Prof. Dr. D. Kranzlmüller, Dr. K. Fürlinger

Parallel Computing

WS 2017/18

Session 4: Loop Tiling, MPI

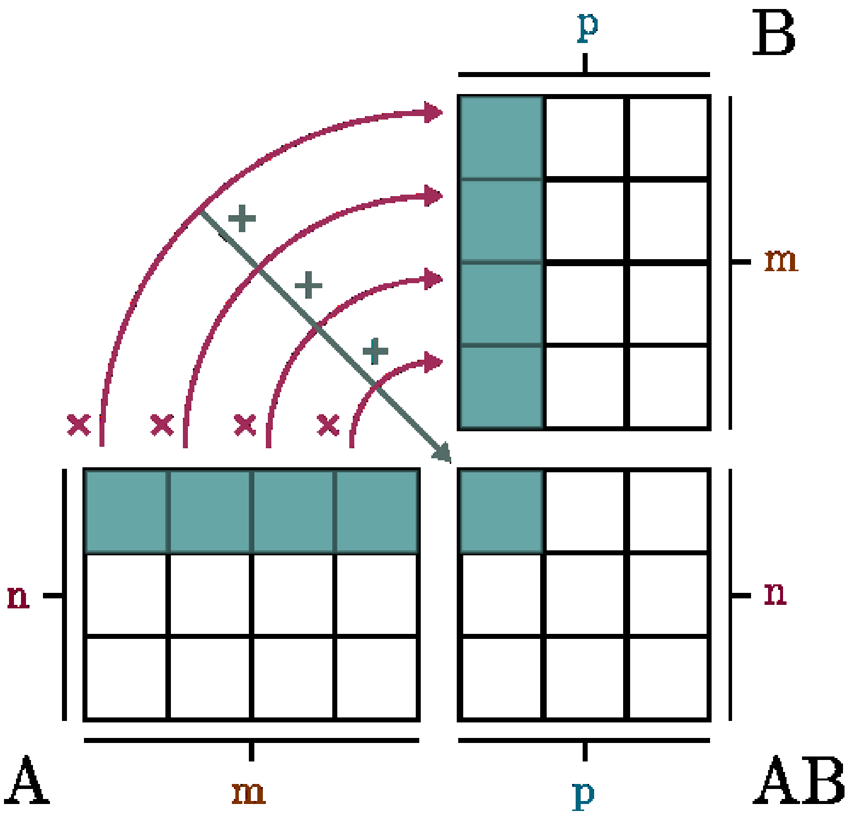
Tobias Fuchs, M.Sc.
tobias.fuchs@nm.ifi.lmu.de

Discussion:

DGEMM Optimization

Recap: Matrix Product

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$



Notation in summation convention:

$$(AB)_{ij} = A_{ik} B_{kj}$$

Matrix Product – Naïve Implementation

```
for (index i = 0; i < n; i++) {
    for (index j = 0; j < p; j++) {
        double sum = 0.0;
        for (index k = 0; k < m; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

How about loop unrolling?

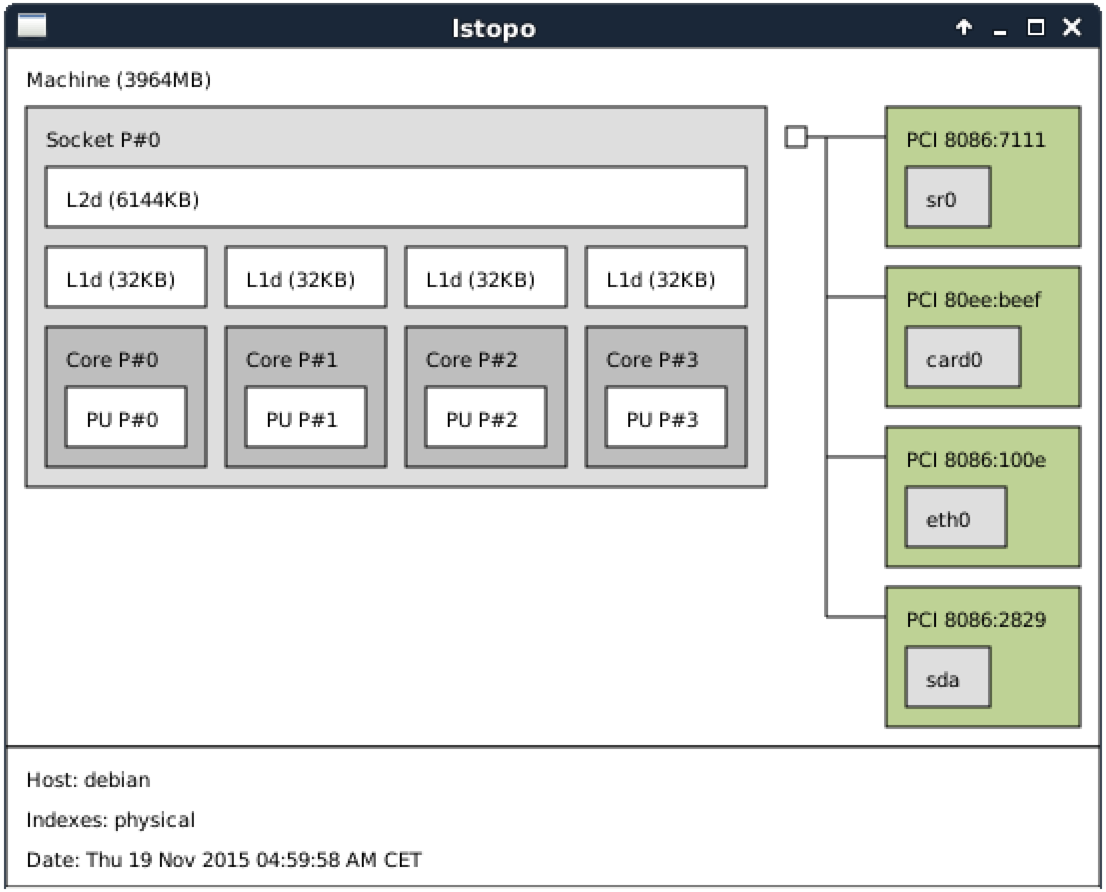
Discussion: DGEMM Optimization

Did you make yourselves familiar with blocking / tiling?

Principle idea:

- Find out the CPU cache size.

Discussion: DGEMM Optimization



Discussion: DGEMM Optimization

Did you make yourselves familiar with blocking / tiling?

Principle idea:

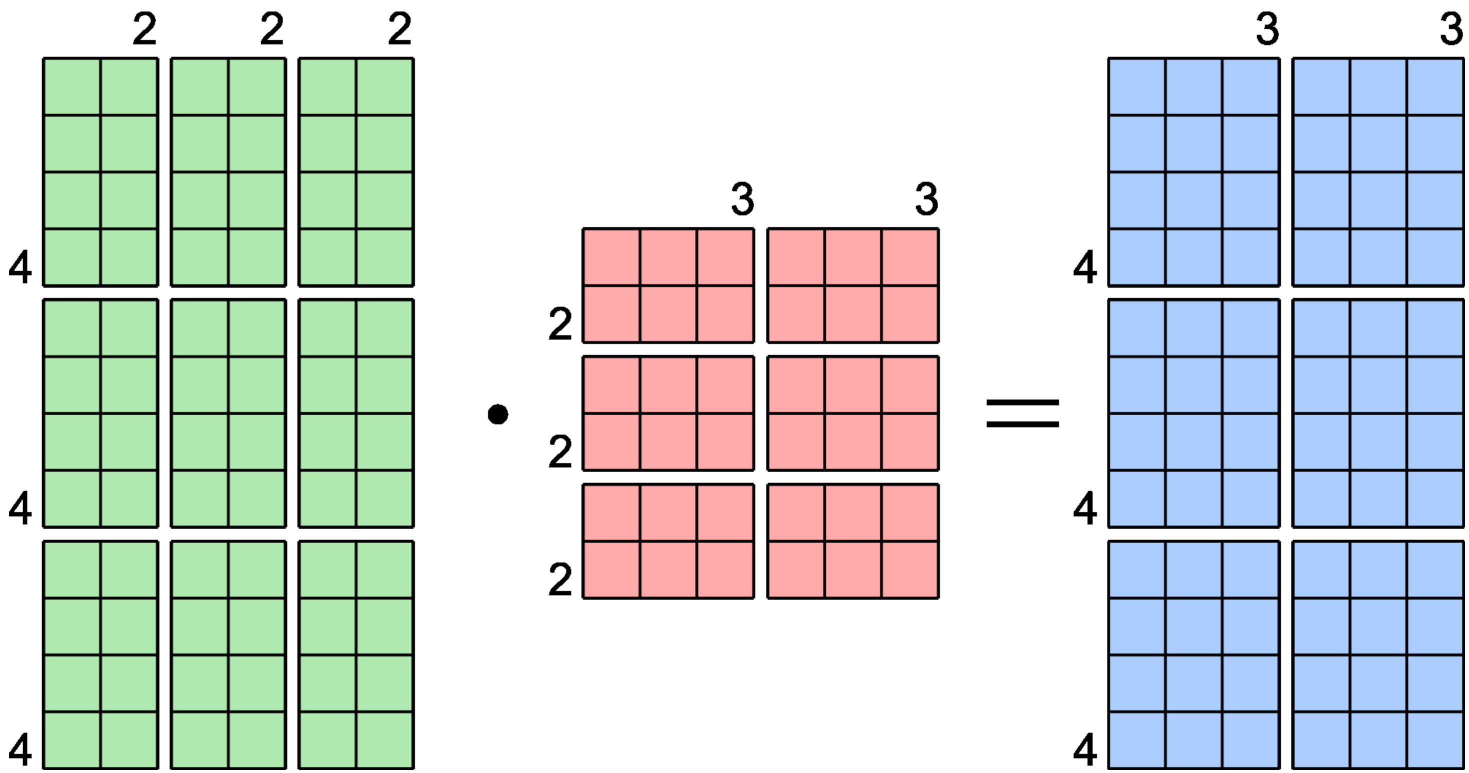
- Find out the CPU cache size.
- Resolve maximum matrix extents $\mathrm{b} \times \mathrm{b}$ so sub-matrices in A, B, C fit into cache, i.e. solve:

  $3(\mathrm{b}^2) \cdot \mathrm{E} = \mathrm{M}$ for cache size M and element size E

- Partition nested for-loops into partial sub-matrix product solutions.

Discussion: DGEMM Optimization

Block matrix multiplication:



Discussion: DGEMM Optimization


```
// Outer loops: Iterate in b x b block steps
for (i0 = 0; i0 < n; i0 += b):
    for (j0 = 0; j0 < n; j0 += b):
        for (k0 = 0; k0 < n; k0 += b):
            // Inner loop: Matrix product of single block
            // -> 2b^3 operations on 3b^2 elements
            for (i = i0; i < min(i0+b, n); i++):
                for (j = j0; j < min(j0+b, n); j++):
                    double sum = C[i][j];
                    for (k = k0; k < min(k0+b, n); k++):
                        sum += A[i][k] * B[k][j];
                    C[i][j] = sum;
```

- Same number of operations, identical result, same round-off errors
- Cache misses: $n^3 / (m \cdot b)$ before: n^3 ($m \cdot b$ is significantly large)

Strassen's algorithm (Volker Strassen, 1969)

- Divide-and-Conquer algorithm
- Multiplying two 2 × 2-matrices using 7 multiplications (instead of 8) with additional add and sub operations
  → Sub-cubic complexity
- **Cache-oblivious**
- But: **numeric stability is inferior** to sequential algorithm
- Used in OpenBLAS in some cases (e.g. finite fields)
- **Complex implementation!**
  (See me after this session if you want to go for it)

Recap: Common MPI Functions

Further Reading

The one MPI tutorial you all want to read:

Basics: https://cvw.cac.cornell.edu/MPI/
P2P: https://cvw.cac.cornell.edu/MPIP2P/
RMA: https://cvw.cac.cornell.edu/MPIoneSided/
Advanced: https://cvw.cac.cornell.edu/MPIAdvTopics/

Official MPI 3.1 documentation (Index):
http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm#Node0

Again, a collection of documented MPI examples:
http://www.mcs.anl.gov/~thakur/sc14-mpi-tutorial/

MPI Recap


```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char * argv[]) {
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("I am %d of %d\n", rank, size );

    MPI_Finalize();
    return 0;
}
```

Point-to-Point vs. Collective

Collective operations are called by all processes in a communicator.

Point-to-point operations are called at sender and receiver process.

Point-to-Point Operations

Blocking	Nonblocking	Somewhat-blocking
`MPI_Send`	`MPI_Isend`	`MPI_Ssend`
`MPI_Recv`	`MPI_Irecv`	`MPI_Bsend`
		`MPI_Sendrecv`

- Transfer of a message from one specific process to another specific process in the communicator.
- Requires action from both the sending and receiving processes.

MPI_Send / MPI_Recv (P2P)

```
MPI_Send(buffer, count, dtype, dst_rank, tag, comm)
MPI_Recv(buffer, count, dtype, src_rank, tag, comm, status)
```

Send from process with rank `src_rank` to process with rank `dst_rank`.

At send process:

```
// buffer out initialized with values before call
MPI_Send(&out, 1, MPI_CHAR, dst_rnk, tag, MPI_COMM_WORLD);
```

At receive processes:

```
MPI_Recv(&in,  1, MPI_CHAR, src_rnk, tag, MPI_COMM_WORLD, &sta);
// buffer in filled with values after call returns
MPI_Get_count(&sta, MPI_CHAR, &count);
```

Collective Operations

Rooted	All-to-all	Other
`MPI_Gather/v`	`MPI_Allgather/v`	`MPI_Scan`
`MPI_Reduce`	`MPI_Alltoall/v/w`	`MPI_Exscan`
`MPI_Scatter/v`	`MPI_Allreduce`	`MPI_Barrier`
`MPI_Bcast`	`MPI_Reduce_scatter`	

- Collective communication routines must involve **all** processes within the scope of a communicator.
- All processes are members in the communicator `MPI_COMM_WORLD` by default.

MPI_Bcast

```
MPI_Bcast(buffer, count, dtype, root, comm)
```

Broadcast from root process to all processes in leaf group.

At root process:

```
// buffer initialized with values before call
MPI_Bcast(buffer, count, root_rank, comm)
```

At leaf processes:

```
MPI_Bcast(buffer, count, root_rank, comm)
// buffer filled with values after call returns
```

Collective Operations

Rooted	All-to-all	Other
`MPI_Gather/v`	`MPI_Allgather/v`	`MPI_Scan`
`MPI_Reduce`	`MPI_Alltoall/v/w`	`MPI_Exscan`
`MPI_Scatter/v`	`MPI_Allreduce`	`MPI_Barrier`
`MPI_Bcast`	`MPI_Reduce_scatter`	

"Messages are ***non-overtaking****: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending."*

MPI_Reduce

```
MPI_Reduce(sbuffer, rbuffer, count, dtype, op, root, comm)
```

Reduce values in sbuffer at processes in leaf group into rbuffer at root process.

At root process:

```
MPI_Reduce(sbuf, rbuf, ..., MPI_SUM, root_rank, comm)
// rbuf filled with values after call returns
```

At leaf processes:

```
// sbuf initialized with values before call
MPI_Reduce(sbuf, rbuf, ..., MPI_SUM, root_rank, comm)
```

MPI_Scatter

```
MPI_Scatter(sbuf, scount, stype,
            rbuf, rcount, rtype, root, comm)
```

Scatter data **in rank order** from root process into receive buffers at leaf processes (i.e. inverse of `MPI_Gather`).

At root process:

```
// sbuf filled before call, receive arguments are ignored
MPI_Scatter(sbuf, scount, ... , root_rank, comm)
```

At leaf processes:

```
// send arguments are ignored
MPI_Scatter(sbuf, scount, ... , root_rank, comm)
```

MPI_Gather

```
MPI_Gather(sbuf, scount, stype,
           rbuf, rcount, rtype, root, comm)
```

Gather data **in rank order** from leaf processes into receive buffer at root process.

At root process:

```
// send arguments are ignored
MPI_Gather(sbuf, scount, ... , root_rank, comm)
```

At leaf processes:

```
// sbuf filled before call, receive arguments are ignored
MPI_Gather(sbuf, scount, ... , root_rank, comm)
```

Blocking vs. Non-blocking

Blocking vs. Non-blocking

Blocking:

Process *waits* to ensure the message data have achieved a particular state before processing can continue.

Non-blocking:

Process merely requests to start an operation and *continues processing*.
Allows overlap of communication and communication.

Blocking, two-sided

```
// blocks until received:
MPI_Recv(...);
```

Non-blocking, two-sided

```
MPI_Request req;
MPI_Irecv(..., &req);
some_local_computation();
// blocks until received:
MPI_Wait(req, &status);
```

Two-sided vs. One-sided

Two-sided

- **Memory is private** to each process.
- When the sender calls the MPI_Send operation and the receiver calls the MPI_Recv operation, data in the sender memory is copied to a buffer then sent over the network, where it is copied to the receiver memory.
- **Drawback:** sender has to wait for the receiver to be ready to receive the data before it can send the data.
- Both sender and receiver have to state a specific call for the communication.

→ **Coupled program flow**

One-sided

- Sections in local memory are made **accessible among processes**.
- Requires **only one process to transfer data**, decouples data transfer from system synchronization.
- **MPI 3.0** supports one-sided passive target communication **without the intervention of the remote process** via Remote Direct Memory Access (RDMA).
  That is: send or receive data without any local action.

→ **Decoupled program flow**

One-sided

- Sections in local memory are made **accessible among processes**.
- Requires **only one process to transfer data**, decouples data transfer from system synchronization.
- **MPI 3.0** supports one-sided passive target communication **without the intervention of the remote process** via Remote Direct Memory Access (RDMA).
  That is: send or receive data without any local action.

→ **Decoupled program flow**

Hard to exploit in the real world

In the real world, passive RDMA in most MPI implementations is either buggy, or inefficient (barrier spin-locks and other delightful hacks) or both, but it's getting better.

One-sided Operations

Standard	Request-based
`MPI_Put`	`MPI_Rput`
`MPI_Get`	`MPI_Rget`
`MPI_Accumulate`	`MPI_Raccumulate`

- All data movement operations are **non-blocking**.
- **Requires explicit synchronization call** to ensure completion, e.g.:

```
MPI_Wait(req)
MPI_Win_fence(win)
MPI_Win_flush(win)
...
```

MPI_Get / MPI_Put

Origin: calling (i.e. local) process
Target: remote process

```
MPI_Get(oaddr, ocount, otype,
        trank, tdisp, tcount, ttype, window)
```

Transfer elements from target in `window[tdisp:tcount]` into local buffer oaddr at origin.

```
MPI_Put(oaddr, ocount, otype,
        trank, tdisp, tcount, ttype, window)
```

Transfer elements from local buffer oaddr at origin to target into `window[tdisp:tcount]`.

(Blocking / Nonblocking) x (One-sided x Two-sided)

	Blocking	**Nonblocking**
Two-sided	MPI_Send	MPI_Isend
One-sided	MPI_Put	MPI_Rput

One-sided communication can be used to implement collective operations.

Pop quiz: How would you implement a reduce operation using one-sided communication?

Example: **Find minimum value in distributed array**.

One-sided true passive RMA Example

```
MPI_Comm comm = MPI_COMM_WORLD;
// Create local window buffer of 4096 integers:
int *    winbuf;
MPI_Alloc_mem(sizeof(int)*4096, MPI_INFO_NULL, &winbuf);
// Create window, collective operation:
MPI_Win  win;
MPI_Win_create(winbuf, sizeof(int)*4096, sizeof(int),
               MPI_WIN_INFO_NULL, comm, &win);
// Start passive RMA epoch, collective operation:
MPI_Win_lock_all(0, win);

...
```

One-sided true passive RMA Example ctd.

```
// rank_0.buf[0:99] <- rank_1.win[1024:1123]

rank 1:                             |   rank 0:
------------------------------------+------------------------------------
for (int i=0; i<128; i++)           |   int buf[100];
  winbuf[i+1024] = 42;              |   MPI_Get(buf, 100, MPI_INT,
}                                   |           1, 1024, comm);
MPI_Win_flush_all(win);             |   MPI_Win_flush(1, win);
```

What could go wrong?

One-sided true passive RMA Example ctd.

```
// rank_0.buf[0:99] <- rank_1.win[1024:1123]

rank 1:                         |   rank 0:
--------------------------------+--------------------------------
for (int i=0; i<128; i++)       |   int buf[100];
  winbuf[i+1024] = 42;          |   MPI_Get(buf, 100, MPI_INT,
}                               |           1, 1024, comm);
MPI_Win_flush_all(win);         |   MPI_Win_flush(1, win);
```

What could go wrong?

One-sided true passive RMA Example ctd.

```
// rank_0.buf[0:99] <- rank_1.win[1024:1123]

rank 1:                           |   rank 0:
----------------------------------+-------------------------------
for (int i=0; i<128; i++)         |   int buf[100];
  winbuf[i+1024] = 42;            |   MPI_Get(buf, 100, MPI_INT,
}                                 |           1, 1024, comm);
MPI_Win_flush_all(win);           |   MPI_Win_flush(1, win);
```

Data races, as known from multi-threading.

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Tobias Fuchs

tobias.fuchs@nm.ifi.lmu.de

www.mnm-team.org/~fuchst