Loop Scheduling in OpenMP

Vivek Kale
University of Southern California/Information Sciences Institute
Overview

• Loop Scheduling in OpenMP.
  • A primer of a loop construct
  • Definitions for schedules for OpenMP loops.

• A proposal for user-defined loop schedule for OpenMP
  • Need to allow for rapid development of novel loop scheduling strategies.
  • We suggest giving users of OpenMP applications control of the loop scheduling strategy to do so.
  • We call the scheme user-defined loop scheduling and propose the scheme as an addition to OpenMP.
OpenMP loops: A primer

• OpenMP provides a loop construct that specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team in the context of their implicit tasks.¹

```c
#pragma omp for [clause[ [,] clause] ... ]
  for (int i=0; i<100; i++){}
```

• Loop needs to be in canonical form.

• The `clause` can be one or more of the following: `private(...)`, `firstprivate(...)`, `lastprivate(...)`, `linear(...)`, `reduction(...)`, `schedule(...)`, `collapse(...)`, `ordered[...]`, `nowait`, `allocate(...)`

• We focus on the clause `schedule(...)` in this talk.

A Schedule for an OpenMP loop

#pragma omp parallel for schedule([modifier [modifier]:]kind[,chunk_size])

• A schedule in OpenMP is:
  • a specification of how iterations of associated loops are divided into contiguous non-empty subsets
    • We call each of the contiguous non-empty subsets a chunk
  • and how these chunks are distributed to threads of the team.¹

• The size of a chunk, denoted as chunk_size must be a positive integer.

The Kind of a Schedule

• A schedule *kind* is passed to an OpenMP loop schedule clause:
  • provides a hint for how iterations of the corresponding OpenMP loop should be assigned to threads in the team of the OpenMP region surrounding the loop.

• Five *kinds of schedules* for OpenMP loop\(^1\):
  • *static*
  • *dynamic*
  • *guided*
  • *auto*
  • *runtime*

• The OpenMP implementation and/or runtime defines how to assign chunks to threads of a team given the kind of schedule specified by as a hint.

---

Modifiers of the Clause Schedule

• **simd**: the chunk_size must be a multiple of the simd width. ¹

• **monotonic**: If a thread executed iteration i, then the thread must execute iterations larger than i subsequently. ¹

• **non-monotonic**: Execution order not subject to the monotonic restriction. ¹
Need Novel Loop Scheduling Schemes in OpenMP

• Supercomputer architectures and applications are changing.
  • Large number of cores per node.
  • Speed variability across cores.
  • Complex dynamic behavior in applications themselves.

• So, we need new methods of distributing an application’s computational work to a node’s cores\textsuperscript{1}, specifically to schedule an application’s parallelized loop’s iterations to cores.

• Such methods need to
  • Ensure data locality and reduce synchronization overhead while maintaining load balance\textsuperscript{2}.
  • Be aware of inter-node parallelism handled by libraries such as MPICH\textsuperscript{3}.
  • Adapt during an application’s execution.

Utility of Novel Strategies Shown

• The utility of novel strategies is demonstrated in published work by V. Kale et al.¹,² and others.

• For example, static-dynamic scheduling mixed strategy with an adjustable static fraction.

  • Motivation: to limit the overhead of dynamic scheduling, while handling imbalances, such as those due to noise.

CALU using static scheduling (top) and \(f_d = 0.1\) (bottom) with 2-level block layout run on AMD Opteron 16 core node.

Diagram of static (top) and mixed static/dynamic scheduling (bottom) where \(f_d\) is the dynamic fraction.

---

Need to Support A User-defined Schedule in OpenMP

• Practice of burying all of the scheduling strategy inside an OpenMP implementation, with little visibility in application code beyond the use of keywords such as ‘dynamic’ or ‘guided’, isn’t adequate for this purpose.

• OpenMP’s implementations, e.g., GCC’s libgomp\(^1\) and LLVM’s OpenMP library\(^2\) are difficult to change, which can hinder development of loop scheduling strategies at a rapid pace.

---

Reasons for User-defined Schedules

• Flexibility.
  • Given the variety of OpenMP implementations, having a standardized way of defining a user-level strategy provides flexibility to implement scheduling strategies for OpenMP programs easily and effectively.

• Emergence of Threaded Runtime Systems.
  • Emergence of threaded libraries such as Argobots\(^1\) and QuickThreads\(^2\) argues in favor of a flexible specification of scheduling strategies also.

• Note that keywords *auto* and *runtime* aren’t adequate.
  • Specifying auto or runtime schedules isn’t sufficient because they don’t allow for user-level scheduling.

---


Scheduling Code of libgomp

• The scheduling code in libgomp, for example, supports the static, dynamic, and guided schedules naturally.

• However, its code structure can’t accommodate the number and sophistication of the strategies that we would like to explore.

→ Adding a user-defined schedule into OpenMP libraries:
  • is possible with effort for a given library.
  • must be done differently for each library.
Specification of User-defined Scheduling Scheme

• We aim to specify a user-defined scheduling scheme within the OpenMP standard\(^1\).
• The scheme should accommodate an arbitrary user-defined scheduler.
• These are the elements required to define a scheduler.
  • Scheduler-specific data structures.
  • History record.
  • Specification of scheduling behavior of threads.

Potential Scheduling Data Structures

• The strategies should be allowed to use a subset or combination of:
  • Shared data structures.
  • Low-overhead steal-queues.
  • Exclusive queues meant for each thread.
  • Shared queues from which multiple threads can dequeue work, each representing a chunk of loop iterations.
History Tracking from Prior Invocations

1. To facilitate the ability to learn from recent history, e.g., values of slack in MPI communication\textsuperscript{1,2} from previous outer iterations, the scheduling scheme should allow for passing a call-site specific history-tracking object\textsuperscript{3} to the scheduler.

2. Examples of history information, i.e., attributes to track via history objects:
   - Previous values of dynamic fraction.
   - Iteration-to-core affinities.
   - Runtime performance profiles.

Example: Library to Support Staggered Scheduling

• In earlier work\(^1\), a loop scheduling library that supports a “staggered” scheduling strategy was implemented.

• It was implemented within an OpenMP parallel region by enclosing the loop body with macros FORALL_BEGIN() and FORALL_END() with appropriate parameters.

```c
int start, end = 0;
static LoopTimeRecord* ltr;
double fd = 0.3;
#pragma omp parallel
{ int tid = omp_get_thread_num();
  int numThrds = omp_get_num_threads();
  FORALL_BEGIN(sds,tid,numThrds,0,n,start,end,fd)
  for(int i=start;i<end;i++)
    c[i] += a[i]*b[i];
  FORALL_END(sds,tid,start,end)
}
```

1. The scheduling strategy’s name and associated parameters are specified in the parameters of the macro.
2. Both macros invoke library functions corresponding to the strategy’s name as specified in the macro call. The macro’s parameters are passed to the user-defined scheduler functions.

---

Example: Library to Support Staggered Scheduling

• In earlier work\(^1\), a loop scheduling library that supports this “staggered” scheduling strategy was implemented.

• It was implemented within an OpenMP parallel region by enclosing the loop body with macros `FORALL_BEGIN()` and `FORALL_END()` with appropriate parameters.

```c
int start, end = 0;
static LoopTimeRecord* ltr;
double fd = 0.3;
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  int numThrds = omp_get_num_threads();
  FORALL_BEGIN(sds, tid, numThrds, 0, n, start, end, fd)
  for(int i=start; i<end; i++)
    c[i] += a[i]*b[i];
  FORALL_END(sds, tid, start, end)
}
```

1. The scheduling strategy’s name and associated parameters are specified in the parameters of the macro.
2. Both macros invoke library functions corresponding to the strategy’s name as specified in the macro call. The macro’s parameters are passed to the user-defined scheduler functions.

Proposal for User-defined Schedule in OpenMP

• We propose a user-defined scheduling scheme that is an adaptation of the above macro-based scheme.
  • Overcomes limitations of simple macro-based scheme.
• The proposed API used for a simple code is illustrated below.

```c
double dynamicFraction = 0.3;
static LoopTimeRecord* ltr; // for history.
int chunkSize = 4;
#pragma omp parallel for schedule(user, staggered:chunkSize:dynamicFraction:ltr)
for(int i = 0; i < n; i++)
    c[i] = a[i]*b[i];
```

• The first parameter of the clause ‘schedule’ specifies a new schedule kind user.
• The second parameter specifies the scheduling strategy’s name staggered, optionally with the strategy-specific parameters.
Implementation of User-defined Schedule

• When a user specifies a schedule kind user and a strategy named X
  • They need to link a library that defines functions:
    • X_loopStart(), X_loopNext() and X_init().
• X_init() allows a user-level scheduler to allocate and initialize its data structures that are to be used commonly across parallel loops that use X.
• The functions X_loopStart() and X_loopNext() determine a loop’s indices that a thread should work on based on the parameter values for the scheduling strategy and of the loop.
• As long as one is allowed to define these functions, one can implement a user-defined scheduler.
• Every thread should call X_loopNext() repeatedly.
Software Architecture for User-defined Schedule

myApplication.C

```c
#include <mpi.h>
#include <omp-mod.h>

int main()
{
  double dynamicFraction = 0.3;
  static LoopTimeRecord* ltr; // for history.
  int chunkSize = 4;
  while(timestep < 1000)
  {
    #pragma omp parallel for schedule(user, staggered:chunkSize:dynamicFraction:ltr)
    for(int i = 0; i < n; i++)
    {
      c[i] = a[i]*b[i];
    }
  }
}
```

omp-mod.h

```c
#include <userDefSched.h>

#define FORALL_BEGIN(strat, ...) strat_##LoopStart(..., &start, &end, ...);

do (loop not done)
{
  #define FORALL_END(strat, ...)
}
while (strat_##LoopNext(..., &start, &end, ...));

barrier();

switch(clause)
{
  'static':
  'dynamic':
  'guided':
  'auto':
  'runtime':
  'user':
}

strat_##LoopStart(..., &start, &end, ...);
```

sd_Init():
- allocate shared data structures for loops that use strategy sd;

sd_LoopStart():
  if I am the master thread,
  - set up a data structure loopParams, along with a lock;
  - signal other threads to start;
  else
  - wait for the signal from master thread;
  use the shared loopParams data structure to calculate my static iterations and execute loop_body for that range;

sd_LoopNext():
  lock loopParams; extract a chunk to work on; unlock loopParams;
  if (no chunk available)
    wait for barrier; done=1;
  else
    execute loop_body for the extracted chunk;
**sd_LoopStart():**

if I am the master thread,
- set up a data structure *loopParams*, along with a lock;
- signal other threads to start;
else
- wait for the signal from master thread;
- use the shared loopParams data structure to calculate my static iterations and execute loop_body for that range;

**sd_LoopNext():**
lock loopParams; extract a chunk to work on;
unlock loopParams;
if (no chunk available)
   wait for barrier; done=1;
else
   execute loop_body for the extracted chunk;
An Alternative Impl. of Static/Dynamic Strategy Using Steal Queues for Dynamic Iterations

```
sd2_LoopStart():
if I am the master thread
   — set up a data structure loopParams with loop parameters;
   — enqueue a single entry corresponding to dynamic range of iterations into thread 0’s steal queue;
   — signal other threads to start;
else
   — wait for the signal from master thread;
- Use the shared data structure to calculate my static iterations and execute loopBody for that range;
```

```
sd2_LoopNext():
- nextRange = myQueue.dequeue();
if (nextRange == NULL) nextRange = steal(random_neighbor);
if (nextRange != NULL)
   L = nextRange->low; U = nextRange->high;
   if ( (U-L) > Threshold )
      - split the range in 2, enqueue them in my steal queue;
   else
      - execute loop body for iterations L:U;
   - update count of iterations completed to set the “done” flag when done;
```
An Implementation of Staggered Static/Dynamic Strategy

staggered_LoopStart():
if I am the master thread
   - set up a data structure loopParams;
   - signal other threads to start;
else
   - wait for the signal from master thread;
   - enqueue entries corresponding to chunks of my dynamic range of iterations into my thread’s steal queue;
   - calculate my static iterations and execute loopBody for that range;

staggered_LoopNext():
nextRange = myQueue.dequeue();
if (nextRange == NULL) nextRange = steal(random_neighbor);
if (nextRange != NULL)
   - L = nextRange->low; U = nextRange->high;
   - execute loop body for my iterations L:U;
   - update count of iterations completed to set the “done” flag when done;
Discussion of API’s details

• We have sketched above a syntax for the API for user-level schedulers.
  • However, we expect that the API’s details will be worked out with the community’s consensus.

• Note that there is a precedent for adding user-defined functions in OpenMP standard.
  • The *combiner* function in user-defined reductions.
Summary

• Need for experimentation with sophisticated loop scheduling strategies.
• OpenMP community should discuss how to allow flexible specification of such strategies in a user’s code and how to design a user-level scheduler library so that it can be portably used with any conforming OpenMP implementation.
• Supporting user-defined schedulers in this way will facilitate rapid development of scheduling strategies.
• I hope the experts will discuss these ideas at this conference.

Acknowledgements: University of Southern California/ISI’s Technical Computing New Directions program.