Assignment 6 – Task 1

```
double x,y,tmp;
int i;

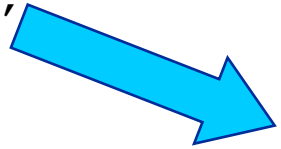
#pragma omp parallel shared(x) private(tmp)
{
  #pragma omp for reduction(+:x) nowait
  for(i=1; i< 100; i++){
    tmp=work1(i);
    x=x+tmp;
  }
  y = x;
}/* end parallel*/
```

Executed by all threads – speeders get wrong x

Assignment 6 – Task 2
Recurrence elimination

- Observation: recurrence is only “pseudo” as array contents can be computed from i directly:

```
for(i=1;i<N;i++) {
  b[i]=1+i;
  c[i]=b[i-1]+i;
  d[i]=c[i-1]+i;
}
```



```
#pragma omp parallel for
for(i=1;i<N;i++) {
  b[i]=1+i;
  c[i]=2*i;
  d[i]=3*i-2;
}
```

Assignment 6 – Task 3

- Eliminate recursion:

```
const double up = 1.00001;
double Sn, origSn=1.0;
double opt[N+1];
int n, lastn;

lastn = -2;
#pragma omp parallel for \
  firstprivate(lastn) lastprivate(Sn)
for (n=0; n<=N; ++n) {
  if (lastn != n - 1)
    Sn = origSn * pow(up, n);
  else
    Sn *= up;
  opt[n] = Sn;
  lastn = n;
}
Sn *= up;
```

Assignment 6 – Task 4

- Pi calculation using a Monte Carlo method and OpenMP

```
cout << "No. of threads: " << numthreads=omp_get_max_threads() << endl;
double rcp = 1./RAND_MAX;
long sum=0;

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

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

}

   double pi = (4.0 * sum) / nIterations;
```

- Performance / accuracy results
  - 10 threads @ 1 sec runtime (2.2 GHz), iterations = 6.2×10^8: relative error = 2.3×10^{-6}
  - Perfect scaling 1→10 threads

Assignment 6 – Task 4

- **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 → no additional statistics, strange random number sequences
  - 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?)
  - Watch your data types – when handling large counts, using `long` may be mandatory
      ```

Assignment 6 – Task 4

- General advice
  - Take care with random numbers; use the available range as far as possible
  - Random seeds should not be too close together (empirical experience)
    - Factor of 2 in accuracy for choosing `seed=6*myID` instead of `myID` alone
    - Correct way: use parallel RNG, by fast-forwarding seeds
  - It is easier to make thread-local automatic variables inside the parallel region than to privatize globals
  - Keep “expensive” divide and sqrt operations out of inner loop