Parallel Programming
Using OpenMP

Tor Sørevik
University of Bergen

Winter School in
Computational Mathematics,
Geilo 27/3 2001


## Outline

1. SMP programming philosophy
2. Directive based OpenMP
   (a) Work sharing constructs
   (b) Data environment constructs
   (c) Synchronization constructs
   (d) Environment variables
   (e) An example
3. Explicit Multi threaded programming with OpenMP
   (a) Environment routines
   (b) Examples
4. Tricks of the trade for efficient OpenMP programming


## Work partition vs Data partition

Parallel programming is about distributing work between cooperating processors.

This can be done implicitly by distributing the data and letting the different processors work on their slice of the data.

When memory is physically distributed you HAVE to do data partition. Synchronization and collaboration is done by exchanging data. (= Message passing)

With a global accessible memory, you might assume all data is available, forget about data partition and zoom in on work partition.

THIS IS THE CORE OF SMP PROGRAMMING.


## Why SMP programming

The sales pitch:

- When inspecting (sequential) code you can (in many case) determine which operations are independent. Thus work sharing is potentially easier to produce parallelizing compilers for.
- Why bother distributing data on a shared memory system?
- You can parallelize your code incrementally
- There is only one code to maintain. The sequential and parallel code is the same.

But don't forget:

- Not all parallel system are shared memory. Thus Message passing applies to a larger class of parallel systems.
- There is no true UMA (Uniform Memory Access) system. This makes data locality crucial for performance. Message passing gives better control over data distribution.

## OpenMP History

- Introduced October 1997
- No new features. Only meant to standardize current practice.
- Designed to be a De facto industry standard. (and it is!)
- Supported by all major vendors (and controlled by the vendors!)
- "lean and mean". ⇒ A small set of directives and only those that everyone agrees on.
- Available for Fortran (version 2.0 released Nov. 2000) and C/C++(version 1.0 released Oct. 1998)
- Supported on all major platforms!

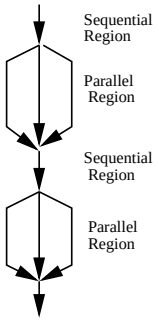

## The "fork and join" model

All OpenMP programs are started on ONE thread.

When an OpenMP directive or a call to an OpenMP function is found multiple threads are created (forking).

The original thread becomes the master thread.

When the parallel region ends. The execution continues sequentially on the master thread.




**NB! Forking and Joining is not without cost.**


## Parallel regions

The magic directive:

FORTRAN version:

```
!$OMP PARALLEL [clause,..]
      A block of code to be executed in parallel
!$OMP END PARALLEL
```

C/C++ version:

```
#pragma omp parallel [clause,..]  {
        A block of code to be executed in parallel
}
```

NOTE: When compiled without parallelization switch, the directives are ignored.


## Work sharing constructs

Do-loops:

```
!$OMP PARALLEL
!$OMP DO
      DO I = 1,N
         A(I) = A(I) + B(I)
      ENDDO
!$OMP END DO ! Optional
!$OMP END PARALLEL
```

Fortran-90 Array syntax:

```
!$OMP WORKSHARE
      A = A + B
!$OMP END WORKSHARE
```

## More Work sharing constructs

Sections:

```
!$OMP SECTIONS
!$OMP SECTION
      CALL SUB1 (A)
!$OMP SECTION
      CALL SUB2 (B)
!$OMP SECTION
      CALL SUB3 (C)
!$OMP END SECTIONS
```


## Variants Work sharing constructs

Multiple constructs:

```
!$OMP PARALLEL
!$OMP WORKSHARE
      A = C + B
!$OMP END WORKSHARE
!!
!!  No need to join and fork the threads
!!
!$OMP DO
      DO I = 1,N
         A(I) = A(I) + B(I)
      ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

Short forms:

```
!$OMP PARALLEL DO
      DO I = 1,N
         A(I) = A(I) + B(I)
      ENDDO
!$OMP END PARALLEL DO
```


## Data scope clauses

What if we do?

```
!$OMP PARALLEL DO
      DO I = 1,N
         TMP = A(I) * C/B(I)
         A(I) = A(I) + TMP
      ENDDO
!$OMP END PARALLEL DO
```

Each thread needs its unique `TMP`!!

Variables which needs to be unique to each thread must be specified as `PRIVATE`

Variables which is shared either because they are read-only or because they are arrays which updates are parallelized must be specified as `SHARED`

```
!$OMP PARALLEL DO PRIVATE(I,TMP),SHARED(A,B,C)
      DO I = 1,N
         TMP = A(I) * C/B(I)
         A(I) = A(I) + TMP
      ENDDO
!$OMP END PARALLEL DO
```

## More Data scope clauses

`LASTPRIVATE` extends `PRIVATE`. Keeps a well defined copy of a `PRIVATE` variable at exit.

```
!$OMP PARALLEL DO PRIVATE(I,J), SHARED (A,C)
!$OMP& LASTPRIVATE(X)
      DO I = 1,N
         X = C(I)**2
         DO J = 1,N
            A(I,J) = A(I,J) + C(I)*X
         ENDDO
      ENDDO
!$OMP END PARALLEL DO
PRINT*,X !!  Will print X = C(N)**2
```

## ..and more Data scope clauses

`FIRSTPRIVATE` extends `PRIVATE`. Initialize all copies of `PRIVATE` variables.

```
!$OMP PARALLEL DO PRIVATE(I, J), SHARED (A,B)
!$OMP& FIRSTPRIVATE(Y)
      DO I = 1,N
         DO J = 1,N
            Y = B(I,J) + Y
            A(I,J) = A(I,J) -Y
         ENDDO
      ENDDO
!$OMP END PARALLEL DO
```

## Default data scope

The default scope is `SHARED`
The exception for this is do-loop control variables.
The default might be overwritten by the `DEFAULT` - clause

```
!$OMP PARALLEL DO DEFAULT(PRIVATE),SHARED(A,B,C)
      DO I = 1,N
         TMP = A(I) * C/B(I)
         A(I) = A(I) + TMP
      ENDDO
!$OMP END PARALLEL DO
```



## The Reduction clause

```
!$OMP PARALLEL DO REDUCTION(+:  A_SUM, B_SUM)
      DO I = 1,N
         A_SUM = A_SUM + A(I)
         B_SUM = B_SUM + B(I)
      ENDDO
!$OMP END PARALLEL DO
```

The following operators and intrinsic functions are allowed:
+, -, *, .AND., .OR., .EQV., .NEQV.
MAX, MIN, IAND, IOR, IEOR

## Synchronization constructs

Suppose we have:

```
DO I = 1,N
   XTMP = FUN1(I) ! Mflops computing
   X(INDEX(I)) = X(INDEX(I)) + XTMP
ENDDO
```

When parallelizing, we might create write conflict if different instance of `I` gives same index for `X`. We do:

```
!$OMP PARALLEL DO PRIVATE(XTMP),SHARED(X,INDEX)
      DO I = 1,N
         XTMP = FUN1(I) ! Mflops computing
!$OMP ATOMIC
         X(INDEX(I)) = X(INDEX(I)) + XTMP
      ENDDO
!$OMP END PARALLEL DO
```

## more synchronization constructs

If we have a block of code need to be computed in a given order:

```
!$OMP PARALLEL DO PRIVATE(XTMP),SHARED(X,INDEX)
      DO I = 1,N
         XTMP = FUN1(I) ! Mflops computing
!$OMP CRITICAL
         X(INDEX(I)) = X(INDEX(I)) + XTMP
!$OMP END CRITICAL
      ENDDO
!$OMP END PARALLEL DO
```

`ATOMIC` applies only to the next statement, while `CRITICAL` applies to a block of code.


## More synchronization constructs

```
!$OMP PARALLEL
!$OMP DO
      Block of code to be executed in parallel
!$OMP END DO [NOWAIT]
!$OMP SINGLE
      BoC to be executed by one thread only
!$OMP END SINGLE
!$OMP BARRIER
!$OMP DO
      BoC to be executed in parallel
      !$OMP CRITICAL
            BoC to be executed sequentially
      !$OMP END CRITICAL
!$OMP END DO [NOWAIT]
!$OMP MASTER
      BoC to be executed by the master thread
!$OMP END MASTER
!$OMP DO
      BoC to be executed in parallel
!$OMP END DO
!$OMP END PARALLEL
```


## Load balancing

```
!$OMP PARALLEL DO PRIVATE(I,J),SHARED(X,L,Y)
      DO I = 1,120
         X(I) =0
         DO J = I,N
            X(I) = X(I) + L(I,J)* Y(J)
         ENDDO
      ENDDO
!$OMP END PARALLEL DO
```

The default scheduling (called `STATIC`) gives on 4 processors:

Thread 0: Does I =1,...,30 $\Rightarrow$ 6330 flops
Thread 1: Does I =31,...,60 $\Rightarrow$ 4530 flops
Thread 2: Does I =61,...,90 $\Rightarrow$ 2730 flops
Thread 3: Does I =91,...,120 $\Rightarrow$ 930 flops


## OTHER SCHEDULINGS

`SCHEDULE(STATIC,[chunk])`:
Divides the iteration in slices of size `chunk` and distribute these cyclic.
Default `chunk` = N/P.

`SCHEDULE(DYNAMIC,[chunk])`:
Divides the iteration in slices of size `chunk`. Each thread starts with the same chunk as in `STATIC,[chunk]`, but the remaining `chunk`'s are processed at a first-come-first-served bases.
Default `chunk` = 1.

`SCHEDULE(GUIDED,[chunk])`:
The size of the `chunk`'s increase exponentially (!?). Default first `chunk` = 1.

The scheduling can be defined as a clause to `!$OMP DO`

## Environment Variables

The scheduling can also be set by an environment variable

```
setenv OMP_SCHEDULE "STATIC, 4"
```

```
setenv OMP_SCHEDULE "DYNAMIC"
```

This is also the best way to set the number of threads

```
setenv OMP_NUM_THREADS 16
```


## Explicit Multi threading

A set of functions to be called from your parallel program is available:

`OMP_SET_NUM_THREADS`: Sets the number of threads to be used in subsequent parallel regions

`OMP_GET_NUM_THREADS`: Returns the number of threads in the team executing the parallel region

`OMP_GET_MAX_THREADS`: Returns the maximum value that can be returned by `OMP_GET_NUM_THREADS`

`OMP_GET_THREAD_NUM`: Returns the thread number of the thread. A number between 0 .. OMP_GET_NUM_THREADS - 1

`OMP_GET_NUM_PROCS` : Returns the number of processors available to the program.


## Ex: Explicit Multi threading

Example:

```
!$OMP PARALLEL
      NTHREADS = OMP_GET_NUM_THREADS ()
      BLOCK = N/NTHREADS
      IF (MOD(N,NTHREADS).NE.0)BLOCK=BLOCK+1
      IAM = OMP_GET_THREAD_NUM()
      DO I = 1+IAM*BLOCK, MIN((IAM+1)*BLOCK,N)
         A(I) = A(I) + B(I)
      ENDDO
!$OMP END PARALLEL
```

## Ex: 1D Poisson equation

```
.......
dx = 1.0/(n+1);
x = 0.0;
for (i = 0; i<=n+1; i++) {
    x += dx;
    rhs[i] = dx*dx*PI*PI*sin(PI*x);
    x_k[i] = x_k1[i] = 0.0;
}
base = norm(n, rhs);
r_norm = 1.0;
i = 0;
/* main loop */
while (i<1000 && (r_norm/base) > 1.0e-4) {
    i++;
    jacobi_iteration(n, rhs, x_k1, x_k);
    calc_residual(n, rhs, x_k1, res);
    r_norm = norm(n,res);
    for (j = 1; j<=n; j++) x_k1[j] = x_k[j];
}
free (x_k); free (x_k1); free (rhs); free (res);
return 0;
```

## Ex continues: Functions

```
void jacobi_iteration (int n, double* rhs,
     double* x_k1, double* x_k){
     int i;
     for (i = 1; i<=n; i++)
         x_k[i] = 0.5*(rhs[i]+x_k1[i-1]+x_k1[i+1]);
}
void calc_residual (int n, double* rhs,
     double* x, double* res){
     int i;
     for (i = 1; i<=n; i++)
         res[i] = rhs[i]+x[i-1]-2*x[i]+x[i+1];
}
double norm(int n, double* x){
     double sum = 0.0;
     int i;
     for (i = 1; i<=n; i++)
         sum += x[i]*x[i];
     return sqrt(sum);
}
```


## OpenMP1; Simple directives

```
void jacobi_iteration (int n, double* rhs,
     double* x_k1, double* x_k){
     int i;
     #pragma omp parallel for private(i)
     for (i = 1; i<=n; i++)
         x_k[i] = (rhs[i]+x_k1[i-1]+x_k1[i+1])/2;
}
void calc_residual (int n, double* rhs,
     double* x, double* res){
     int i;
     #pragma omp parallel for private(i)
     for (i = 1; i<=n; i++)
         res[i] = rhs[i]+x[i-1]-2*x[i]+x[i+1];
}
double norm(int n, double* x){
     double sum = 0.0;
     int i;
     #pragma omp parallel for reduction(+:sum)
     for (i = 1; i<=n; i++)
         sum += x[i]*x[i];
     return sqrt(sum); }
```

## OpenMP1; Simple directives, comments

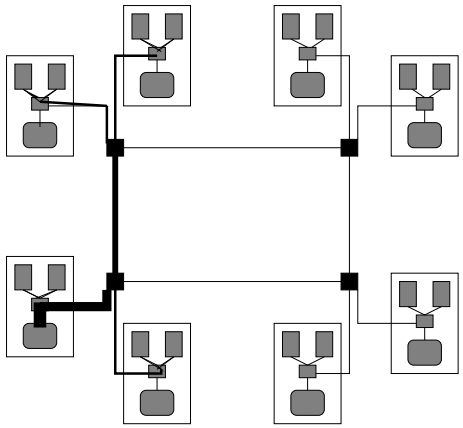
**Problem 1:** Threads are created 4 times in every iteration!

We would like to set `#pragma omp parallel` in the main routine and only `#pragma omp for` in front of each loop of the subroutine.
But than each thread is making its individual call to the different functions.
In that case all local variables within the functions are **private** to the calling thread. This prohibits reduction on `sum`!!

**Problem 2:** Data distribution! This is a cc-NUMA problem.
When memory is physical distributed it is important that data is allocated physically close to the processor using it.


## Memory bottleneck in cc-NUMA

This is a 16 CPU Origin 2000 system. 8 CPUs are fetching data from the same memory location.



The Origin 2000's policy for data layout is: *first touch policy*
Thus sequential initialization puts all data in one memory module!!
**Solution: Parallelize initialization**

## OpenMP; Explicit Threading

```
#pragma omp parallel private(id,j,iter,step) {
threads = omp_get_max_threads();
id = omp_get_thread_num();
n_del = (n+1)/threads;
for (i = 0; i<=threads-1; i++) {
   ibeg[i] = i*n_del + 1;
   iend[i] = (i+1)*n_del; }
#pragma omp barrier
while (iter++<1000 && (r_norm[0]/base)>1.0e-4){
   jacobi_iteration(id, ibeg, iend, n, rhs, ...);
   calc_residual(id, ibeg, iend, n, rhs, ...);
   r_norm[id] = norm(id, ibeg, iend, n, res);
   for (step = 1; step < threads; step *=2){
   #pragma omp barrier
       if (id%(step*2) == 0)
          r_norm[id] += r_norm[id+step]; }
   if (id == 0) r_norm[id] = sqrt(r_norm[id]);
   for (j = ibeg[id]; j<=iend[id]; j++)
       x_k1[j] = x_k[j];
   #pragma omp barrier
}
```

## OpenMP; Explicit Threading

```
void jacobi_iteration(int id, int* ibeg, int* iend,
     int n, double* rhs, double*x_k1, double* x_k)
{
     int i;
     for (i = ibeg[id]; i<=iend[id]; i++)
         x_k[i] = 0.5*(rhs[i]+x_k1[i-1]+x_k1[i+1]);
}
void calc_residual(int id, int* ibeg, int* iend,
     int n, double* rhs, double* x, double* res){
     int i;
     for (i = ibeg[id]; i<=iend[id]; i++)
         res[i] = rhs[i]+x[i-1]-2*x[i]+x[i+1];
}
double norm(int id, int* ibeg, int* iend, int n,
     double* x){
     double sum = 0.0;
     int i;
     for (i = ibeg[id]; i<=iend[id]; i++)
         sum += x[i]*x[i];
     return sum;/* NB only a local sum!*/
}
```


## Speed up results

The Jacobi-iteration for 1-D Poisson equation.
N = 99999

| No of CPUc | Directives Seq. Init | Directives Par. Init | Explicit Multi threading |
|---|---|---|---|
| 1 | 1.00 | 1.00 | 1.00 |
| 2 | 1.95 | 2.01 | 2.05 |
| 4 | 3.88 | 3.98 | 3.99 |
| 8 | 6.62 | 7.50 | 7.60 |
| 16 | 11.50 | 12.30 | 14.30 |


## Auto parallelizers

There are tools which inspect a code, finds parallel construct and inserts directives.

Most vendors have bundled this with their compiler.

+ Saves a lot of work
+ What they, they do right (Put the right variables in `PRIVATE` or `SHARED` clauses.
+ Gives good hints on why a specific loop is not parallelizable
- Can't handle subroutine calls within a loop
- Don't know when

```
DO I = 1,N
    A(INDEX(I)) = A(INDEX(I)) + B(I)
ENDDO
```

Good strategy: Start with auto parallelizer, continue with inserting directives.

## Tricks of the trade

- Parallel bugs are much more difficult to find than sequential. Therefor: **Do your parallelization step by step**
- Try always to parallelize **outer loop.**
- For efficiency you have to be aware of **data access**. But in OpenMP you don't have the same explicit control over it as in MPI.
- Parallelization directives very often prohibits the compiler to **do sequential optimization**. (Interchanging nested loops, loop fission, ...)
- Watch out for **false sharing**. i.e. Two (or more) CPUs are updating different data on the same cache line.

Exercise: Interchange the loops in the Triangular matrix-vector code and parallelize it.