Introduction to High Performance Computing

OpenMP

non iterative work sharing

```
#pragma omp sections [clause]
{
  #pragma omp section
        block
 #pragma omp section
        block
  ……
}
```

Possible clauses:

firstprivate(list)	initialization
lastprivate(list)	value of last iteration is copied to master thread
private(list)	variables are treated as private
reduction(op:list)	declares reduction variable and operation
nowait	no barrier at end

sections

Sections may be used to exploit functional decomposition into parallel work. It is static in it‘s use. Example:

```
#pragma omp parallel
{
   #pragma omp sections
  {
     #pragma omp section
        functionA;
     #pragma omp section
         functionB;
  }  /* end of sections */
}  /* end of parallel */
```

specific work sharing

```
#pragma omp single[clause]
{
}
```

One of the threads in the team – not necessarily the master thread – is executing the subsequent block.
Possible clauses:

firstprivate(list)	initialization
private(list)	variables are treated as private
nowait	no barrier at end

specific work sharing

#pragma omp parallel for

#pragma omp parallel sections

Combine work sharing with forking into a parallel region

vectorization

```
#pragma omp simd[clause]
 for-loops
```

Loop may be vectorized.
Possible clauses:

aligned(list:al)	alignment of variable (list) at adress (al)
collapse(n)	extend to n nested loops
lastprivate(list)	value of last iteration is copied to master thread
linear(list:step)	list is treated as private vector
private(list)	variables are treated as private
reduction(op:list)	declares reduction variable and operation
safelen(len)	maximal vectorization degree
simdlen(len)	preferred vectorization degree

example for omp simd

```
#define N 1000000
float x[N][N], y[N][N];
#pragma omp parallel
{
 #pragma omp for
  for (int i=0; i<N; i++) {
  #pragma omp simd safelen(18)
  for (int j=18; j<N−18; j++) {
    x[i][j] = x[i][j−18] + sinf(y[i][j]);
    y[i][j] = y[i][j+18] + cosf(x[i][j]);
}}}
```

assures no recursion as long as vector length is below 19

vectorization

```
#pragma omp declare simd[clause]
 function
```

Function may be called in a vectorized loop.
Possible clauses:

aligned(list:al)	alignment of variable (list) at adress (al)
inbranch	called in a branch
linear(list:step)	list is treated as private vector
simdlen(len)	preferred vectorization degree
uniform(list)	declares variable not being a vector

vectorization

```
#pragma omp declare simd
float min(float a, float b) {
return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
for (i=0; i<N; i++) {
d[i] = min(distsq(a[i], b[i]), c[i]);
} }
```

Function call – not vectorizable, no AVX

vectorization

```
#pragma omp declare simd
float min(float a, float b) {
return a < b ? a : b;
}
float *vec_distsq(int n,float *x, float *y) {
 if(n==1) distsq(x[0],y[0]);
for(int i=0;i<n;i++) {
 float tmp[i]=(x[i]-y[i])*(x[i]-y[i])
}
return *tmp;}
void example() {
#pragma omp parallel for simd
for (i=0; i<N; i++) {
d[i] = min(distsq(a[i], b[i]), c[i]);
} }
```

vectorization

Usage of declare SIMD :

- Called in outer loop
- loop split into SIMD-loop and iteration with simdlen
- loop reorganized – pre/main/post part
- SIMD loop moved into function
- compiler introduces alternative function with loop inside
- inside loop is vectorized – if possible

sharing unstructured work

```
#pragma omp task [clause …]
```

Possible clauses:

if(scalar_exp)	execute task immediately if false, else the task may be executed later by another thread (deferred)
depend(type:list)	dependencies to respect flow graphs
final(scalar_exp)	final task if true
untied	if suspended, any other thread may resume this task
mergeable	an undeferred task may merge this task
priority(value)	the higher value, the sooner this task is executed.
private(list)	variables are treated as private
shared(list)	variables are treated as shared
firstprivate(list)	initialization

example task

```
#pragma omp parallel
{
#pragma omp single private(p)
{
while (p) {
#pragma omp task
{
processwork(p);
}
p = p->next;
}
}
}
```

team of threads with assumed 8 threads

one thread executes a while loop, rest does nothing

the single thread creates a variable number of tasks which may be executed by free threads in the team

sharing unstructured work

```
#pragma omp taskloop [clause …]
```

Possible clauses:

if(scalar_exp)	execute task immediately if false, else the task may be executed later by another thread (deferred)
grainsize(grain)	minimal number of loop iterations per task
num_tasks(num)	number of tasks to be generated.
collapse(n)	specifies how many loops are associated
	task clauses: priority, depend, untied, mergeable, final
	data clauses: shared, private, firstprivate, lastprivate

taskloop

The taskloop directive may be used to tackle unstructured and unbalanced loops. The iterations are cut into chunks according to grainsize and are assigned to tasks.

tasking

#pragma omp taskyield

task may be suspended in favor of a different task for optimization and/or deadlock prevention.

Example

```
#pragma omp parallel
{
#pragma omp single
{
  while (( dirp=readdir(d)) != NULL) {
      int i = strlen(dirp->d_name);
      if((i>2) && strcmp(&(dirp->d_name[i-2]),".o") == 0) {
           sprintf(fname,"%s/%s",path,dirp->d_name);
           istat = stat(fname,&sbuf);
           isize = (int) sbuf.st_size;
#pragma omp task private(fd) untied if(1)\
        firstprivate(fname,ibuf,isize) shared(MEMBUF)
           {
              fd=open(fname,O_RDONLY);
              read(fd,&(MEMBUF[ibuf],isize);
              close(fd);
           } // end of task
           ibuf+=(isize+63) & 0xffffffc0;
      } }
} // end of single region
} // end of parallel region
```

Example

```
#pragma omp parallel
#pragma omp single
{
 while (( dirp=readdir(d)) != NULL) {
      int i = strlen(dirp->
      if((i>2) && strcmp(&(dirp->d_name[i-2]),".o") == 0)
{
#pragma omp task private(fd) untied if(1)\
        firstprivate(fname,ibuf,isize) shared(MEMBUF)
{
                fd=open(fname
                read(fd,&(MEM
                close(fd);
             } // end of task
           ibuf+=(isize+63)
} // end of single region
} // end of parallel region
```

Fork into a team of threads

Execute with a single thread of the team

Save file characteristics
Setup a deferred task, resumable by any thread

open, read the entire file
and close it

update (single) location in memory

master and synchronization

#pragma omp master

executed by the master thread only – no implied barrier at entry or end of the block.

#pragma omp barrier

All threads and explicit tasks must execute the barrier until anyone may proceed beyond.

#pragma omp flush

explicit synchronization of the cache of the encountering thread

master and synchronization

#pragma omp taskwait

All child tasks before the taskwait are completed.

#pragma omp taskgroup

All child tasks and their descendent tasks are completed.

master and synchronization

#pragma omp atomic read | write | update | capture

Exclusive access to a storage location for different operations. A capture is an update of a variable plus writing the update result to a different place.
The storage location is implicitely flushed afterwards.

#pragma omp cancel

Allows leaving of innermost enclosing region.

#pragma omp critical

Exclusive access to the following block.

Examples

Numerical integration:

$$\int_0^1 F(x)dx = \int_0^1 \frac{4,0}{1+x^2}dx = \pi$$

approximated by a finite sum:

$$\sum_{i=0}^{N} F(x_i)\Delta x$$

Examples

```
#ifdef _OPENMP
#include <omp.h>
#endif
#define intervals 10000
void main(int argc, char **argv) {
 int i;
 double step,x,pi,sum;
 step=1./Intervalle;
 /* Fork and work sharing combined */


   for(i=0,sum=0.;i<intervals;i++) {
     x=(i+0.5)*step;
     sum+=4.0/(1.0+x*x);
   } /* end of parallel region */
 pi=step*sum;
}
```

Examples

```
#ifdef _OPENMP
#include <omp.h>
#endif
#define intervals 10000
void main(int argc, char **argv) {
 int i;
 double step,x,pi,sum;
 step=1./Intervalle;
 /* Fork and work sharing combined */
#pragma omp parallel for private(x,i) shared(sum)
for(i=0,sum=0.;i<Intervalle;i++) {
     x=(i+0.5)*step;
#pragma omp atomic update
     sum+=4.0/(1.0+x*x);
   } /* end of parallel region */
 pi=step*sum;
}
```

Will do, but very very slow
All iterations are synchronized
memory access is serialized

Examples

```
#define intervals 10000
void main(int argc, char **argv) {
 int i;
 double step,x,pi,sum=0.,psum;
 step=1./Intervalle;
 /* Fork and work sharing combined */
#pragma omp parallel shared(sum)
{
#pragma omp for private(x,i,psum) nowait
for(i=0,psum=0.;i<Intervalle;i++) {
     x=(i+0.5)*step;
     psum+=4.0/(1.0+x*x);
   }
#pragma omp critical
   sum+=psum;
} /* end of parallel region */
 pi=step*sum;
}
```

Fine grain synchronization

```
omp_init_lock
omp_destroy_lock
omp_set_lock
omp_unset_lock
omp_test_lock

omp_init_nest_lock
omp_destroy_nest_lock
omp_set_nest_lock
omp_unset_nest_lock
omp_test_nest_lock
```

Examples

```
 omp_lock_t lck;
 omp_init_lock(&lck);
 step=1./Intervalle;
 /* Fork and work sharing combined */
#pragma omp parallel shared(sum)
{
#pragma omp for private(x,i,psum) nowait
for(i=0,psum=0.;i<Intervalle;i++) {
     x=(i+0.5)*step;
     psum+=4.0/(1.0+x*x);
   }
   omp_set_lock(&lck);
   sum+=psum;
   omp_unset_lock(&lck);
} /* end of parallel region */
 omp_destroy_lock(&lck);
 pi=step*sum;
```

Examples

```
 omp_lock_t lck;
 omp_init_lock(&lck);
 step=1./Intervalle;
 /* Fork and work sharing combined */
#pragma omp parallel shared(sum)
{
#pragma omp for private(x,i,psum) nowait
for(i=0,psum=0.;i<Intervalle;i++) {
     x=(i+0.5)*step;
     psum+=4.0/(1.0+x*x);
   }
   omp_set_lock(&lck);
   sum+=psum;
   omp_unset_lock(&lck);
} /* end of parallel region */
 omp_destroy_lock(&lck);
 pi=step*sum;
```

get the key to access sum . If the key is not available – wait.

Examples

```
 omp_lock_t lck;
 omp_init_lock(&lck);
 step=1./Intervalle;
 /* Fork and work sharing combined */
#pragma omp parallel shared(sum)
{
#pragma omp for private(x,i,psum) nowait
for(i=0,psum=0.;i<Intervalle;i++) {
     x=(i+0.5)*step;
     psum+=4.0/(1.0+x*x);
   }
   omp_set_lock(&lck);
   sum+=psum;
   omp_unset_lock(&lck);
} /* end of parallel region */
 omp_destroy_lock(&lck);
 pi=step*sum;
```

Release the key and continue.

Not covered

Extension of OpenMP to accelerator cards

- target directive
- data management with accelerators

User defined reduction

- not fully implemented yet

Lots of runtime get and set functions

Einführung in das Hochleistungsrechnen

Introduction to High Performance Computing

VIELEN DANK
THANK YOU