ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Systems Lab

Stefan M. Freudenberger
Computer Systems, ETH Zürich

OpenMP

Open Multi-Processing

- API supporting multiplatform shared memory multiprocessing
- (Unix and Windows)
- Set of: compiler directives, libraries and environment variables
- Parallel code sections are executed in parallel using several threads and are managed by the runtime environment

Implementations

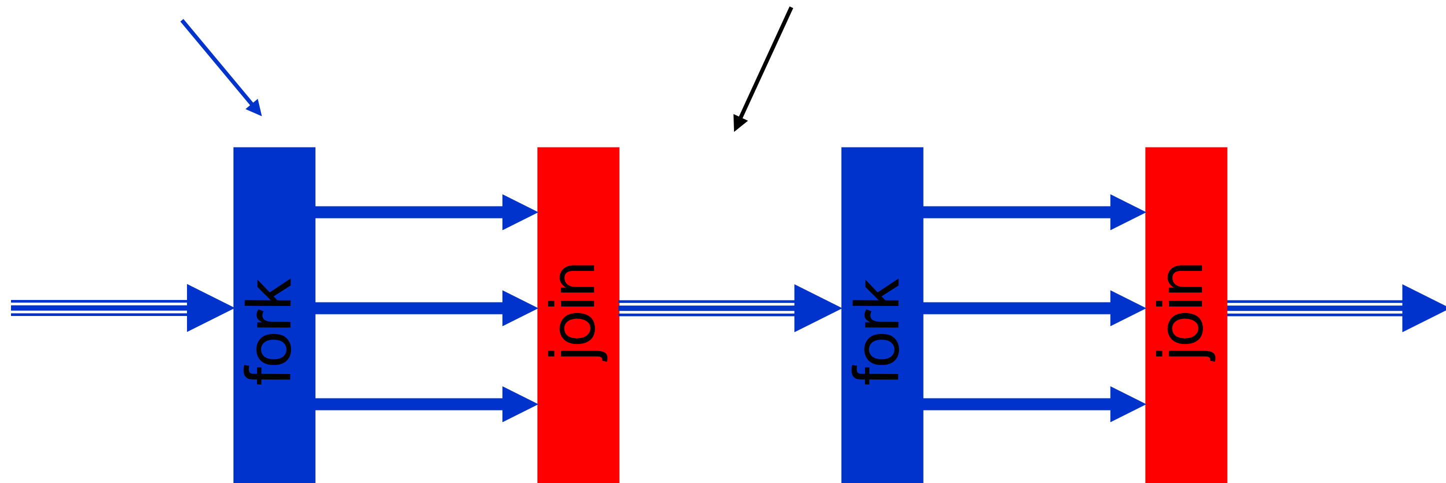
- Visual C++ 2005
- Intel compilers
- Sun Studio
- GCC 4.2 (or GCC 4.1 on some RH platforms)

OpenMP

- Advantages
  - Simple: need not deal with message passing
  - Automatic data layout and decomposition
  - Incremental parallelism
  - Unified code for both serial and parallel applications
- Disadvantages
  - Only runs efficiently in shared-memory multiprocessor platforms
  - Requires a compiler that supports OpenMP
  - Scalability is limited by memory architecture
  - Synchronization between a subset of threads is not allowed

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallelism in OpenMP



OpenMP Components

- Compiler directives
  - Creating teams of threads
  - Sharing the work among threads
  - Synchronizing the threads
- Library routines
  - To set and query thread attributes
- Environment variables
  - To control run-time behavior of the parallel program

OpenMP Directives

- C / C++ Syntax
  - OpenMP directives are expressed using *pragmas*:

```
#ifdef _OPENMP
#pragma omp directive [clause[[,]clause...]] new-line
#endif
```

  - Applies to the succeeding structured block or OpenMP construct
    - Can be single statement or compound statement
    - Must have single entry at top, single exit at bottom
- Also structured comments for FORTRAN

Parallelism in OpenMP

- The *parallel* construct forms a team of threads in an OpenMP program and starts parallel execution
  - A team of threads is created at run time for the parallel region
  - The work is shared among the threads
  - A nested parallel region is allowed
    - May contain a team of one thread
  - Nested parallelism is enabled with
    `setenv OMP_NESTED TRUE`

```
#pragma omp parallel
{
  /* Statement executed by all threads */
} /* Implicit barrier */
```

Example

```
double xyz[5000][3];

printf("entering parallel region\n");
#pragma omp parallel
{
    int tid;
    tid = omp_get_thread_num();
    compute_edges( tid, xyz );
}
printf("parallel computation completed\n");
```

- `double xyz[5000][3];` — Shared between all threads!
- first `printf` — Master only
- `#pragma omp parallel` — Thread Forks
- `{` — Thread Private Space
- `}` — Implicit barrier: Thread Join
- second `printf` — Master only

OpenMP Directives

- Basic Work Sharing Directives:
  - `#pragma omp for`
    - Each thread receives a portion of work to accomplish – data parallelism
  - `#pragma omp sections`
    - Each section is executed by a different thread – functional parallelism
  - `#pragma omp single`
    - Serialize a section of code, only one thread executes code block
      - Need not be master thread
      - (*e.g.*, good for I/O)
- Can combine parallel construct with one work sharing construct:
  - `#pragma omp parallel for`
  - `#pragma omp parallel sections`

OpenMP Directives (cont.)

- `#pragma omp master`
  - Block will be executed by the master thread of the team
- `#pragma omp critical` *`[(name)]`*
  - Block will be executed by single thread at a time
- `#pragma omp barrier`
  - Explicit barrier (wait for all threads in team)
- `#pragma omp atomic`
  `expr-statement`
  - Guarantee that specified storage location is updated atomically
- …

OpenMP Worksharing Constructs

- Loops can be automatically parallelized (data parallelism)

```
#pragma omp parallel
#pragma omp for shared(A, row, col)
for (i = k+1; i<SIZE; ++i) {
  for (j = k+1; j<SIZE; ++j) {
    A[i][j] = A[i][j] - row[i] * col[j];
  }
}
```

- Data reduction: Data from different threads can be merged

```
sum = 0;
#pragma omp parallel for reduction(+: sum)
for (i = 0; i<NUM_STEPS; ++i) {
  x = 2.0 * (double)i / (double)(NUM_STEPS);
  sum += x * x / NUM_STEPS;
}
```

OpenMP Worksharing Constructs

- **`sections`** and **`section`**
  - Each section is executed by a different thread (functional parallelism)

```
#pragma omp parallel sections num_threads(2)
{
  #pragma omp section
  { /* thread-1 */ }
  #pragma omp section
  { /* thread-2 */ }
}
```

- **`single`**
  - Serialize a section of code, only one thread executes code block (good for I/O)

OpenMP Synchronization Constructs

- **`critical`**
  - Defines a critical section (only one thread at a time)
  - All **`critical`** constructs without a name are considered to have the same unspecified name

```
#pragma omp critical
{
  /* critical section */
}
```

- **`barrier`**
  - A thread reaching a barrier must wait all the other threads of the team

```
#pragma omp barrier
```

OpenMP Synchronization Constructs

- **`ordered`**
  - Execute the block in the order it would be executed in a sequential execution of the loop

```
#pragma omp parallel for
for (i = 0; i < 1000; i++) {
  for (j = 0; j < 1000; j++)
    res = foo();
  #pragma omp ordered
  {
    if (i<5)
      printf("%i: %i\n", i, res);
  }
}
```

Schedule Types for Loop Constructs

- `schedule(static[,chunk])`
  - Threads get a chunk of data to iterate over
  - Chunks are assigned in round-robin fashion
- `schedule(dynamic[,chunk])`
  - Each thread executes a chunk of iterations, then requests another chunk until no chunks remain
- `schedule(guided[,chunk])`
  - Each thread executes a chunk of iterations, then requests another chunk until no chunks remain
  - Chunk sizes start large and then decrease to specified chunk size as the computation progresses
- `auto`
  - Leave decision to compiler and/or runtime system
- `schedule(runtime)`
  - Use the schedule defined at runtime by the `OMP_SCHEDULE` environment variable

Data Sharing Attribute Clauses

- These apply only to variables visible in construct
  - `default(shared|none)`
    - Controls the default data-sharing attributes
  - `shared(list)` / `private(list)`
    - Variables in list are shared / private
  - `firstprivate(list)`
    - Variables in list are private to each thread
    - Variables are initialized to value of corresponding original items
  - `lastprivate(list)`
    - Variables in list are private to each thread
    - Corresponding original items will be updated after the end of region
  - `reduction(operator:list)`
    - Accumulate into list items using indicated associative operator
  - ...

gcc

Compile and link with:

```
gcc -fopenmp -lgomp
```

- The number of threads is determined by the runtime environment or can be set with the `OMP_NUM_THREADS` environment variable