Analysis of Explicit vs. Implicit Tasking in OpenMP using Kripke

Charles Jin*, Muthu Baskaran

ESPM2 Workshop
in conjunction with SuperComputing 2018
Nov 12, 2018

Background

- Task-based parallelism is a leading contender as a paradigm for exascale computing
  - Dynamic scheduling can react to runtime variations (clock speeds, node failures)
  - Legion, OpenMP, Charm++, PaRSEC, Kokkos, OCR, CnC, etc.
- Case study: Kripke benchmark (LLNL)
  - Base implementation in MPI + OpenMP
  - Scope of talk is the OpenMP component (i.e. performance for shared memory parallelism)

Central question: compare the flexibility of dynamic task-based parallelism against the runtime overhead savings of do-all parallelism

Tasking in OpenMP

Implicit Tasking

- Available since initial release
- Tells compiler that the loop body can be safely workshared, i.e. no restrictions when scheduling iterations

```
#pragma omp for
for (i = 0; i < iMax; i++) {
  // loop body
}
```

Explicit Tasking

- Available in OpenMP >= 3.0
- Whenever the construct is encountered, the runtime creates a deferrable task (overhead is incurred per task)
- Runtime free to schedule subject to dependences specified via data read/writes

```
for (i = 0; i < iMax; i++) {
#pragma omp task depend(in:  a)
                 depend(out: b)
  // loop body
}
```

Kripke

- Mini-app developed by LLNL as a proxy for particle transport codes
- Designed to explore how data layout affects performance
- 3D position, 2D direction, 1D group => 6D unknown space for phase, so standard approach is to discretize and solve via iterative methods

$$[\Omega \cdot \nabla + \sigma(r, E)]\, \psi(r, \Omega, E) = \int dE' \int d\Omega' \sigma_s(r, \Omega' \cdot \Omega, E' \rightarrow E) \psi(r, \Omega', E') + q(r, \Omega, E)$$

Steady state form of the Boltzmann transport equation in three dimensional geometry.
See https://computation.llnl.gov/projects/co-design/kripke for additional references and details.

Kripke

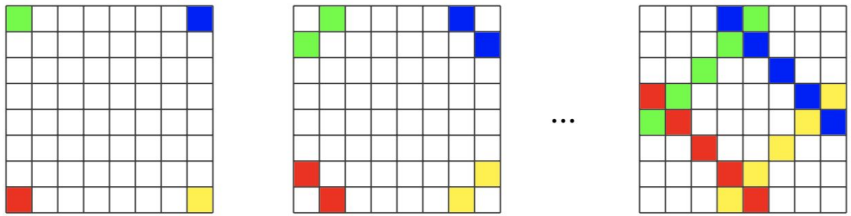
2 / 2

- Mini-app developed by LLNL as a proxy for particle transport codes
- Designed to explore how data layout affects performance
- 3D position, 2D direction, 1D group => 6D unknown space for phase, so standard approach is to discretize and solve via iterative methods
- Scalability is determined by the matrix inversion of H on the LHS, implemented using diamond differences method in the sweep kernel

$$H\Psi = L^{+}\Sigma_{s}L\Psi + Q$$

Sweep kernel

- Sweep in 2D: 4 independent wavefronts starting from each corner
- e.g. for the sweep starting from `(0,0)`, `(i,j)` depends on
  - `(i-1,j)` and `(j-1,i)` from current iteration
  - `(i+1,j)` and `(j+1,i)` from previous iteration
- Kripke sweeps over 3D position, with one independent wavefront per (group, direction) pair (termed “subdomain”)



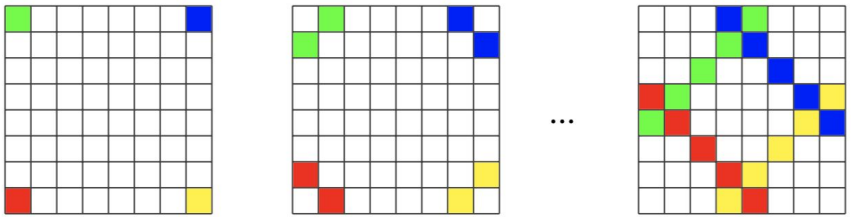
Implicit implementation

1 / 3

```
// nIter    number of iterations
// nSdom    number of subdomains
// iMax     length in i dimension
// jMax     length in j dimension
// kMax     length in k dimension
diags = iMax + jMax + kMax - 2;

for (n = 0; n < nIter; n++)
#pragma omp for
  for (s = 0; s < nSdom; s++)
    // sweep kernel
    for (d = 0; d < diags; d++)
#pragma omp for
      for (i + j + k = d)
        sweepInner(s, i, j, k);
```

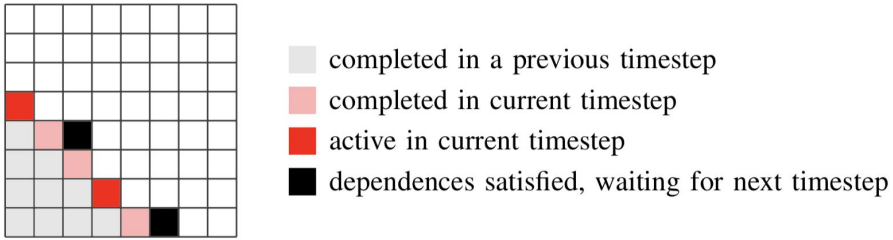
- The implicit implementation uses the `omp for` construct
- Independent sweep per subdomain
- Within each sweep diagonal, the the blocks are independent
- Overhead for a given subdomain is incurred once per diagonal



Implicit implementation

2 / 3

- The `omp for` construct contains an implicit barrier at the end of the block
- => artificial dependence between the sweep diagonals
- => limits the amount of parallelism the runtime can exploit



Implicit implementation

3 / 3

- Thread affinity is an important consideration on NUMA machines
- Pin threads for a given subdomain (= an independent sweep) to the same socket to reduce overhead
- Share work between threads on the same socket to increase parallelism

```
// export OMP_PLACES=sockets
// export OMP_PROC_BIND=spread,master

for (n = 0; n < nIter; n++)
#pragma omp parallel for num_threads(8)  // spread over sockets
  for (s = 0; s < nSdom; s++)
    for (d = 0; d < diags; d++)
#pragma omp parallel for num_threads(8)  // tied to master socket
      for (i + j + k = d)
        sweepInner(s, i, j, k);
```

Explicit implementation

1 / 3

```
// there are 8 sweep directions
// sweep from (0,    0,    0)
//       to   (iMax, jMax, kMax)

for (i = 0; i < iMax; i++)
  for (j = 0; j < jMax; j++)
    for (k = 0; k < kMax; k++)
#pragma omp task
  depend(in: A[i-1][j][k],
             A[i][j-1][k],
             A[i][j][k-1])
  depend(out:A[i][j][k])
      sweepInner(sdom, i,j,k);
```

- The explicit implementation uses the `omp task` construct
- Each subdomain has one sweep direction, and each subdomain can be done in parallel (not shown)
- The runtime is free to schedule the tasks subject to the data dependencies
- Overhead incurred once per task

Explicit implementation

2 / 3

```
// first jMax tasks are not
// parallel
for (i = 0; i < iMax; i++)
  for (j = 0; j < jMax; j++)
#pragma omp task
  depend(in: A[i-1][j],
             A[i][j-1])
  depend(out:A[i][j])
    sweepInner(i,j);
```

- Problem: overhead to create the entire dependence graph at startup can be very high
- OpenMP's solution: if the queue of deferred tasks is large, the master thread may stop spawning tasks and switch to execution
- But this means that if the parallelism isn't exposed until later tasks, you may not get any parallelism at all!

Explicit implementation

3 / 3

```
// 2nd and 3rd tasks are already
// parallel
for (d = 0; d < diags; d++)
  for (i + j = d)
#pragma omp task
  depend(in: A[i-1][j],
             A[i][j-1])
  depend(out:A[i][j])
      sweepInner(i,j);
```

- Better solution: apply loop transformations to help the runtime extract parallelism
- Conclusion: though OpenMP pragmas make it easy to ensure correctness, getting optimal performance still requires non-trivial tuning of source

Results

1 / 2

- Hybrid variant uses both implicit and explicit tasking to reduce overhead to one parallel region per iteration, but still synchronizes after diagonals

	Small Problem	Large Problem
Serial	17.036	485.348
Explicit tasking*	**2.760***	**70.841***
Implicit tasking (pure)	4.044	77.073
Implicit tasking (hybrid)	3.178	75.204

* best performance over all explicit tasking variants (see paper for further details).

Results are average execution time (in seconds) over 10 runs on an 8-core (16 threads) quad socket Intel Xeon (Ivy Bridge) server using 64 threads; compiled with GCC 7.3 (OpenMP 4.5). Large problem is ~32x size of small problem.

Results

2 / 2

Overhead

- Hybrid: once per subdomain, per iteration (**GOOD**)
- Implicit: once per diagonal, per subdomain, per iteration (**OKAY**)
- Explicit: once per task (**NOT GOOD**)

Parallelism

- Explicit: complete dependence graph available to runtime (**GREAT**)
- Implicit, Hybrid: synchronization barrier after each diagonal limits parallelism, load balancing (**NOT GOOD**)

Kripke demonstrates that optimal application of explicit tasking can outperform implicit parallelism in high-performance codes with irregular dependences.

Future Work

- More generally, how to determine the correct form of parallelism?
- One approach: build tools for automatic extraction of parallelism from sequential source
- R-Stream is a source-to-source compiler for loop optimization via polyhedral techniques
  - Extend support to distributed memory systems via cross-runtime interactions and hierarchical parallelism
  - Extend support to heterogeneous architectures (e.g. GPUs)
- OpenMP 5.0: task `affinity`, more expressive `depend` clause

Questions?