Enhancing Support in OpenMP to Improve Data Locality in Application Programs Using Task Scheduling

Martin Kong and Vivek Kale

September 25, 2018

Presenter: Lingda Li

Motivation

- In an OpenMP application in which work is scheduled to threads dynamically, data locality is important for efficient execution of the application.
- Using the clause `affinity` for task scheduling proposed for OpenMP 5.0 can improve data locality [7].
- However, strategies for tasking are fixed by OpenMP's runtime system, even with hints to the affinity clause.
- One can argue that this small set of strategies isn't beneficial for all application-architecture pairs [1, 5].

A Possible Solution

- OpenMP needs an adequate amount of support to maintain high levels of data locality when scheduling tasks to threads.
- Specifically, we need task-to-thread affinity in OpenMP to reduce
  1. capacity cache misses on a multi-core node, or *locality-awareness*, and
  2. coherence cache misses on a multi-core node, or *locality-sensitivity*.
- We need to provide more hints to OpenMP's runtime for assigning OpenMP's tasks to threads in a way that preserves data locality.

Contribution

- Our solution builds on the `affinity` clause for OpenMP 5.0 [2] → the user provides input to the clause as hints on
  1. *what* data needs to be localized
  2. the *degree* to which the data should be localized
- Prior work on the degree to which the data should be localized has been shown to improve performance [3].
- *Contribution*: the addition of constructs to OpenMP that provides and allow for a rich set of task scheduling schemes having (a) locality-awareness or (b) locality-sensitivity.
- This work develops ideas of (a) for the affinity clause, and building on (b) from previous work for the affinity clause.

Scheduling Data Access

- OpenMP lacks a mechanism for allowing the thread identifier to affect the scheduling of inner loops (when this is legal)
- Here we show two examples of how such mechanism can be used
- Benefits: Improve execution time, energy consumption and make better usage of available bandwidth
- We show the results of some preliminary experiments conducted to show the benefits of the proposed directive

Proposal

- Add **loopshift** directive
- Must be nested within a work-sharing directive and parallel region
- Allow to map iterator of some inner loop of the work-sharing loop with some arithmetic expression
- Can use pre-defined variables such as thread identifier (tid) and number of threads (numthreads)

```
#pragma omp parallel for
for (i = lbi; i < ubi; i++)
{
  int j;
  pragma omp loopshift(j = (i + tid) % numthreads)
  for (j = lbj; j < ubj; j++)
  {
    /* do work */
  }
}
```

Listing 1: OpenMP LoopShift Directive

Example: Matrix-Multiply Loop Shift

- First example: Matrix-Multiply
- Shift loop-K w.r.t outer parallel and worksharing loop-i
- Effect: Each thread accesses a different part of array B
- Example shows the semantics of **loopshift** in terms of a more explicit worksharing loop
- Perform explicit partition of rows of B according to value of cc (core/thread)
- Could potentially use a renaming mechanism

```
#pragma omp parallel
{
 #pragma omp for private (i,j,kk)
 for (cc = 0; cc < CORES; cc++) {
  int lb = (ni/CORES) * cc;
  int ub = (ni/CORES) * (cc + 1);
  for (i = lb; i < ub; i++)
   for (kk = 0; kk < nk; kk++) {
    int k = (kk + cc*ni/CORES) % nk;
    for (j = 0; j < nj; j++)
      C[i][j] += A[i][k]  * B[k][j];
   }
 }
}
```

Listing 2: MatMul Loop Shift Semantics

```
#pragma omp parallel
{
 #pragma omp for private (i,j,kk)
 for (cc = 0; cc < CORES; cc++) {
  int lb = (ni/CORES) * cc;
  int ub = (ni/CORES) * (cc + 1);
  for (i = lb; i < ub; i++)
   for (kk = 0; kk < nk; kk++) {
    #pragma omp loopshift \
      (k = (kk+cc*ni/CORES)%nk)
    for (j = 0; j < nj; j++)
     C[i][j] += A[i][k]  * B[k][j];
   }
 }
}
```

Listing 3: MatMul LoopShift Directive

Example: Jacobi 2D Stencil Loop Shift

- Second example: Jacobi-2D
- Shift loop-i w.r.t to thread number
- Effect: Ideally, user-provided mapping function should attempt to reuse the data already brought into cache by the thread

```
for (t = 0; t < TSTEPS; t++) {
 #pragma omp parallel
 {
  int ii;
  #pragma omp for private (j)
  for (ii = 1; ii < n-1; ii++) {
   int tid = omp_get_thread_num ();
   int i = (tid + ii) % (n-2) + 1;
   for (j = 1; j < n-1; j++)
    ref(B,i,j) = 0.2 * (
     ref(A,i,j) + ref(A,i-1,j) +
     ref(A,i+1,j) + ref(A,i,j-1) +
     ref(A,i,j+1));
  }
 }
 /* pointer swap */
 temp = B; B = A; A = temp;
}
```

Listing 4: Jacobi Stencil LoopShift Semantics

```
for (t = 0; t < TSTEPS; t++) {
 #pragma omp parallel
 {
  // Assume tid is an ICV
  #pragma omp for private (j) \
    loopshift(i=(tid+ii)%(n-2)+1)
  for (int ii = 1; ii < n-1; ii++) {
   for (j = 1; j < n-1; j++)
    ref(B,i,j) = 0.2 * (
      ref(A,i,j) + ref(A,i-1,j) +
      ref(A,i+1,j) + ref(A,i,j-1) +
      ref(A,i,j+1));
  }
 }
 /* pointer swap */
 temp = B; B = A; A = temp;
}
```

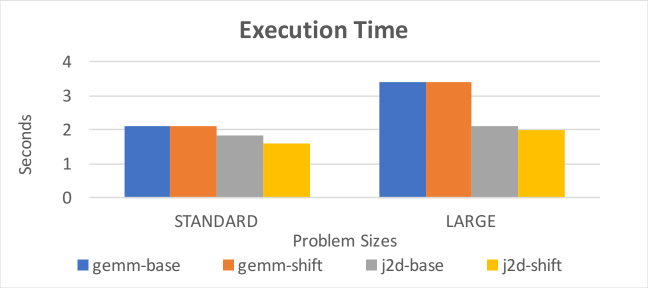
Listing 5: Jacobi Stencil LoopShift Directive

Preliminary Experiments

- Performed some preliminary experiments on Intel Core i9-7900X (10 core)
- Used Clang v7.0 (llvm/trunk)
- Experiments show that the **loopshift** directive can be used to reduce execution time, improve bandwidth usage and/or reduce energy consumption
- We evaluate the kernels previously shown (matmul and jacobi-stencil 2D)
- problem sizes (750^2 and 1000^3)
- the stencil iterates for 200 steps, we repeat the matmul kernel 10 times
- Baseline versions assume static schedule

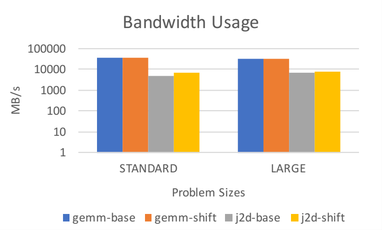
Results: Execution Time

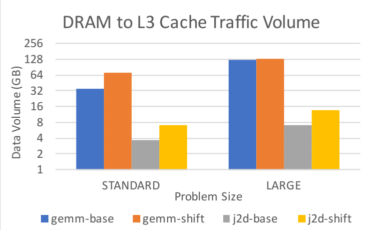
- Impact on execution time varies, in some cases we observe speedups, and in others the runtime remains constant
- We didn't observe slowdowns
- Need to perform a few more experiments



Results: Bandwidth and Data Volume

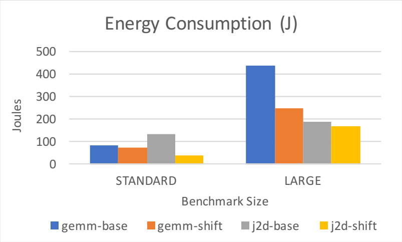
- New directive allows multiple threads to better exploit bandwidth
- Each threads can access disjoint memory regions
- Common to observe higher data-volume movement, but observe no loss in performance
- BW usage remains almost constant
- Loopshift directive allows to affect memory traffic between L3 and DRAM
- Have not observed effects between L1 and L2, nor L2 and L3
- Require more experiments

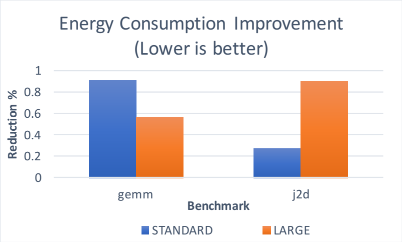




Results: Energy Consumption

- Loopshift directive allows to reduce the energy consumption
- Small exploration shows energy reductions from 10% to almost 70%
- Does not compromise performance (from previous slides)





Additional Notes

- New directive can alleviate different performance factors such as: execution time, bandwidth usage, memory traffic and energy consumption
- Directive, depending on application and code, can help emulate GPU SIMT access.
- Also performed experiments with Pthreads: observed same behavior.
- Also tested OpenMP with GCC 7.2, Clang runtime much faster in many cases
- GCC's OpenMP showed to be less sensitive to loopshifting (likely due to some under-the-hood implementation)

Need to Use Task-to-thread Assignment History

- Consider an OpenMP application code using a `taskloop` construct with multiple outer iterations and that computation load balanced across cores in a timestep.
- If task scheduled to core different than the one in the previous outer iteration, application code retrieves data from cache of the other core, causing a coherence cache miss $\rightarrow$ note that cost is high with more cores such as Intel Xeon Phi 64-core node.
- Such performance degradation is non-trivial if the **cost** of *moving the data between the two cores* exceeds the **benefit** of *load balancing obtained from migrating the data to another core*.
- Application programmer can improve data locality much more than just using the affinity clause may reduce such a cache miss in this case through hints to OpenMP runtime about how task affinity should be done.

Proposal for Locality-sensitivity

We propose adding a new scheduling strategy clause, `schedstrat`, in which one uses the following parameters within the clause to specify how task scheduling ought to be done:

- `history(tid, mode)`: Specifies the mode, or methodology (from a pre-specified set of methodologies) in which history is used to select a task from the shared queue, given a thread ID. If no mode is chosen, the task is chosen based on whether it ran on a given thread ID in the previous outer iteration.
- `randomizationFactor`: Reduces coherence cache misses by having an adjustable parameter for the probability, between 0.0 and 1.0, that a task is chosen according to history from the previous outer iteration.

Proposal for Handling Locality-sensitivity

Example illustrating Approach

- Consider the Barnes-Hut code below run on a node of a supercomputer of four cores.

```
Process(void * arg)
{
  register const int slice = (long) arg;
  int tid = (long) arg;
  int i;
  #pragma omp taskloop affinity schedstrat(history(tid):randomizationFactor) grainsize(4)
  for(i=0; i<n; i++)
     body[i]->ComputeForce(groot, gdiameter);
}
```

Listing 6: Barnes-Hut user code using proposed locality-sensitive tasking

Support by Runtime

- When each of the the four threads each pick up work from the shared work queue, a thread first generates a random number between 0.0 <= P <= 1.0. A user sets a threshold r.
- 1. If p > r, dequeue a task that ran on thread X. 2. If p <= r, choose a random thread that's not X and dequeue a task from that thread.
- If the number generated determines (1), the thread searches for the first task in the queue which has run on that thread in the previous invocation of the taskloop computation region.
- If the thread finds such a task, the thread dequeues and the executes the task. If the thread doesn't find such a task, the thread will dequeue the task at the head of the queue.

→ Options passed to the affinity scheduling clause tunes the degree to which load balancing is done with respect to data locality.

Implementation Guidelines

The implementation:

- needs to not create false sharing in misalignment of shared queue;
- should minimize time to search for a task in queue that match the locality tag;
- should reduce synchronization overheads by supporting a and tuning of parameter for number of queues;
- should use an efficient implementation of work stealing[1];
- shall ideally have an automatic determination of parameters of the task scheduling strategy;
- should support history from previous outer iterations for per-outer-iteration adjustment of parameters of task scheduling strategy during runtime.

Performance Expectations

- There will be fewer coherence cache misses and less capacity cache misses with more memory bandwidth on the bus.
- Some benefits not addressed here but that can be addressed are:
    1. The idea won't decrease synchronization overheads.
    2. The prefetching engine still can't be beneficial for constrained dynamic task scheduling because of the randomized branch involved in the strategy.

Conclusions

1. Need mechanism to enable locality-aware and data-oriented task and thread scheduling in OpenMP 5.0
2. Propose clause `affinity` and through using parameters and hints to the clause; propose `loopshift` directive to affect inner worksharing loops
3. Propose new types of hints for locality-aware task scheduling's clause `affinity` that specify
   - what data should be associated with a particular thread, or privatized
   - the degree to which that data should be privatized.
4. We believe that such support in OpenMP will improve performance of many OpenMP application codes on current and future architectures.
5. We'll take feedback to add the ideas to OpenMP version 5.1 or a version of OpenMP immediately succeeding OpenMP version 5.1.
6. Please email us questions and inquiries :-)

Acknowledgements

- This work was supported in part by funding Exascale Computing Project under Grant Number 17-SC-20-SC.
- We thank input from Oscar Hernandez from Oak Ridge National Laboratory for initial input of the ideas for locality in tasking and formulating the ideas.

Questions?

Contacting Authors

- Vivek Kale: `vkale@isi.edu`
- Martin Kong: `mkong@bnl.gov`
- Lingda Li: `lli@bnl.gov`

Bibliography I

[1] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of ACM*, 46(5):720–748, 1999.

[2] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.

[3] V. Kale and W. Gropp. Load Balancing for Regular Meshes on SMPs with MPI. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI '10, pages 229–238, Stuttgart, Germany, 2010. Springer-Verlag.

[4] V. Kale, S. Donfack, L. Grigori, and W. D. Gropp. Lightweight Scheduling for Balancing the Tradeoff Between Load Balance and Locality. 2014.

[5] M. Kulkarni, P. Carribault, and K. Pingali. Scheduling Strategies for Optimistic Parallel Execution of Irregular Programs. In *ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, Jun 2008.

Bibliography II

[6] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins. Characterizing and Mitigating Work Time Inflation in Task Parallel Programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 65:1–65:12, Salt Lake City, UT, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389085.

[7] C. Terboven, J. Hahnfeld, X. Teruel, S. Mateo, A. Duran, M. Klemm, S. L. Olivier, and B. R. de Supinski. Approaches for task affinity in openmp. 9903, 7 2016. doi: 10.1007/978-3-319-45550-1_8.