Erlangen Regional Computing Center

UNIVERSITÄT GREIFSWALD
Wissen lockt. Seit 1456

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Winter term 2020/2021

# Parallel Programming with OpenMP and MPI


Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald


## Lecture 7: ccNUMA and wavefront parallelization with OpenMP

High Performance Computing

# Outline of course

- Basics of parallel computer architecture
- Basics of parallel computing
- Introduction to shared-memory programming with OpenMP
- OpenMP performance issues
- Introduction to the Message Passing Interface (MPI)
- Advanced MPI
- MPI performance issues
- Hybrid MPI+OpenMP programming

Erlangen Regional Computing Center

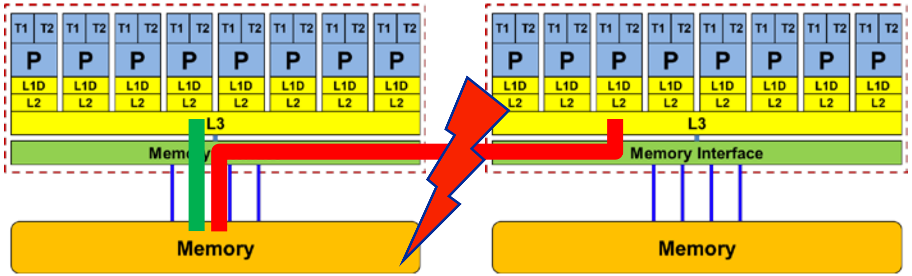
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

# Efficient programming of ccNUMA nodes

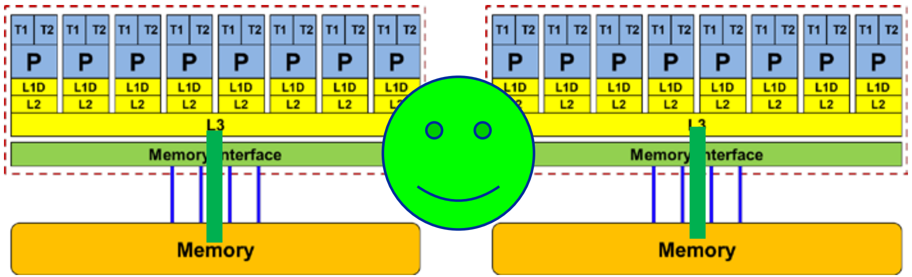
High Performance Computing

# ccNUMA – The other affinity to care about

- ccNUMA:
  - Whole memory is transparently accessible by all cores
  - but physically distributed across multiple locality domains (LDs)
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)
- How do we make sure that memory access is always as "local" and "distributed" as possible?



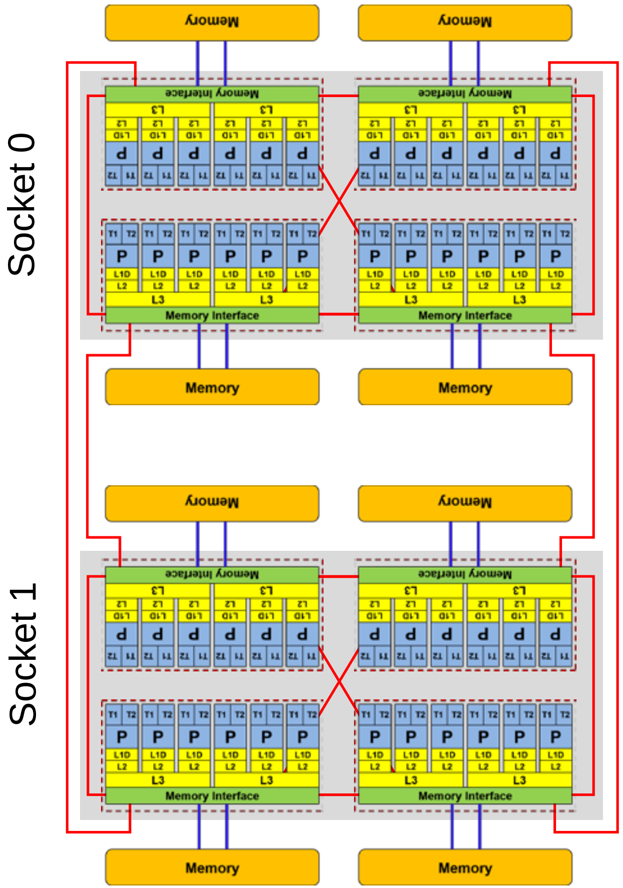

Note: Page placement is implemented in units of OS pages (often 4kB, possibly more)



# How much does nonlocal access cost?

Example: AMD “Naples” (Zen) 2-socket system
(8 chips, 2 sockets, 48 cores):
STREAM Triad bandwidth measurements [Gbyte/s]

| MEM node \ CPU node | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 32.4 | 21.4 | 21.8 | 21.9 | 10.6 | 10.6 | 10.7 | 10.8 |
| 1 | 21.5 | 32.4 | 21.9 | 21.9 | 10.6 | 10.5 | 10.7 | 10.6 |
| 2 | 21.8 | 21.9 | 32.4 | 21.5 | 10.6 | 10.6 | 10.8 | 10.7 |
| 3 | 21.9 | 21.9 | 21.5 | 32.4 | 10.6 | 10.6 | 10.6 | 10.7 |
| 4 | 10.6 | 10.7 | 10.6 | 10.6 | 32.4 | 21.4 | 21.9 | 21.9 |
| 5 | 10.6 | 10.6 | 10.6 | 10.6 | 21.4 | 32.4 | 21.9 | 21.9 |
| 6 | 10.6 | 10.7 | 10.6 | 10.6 | 21.9 | 21.9 | 32.3 | 21.4 |
| 7 | 10.7 | 10.6 | 10.6 | 10.6 | 21.9 | 21.9 | 21.4 | 32.5 |



# Enforcing memory locality with **numactl**

- **numactl** can influence the way a binary maps its memory pages:

```
numactl --membind=<nodes>  a.out          # map pages only on <nodes>
        --preferred=<node> a.out          # map pages on <node>
                                          # and others if <node> is full
        --interleave=<nodes> a.out        # map pages round robin across
                                          # all <nodes>
```

- Examples:

```
for m in `seq 0 7`; do
  for c in `seq 0 7`; do
    env OMP_NUM_THREADS=6 \
      numactl --membind=$m likwid-pin –c M${c}:0-5 ./a.out
  done
done
```

ccNUMA map scan

```
numactl --interleave=0-7 likwid-pin -c E:N:8:1:12 ./a.out
```

Advanced affinity enforcement with LIKWID → see separate lectures

- But what is the default without **numactl?**

# ccNUMA default placement policy

"Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that touches it first!

(Except if there is not enough local memory available)

- Caveat: "to touch" means "to write," not "to allocate"
- Example:

```
double *huge = (double*)malloc(N*sizeof(double));

for(i=0; i<N; i++) // or i+=PAGE_SIZE/sizeof(double)
   huge[i] = 0.0;
```

Memory not mapped here yet

Mapping takes place here

- It is sufficient to touch a single item to map the entire page

# Coding for ccNUMA data locality

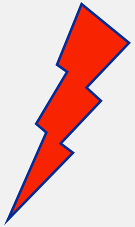
## Most simple case: explicit initialization

```
const int n=10000000;
a=(double*)malloc(n*sizeof(double));
b=(double*)malloc(n*sizeof(double));

...


for(int i=0; i<n; ++i)
    a[i] = 0.;
...

#pragma omp parallel for
for(int i=0; i<n; ++i)
    b[i] = function(a[i]);
```



```
const int n=10000000;
a=(double*)malloc(n*sizeof(double));
b=(double*)malloc(n*sizeof(double));
...

#pragma omp parallel
{
#pragma omp for schedule(static)
for(int i=0; i<n; ++i)
    a[i] = 0.;
...
#pragma omp for schedule(static)
for(int i=0; i<n; ++i)
    b[i] = function(a[i]);
}
```

# Coding for Data Locality

- Required condition: OpenMP loop schedule of initialization must be the same as in all computational loops
  - Only choice: **`static`**! Specify explicitly on all NUMA-sensitive loops, just to be sure…
  - Imposes some constraints on possible optimizations (e.g., load balancing)
  - Presupposes that all worksharing loops with the same loop length have the same thread-chunk mapping
  - If dynamic scheduling/tasking is unavoidable, the problem cannot be solved completely if a team of threads spans more than one LD
    - Static parallel first touch is still a good idea
- How about global objects?
  - If communication vs. computation is favorable, might consider properly placed copies of global data
- C++: Arrays of objects and **`std::vector<>`** are by default initialized sequentially
  - STL allocators provide an elegant solution

# NUMA-aware allocator for C++ **std::vector<>**

```
template <class T> class NUMA_Allocator {
public:
  T* allocate(size_type numObjects, const void
                 *localityHint=0) {
    size_type ofs,len = numObjects * sizeof(T);
    void *m = malloc(len);
    char *p = static_cast<char*>(m);
    int i,pages = len >> PAGE_BITS;
#pragma omp parallel for schedule(static) private(ofs)
    for(i=0; i<pages; ++i) {
      ofs = static_cast<size_t>(i) << PAGE_BITS;
      p[ofs]=0;
    }
    return static_cast<pointer>(m);
  }
...
};
```

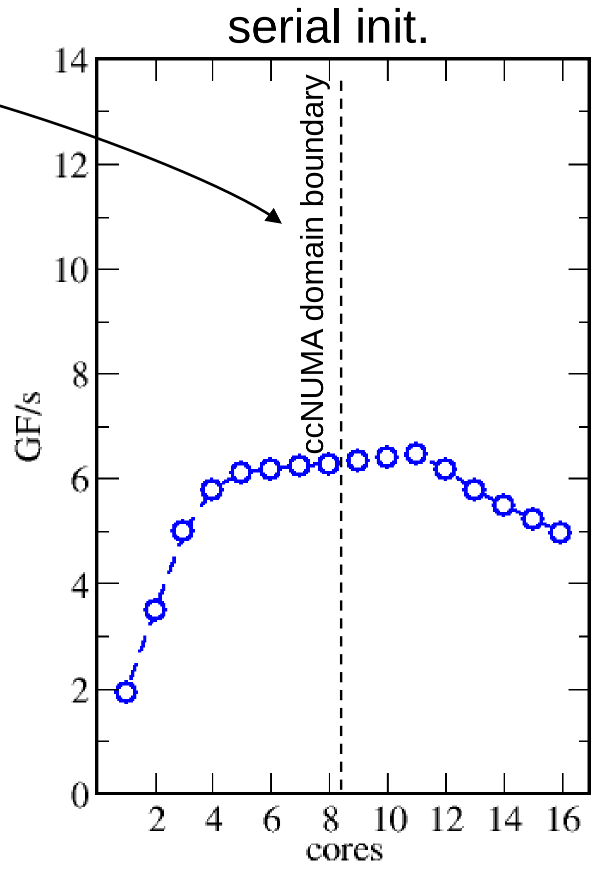
Application:

```
vector<double,NUMA_Allocator<double> > x(10000000);
```

# Diagnosing bad ccNUMA locality

- Bad locality limits scalability (whenever a ccNUMA node boundary is crossed)
  - Just an indication, not a proof yet
- Running with `numactl --interleave` might give you a hint

- Important:
  This is all only relevant if the code is actually sensitive to memory access!



# Using performance counters for diagnosis

- Intel Ivy Bridge EP node (running 2x5 threads):
  measure NUMA traffic per core with **`likwid-perfctr`**

```
$ likwid-perfctr -g NUMA –C M0:0-4@M1:0-4 ./a.out
```

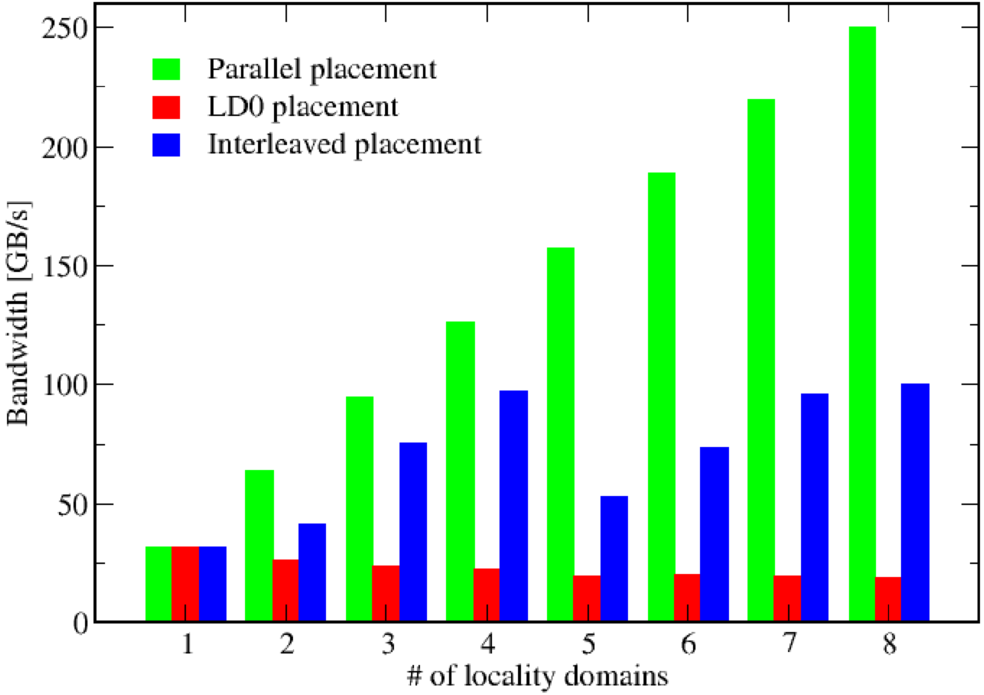
- Summary output:

| Metric | Sum | Min | Max | Avg |
|---|---|---|---|---|
| Runtime (RDTSC) [s] STAT | 4.050483 | 0.4050483 | 0.4050483 | 0.4050483 |
| Runtime unhalted [s] STAT | 3.03537 | 0.3026072 | 0.3043367 | 0.303537 |
| Clock [MHz] STAT | 32996.94 | 3299.692 | 3299.696 | 3299.694 |
| CPI STAT | 40.3212 | 3.702072 | 4.244213 | 4.03212 |
| Local DRAM data volume [GByte] STAT | 7.752933632 | 0.735579[illegible] | [illegible] | [illegible]2 |
| Local DRAM bandwidth [MByte/s] STAT | 19140.761 | 1816.0[illegible] | [illegible] | [illegible] |
| Remote DRAM data volume [GByte] STAT | 9.16628352 | 0.86682[illegible] | [illegible] | [illegible]2 |
| Remote DRAM bandwidth [MByte/s] STAT | 22630.098 | 2140.0[illegible] | [illegible] | [illegible] |
| Memory data volume [GByte] STAT | 16.919217152 | 1.690376[illegible] | [illegible] | [illegible]2 |
| Memory bandwidth [MByte/s] STAT | 41770.861 | 4173.27 | 4180.714 | 4177.0861 |

About half of the overall memory traffic is caused by the remote domain!

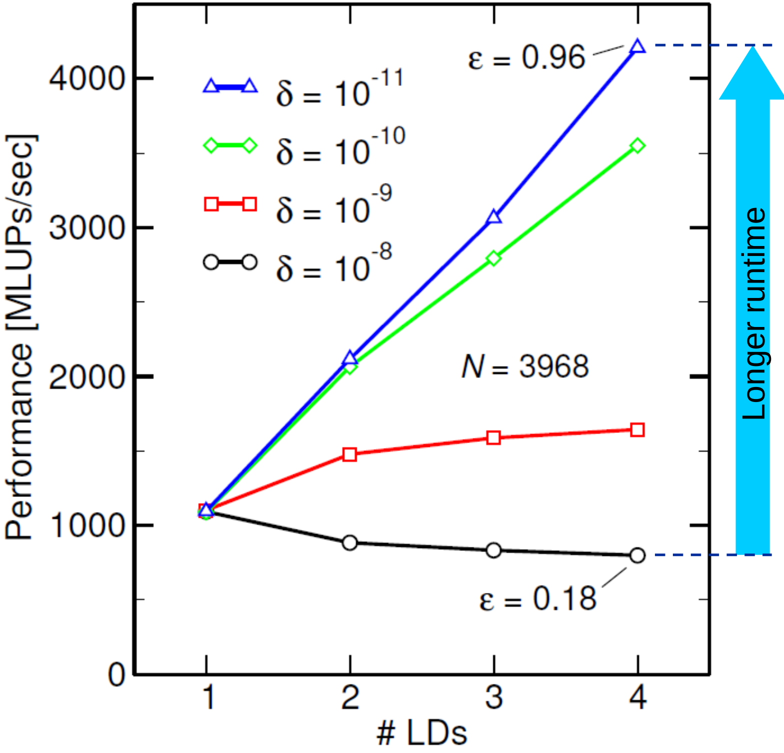
# OpenMP STREAM triad on AMD Epyc 7451 (6 cores per LD)

- Parallel init: Correct parallel initialization
- LD0: Force data into LD0 via **`numactl –m 0`**
- Interleaved:
  **`numactl --interleave ...`**



# A weird observation

- Experiment: memory-bound Jacobi solver with sequential data initialization
  - No parallel data placement at all!
  - Expect no scaling across LDs
- Convergence threshold $\delta$ determines runtime
  - The smaller $\delta$, the longer the run
- Observation
  - No scaling across LDs @ large $\delta$ (runtime 0.5 s)
  - Scaling gets better with smaller $\delta$ up to almost perfect efficiency $\varepsilon$ (runtime 91 s)




- Conclusion
  - Something seems to "heal" the bad access locality on a time scale of tens of seconds

# Riddle solved: NUMA balancing

- Linux kernel supports automatic page migration:

```
$ cat /proc/sys/kernel/numa_balancing
0
$ echo 1 > /proc/sys/kernel/numa_balancing    # activate
```

- Active on current Linux distributions
- Parameters control aggressiveness

```
$ ll /proc/sys/kernel/numa*
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_delay_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_period_max_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_period_min_ms
-rw-r--r-- 1 root root 0 Jun 26 09:16 numa_balancing_scan_size_mb
```

- Default behavior is “take it slow” → it takes some time to “kick in”
- Do not rely on it! Parallel first touch is still a good idea!

# Summary on ccNUMA issues

- Identify the problem
  - Is ccNUMA an issue in your code?
  - Simple test: run with **`numactl --interleave`**
- Apply first-touch placement
  - Look at initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care
- NUMA balancing is active on many Linux systems today
  - Slow process, may take many seconds (configurable), not a silver bullet
  - Still a good idea to do parallel first touch
- If dynamic scheduling cannot be avoided
  - Still a good idea to do parallel first touch

Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

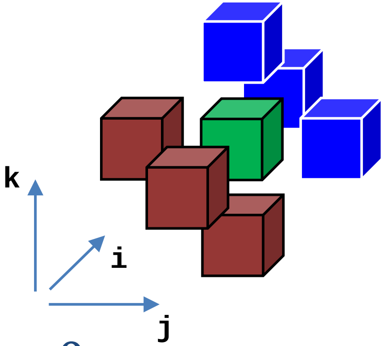
# Case study: Parallelizing a Gauss-Seidel Solver

High Performance Computing

# 3D matrix-free Gauss-Seidel smoother

- Matrix-free iterative solver for $Ax = b$

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(-\sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)} + b_i\right)$$




- Here used for Dirichlet boundary value (PDE) problem $\Delta x = 0$

```
for(it=0; it<itmax; ++it) { // or convergence check
  for(k=1; k<kmax-1; ++k) {
    for(j=1; j<jmax-1; ++j) {
      for(i=1; i<imax-1; ++i) {
        x[k][j][i] = ( x[k][j][i-1] + x[k][j][i+1]
                     + x[k][j-1][i] + x[k][j+1][i]
                     + x[k-1][j][i] + x[k+1][j][i])/6.0;
} } } }
```

"new data" "old data"

# OpenMP parallelization?

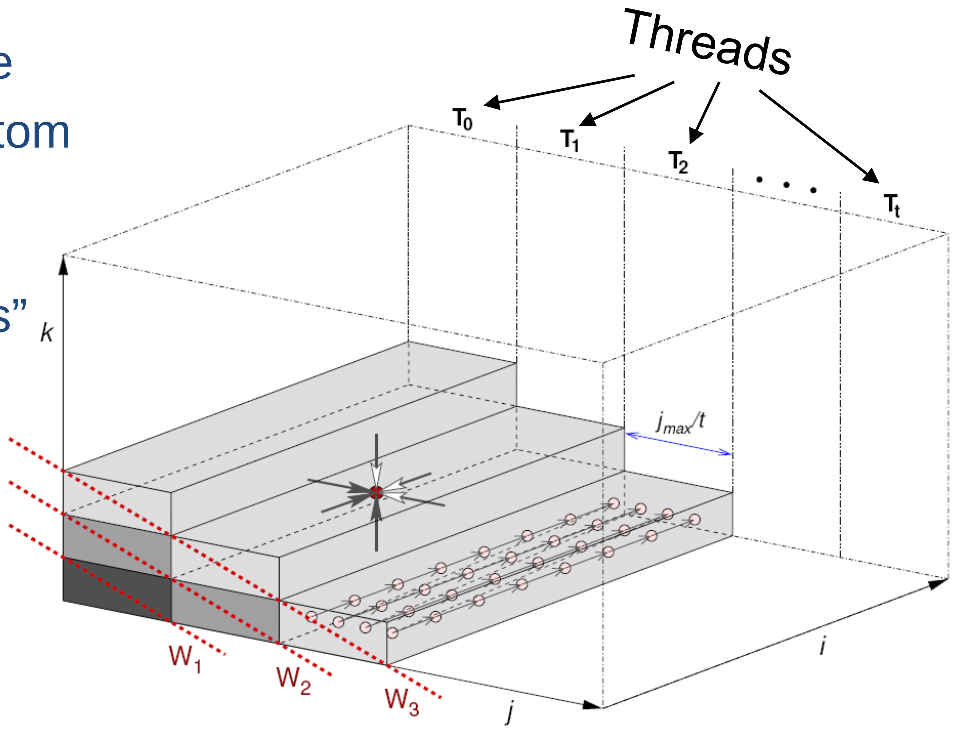
- Naïve OpenMP loop parallelzation impossible due to loop-carried dependency on all spatial loop levels

```
for(t=0; t<itmax; ++t) {
  for(k=1; k<kmax-1; ++k) {
    for(j=1; j<jmax-1; ++j) {
      for(i=1; i<imax-1; ++i) {
        x[k][j][i] = ( x[k][j][i-1] + x[k][j][i+1]
                     + x[k][j-1][i] + x[k][j+1][i]
                     + x[k-1][j][i] + x[k+1][j][i])/6.0;
} } } }
```

- Can we solve this in parallel but still keep the dependencies intact?

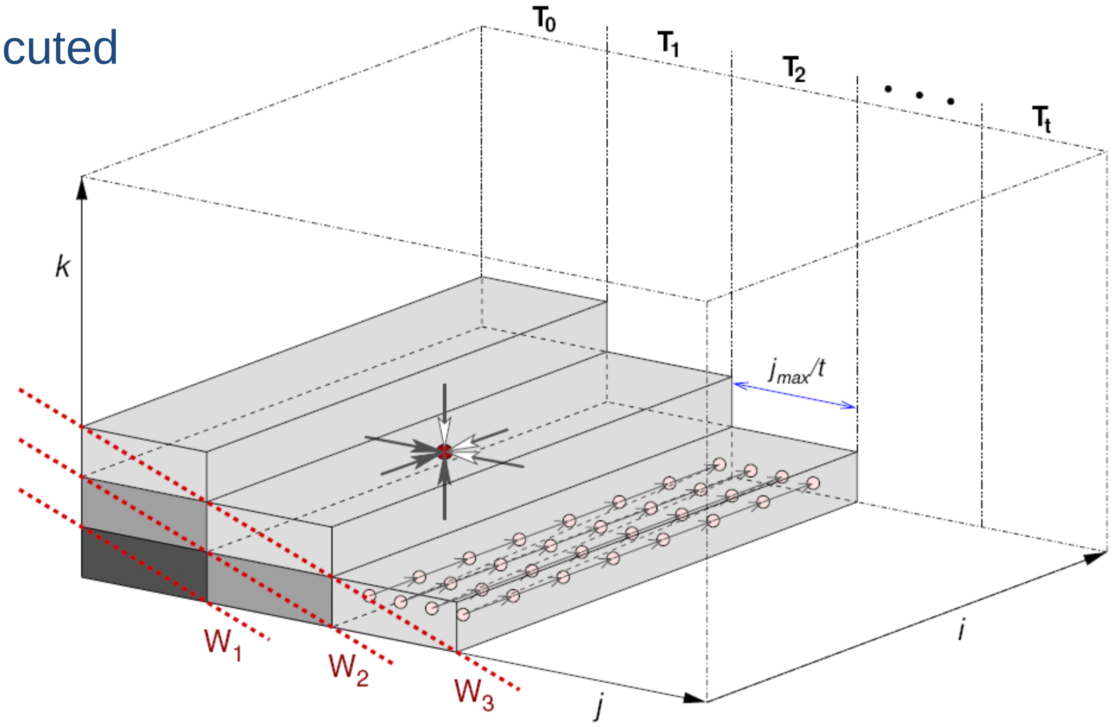
# Idea: wavefront parallelization

- Parallelization approach
  - Middle (j) loop is parallel
  - Outer dimension: wavefront scheme
  - Each block can be updated iff if bottom neighbor (same threadID) and left neighbor (threadID-1) are done
  - $W_i$: independent blocks, “wavefronts”
  - After each wavefront: synchronization to maintain ordering



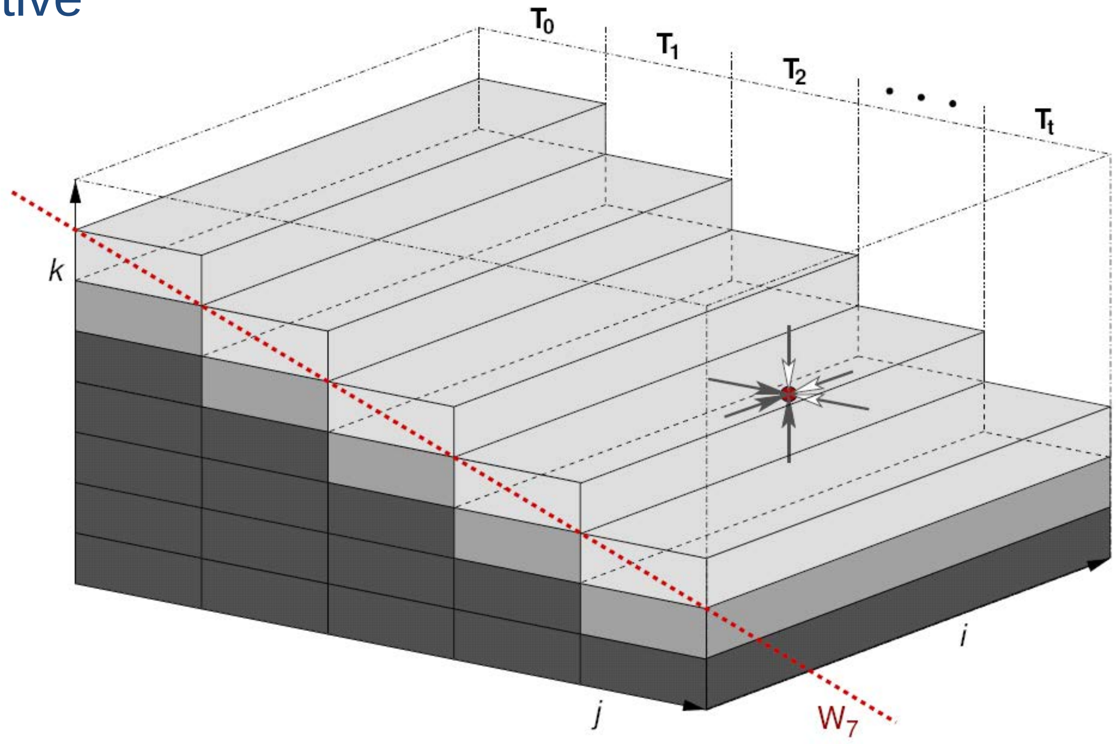
# Wavefront parallelization

- Wind-up phase
  - Not all threads active
  - Each wavefront ($W_i$) is executed by i threads concurrently



# Wavefront parallelization

- "Full pipeline": All threads active




- Wind-down phase starts after $T_0$ has completed its k loop (not shown)

# Wavefront parallelization with OpenMP in 3D

```
#pragma omp parallel private(nthreads,istart,iend,tid,kk,it,k,i){
  nthreads = omp_get_num_threads();
  tid = omp_get_thread_num();
  jstart= (jmax-2)/nthreads * tid;
  jend  = jstart+(jmax-2)/nthreads-1
  for(t=0; t<itmax; ++t) {
    for(k=1; k<kmax-1+nthreads-1; ++k) {
      kk = k – tid;
      if(kk>=1 && k<=kmax-1) {
        for(j=jstart; j<=jend; ++j) {
          for(i=1; i<kmax-1; ++i) {
            x[kk][j][i] = ( x[kk][j][i-1] + x[kk][j][i+1]
                          + x[kk][j-1][i] + x[kk][j+1][i]
                          + x[kk-1][j][i] + x[kk+1][j][i])/6.0;
          }
        }
      }
      #pragma omp barrier
    }
  }
}
```

Chop **j** loop into **nthreads** chunks

Wind-up/-down

Wavefront sync

# Wavefront parallelization – open questions

- Global barrier per middle loop sweep (i.e., **`kmax-2`** barriers overall)
  - Remedy?

- Is there a global performance limit?
  - Minimum data traffic: update whole array once
    → minimum traffic = read and write **`imax*jmax*kmax`** elements
    → 16 byte/update
- Should SMT give better performance?
  - There's a dependency after all…
- How about ccNUMA?
  - Is placement an issue here?