NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Programming Techniques for Supercomputers Tutorial

Erlangen National High Performance Computing Center

Department of Computer Science

FAU Erlangen-Nürnberg

Sommersemester 2026

Assignment 7 – Task 1

OpenMP fun: correctness

- Problem: tmp is not private

```
double x[100], y[100], tmp;
int i;
double work(int);

// initialization code etc. omitted

#pragma omp parallel
{
  #pragma omp for
  for(i=0; i<100; i++) {
    tmp = work(i);
    x[i] = x[i] + tmp;
  }
  #pragma omp for
  for(i=1; i<100; i++) {
    y[i] = x[i-1] * y[i];
  }
}
```

Possible solutions:

- Make **`tmp`** private on parallel or for construct:
  `#pragma omp parallel for private(tmp)`
- Declare **`tmp`** within the parallel region or loop body
  `double tmp = work(i);`
- Eliminate **`tmp`** altogether
  `x[i] += work(i);`

Assignment 7 – Task 2

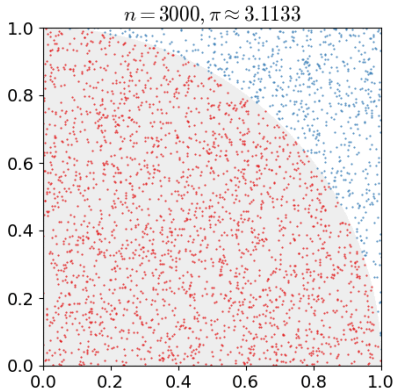
Parallel π by Monte-Carlo method

```
long i, sum=0, nn = 4000000000; // must be long

#pragma omp parallel reduction(+:sum)
{
   unsigned int seed = 6 * omp_get_thread_num(); // per thread
   double x, y;

   #pragma omp for
   for (i = 0; i < nn; i++) {
      x = (rand_r(&seed)/RAND_MAX); y = (rand_r(&seed)/RAND_MAX);
      if (sqrt(x*x + y*y) <= 1.0) ++sum;
   }

}
double pi = (4.0 * sum) / nn;
```



Assignment 7 – Task 2

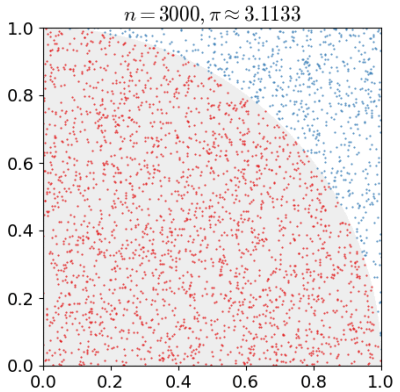
Parallel π by Monte-Carlo method

```
long i, sum=0, nn = 4000000000; // must be long
double rcp = 1./RAND_MAX;

#pragma omp parallel reduction(+:sum)
{
   unsigned int seed = 6 * omp_get_thread_num(); // per thread
   double x, y;

   #pragma omp for
   for (i = 0; i < nn; i++) {
      x = (rand_r(&seed)*rcp); y = (rand_r(&seed)*rcp);
      if (x*x + y*y <= 1.0) ++sum;
   }

}
double pi = (4.0 * sum) / nn;
```



Assignment 7 – Task 2

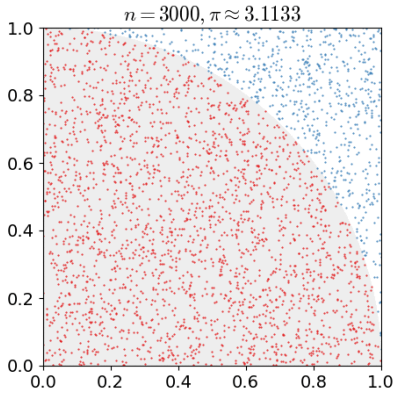
Parallel π by Monte-Carlo method

```
#pragma omp parallel //warm up
{ <do something> }

long i, sum=0, nn = 4000000000; // must be long
double rcp = 1./RAND_MAX;
wcstart = getTimeStamp();
#pragma omp parallel reduction(+:sum)
{
   unsigned int seed = 6 * omp_get_thread_num(); // per thread
   double x, y;

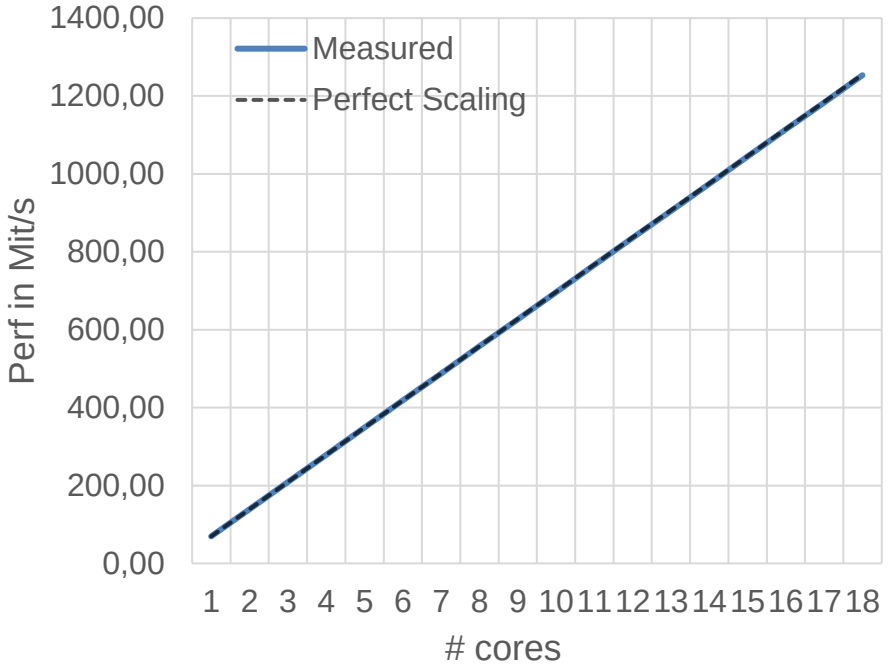
   #pragma omp for
   for (i = 0; i < nn; i++) {
      x = (rand_r(&seed)*rcp); y = (rand_r(&seed)*rcp);
      if (x*x + y*y <= 1.0) ++sum;
   }
}
wcend = getTimeStamp();

double pi = (4.0 * sum) / nn;
printf("Perf: %.3lf Mit/sec\n", (double)nn/(wcend-wcstart)/1.0e6);
```



Assignment 7 – Task 2

Parallel π by Monte-Carlo method



- 1..18 cores of Fritz, fixed 2.4 GHz
- Code scales perfectly across cores if problem is large enough ($4*10^9$ iterations)
- Relative error with 72 threads and nn=4×10^9: 3.2×10^{-6}

Assignment 7 – Task 2

Common errors

- **omp_get_thread_num()** and **omp_get_num_threads()** are only useful inside parallel regions
  - ```
    #pragma omp parallel
    {
      if(omp_get_thread_num()==0)
        numthreads = omp_get_num_threads();
    }
    ```
- Forgot to privatize random seeds with different values→ “incorrect” random sequences, slow code
- Put random seeds into small array:
  x = (rand_r(&seeds[myID])*rcp);
  → massive false sharing leads to extremely bad scalability
- **rand()** works correctly, but very slowly (internal lock on the seed? Depends on the compiler.)
- Watch your data types – when handling large counts, using long may be mandatory
- Forgetting the warmup parallel region (leads to overhead when actual loop is entered, especially with likwid-pin)
    ```

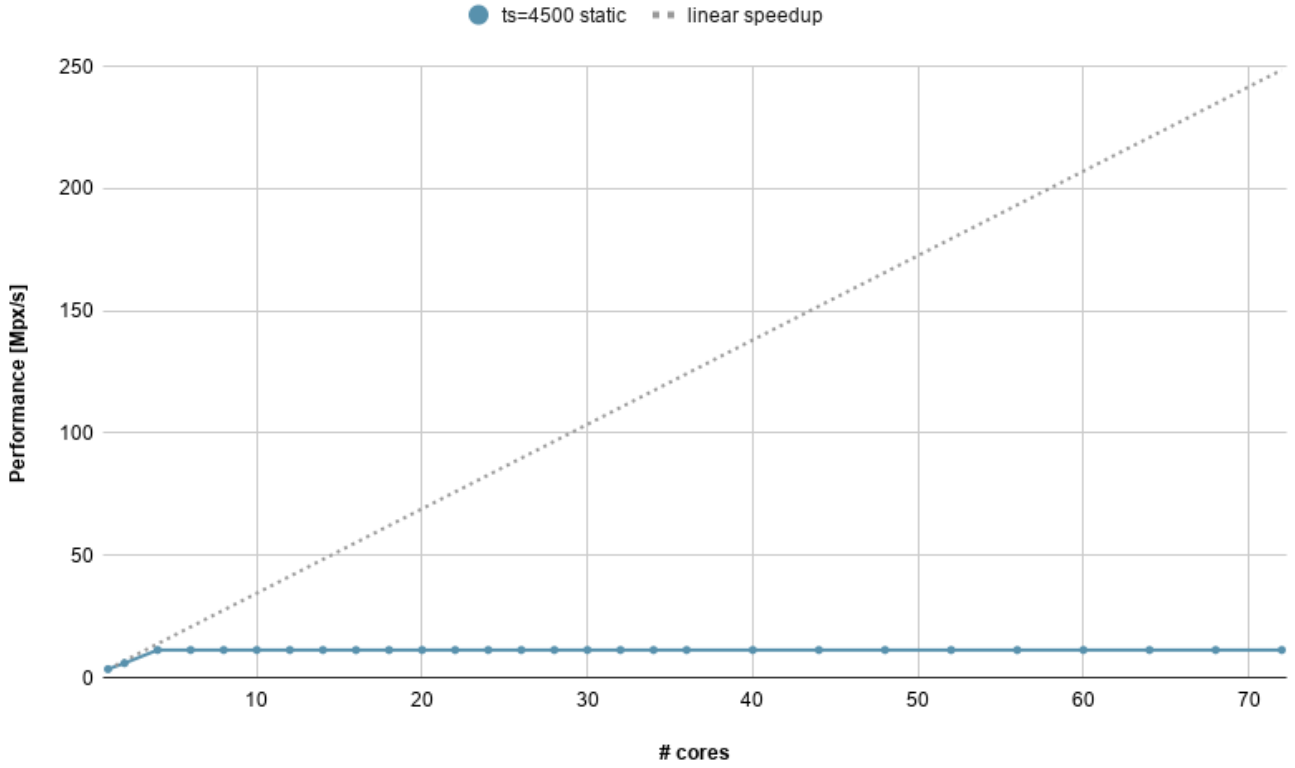
Assignment 7 – Task 3

```
// ...
#pragma omp parallel private(tile)
{
tile = (char*) malloc(tilesize * tilesize * sizeof(char));

//..

count = 0;
#pragma omp for private(xc,i) reduction(+:count)
for(yc=0; yc<ytiles; yc++) {
  for(xc=0; xc<xtiles; xc++) {
     /* calc one tile */
     calc_tile(size, xc*tilesize, yc*tilesize, tilesize, tile);
     /* copy to picture buffer */
     for(i=0; i<tilesize; i++) {
         int tilebase = yc * tilesize * tilesize * xtiles + xc * tilesize;
         memcpy((void*)(picture + tilebase + i * tilesize * xtiles),
                (void*)(tile + i * tilesize),
                tilesize * sizeof(char));
     }
     count++;
   }
}
}
```

Assignment 7 – Task 3



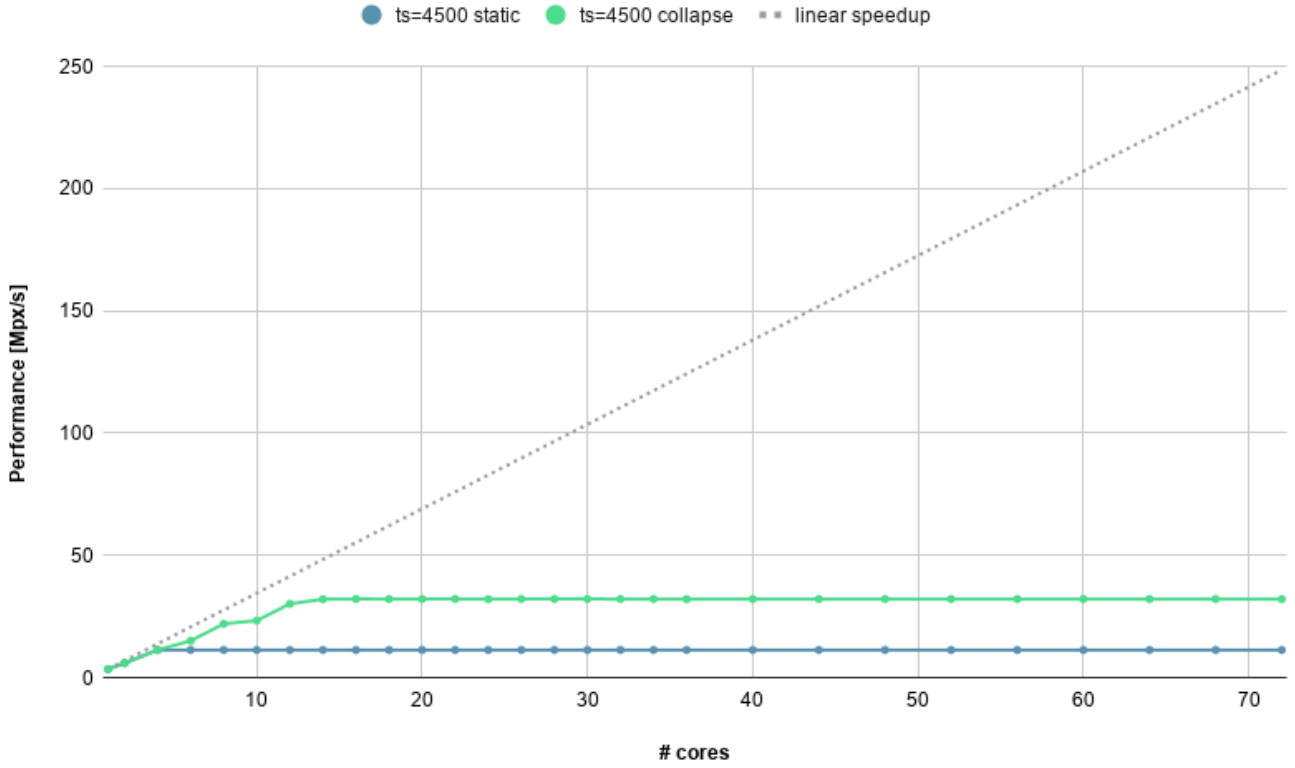
Assignment 7 – Task 3

```
// ...
#pragma omp parallel private(tile)
{
tile = (char*) malloc(tilesize * tilesize * sizeof(char));

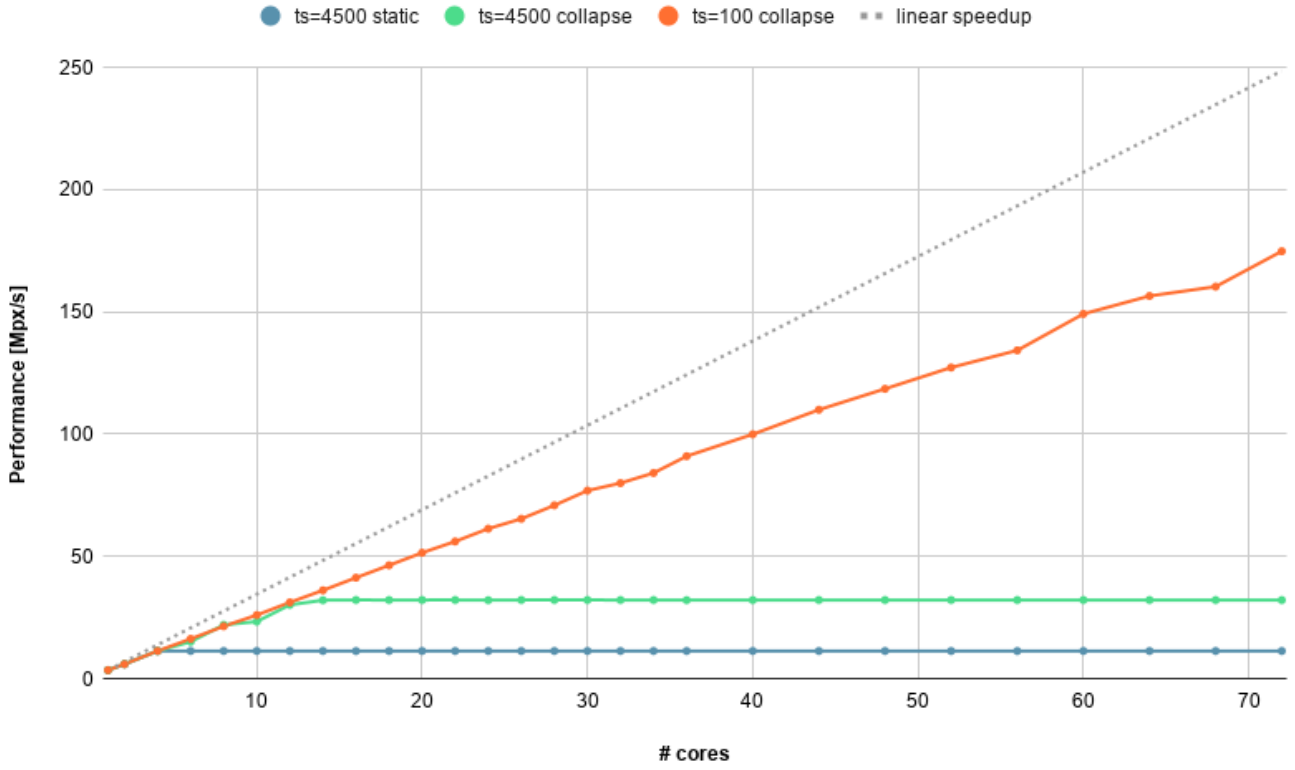
//..

count = 0;
#pragma omp for collapse(2) private(xc,i) reduction(+:count)
for(yc=0; yc<ytiles; yc++) {
  for(xc=0; xc<xtiles; xc++) {
     /* calc one tile */
     calc_tile(size, xc*tilesize, yc*tilesize, tilesize, tile);
     /* copy to picture buffer */
     for(i=0; i<tilesize; i++) {
         int tilebase = yc * tilesize * tilesize * xtiles + xc * tilesize;
         memcpy((void*)(picture + tilebase + i * tilesize * xtiles),
                (void*)(tile + i * tilesize),
                tilesize * sizeof(char));
     }
     count++;
   }
}
}
```

Assignment 7 – Task 3



Assignment 7 – Task 3



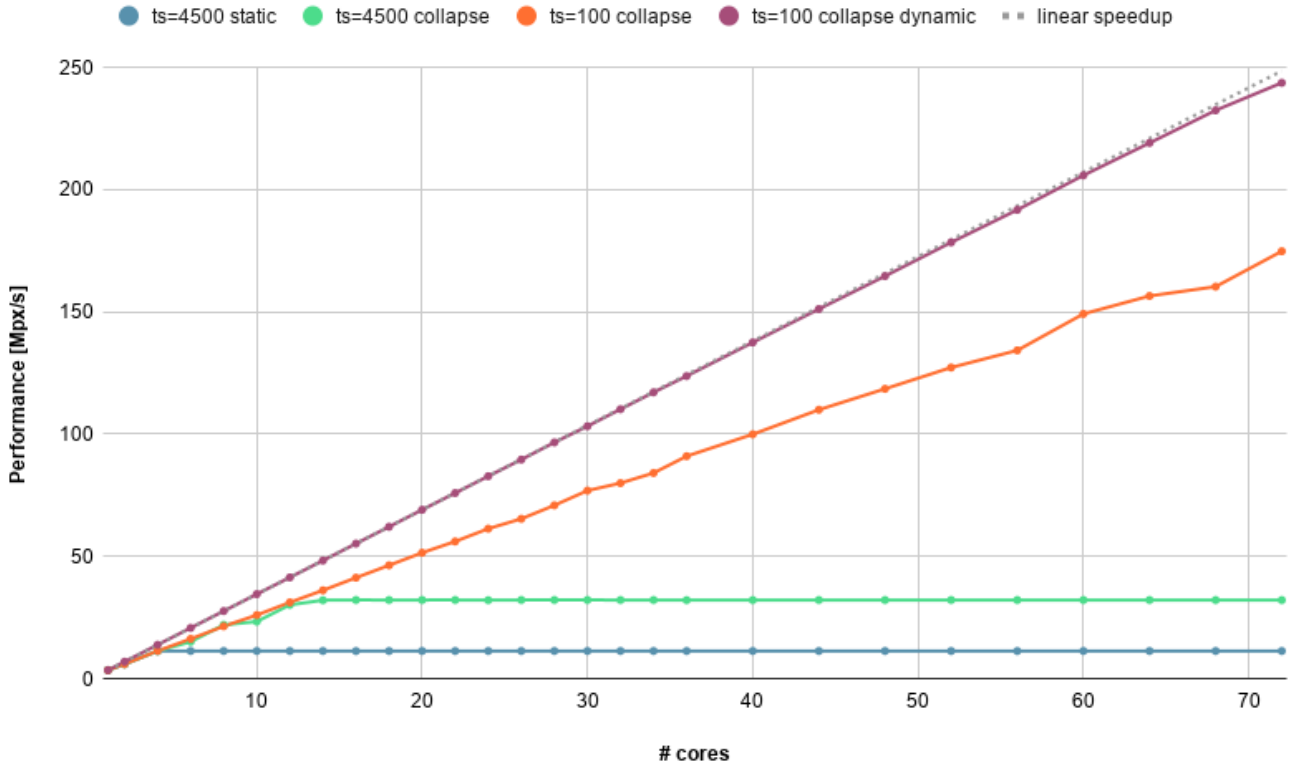
Assignment 7 – Task 3

```
// ...
#pragma omp parallel private(tile)
{
tile = (char*) malloc(tilesize * tilesize * sizeof(char));

//..

count = 0;
#pragma omp for schedule(dynamic) collapse(2) private(xc,i) reduction(+:count)
for(yc=0; yc<ytiles; yc++) {
  for(xc=0; xc<xtiles; xc++) {
     /* calc one tile */
     calc_tile(size, xc*tilesize, yc*tilesize, tilesize, tile);
     /* copy to picture buffer */
     for(i=0; i<tilesize; i++) {
         int tilebase = yc * tilesize * tilesize * xtiles + xc * tilesize;
         memcpy((void*)(picture + tilebase + i * tilesize * xtiles),
                (void*)(tile + i * tilesize),
                tilesize * sizeof(char));
     }
     count++;
   }
}
}
```

Assignment 7 – Task 3



Assignment 7 – Task 3 b)

Does barrier hurt?

- The barrier cost on the full node (72 cores) is about 10000 cycles (or 50000) ➔ 4.2 µs (21 µs) at 2.4 GHz
- The whole loop takes ≈ 1.3 s with 18000x18000 pixels on 72 cores
- One barrier for whole loop
- Therefore, barrier cost is negligible