

Programming Techniques for Supercomputers: Shared-memory parallel processing with OpenMP (II)

OpenMP reductions

OpenMP synchronization

OpenMP basic overheads

OpenMP affinity

Prof. Dr. G. Wellein^(a,b), Dr. G. Hager^(a)

^(a) Erlangen National High Performance Computing Center (NHR@FAU)

^(b) Department für Informatik

Friedrich-Alexander-Universität Erlangen-Nürnberg, Sommersemester 2025



Shared-memory parallel processing with OpenMP (II)

OpenMP reductions

OpenMP synchronization

OpenMP basic overheads

OpenMP affinity



Operations on data across threads

- Recurring problem: Operations across thread-local instances of a variable

```
int i,N;
double a[N], b[N];
...
s=0.;
#pragma omp parallel firstprivate(s)
{
    #pragma omp for
    for(i=0; i<N; ++i)
        s = s + a[i] * b[i];
    // How to sum up the different s?
}
```

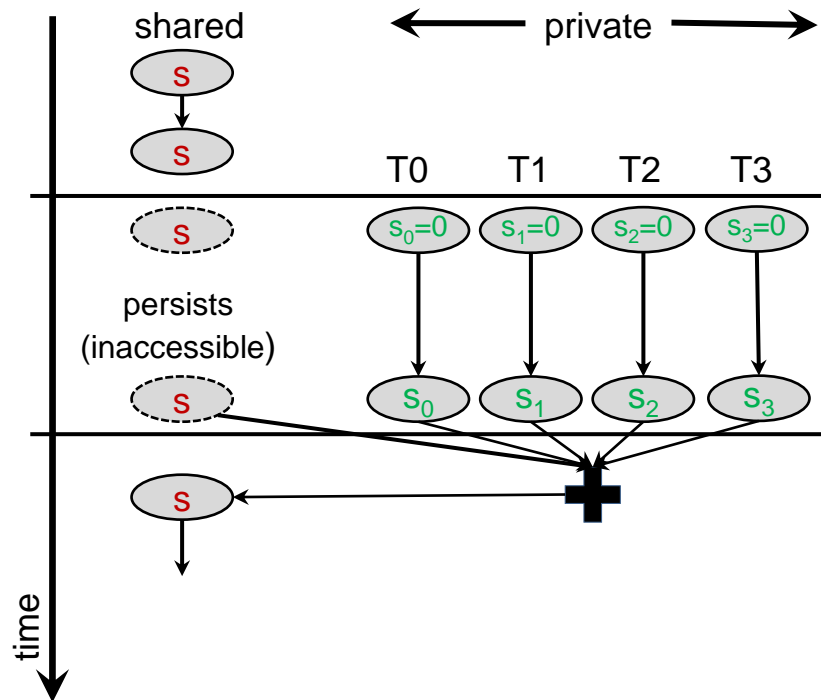
- Solution: **reduction** clause

Reduction clause on parallel region or workshared loop

```
int i,N;
double a[N], b[N];
...
s=0.;
#pragma omp parallel
{
  // s is still shared here
  #pragma omp for reduction(+:s)
  for(i=0; i<N; ++i)
    s = s + a[i] * b[i];
  // s is shared again here
}
```

At synchronization point:

- reduction operation is performed
- result is transferred to master copy
- restrictions similar to **firstprivate**



Reduction variable **must be shared** in enclosing context!

Reduction operations: general considerations

Fortran has an analogous set

Operation	Initial value
+	0
-	0
*	1
&	~0
	0
^	0
&&	1
	0
max	MINVAL (type)
min	MAXVAL (type)

Multiple reductions:

```
float x, y, z;  
#pragma omp for reduction(+:x, y, z)
```

```
#pragma omp for reduction(+:x, y) \  
reduction(*:z)
```

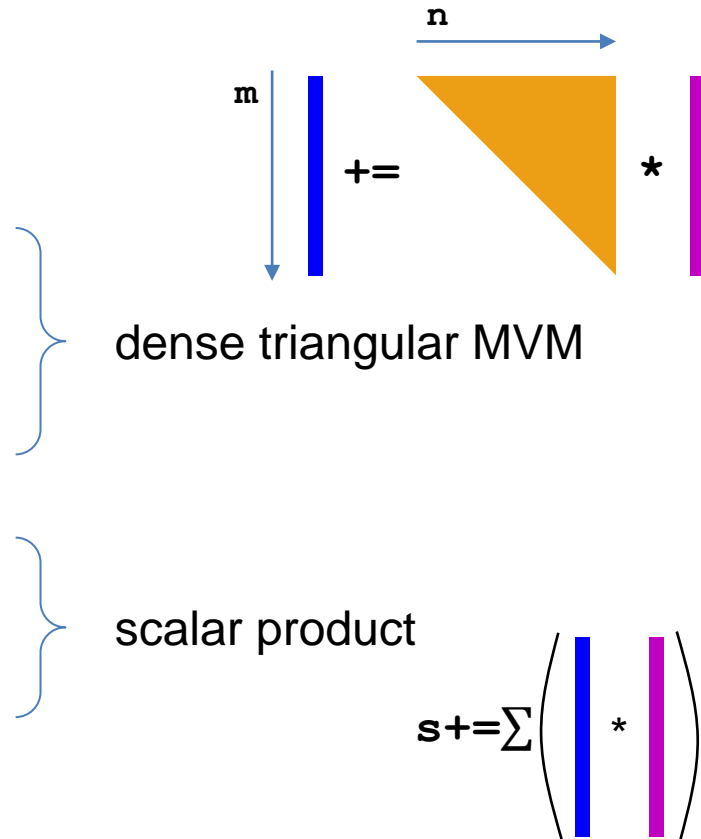
Consistency required!

X = expr - X is not allowed

Don't lie.

Reduction operations: Example

```
double s, a[size*size], x[size], y[size];
...
s=0.;
#pragma omp parallel
{
#pragma omp for schedule(???)
    for(int m=0; m<size; m++){
        for(int n=m; n<size; n++){
            y[m] += a[m*size+n] * x[n];
        }
    }
    ...
#pragma omp for reduction(+:s)
    for(int m=0; m<size; m++) {
        s += x[m] * y[m];
    }
    ...
}
```



Reductions on arrays

- Elementwise reductions on arrays (or slices thereof)

```
#pragma omp parallel for reduction(+:y[0:rows])
for(int c=0; c<cols; ++c)
    for(int r=0; r<rows; ++r)
        y[r] += a[r+c*rows] * x[c];
```

C/C++: Array slice
syntax is mandatory

Fortran: No slice
necessary on full array
reduction

```
!$omp parallel do reduction(+:y)
do c = 1 , C
    do r = 1 , R
        y(r) = y(r) + A(r,c) * x(c)
    enddo
enddo
!$omp end parallel do
```

Shared-memory parallel processing with OpenMP (II)

OpenMP reductions

OpenMP synchronization:

OpenMP basic overheads

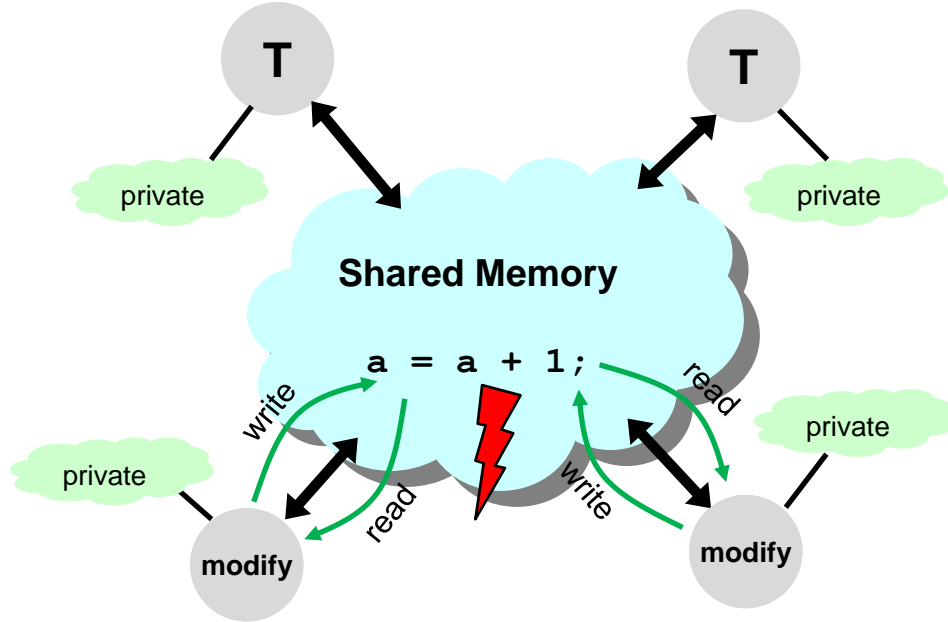
OpenMP affinity

Ensuring consistency



Why synchronization?

Example: variable update (read – modify – write)



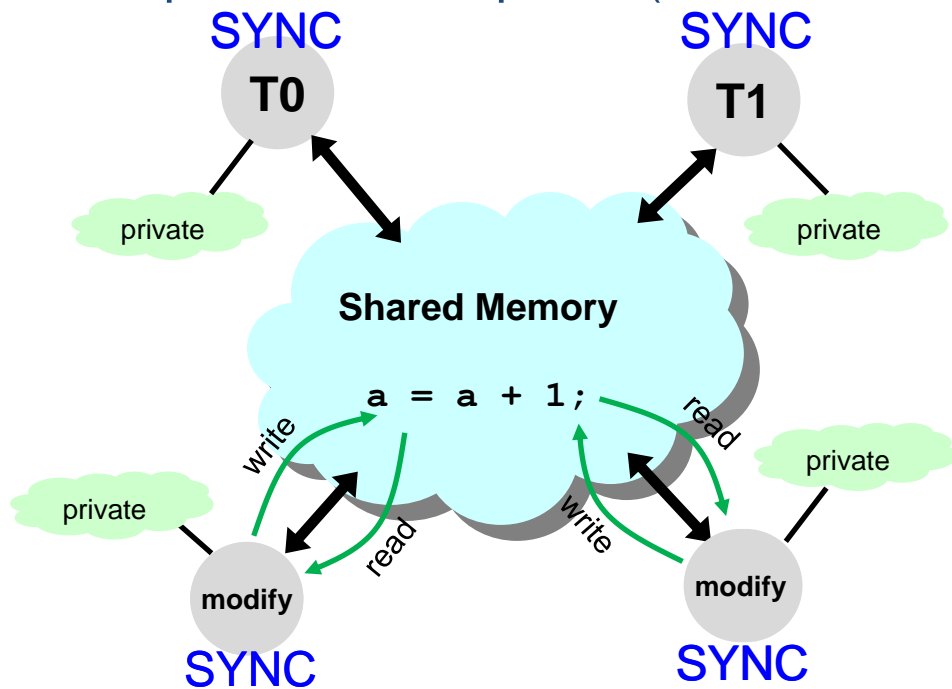
Multiple threads access shared variable, and at least one writes to it

→ “**race condition**”

Synchronization = means to manage conflicting/uncontrolled accesses

Why synchronization?

Example: variable update (read – modify – write)

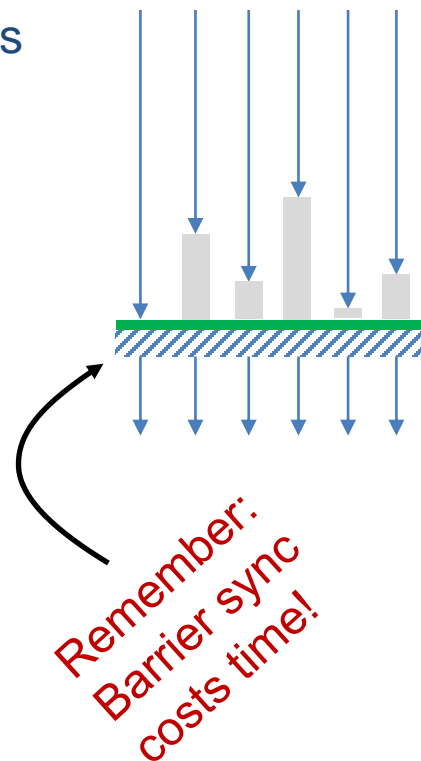


Synchronization: All threads need to wait until last thread enters synchronization

T2: read a
T2: `a=a+1`
T2: write a
SYNCHRONIZATION
T3: read a
T3: `a=a+1`
T3: write a
SYNCHRONIZATION

Barrier synchronization

- **#pragma omp barrier**
 - Each thread blocks upon reaching the barrier until all threads have reached the barrier
 - All accessible shared variables are flushed to the memory hierarchy (similar to **volatile** attribute in C/C++)
 - barrier may not appear within work-sharing construct (e.g., **omp for** block) → potential of **deadlock**
- **Implicit barrier:**
 - at the beginning and end of parallel regions
 - at the end of worksharing constructs unless a **nowait** clause is present



Relaxing synchronization requirements

- The **nowait** clause
 - removes the implicit barrier at end of worksharing construct
 - potential performance **improvement** (especially if load imbalance occurs within construct)
 - Programmer is responsible for preventing race conditions!

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(int i=0; i<N; ++i) {
        a[i] = some_stuff(i);
    }

    // ... More parallel work (don't reference a[])

    #pragma omp barrier
    ... = a[i];    // after deferred barrier
}
```

No barrier here

Case study: reducing barrier cost for dense MVM

- General advice: Parallelize as far out as possible!

```
void dmvm(int n, int m, double *lhs,  
          double *rhs, double *mat){  
    ...  
    #pragma omp parallel for  
    for(int c=0; c<n; ++c)  
        int offset = m * c;  
        for(int r=0; r<m; ++r)  
            lhs[r] += mat[r + offset] * rhs[c];  
}
```



Only one barrier...

... but race condition
on lhs[]

Reducing barrier cost: dense MVM

- Inner loop parallel → correct result

```
void dmvm(int n, int m, double *lhs,  
          double *rhs, double *mat){  
    ...  
    #pragma omp parallel  
    {  
        for(int c=0; c<n; ++c)  
            int offset = m * c;  
            #pragma omp for  
            for(int r=0; r<m; ++r)  
                lhs[r] += mat[r + offset] * rhs[c];  
    }  
}
```

Only one parallel region

... but n implicit barriers

Result is correct: threads work
on separate parts of `lhs[]`

Reducing barrier cost: dense MVM

- Inner loop parallel → correct result, and use `nowait` to avoid barriers

```
void dmvm(int n, int m, double *lhs,  
          double *rhs, double *mat){  
    ...  
    #pragma omp parallel  
    {  
        for(int c=0; c<n; ++c)  
            int offset = m * c;  
            #pragma omp for schedule(static) nowait  
            for(int r=0; r<m; ++r)  
                lhs[r] += mat[r + offset] * rhs[c];  
    }  
}
```

Only one parallel region

No implicit barriers on
workshared loop

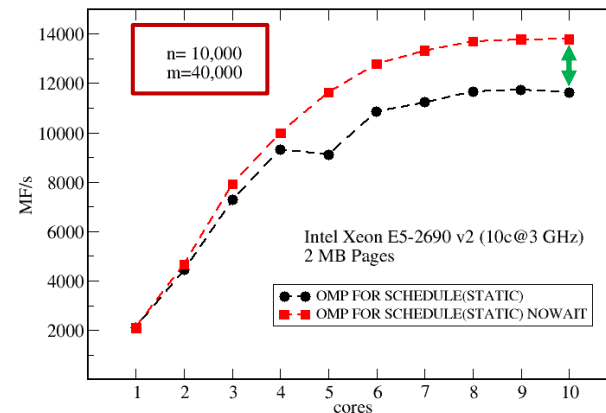
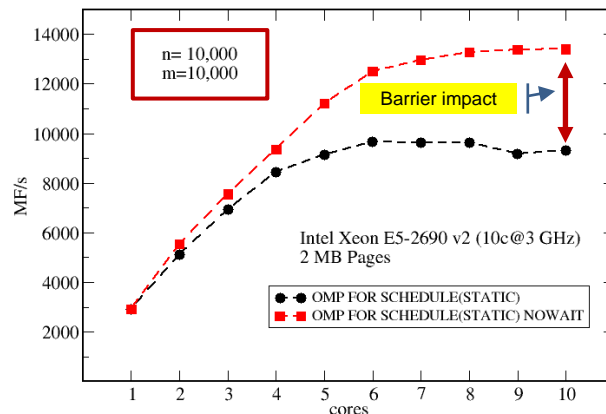
Ensure same iteration-to-
thread mapping

One implicit barrier

Result is correct: threads work
on separate parts of `lhs[]`

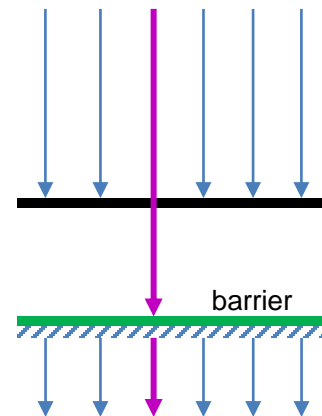
Reducing barrier cost: dense MVM

- **Barrier overhead** may substantially decrease performance
- Performance impact decreases as inner loop length (work per barrier) increases (see $m=40,000$ vs. $m=10,000$)
- Use `nowait` with due care (correctness)!
- Is the performance as expected? What does the barrier cost?
 - → homework



The `single` directive

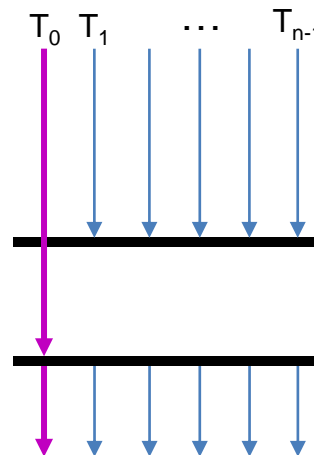
- `#pragma omp single [clause[[,]clause]...]`
`structured-block`
- Structured block is **executed by exactly one thread**, which one is unspecified
 - Actually a **worksharing** directive
- Remaining threads skip the structured block and continue execution.
- Implied barrier at the exit of the single section!
- **Do not use within another worksharing construct (deadlock!)**
- `nowait` clause suppresses barrier



The master directive

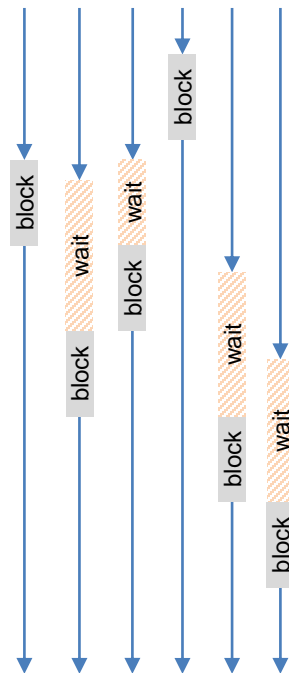
- `#pragma omp master [clause[[,]clause]...]`
`structured-block`
- Only thread zero executes the structured block
- Other threads continue **without synchronization**
- Not all threads have to reach the construct
- Essentially equivalent to:

```
#ifdef _OPENMP
if(omp_get_thread_num()==0)
#endif
    structured-block;
```



Critical region

- `#pragma omp critical`
 `structured-block`
- Only **one thread at a time** can execute the block
- ... but every thread that encounters it will eventually execute it
- **Order of execution is undefined!**
- All unnamed critical regions are mutually exclusive across the whole program
 - **Beware of deadlocks!**



Named critical regions

- What if I want several independent critical regions?
 - Named critical regions to the rescue!
- Regions with different names are mutually independent
- Name can be chosen freely
 - No association with data to be “protected”
- Unnamed critical regions share the same (invisible) name

```
double func(double v) {  
    double x;  
    #pragma omp critical(prand)  
        x = v + random_func();  
    return x;  
}  
...  
#pragma omp parallel for private(x)  
for(int i=0; i<N; ii++) {  
    x = sin(2.*M_PI*i/N);  
    #pragma omp critical(psum)  
        sum += func(x);  
}
```

Protect lib-call
(random_func)

Atomic updates

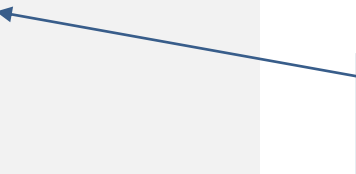
- `#pragma omp atomic [clause[[,] clause] ...]
expression-stmt`
- Ensures that a storage location is **accessed atomically**, i.e., the full access cannot be interrupted
- Applies only to the statement immediately following it
- `expression-stmt` can be:
 - `x++;`
 - `x--;`
 - `++x;`
 - `--x;`
 - `x binop= expr;`
 - `x = x binop expr;`
 - `x = expr binop x;`
- Variants of `atomic` for pure read, pure write, and capture are also available

Why `atomic`?

Can't I just use a critical region?

1. `atomic` *may* be **more efficient** due to hardware support (no guarantee!)
2. `atomic` allows for protecting updates to **individual data elements**

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    double t = func(table[i]);
    if(t < 0.) {
        #pragma omp atomic
        x[table[i]]++;
    }
    y[i] += other(i);
}
```



Updates of different `x[]` entries do not block each other

Shared-memory parallel processing with OpenMP (II)

- OpenMP reductions
- OpenMP synchronization
- OpenMP basic overheads
- OpenMP affinity



Basic OpenMP overheads



“Wake up” team
of threads

```
! $OMP PARALLEL PRIVATE (k)  
do k=1,NITER
```

Workload
distribution

Loop
parallelization

```
! $OMP DO SCHEDULE (...)  
do i=1,N  
A(i)=B(i)+C(i)*D(i)  
enddo
```

Implicit barrier /
synchronization

```
! $OMP END DO
```

“Retire” team of
threads

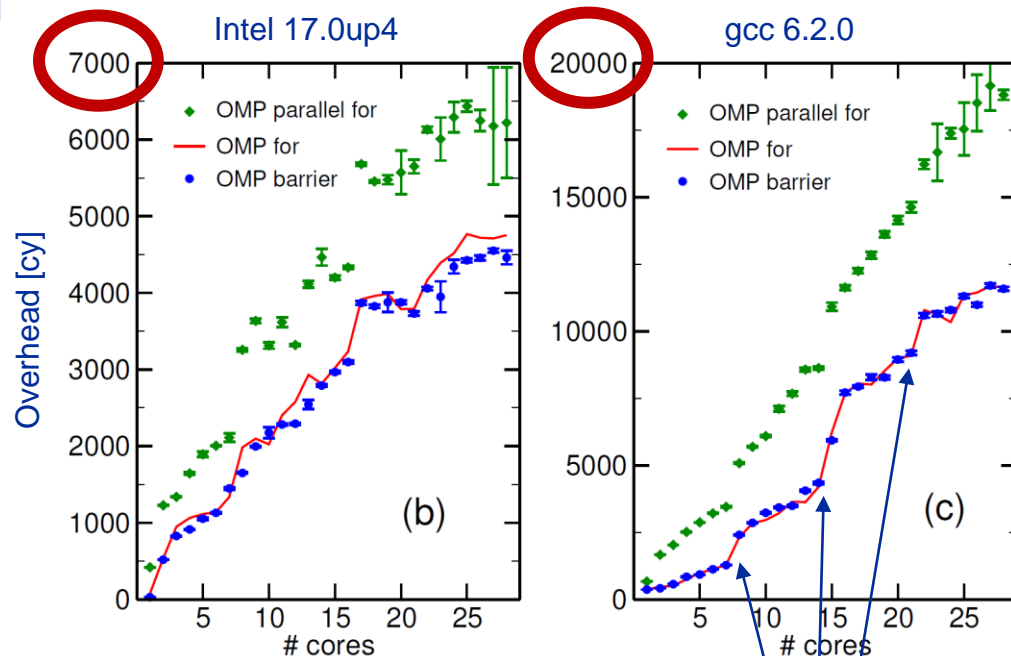
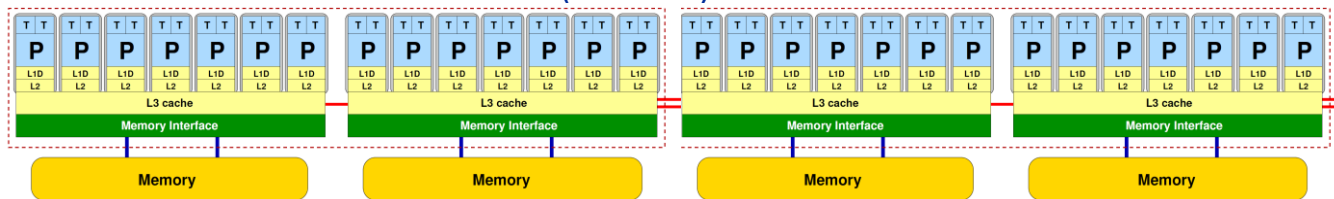
```
! $OMP END PARALLEL
```


OpenMP overheads: loops and barriers

■ Benchmarking OpenMP overhead

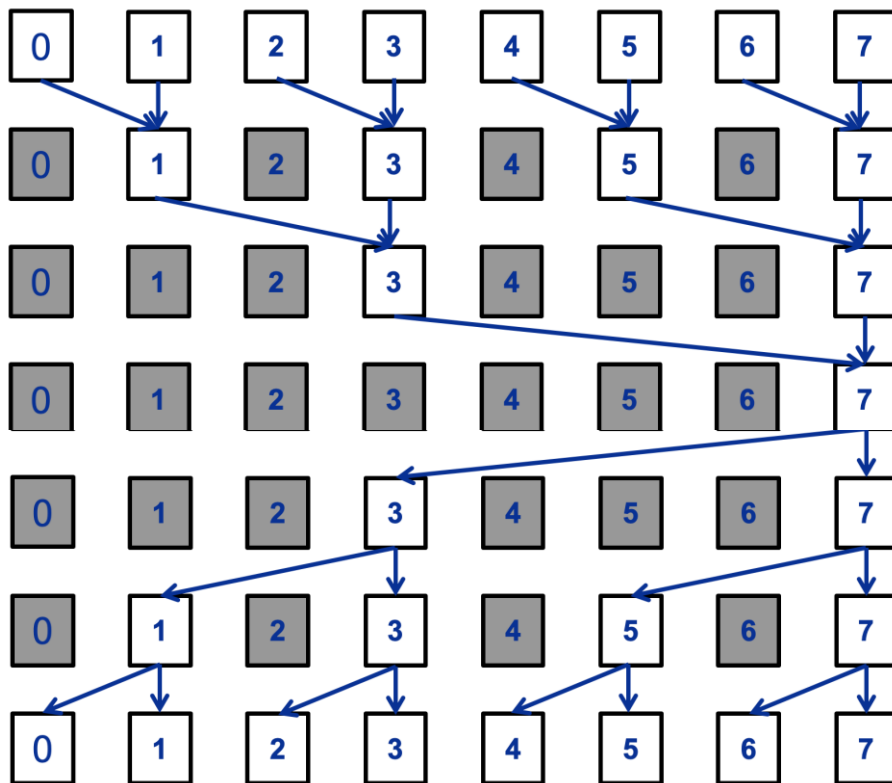
- OpenMP parallel for
 - OpenMP for (w/o parallel)
 - OMP barrier
-
- Static scheduling
 - Compact pinning (physical cores only)

Intel Xeon “Haswell” E5-2695v3 (2.3GHz) CoD



Node topology

OpenMP overheads: Barrier implementation (reminder)

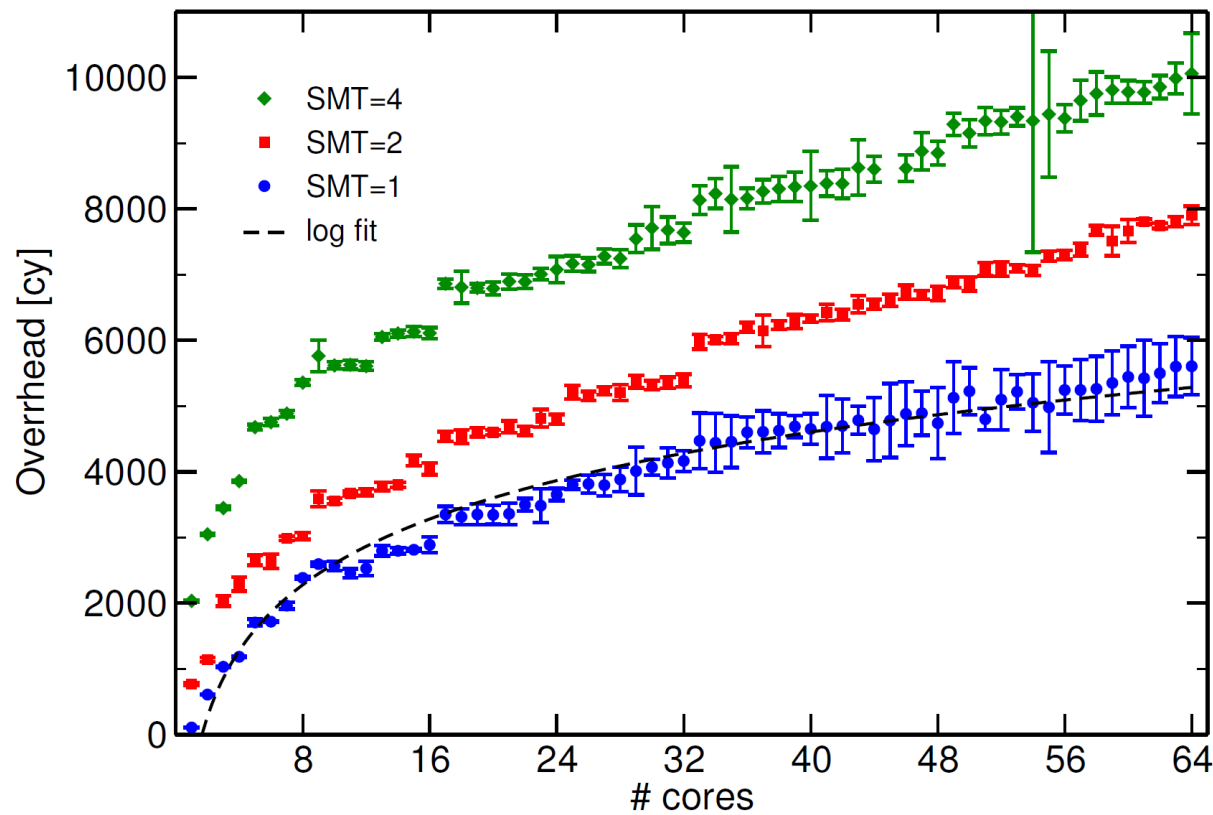


How does a “barrier” scale (best case)?

$$\text{Time}(N) = \text{const} \times 2 \times \log_2 N$$

Where N is number of threads/processes in the barrier

OpenMP overheads: Barrier cost on Intel Xeon Phi (KNL)



Intel Xeon Phi
("Knights Landing"):

64 cores@1.3GHz

1,2,4 SMT per core

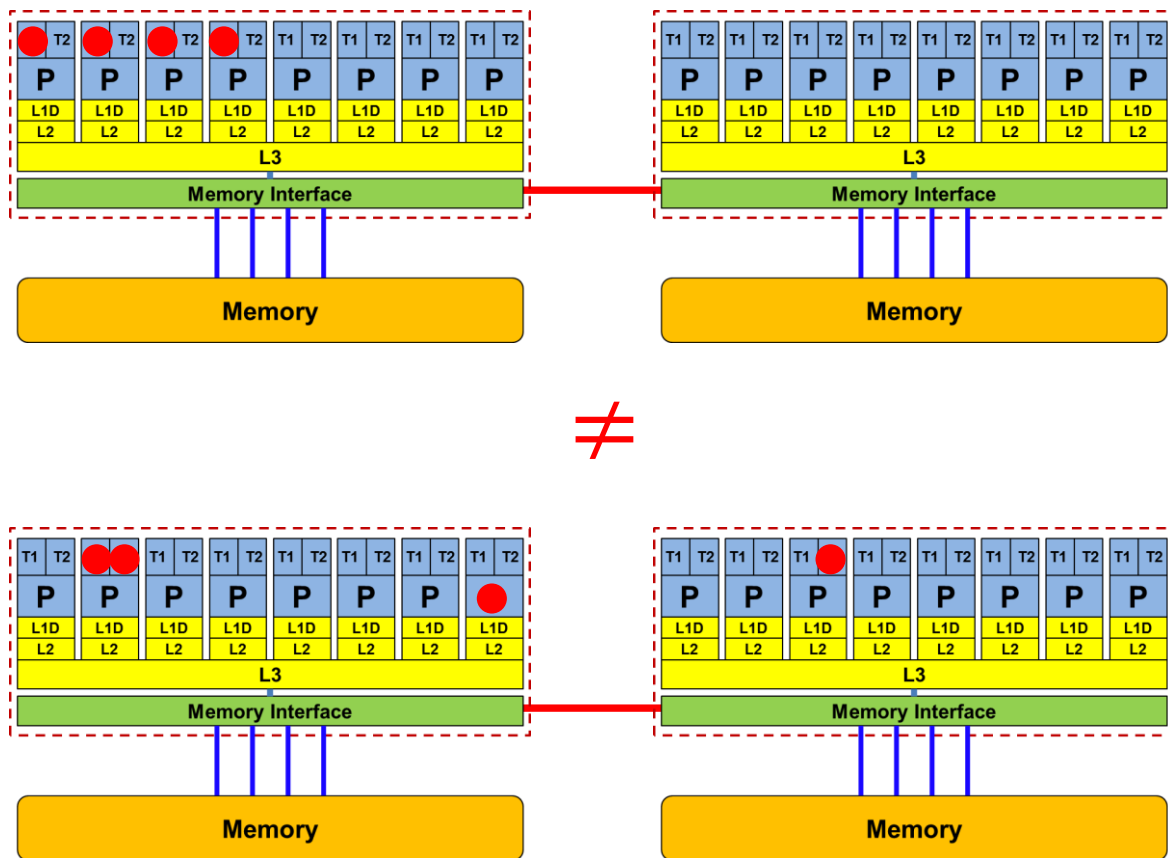
Shared-memory parallel processing with OpenMP (II)

- OpenMP reductions
- OpenMP synchronization
- OpenMP basic overheads
- OpenMP affinity

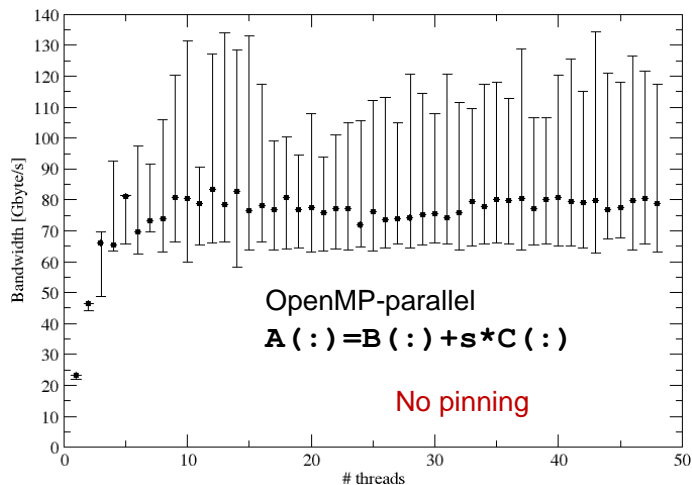


OpenMP affinity: it matters!

- Remember all the hardware bottlenecks!
- It *does* matter where the threads are running
- Yes, it's up to you
- No, the system will not magically guess what's best

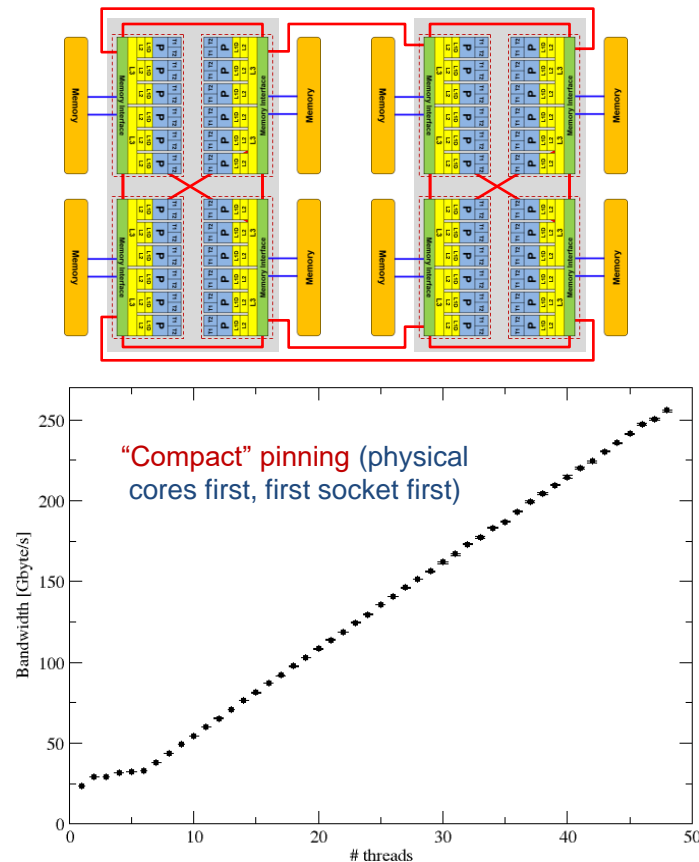


STREAM benchmark on 2x24-core AMD “Naples”: Anarchy vs. thread pinning



There are several reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention



OMP_PLACES and Thread Affinity

- **Processor**: smallest entity able to run a thread or task (SMT/hyper-thread)
- **Place**: one or more processors → thread pinning is done place by place
- Free migration of the threads on a place between the processors of that place.

OMP_PLACES	Place ==
threads	Hardware thread (hyper-thread)
cores	All HW threads of a single core
sockets	All HW threads of a socket
abstract_name (num_places)	Restrict # of places available

abstract name

Or use explicit numbering, e.g. 8 places, each consisting of 4 processors:

- `OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11}, ... {28,29,30,31}"`
- `OMP_PLACES="{0:4},{4:4},{8:4}, ... {28:4}"` — `<lower-bound>:<number of entries>[:<stride>]`
- `OMP_PLACES="{0:4}:8:4"`

Caveat: Actual behavior is implementation defined!

OMP_PROC_BIND variable / proc_bind() clause

Determines how places are used for pinning:

OMP_PROC_BIND	Meaning
FALSE	Affinity disabled
TRUE	Affinity enabled, implementation defined strategy
CLOSE	Threads bind to consecutive places
SPREAD	Threads are evenly scattered among places
MASTER	Threads bind to the same place as the master thread that was running before the parallel region was entered

If there are more threads than places, consecutive threads are put into individual places (“balanced”)

Example:

```
$ OMP_NUM_THREADS=4 OMP_PROC_BIND=close OMP_PLACES=cores ./a.out
```


Some simple OMP_PLACES examples

Intel Xeon w/ SMT, 2x10 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=10  
OMP_PLACES=cores  
OMP_PROC_BIND=close
```

**Always prefer abstract places
instead of hardware thread
IDs!**

Intel Xeon Phi with 72 cores, 4-way SMT

32 cores to be used, 2 threads per physical core

```
OMP_NUM_THREADS=64  
OMP_PLACES=cores(32)  
OMP_PROC_BIND=close      # spread will also do
```

Intel Xeon, 2 sockets, 4 threads per socket (no binding within socket!)

```
OMP_NUM_THREADS=8  
OMP_PLACES=sockets  
OMP_PROC_BIND=close      # spread will also do
```

Intel Xeon, 2 sockets, 4 threads per socket, binding to cores

```
OMP_NUM_THREADS=8  
OMP_PLACES=cores  
OMP_PROC_BIND=spread
```

Wrap-up: beginner's OpenMP toolbox

- Parallel region
- Workshared loop construct
- Data scoping (shared, private, firstprivate)
- Basic reductions with standard operators
- Simple synchronization constructs
 - barrier, nowait
 - (named) critical, atomic
 - single (actually worksharing), master
- OpenMP affinity as defined in the standard

- But wait, there's more...