ'}, x, y, mtx);
```

Solution 3

Lock-free approach by block-row oriented recursion.

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:		

```
function MATVEC(M_{τ,σ}, x, y)
    if (τ, σ) ∈ L(T) then
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    for all σ' ∈ S(σ) do
                        matvec(M_{τ',σ'}, x, y);
```

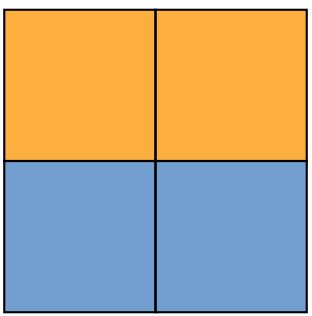


Solution 3

Lock-free approach by block-row oriented recursion.

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:		

```
function MATVEC(M_{τ,σ}, x, y)
    if (τ, σ) ∈ L(T) then
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    for all σ' ∈ S(σ) do
                        matvec(M_{τ',σ'}, x, y);
```

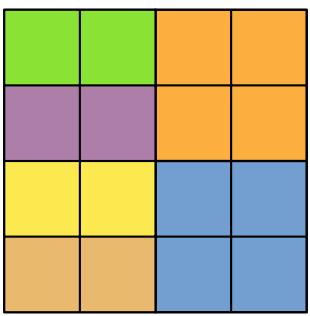


Solution 3

Lock-free approach by block-row oriented recursion.

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:		

```
function MATVEC(M_{τ,σ}, x, y)
    if (τ, σ) ∈ L(T) then
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    for all σ' ∈ S(σ) do
                        matvec(M_{τ',σ'}, x, y);
```

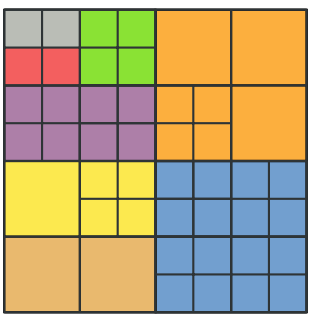


Solution 3

Lock-free approach by block-row oriented recursion.

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:		

```
function MATVEC(M_{τ,σ}, x, y)
    if (τ, σ) ∈ L(T) then
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    for all σ' ∈ S(σ) do
                        matvec(M_{τ',σ'}, x, y);
```

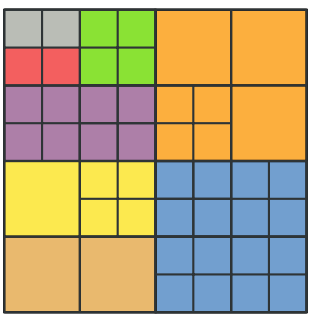


Solution 3

Lock-free approach by block-row oriented recursion.

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:	3.62s / 2.5x	0.98s / 15.2x

```
function MATVEC(M_{τ,σ}, x, y)
    if (τ, σ) ∈ L(T) then
        y|_τ := y|_τ + M_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    for all σ' ∈ S(σ) do
                        matvec(M_{τ',σ'}, x, y);
```

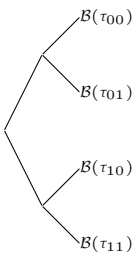


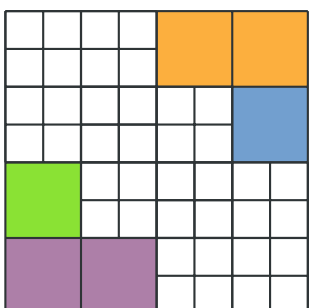
Solution 3 (optimised)

Keep list of matrix blocks per row cluster an handle all such blocks in single task..

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:		

```
function MATVEC(τ, B, x, y)
    if B(τ) ≠ ∅ then
        for all M' ∈ B(τ) do
            y|_τ := y|_τ + M'_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    matvec(τ', B, x, y);
```



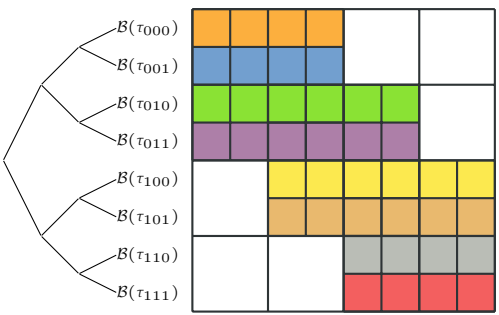


Solution 3 (optimised)

Keep list of matrix blocks per row cluster an handle all such blocks in single task..

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:		

```
function MATVEC(τ, B, x, y)
    if B(τ) ≠ ∅ then
        for all M' ∈ B(τ) do
            y|_τ := y|_τ + M'_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    matvec(τ', B, x, y);
```

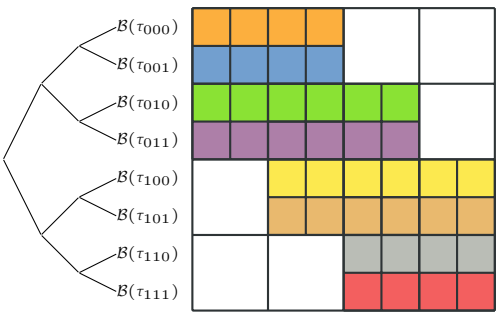


Solution 3 (optimised)

Keep list of matrix blocks per row cluster an handle all such blocks in single task..

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:	3.45s / 2.6x	0.56s / 26.6x

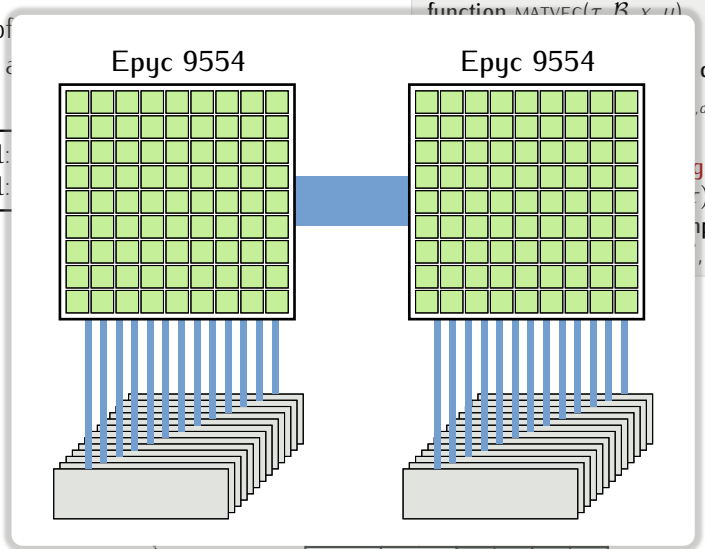
```
function MATVEC(τ, B, x, y)
    if B(τ) ≠ ∅ then
        for all M' ∈ B(τ) do
            y|_τ := y|_τ + M'_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    matvec(τ', B, x, y);
```



Solution 3 (optimised)

Keep list of
an handle a

sequential:
parallel:



function MATVEC(τ, $\mathcal{B}$, x, y)

do
$_\sigma x|_\sigma$;

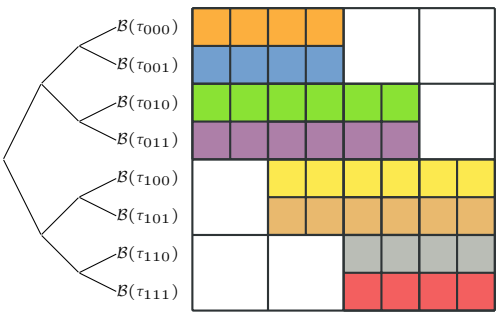
group
τ) do
np task
$\mathcal{B}, x, y$);

Solution 3 (optimised)

Keep list of matrix blocks per row cluster an handle all such blocks in single task..

	i7-13700	2x Epyc 9554
sequential:	9.12s	14.89s
parallel:	3.45s / 2.6x	0.56s / 26.6x

```
function MATVEC(τ, B, x, y)
    if B(τ) ≠ ∅ then
        for all M' ∈ B(τ) do
            y|_τ := y|_τ + M'_{τ,σ} x|_σ;
    else
        #pragma omp taskgroup
            for all τ' ∈ S(τ) do
                #pragma omp task
                    matvec(τ', B, x, y);
```



Implementation

Sequential multiplication in `hmatrix.c`:

```
void fastaddeval_hmatrix_avector ( field alpha, pchmatrix hm, pcavector x, pavector y )
{
  if      (hm->r) { addeval_rkmatrix_avector(alpha, hm->r, x, y); }
  else if (hm->f) { mvm_amatrix_avector(alpha, false, hm->f, x, y); }
  else {
    uint  rsons = hm->rsons;
    uint  csons = hm->csons;
    uint  xoff  = 0;

    for ( uint j = 0; j < csons; j++) {
      avector   xtmp;
      pavector  x1   = init_sub_avector(&xtmp, (pavector) x, hm->son[j * rsons]->cc->size, xoff);
      uint      yoff = 0;

      for (uint i = 0; i < rsons; i++) {
        avector   ytmp;
        pavector  y1    = init_sub_avector(&ytmp, y, hm->son[i]->rc->size, yoff);

        fastaddeval_hmatrix_avector(alpha, hm->son[i + j * rsons], x1, y1);
        uninit_avector(y1);
        yoff += hm->son[i]->rc->size;
      }

      uninit_avector(x1);
      xoff += hm->son[j * rsons]->cc->size;
} } }
```

Implementation

Guard leaf block multiplication by mutex:

```
void fastaddeval_hmatrix_avector_omp ( field alpha, ..., pavector y, omp_lock_t * mtx )
{
  if ( hm->r || hm->f ) {
    omp_set_lock( mtx );
    if ( hm->r ) addeval_rkmatrix_avector(alpha, hm->r, x, y);
    else         mvm_amatrix_avector(alpha, false, hm->f, x, y);
    omp_unset_lock( mtx );
  } else {
    ...
    for ( uint j = 0; j < csons; j++) {
      ...
      for (uint i = 0; i < rsons; i++) {
        ...
        fastaddeval_hmatrix_avector(alpha, hm->son[i + j * rsons], x1, y1, mtx);
    ...
} } } }
```

Implementation

Split computation and application of local update:

```
void fastaddeval_hmatrix_avector_omp ( field alpha, ..., pavector y, omp_lock_t * mtx )
{
  if ( hm->r || hm->f ) {
    pavector t = new_avector( y->dim );
    clear_avector( t );
    if ( hm->r ) addeval_rkmatrix_avector(alpha, hm->r, x, t);
    else         mvm_amatrix_avector(alpha, false, hm->f, x, t);

    omp_set_lock( mtx );
    add_avector( 1.0, t, y );
    omp_unset_lock( mtx );
    del_avector( t );
  } else {
    ...
    for ( uint j = 0; j < csons; j++) {
      ...
      for (uint i = 0; i < rsons; i++) {
        ...
        fastaddeval_hmatrix_avector(alpha, hm->son[i + j * rsons], x1, y1, mtx);
    ...
} } } }
```

Implementation

Define tasks:

```
void fastaddeval_hmatrix_avector_omp ( field alpha, ..., pavector y, omp_lock_t * mtx )
{ ...
  } else {
    uint  rsons = hm->rsons;
    uint  csons = hm->csons;
    uint  xoff  = 0;


    for ( uint j = 0; j < csons; j++) {
      avector   xtmp;
      pavector  x1   = init_sub_avector(&xtmp, (pavector) x, hm->son[j * rsons]->cc->size, xoff);
      uint      yoff = 0;

      for (uint i = 0; i < rsons; i++) {
        #pragma omp task
        {
          avector   ytmp;
          pavector  y1 = init_sub_avector(&ytmp, y, hm->son[i]->rc->size, yoff);

          fastaddeval_hmatrix_avector(alpha, hm->son[i + j * rsons], x1, y1, mtx);
          uninit_avector(y1);
          yoff += hm->son[i]->rc->size;
        }

      }
      uninit_avector(x1);
      xoff += hm->son[j * rsons]->cc->size;
} } }
```

Implementation

Define tasks:

```
void fastaddeval_hmatrix_avector_omp ( field alpha, ..., pavector y, omp_lock_t * mtx )
{ ...
  } else {
    uint  rsons = hm->rsons;
    uint  csons = hm->csons;
    uint  xoff  = 0;


    for ( uint j = 0; j < csons; j++) {
      avector   xtmp;
      pavector  x1   = init_sub_avector(&xtmp, (pavector) x, hm->son[j * rsons]->cc->size, xoff);
      uint      yoff = 0;

      for (uint i = 0; i < rsons; i++) {
        #pragma omp task shared(alpha,y,mtx,rsons,csons,xoff,xtmp,x1,yoff,j,i)
        {
          avector   ytmp;
          pavector  y1 = init_sub_avector(&ytmp, y, hm->son[i]->rc->size, yoff);

          fastaddeval_hmatrix_avector(alpha, hm->son[i + j * rsons], x1, y1, mtx);
          uninit_avector(y1);
          yoff += hm->son[i]->rc->size;
        }

      }
      uninit_avector(x1);
      xoff += hm->son[j * rsons]->cc->size;
} } }
```

Implementation

Define tasks:

```
void fastaddeval_hmatrix_avector_omp ( field alpha, ..., pavector y, omp_lock_t * mtx )
{ ...
  } else {
    uint  rsons = hm->rsons;
    uint  csons = hm->csons;
    uint  xoff  = 0;


    for ( uint j = 0; j < csons; j++) {
      avector   xtmp;
      pavector  x1   = init_sub_avector(&xtmp, (pavector) x, hm->son[j * rsons]->cc->size, xoff);
      uint      yoff = 0;

      for (uint i = 0; i < rsons; i++) {
        #pragma omp task firstprivate(j,i,xoff,yoff,rsons,csons,alpha,y)
        {
          avector   ytmp;
          pavector  y1 = init_sub_avector(&ytmp, y, hm->son[i]->rc->size, yoff);

          fastaddeval_hmatrix_avector(alpha, hm->son[i + j * rsons], x1, y1, mtx);
          uninit_avector(y1);

        }
        yoff += hm->son[i]->rc->size;
      }
      uninit_avector(x1);
      xoff += hm->son[j * rsons]->cc->size;
} } }
```

Implementation

Define tasks:

```
void fastaddeval_hmatrix_avector_omp ( field alpha, ..., pavector y, omp_lock_t * mtx )
{ ...
  } else {
    uint  rsons = hm->rsons;
    uint  csons = hm->csons;
    uint  xoff  = 0;

    #pragma omp taskgroup
    for ( uint j = 0; j < csons; j++) {
      avector   xtmp;
      pavector  x1   = init_sub_avector(&xtmp, (pavector) x, hm->son[j * rsons]->cc->size, xoff);
      uint      yoff = 0;

      for (uint i = 0; i < rsons; i++) {
        #pragma omp task firstprivate(j,i,xoff,yoff)
        {
          avector   ytmp;
          pavector  y1 = init_sub_avector(&ytmp, y, hm->son[i]->rc->size, yoff);

          fastaddeval_hmatrix_avector(alpha, hm->son[i + j * rsons], x1, y1, mtx);
          uninit_avector(y1);

        }
        yoff += hm->son[i]->rc->size;
      }
      uninit_avector(x1);
      xoff += hm->son[j * rsons]->cc->size;
} } }
```

Implementation

Task private data:

```
void fastaddeval_hmatrix_avector_omp ( field alpha, ..., pavector y, omp_lock_t * mtx )
{
  ...
  } else {
    uint    rsons = hm->rsons;
    uint    csons = hm->csons;
    uint *  rofs  = allocuint( rsons );
    uint *  cofs  = allocuint( csons );

    for ( uint j = 0; j < csons; j++ ) { cofs[j] = xofs; xofs += hm->son[j * rsons]->cc->size; }
    for ( uint i = 0; i < rsons; i++ ) { rofs[i] = yofs; yofs += hm->son[i]->rc->size; }

    #pragma omp taskgroup
    {
      #pragma omp taskloop collapse(2)
      for ( uint j = 0; j < csons; j++) {
        for ( uint i = 0; i < rsons; i++) {
          avector   xtmp, ytmp;
          pavector  x1 = init_sub_avector( &xtmp, (pavector) x, hm->son[j * rsons]->cc->size, cofs[j] );
          pavector  y1 = init_sub_avector( &ytmp, y, hm->son[i]->rc->size, rofs[i] );

          fastaddeval_hmatrix_avector_omp( alpha, hm->son[i + j * rsons], x1, y1, mtx );

          uninit_avector( y1 );
          uninit_avector( x1 );
} } } } }
```

$\mathcal{H}$-Matrix Multiplication

$\mathcal{H}$-Matrix Multiplication

We consider the general update

$$C := C + \alpha A \cdot B$$

A full parallel approach with tasks for *each* sub-block multiplication looks as

function MUL($\alpha, A_{\tau,\rho}, B_{\rho,\sigma}, C_{\tau,\sigma}$)
  if $A_{\tau,\rho}, B_{\rho,\sigma}, C_{\tau,\sigma}$ are block matrices **then**
    #pragma omp taskloop collapse(3) firstprivate($\alpha, A_{\tau,\rho}, B_{\rho,\sigma}, C_{\tau,\sigma}$)
    for $\tau' \in \mathcal{S}(\tau)$ **do**
      for $\sigma' \in \mathcal{S}(\sigma)$ **do**
        for $\rho' \in \mathcal{S}(\rho)$ **do**
          `mul(` $\alpha, A_{\tau',\rho'}, B_{\rho',\sigma'}, C_{\tau',\sigma'}$ `);`
  else
    $C_{\tau,\sigma} := C_{\tau,\sigma} + \alpha A_{\tau,\rho} B_{\rho,\sigma}$; // *critical section*

$\mathcal{H}$-Matrix Multiplication

We consider the general update

$$C := C + \alpha A \cdot B$$

A full parallel approach with tasks for *each* sub-block multiplication looks as

```
function MUL(α, A_{τ,ρ}, B_{ρ,σ}, C_{τ,σ})
    if A_{τ,ρ}, B_{ρ,σ}, C_{τ,σ} are block matrices then
        #pragma omp taskloop collapse(3) firstprivate(α, A_{τ,ρ}, B_{ρ,σ}, C_{τ,σ})
        for τ' ∈ S(τ) do
            for σ' ∈ S(σ) do
                for ρ' ∈ S(ρ) do
                    mul( α, A_{τ',ρ'}, B_{ρ',σ'}, C_{τ',σ'} );
    else
        omp_set_lock(mutex(C_{τ,σ}));
        C_{τ,σ} := C_{τ,σ} + αA_{τ,ρ}B_{ρ,σ};          // critical section
        omp_unset_lock(mutex(C_{τ,σ}));
```

Each matrix block has an associated mutex.

$\mathcal{H}$-Matrix Multiplication

C++ implementation with OpenMP:

```
void multiply_task ( alpha, op_A, A, op_B, B, C, acc, approx ) {
  if ( is_blocked_all( A, B, C ) ) {
    auto  BA, BB, BC = ... ;

    #pragma omp taskloop collapse(3) firstprivate(alpha,op_A,op_B)
    for ( uint  i = 0; i < BC->nblock_rows(); ++i ) {
      for ( uint  j = 0; j < BC->nblock_cols(); ++j ) {
        for ( uint  l = 0; l < BA->nblock_cols( op_A ); ++l ) {
          auto  C_ij = BC->block(i,j);
          auto  A_il = BA->block( i, l, op_A );
          auto  B_lj = BB->block( l, j, op_B );

          multiply_task< value_t >( alpha, op_A, *A_il, op_B, *B_lj, *C_ij, acc, approx );
  } } } }
  else
    multiply( alpha, op_A, A, op_B, B, C, acc, approx );
}
```

```
void multiply ( ..., lrmatrix< value_t > &  A, lrmatrix< value_t > &  B, dense_matrix< value_t > & C ) {
  auto  T  = blas::prod( value_t(1), blas::adjoint( A.V( op_A ) ), B.U( op_B ) );
  auto  UT = blas::prod( value_t(1), A.U( op_A ), T );

  std::scoped_lock  lock( C.mutex() );
  auto              DC = C.mat();

  blas::prod( alpha, UT, blas::adjoint( B.V( op_B ) ), value_t(1), C.mat() );
}
```

Results for Laplace-SLP, $n = 131.072$

Intel i7-13700

# Cores	Runtime	Speedup
1	145.9s	
2	76.8s	1.90x
4	38.7s	3.77x
8	21.6s	6.75x
16	19.4s	7.52x
24	17.4s	8.38x

Results for Laplace-SLP, $n = 131.072$

Intel i7-13700

# Cores	Runtime	Speedup
1	145.9s	
2	76.8s	1.90x
4	38.7s	3.77x
8	21.6s	6.75x
16	19.4s	7.52x
24	17.4s	8.38x

AMD Epyc 9554

# Cores	Runtime	Speedup
1	202.80s	
64	3.54s	57.29x
128	1.88s	107.87x

Literature

Literature

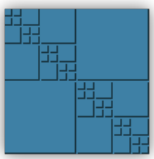
S. Börm, S. Christophersen and R. Kriemann:
Semi-automatic task graph construction for H-matrix arithmetic,
SIAM Journal on Scientific Computing, Volume 44(2), pp. 77–98, 2022.

R. Kriemann,
$\mathcal{H}$-LU Factorization on Many-Core Systems,
Computing and Visualization in Science, 16, pp. 105–117, 2015.

L. Grasedyck, R. Kriemann and S. Le Borne:
Parallel blackbox H-LU preconditioning for elliptic boundary value problems,
Computing and Visualization in Sciences, Vol. 11, pp 273-291, 2007.

R. Kriemann,
Parallel $\mathcal{H}$-Matrix Arithmetics on Shared Memory Systems,
Computing, 74:273–297, 2005.

hlibpro.com



libHLR.org