A Multithreaded Compiler Backend based on OpenMP

ZhengZhangzheng
University of Amsterdam
Motivation 1:

- PTHREAD solution is based on the assumption that the underlying architecture implements cache coherent protocols.
- New architectures might not support cache coherent protocols.
- But OpenMP has its own set of rules on how data is shared among the threads running on different processors.
- So we can rely on OpenMP to do this job for us on the new architecture.
Motivation 2:

- OpenMP has a versatile scheduling techniques (such as static, dynamic and guided) and other advanced features
- It has already been proven a success in the industry.
- Thus we want to know the efficiency discrepancy between OpenMP solution and the PTHREAD solution, whether OpenMP solution can outperform PTHREAD solution or not.
Overview of the solution to data parallelism

- Add a new target “omp”
- Reuse sub-phase “MTCMdoRunCostModel” (Running multithreading cost model)
- Reuse sub-phase “MTSTFdoCreateMtStFuns” (Creating MT and ST functions)
- Add a new sub-phase in PC (Prepare C code generation) to find all the OpenMP private variables and store them in a link list.
- For code generation, make a distinction between mt mode, st mode and at mode.
- For compilation scheme, make a distinction between index vector removed and not removed.
Add a new target “omp”

- Add a new target “omp”
- Add command line flag “-omp_schedule” to decide the scheduling technique of OpenMP, default is static
- Add command line flag “-omp_nestlevel” to set the active nesting level in OpenMP, default is 1
- To generate C code with OpenMP directives
- Reuse -numthreads command line flag, and call “omp_set_num_threads” to set the number of threads
Reuse two sub-phases

- Reuse sub-phase “MTCMdoRunCostModel” (Running multithreading cost model)
- Reuse sub-phase “MTSTFdoCreateMtStFuns” (Creating MT and ST functions)
Add a new sub-phases to find OpenMP private variables

- Add a new sub-phase in PC (Prepare C code generation) to find all the OpenMP private variables, switched on by “omp” backend.
- All the variables which appear on the left hand side of assignments inside WITH-loop are regarded as OpenMP private variables.
- Create one link list for each WITH-loop to store all OpenMP private variables in it.
The same as the present MT solution
Can have three modes: mt mode, st mode and at mode
For st mode, just emit sequential code
For mt mode, generate sequential code with OpenMP directives and other necessary statements in appropriate place
For at mode, a “if-else” statement will include both mt code and st code, at run time, one of them will be chosen
Compilation scheme

- Index vector is used to store the offset in each dimension of the array during the traversal.
- It can be replaced by some offset variables after optimization.
- But there is a case in which index vector could not be removed: index vector is a parameter of a function inside WITH-loop.
- Thus make a distinction between index vector removed and not removed.
Compilation scheme: Index vector removed

A = with {
    (.<=[i,j]<=.) : tod(i+j);
} : genarray( [6,6]);

- This is a typical case in which index vector will be removed
Compilation scheme:
Index vector removed

#pragma omp parallel private (SACl_i, SACl_j, SACp_wlidx_1613_A )
{
    #pragma omp parallel for
    for (SACl_i = 0; SACl_i < 6; SACl_i = SACl_i + 1)
    {
        SACp_wlidx_1613_A = .....;
        for (SACl_j = 0; SACl_j < 6; SACl_j = SACl_j + 1)
        {
            .................
            SACp_wlidx_1613_A = SACp_wlidx_1613_A + 1;
            .................
        }
    }
}

The variables in private clause are got from the link list
Compilation scheme: Index vector not removed

\[ a = \text{with}([,] \leq \text{iv} < [,] ) \]
\[ \{ \text{b} = \text{FOO(iv)}; \} \]
\[ \text{modarray}(a, \text{iv}, \text{b}); \]

- This is a typical case in which index vector could not be removed
- The observation is that in the C code generated, index is a pointer
- Then even every thread has a private copy of index vector, they will still write to the same location
- Thus we have to allocate additional memory space of index vector for each thread
Compilation scheme: Index vector not removed

```c
int thread_len_iv = (LEN_IV / LEN_CACHE + 1 + 1) * LEN_CACHE;
int thread_len_iv_desc = (LEN_IV_DESC / LEN_CACHE + 1 + 1) * LEN_CACHE;

int* iv_ptr = (int*)malloc(N * thread_len_iv);
int* iv_desc_ptr = (int*)malloc(N * thread_len_iv_desc);

for (int i = 0; i < N; i++)
{
    *(iv_desc_ptr + thread_len_iv_desc * i) = 1;
}

static const SAC_HM_thread_status_t
SAC_OMP_HM_thread_status = SAC_HM_multi_threaded;
int *thread_prv_iv;
int *thread_prv_iv_desc;
```
Compilation scheme: Index vector not removed

```c
#pragma omp parallel private (iv0, iv1, thread_prv_iv, thread_prv_iv_desc, thread_id, ... )
{
    int thread_id = omp_get_thread_num();
    thread_prv_iv = iv_ptr + thread_id * thread_len_iv;
    thread_prv_iv_desc = iv_desc_ptr + thread_id * thread_len_iv_desc;
#pragma omp for
    for (iv0 = 0; ..... ; iv0 = iv0 + 1)
    {
        thread_prv_iv[0] = iv0;
        ..... 
        for (iv1 = 0; ..... ; iv1 = iv1 + 1)
        {
            ..... 
            thread_prv_iv[1] = iv1;
            thread_prv_iv_desc[0] += 1;
            ..... 
            FOO(thread_id, thread_prv_iv, thread_prv_iv_desc);
        }
    }
}
free(iv_ptr);
free(iv_desc_ptr);
```
Compilation scheme: another situation to notice

B[]

with (<iv>)
{
    foo(B)
} genarray();

• If execute in mt environment, descriptor of B will also be written by multi-threads. This again will lead to chaos.

• The solution is the same to the case in which index vector is not removed, to allocate additional memory for descriptor for different threads.
Code generation in conclusion

- It is the mapping SAC construct to some ICMs especially designed for OpenMP backend.
- These ICMs are just sequential C code with OpenMP directives and some necessary statements.
- Now the work is the details ……
Overview of the solution to task parallelism

- Very brief introduction to task in OpenMP 3.0
- Two possible solutions
Task in OpenMP 3.0

- The most important achievement in OpenMP 3.0
- Target to parallelize the irregular parallelism, such as while loop and recursive algorithms
task Construct

#pragma omp task [clause[[,]clause] ...] structured-block

where clause can be one of:

if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none )
Task Synchronization

- **Barriers (implicit and explicit)**
  - All tasks created by any thread of the current team are guaranteed to be completed at barrier exit

- **Task barriers**
  - `#pragma omp taskwait`
  - Encountering tasks suspends until child tasks complete (only direct child, not descendants)
Solution 1

- map Aram’s keywords into OpenMP task directives
- \texttt{Spawn()} → \#pragma omp task \[\texttt{clause[[,\texttt{clause}] ...]}
- \texttt{Sync()} \rightarrow \#pragma omp taskwait
Solution 2

- Create a task model for task parallelism and automatically add task directives into the generated C code
• Questions?
• Criticisms?
• Advices?