OPENMP

Course “Parallel Computing”

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at
OpenMP (OMP)

- An API for portable shared memory parallel programming.
  - Compiler directives (pragmas), library routines, environment variables.
- Targets are C, C++, Fortran.
  - Often used in combination with MPI (Message Passing Interface) for hybrid MPP/SMP programs.
- Widely supported.
  - Commercial compilers: Intel, IBM, Oracle, . . .
  - Free compilers: GCC, Clang.
- Maintained by the OpenMP ARB.
  - Architecture Review Board.

Master thread executes program in sequential mode.

- Reaches code section marked with OMP directive:
  - Execution of section is distributed among multiple threads.
  - Main thread waits for completion of all threads.
  - Execution is continued by main thread only.

A fork-join model of parallel execution.
Shared versus Private Variables

The default context of a variable is determined by some rules.

- Static variables and heap-allocated data are shared.
- Automatically allocated variables are
  - Shared, when declared outside a parallel region.
  - Private, when declared inside a parallel region.
- Loop iteration variables are private within their loops.
  - After the loop, the variable has the same value as if the loop would have been executed sequentially.

... OpenMP clauses may specify the context of variables directly.
Controlling the Number of Threads

- Default set by environment variables:
  ```
  export OMP_DYNAMIC=FALSE
  export OMP_NUM_THREADS=4
  ```

- May be overridden for all subsequent code sections:
  ```
  omp_set_dynamic(0);
  omp_set_num_threads(4);
  ```

- May be overridden for specific sections:
  ```
  #pragma omp parallel ... num_threads(4)
  ```

If dynamic adjustment is switched on, the actual number of threads executing a section may be smaller than specified.
Controlling the Affinity of Threads to Cores

- Pin threads to cores:
  ```
  export OMP_PROC_BIND=TRUE
  ```

- Specify the cores (GCC, Intel Compilers):
  ```
  export GOMP_CPU_AFFINITY="64-127:2"
  ```

- More flexible alternative for Intel compilers:
  ```
  export KMP_AFFINITY="verbose,granularity=core,explicit,proclist=[64-127:2]"
  ```
Compiling and Executing OpenMP

- **Source**
  
  ```cpp
  #include <omp.h>
  ```

- **Intel Compiler:**
  
  ```bash
  module load intelcompiler
  icc -Wall -O3 -openmp -openmp-report2 matmult.c -o matmult
  ```

- **GCC:**
  
  ```bash
  module load GnuCC
  gcc -Wall -O3 -fopenmp matmult.c -o matmult
  ```

- **Execution:**
  
  ```bash
  export OMP_DYNAMIC=FALSE
  export OMP_NUM_THREADS=32
  export GOMP_CPU_AFFINITY="64-127:2"
  ./matmult
  ```
Parallel Loops

```c
#pragma omp parallel for private(j,k)
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        for (k=0; k<N; k++) {
            a[i,j] += b[i,k]*c[k,j];
        }
    }
}
```

- Iterations of $i$-loop are executed by parallel threads.
- Matrix $a$ is shared by all threads.
- Every thread maintains private instances of $i, j, k$.

Most important source of scalable parallelism.
Load Balancing

#pragma omp parallel for ... schedule(kind [, chunk size])

- Various kinds of loop scheduling:
  - **static**: Loop is divided into equally sized chunks which are interleaved among threads; default chunk size is $N/T$.
    - Number of loop iterations $N$ and number of threads $T$.
  - **dynamic**: Threads retrieve chunks from a shared work queue; default chunk size is 1.
  - **guided**: Like “dynamic” but chunk size starts large and is continuously decremented to specified minimum (default 1).
  - **auto**: One of the above policies is automatically selected (same as if no schedule is given).
  - **runtime**: Schedule taken from environment variable `OMP_SCHEDULE`.
    - `export OMP_SCHEDULE="static,1"`
Example: Matrix Multiplication

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 2000
double A[N][N], B[N][N], C[N][N];

int main(int argc, char *argv[]) {
    int i, j, k;
    double s;

    for (i=0; i<N; i++)
    {
        for (j=0; j<N; j++)
        {
            A[i][j] = rand();
            B[i][j] = rand();
        }
    }

    printf("%f %f\n", A[0][0], B[0][0]);
    double t1 = omp_get_wtime();

    #pragma omp parallel for private(j,k,s) schedule(runtime)
    for (i=0; i<N; i++)
    {
        for (j=0; j<N; j++)
        {
            s = 0;
            for (k=0; k<N; k++)
            {
                s += A[i][k]*B[k][j];
            }
            C[i][j] = s;
        }
    }

    double t2 = omp_get_wtime();
    printf("%f (%f s)\n", C[0][0], t2-t1);
    return 0;
}
```
Parallel Sections

```c
int found1, found2, found3;

#pragma omp parallel sections
{
    #pragma omp section
    found1 = search1();
    #pragma omp section
    found2 = search2();
    #pragma omp section
    found3 = search3();
}

if (found1) printf("found by method 1\n");
if (found2) printf("found by method 2\n");
if (found3) printf("found by method 3\n");
```

- Each code section is executed by a thread in parallel.

Parallel sections and loops may be also nested.
int n, a[n], t, i;

#pragma omp parallel private(t, i)
{
    t = omp_get_num_threads(); // number of threads
    i = omp_get_thread_num(); // 0 <= i < t
    compute(a, i*(n/t), min(n, (i+1)*(n/t)));
}

- Every thread executes the annotated block.
- Array $a$ and length $n$ are shared by all threads.
- Every thread maintains private instances of $t$ and $i$.

Parallelism on the lowest level.
Critical Sections

```c
int n, a[n], t = 0, i;

#pragma omp parallel private(i)
{
    #pragma omp critical(mutex_i)
    {
        i = t; t++;
    }
    if (i < n) compute(a, i);
}
```

- No two threads can simultaneously execute a critical section with the same name.

High-level but restricted synchronization.
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 2000
double A[N][N], B[N][N], C[N][N];

int main(int argc, char *argv[])
{
    int i, j, k, row;
    double s;

    for (i=0; i<N; i++)
    {
        for (j=0; j<N; j++)
        {
            A[i][j] = rand();
            B[i][j] = rand();
        }
    }

    printf("%f %f\n", A[0][0], B[0][0]);
    double t1 = omp_get_wtime();

    row = 0;
    #pragma omp parallel private(i,j,k,s)
    {
        while (1)
        {
            #pragma omp critical(getrow)
            {
                i = row;
                row++;
            }
            if (i>=N) break;
            for (j=0; j<N; j++)
            {
                s = 0;
                for (k=0; k<N; k++)
                {
                    s += A[i][k]*B[k][j];
                }
                C[i][j] = s;
            }
        }
    }

    double t2 = omp_get_wtime();
    printf("%f (%f s)\n", C[0][0], t2-t1);
    return 0;
}
Lock Variables

```c
int n, a[n], t = 0, i;
omp_lock_t lock;
omp_init_lock(lock);

#pragma omp parallel private(i)
{
    omp_set_lock(lock);
    i = t; t++;
    omp_unset_lock(lock);
    if (i < n) compute(a, i);
}
```

- Only one thread can set a lock at a time.

Flexible but low-level synchronization.
#include <omp.h>

int compute(int*a, int begin, int end)
{
    int n = end-begin;
    if (n < 0) return;
    if (n == 1) return f(begin);
    int mid = (begin+end)/2;
    #pragma omp task shared(r1)
    int r1 = compute(a, begin, mid);
    #pragma omp task shared(r2)
    int r2 = compute(a, mid, end);
    #pragma omp taskwait
    return r1+r2;
}

Create two tasks and wait for their completion.

Task parallelism possible, but may become cumbersome.