Auto-tuned Low-cost Algorithmic Strategies for HPC: A Study with OpenMP

Vivek Kale
Sandia National Laboratories
SC '22 OpenMP Booth Talk
November 15, 2022

"Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525."

Motivating Example Code Structure for Clusters of SMPs

OpenMP

assume mpich's MPI

assume LLVM's OpenMP

BSP Timestepping Loop

Threaded computation region

A Loosely Synch MPI Communication

OpenMP parallel loop with static schedule; threads in team share `u` and `unew`.

```
#include <mpi.h>
#include <omp.h>
int main(int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  int procsPerNode = 1; // OpenMP manages entire (logical) node
  // read input size n per process
  double *u; // input data array of size n
  double *unew; // output data array
   // allocate and initialize u and unew
  for(int step=0; step < numSteps; step++)
  {
   MPI_Isend()/MPI_Irecv()/MPI_Waitall(); // border exchange
   #pragma omp parallel for schedule(static) shared(u, unew)
    {
     for (int i = 0; i < n; i++) {
        unew[i] = (u[i-1] + u[i] + u[i+1])/3.0; // stencil
      // debugging + profiling
     }
   std::swap(u, unew);
  }
  // assess ease of use, correctness, performance, energy
  MPI_Finalize();
}
```

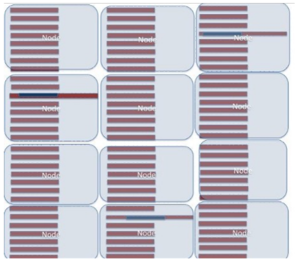
U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

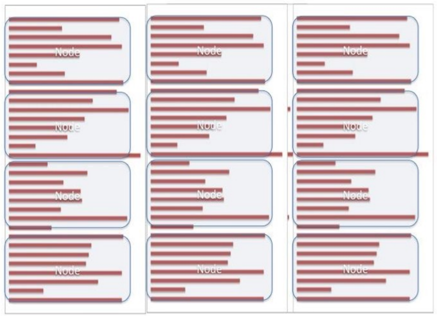
Within-node Load Imbalance Slows Applications

Time

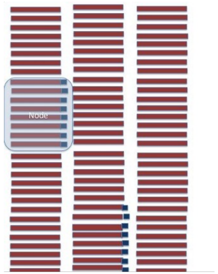
Time



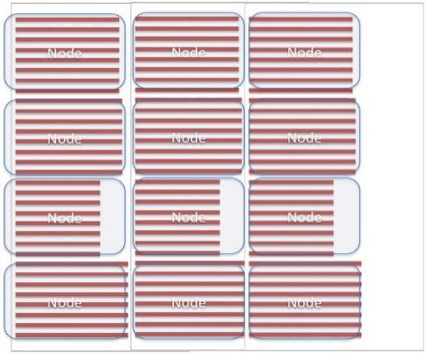
Noise delays every timestep on some node



Application-created imbalances.



Performance improves if we can perfectly redistribute work within each node



These can also be handled if we had a *perfect* within-node load redistribution.

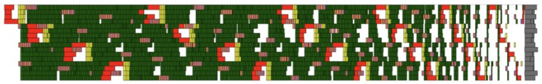
Noise amplification:

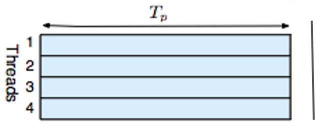
[PetriniSC03theCase] , [hoeflerSC10noise]

Sophisticated Loop Scheduling Pays Off

Statically Scheduled Work | Dynamically Scheduled Work | Dequeue Overhead | Thread barrier

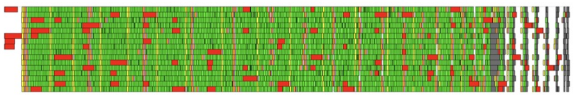
- Static scheduling
  - + Good locality of data
  - - Ignores OS jitter

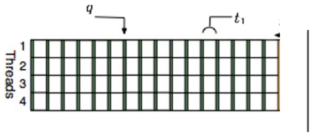





```
#pragma omp parallel for schedule(static)
      for(int i=0; i<n; i++)
                 loop_body(i);
```

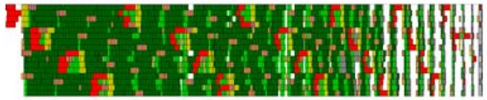
- Dynamic scheduling
  - + Keeps cores busy
  - - Poor usage of data locality
  - - Can lead to large overhead

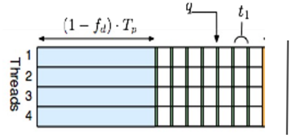





```
#pragma omp parallel for schedule(dynamic)
      for(int i=0; i<n; i++)
                 loop_body(i);
```

Static + 10% dynamic scheduling






```
double fd = predict_dynamic_fraction();
#pragma omp parallel for nowait
      for(int i=0; i< ceil((1.0-fd)*n); i++)
                 loop_body(i);
#pragma omp parallel for schedule(dynamic)
      for(int i= ceil((1.0-fd)*n); i<n; i++)
                 loop_body(i);
```

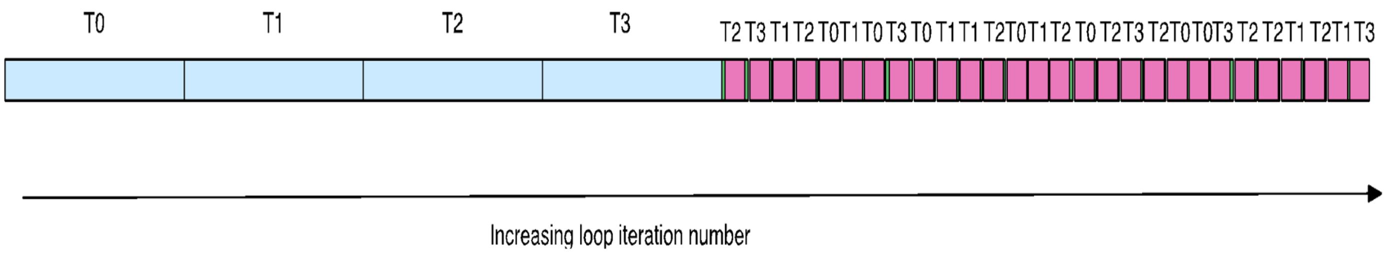
Time

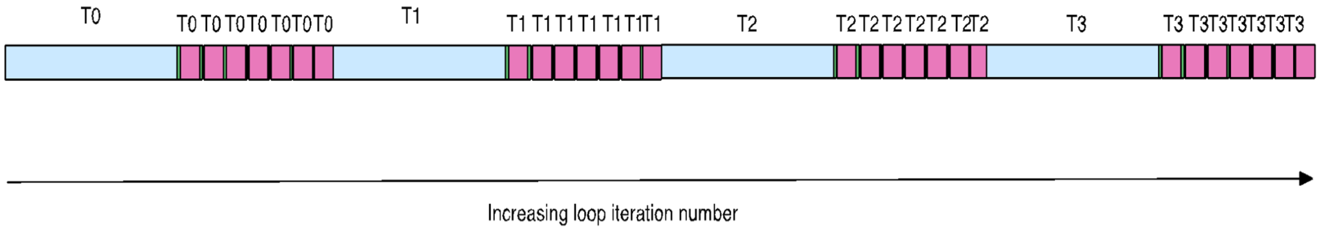
$$f_s \le 1 - \frac{\delta_{total}}{T_P}$$

→ We can guess that average delta increases with a larger number nodes

→ Implies optimal dynamic fraction for a larger number of nodes should increase.

Enhancing with Data Layout and Loop Transformations

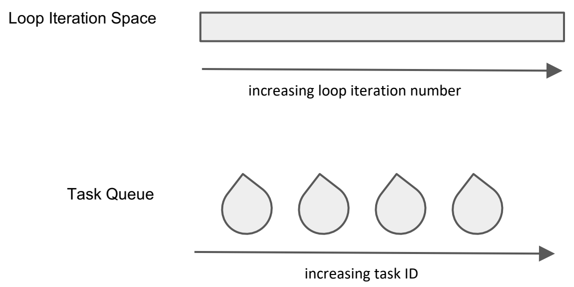


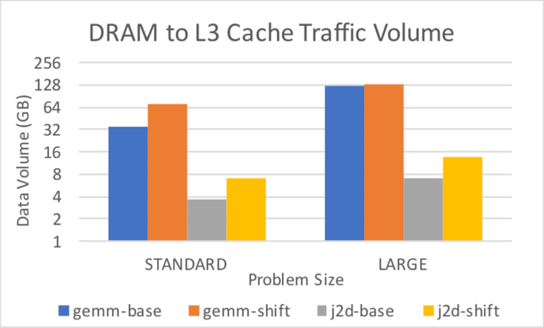


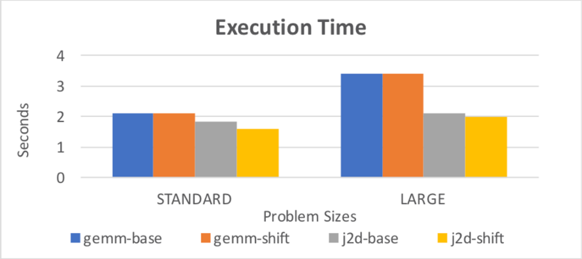
- Each thread finishes its static iterations.
- Then does dynamic iterations marked for it if available.
- Only if not, looks for other dynamic work from other threads to steal.

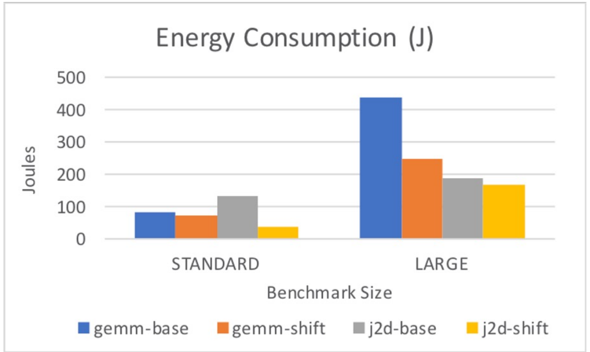
→ Improves Spatial Locality
→ Other loop transformations can be composed with schedules

Data Locality in Tasking in OpenMP









- Lower *execution time* for shifted versions due to threads taking in more data from DRAM to L3 cache.
- Lower *energy consumption* for shifted versions due to more efficient data movement.

Martin Kong and Vivek Kale. Enhancing Support in OpenMP to Improve Data Locality in Application Programs Using Task Scheduling. OpenMPCon 2018. September 2018. Barcelona, Spain.

User-defined Multi-core Loop Schedule

```
#include <ompFromTheFuture.h>
int main(int argc, char** argv)
{
 struct loop_record_t {double fd; int counter; int chunksz; int increment;};
 void mydyn_start(int lb, int ub, int incr, int chunksz, loop_record_t * lr) { ... }
 void mydyn_next(int lb, int ub, int incr, int chunksz, loop_record_t * lr) { ... }
 void mydyn_fini(loop_record_t * lr) { ... }
 #pragma omp declare schedule(mydyn) start(mydyn_start(omp_lb, omp_ub, omp_incr, omp_chunksz, ptr)) \
                                     next(mydyn_next(omp_lb, omp_ub, omp_incr, omp_chunksz, ptr)) \
                                     fini(mydyn_fini(ptr))

static loop_record_t lr;
 for (tstep =0; tstep < numSteps; tstep++)
  {
  lr->chunksz = 1; lr->fd = 0.45;
  #pragma omp parallel num_threads(8)
   {
   #pragma omp for schedule(mydyn, &lr)
     for (int i = 0; i < n; i++) {
         unew[i] = (u[i] + u[i-1] + u[i+1])/3.0;
     }
     std::swap(u,new);
   }
  } //end timestep loop
}
```

OpenMP arguments need to go first, omp_* are used to let the compiler insert actuals at the right formal parameter of the function.

Struct to contain loop information; provided by UDS, instantiated by programmer.

bit.ly/ompuds

Low-cost Schedules for Heterogeneous Nodes

Each device task has three CPU-side subtasks: pre-task to choose device, a task to fire GPU kernel, post-task to deal with completion of task and release work. Not every strategy needs all three.

```
#pragma omp taskloop grainsize (gsz)
for (i = 0; i < numTasks; i++)
{

#pragma omp task depend (out: chosen[i])
{
…
}

#pragma omp task depend(chosen[i]) depend(out: success[i])
   {
  …
#pragma omp target … device(chosen[i])

  }

#pragma omp task depend(in: success[i])
 {
  …
 }

}
```

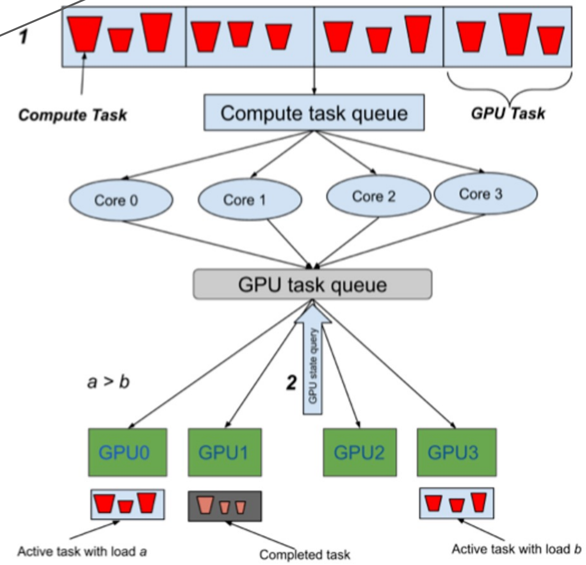
GPU Task

Pre

Fire

Post

i^{th} GPU task is assigned to chosen[i]



Relevant Paper: Mathialakan Thavappiragasm, Vivek Kale, Oscar Hernandez, Ada Sedova. *Addressing Load Imbalance across Multi-GPU Systems for Bioinformatics Applications.* BioHPC 2021. December 2021.

User-defined Hierarchical Multi-xPU Schedule

OpenMP

```
#include <ompFromTheFuture.h>
int main(int argc, char** argv)
{
 struct loop_record_t {double fd; int counter; int chunksz; int increment;};
 void mydyn2_start(int lb, int ub, int incr, int chunksz, loop_record_t * lr) { ... }
 void mydyn2_next(int lb, int ub, int incr, int chunksz, loop_record_t * lr) { ... }
 void mydyn2_fini(loop_record_t * lr) { ... }
 #pragma omp declare schedule(mydyn2) level(4:spread)
                                    start(mydyn2_start(omp_lb, omp_ub, omp_incr, omp_chunksz, ptr)) \
                                    next(mydyn2_next(omp_lb, omp_ub, omp_incr, omp_chunksz, ptr)) \
                                    fini(mydyn2_fini(ptr))
 static loop_record_t lr;
for (tstep =0; tstep < numSteps; tstep++)
 {
  lr->chunksz = 1024; lr->fd = 0.45;
  #pragma omp parallel num_threads(8)
  #pragma omp single
  {
   #pragma omp target spread teams distribute parallel for schedule(mydyn2, &lr) is_device_ptr(u, unew)
     for (int i = 0; i < n; i++) {
         unew[i] = (u[i] + u[i-1] + u[i+1])/3.0;
     }
     swap(u, new);
   }
  }
}
```

Level in hierarchy of schedule, along with optional type

Belongs to level 4 parallelism

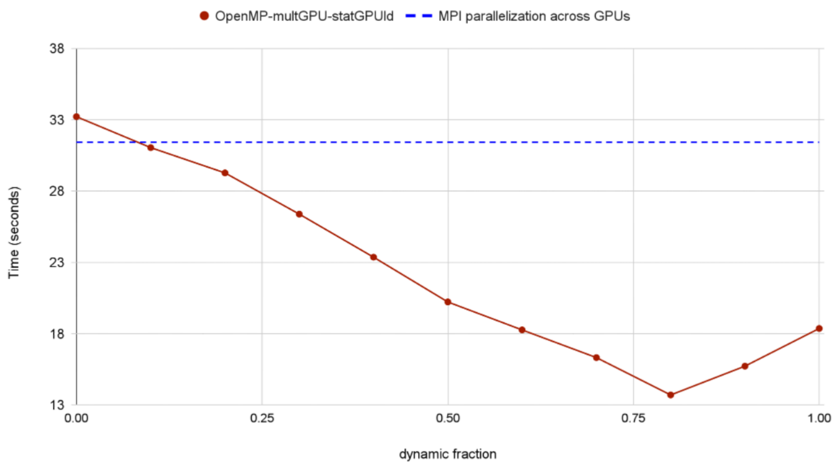
Experimentation with Different Locality Strengths Through User-defined Schedule

Ongoing Work: Can we reduce dynamic scheduling overhead?

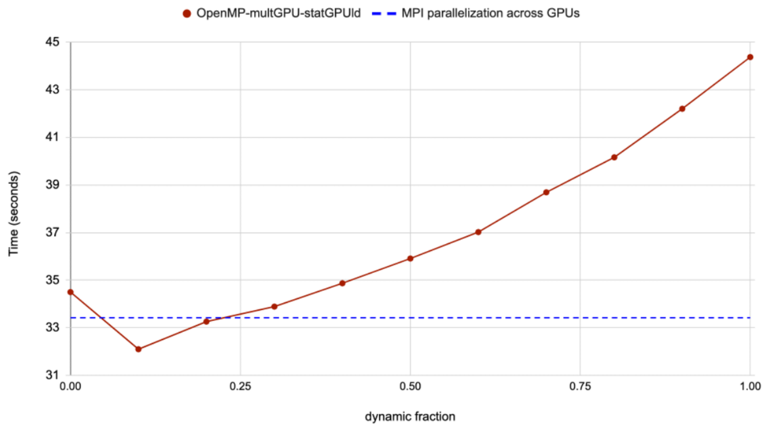
Dynamic Fraction: fraction of iterations that are scheduled to *GPUs* dynamically

- Experiments are done with mixed static/dynamic with dynamic fraction being *probability* of picking GPU with static-adaptive.
- Use probability because currently, can't use event-based trigger which allows a child task on the GPU to tell its parent task on the CPU that the GPU is ready for more work
- If static-adaptive isn't triggered, then the static-default strategy of pick the GPU based on thread ID is used.

Randomized Mat Mul on node of Summit



Stencil on node of Summit



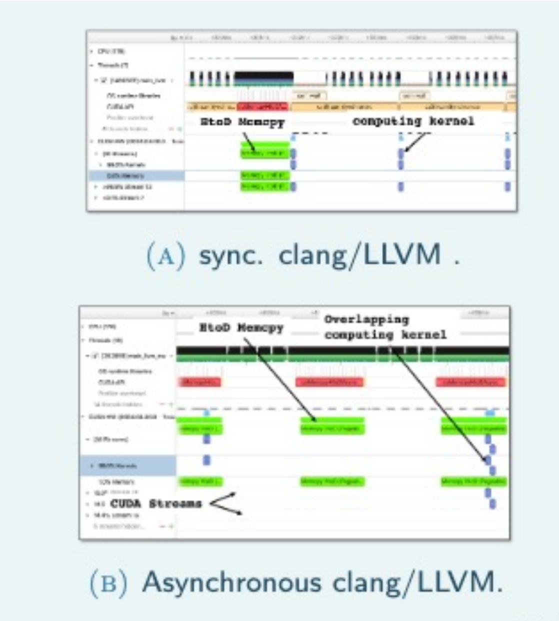
- Mixed static/dynamic scheduling improves performance over OpenMP static by 25.6% and MPI version by 16.8%. Overhead of task contexts likely reduced at fd=0.8
- Mixed static/dynamic scheduling improves performance over OpenMP static by just 6.9% and MPI version by 3.4%. Overhead of task contexts likely reduced at fd=0.1.

→ Mixed static/dynamic beneficial both for data locality and thread-to-device affinity.

github.com/vlkale/TaskGPUSched

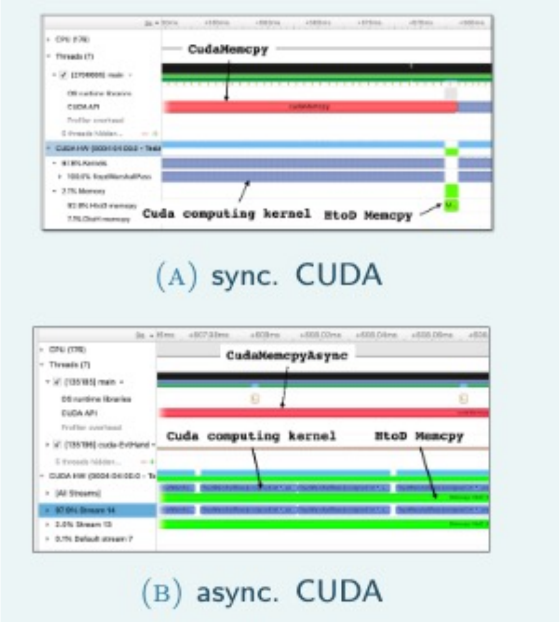
Tunable Low-Cost Asynchronous Offloading

- **<u>Problem:</u>** When using clang/LLVM OpenMP asynchronous offload for Floyd-warshall, register file usage is high on Summit.
- **<u>Hypothesis:</u>** Reduce overhead of it through –fopenmp-no-thread-state; tune GPU parallelization parameters of `thread_limit` and `num_threads`



(A) sync. clang/LLVM .

(B) Asynchronous clang/LLVM.



(A) sync. CUDA

(B) async. CUDA

→ **<u>Result from Experiments</u>:** Tuning asynchronous offload when using clang/LLVM provides performance close to that of CUDA.

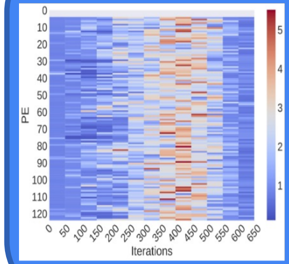
<u>Relevant Paper:</u> Mathialakan Thavappiragasm, Vivek Kale. **OpenMP's Asynchronous Offloading for Combinatorial Scientific Computations.** HiPar 2022 at SC '22. November 18, 2022. Dallas, Texas.

U.S. DEPARTMENT OF ENERGY

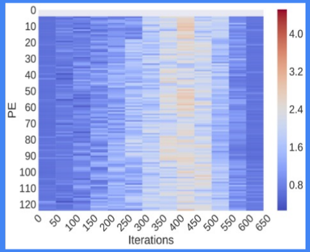
National Nuclear Security Administration

Multi-level Load Balancing

Inter-node load balancing

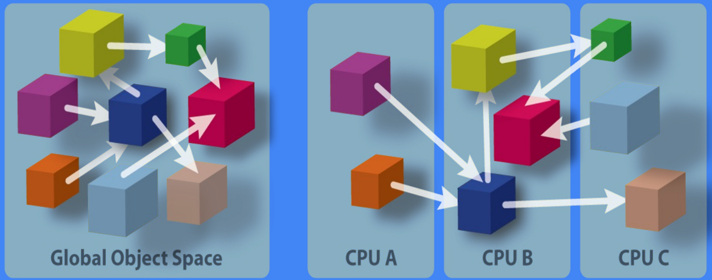


Inter- + intra- node load balancing

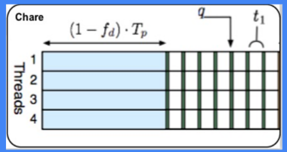


Courtesy: Harshita Menon and B. McCandles

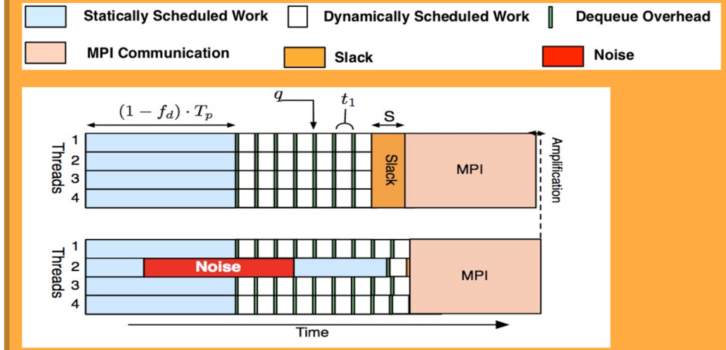
Charm++



Charm++ + CkLoopHybrid



github.com/vlkale/charm/tree/main/src/libs/ck-libs/ckloop and part of Charm 7.0.0



Dynamic fraction

$$f_d = \frac{p\delta}{N(t_1+q)}$$

Dynamic fraction adjusted for MPI process slack

$$f'_d = f_d - \frac{S}{(p-1)\frac{Nt_1}{p} - Nq}$$

Code Transformation for Technique with ROSE

```
#include <omp.h>

int main(int argc, char* argv[])
{
  int timestep = 0;
  void *status;
  int numprocs;
  int rank;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // ...

  while(timestep < 1000)
   {
#pragma omp parallel
    {
#pragma omp for
      for(int i = 0; i<n; i++)
  c[i] += a[i]*b[i];
    }
    MPI_Allreduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM
       , MPI_COMM_WORLD);
    timestep++;
   }

  MPI_Finalize();
}
```

Figure 6.2: Original code with OpenMP loop.

ROSE-based Clang/LLVM ASTRewriter

```
#include "mpi.h"
#include <omp.h>
#include "vSched.h"
// ...

// In the below macros, strat is how we specify the library
 .
#define FORALL_BEGIN(strat, s,e, start, end, tid, numThds )
     loop_start_ ## strat (s,e ,&start, &end, tid, numThds)
   ;  do {

#define FORALL_END(strat, start, end, tid)   } while(
    loop_next_ ## strat (&start, &end, tid));

int main(int argc, char* argv[])
{
  int timestep = 0;
  int rank, numprocs;
  int numThrds;
  int start, end = 0;
  double fd, fs;
  static LoopTimeRecord *record = NULL;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  vSched_init(numThrds);
  // ...

  while(timestep < 1000)
  {
    fd = predict_dynamic_fraction(&record); fs = 1.0 - fd;
#pragma omp parallel
    {
  int tid = omp_get_thread_num();
  int numThrds = omp_get_num_threads();
  FORALL_BEGIN(sds,tid,numThrds, 0, n, start, end, fs)
    for(int i=start;i<end;i++)
      c[i] += a[i]*b[i];
  FORALL_END(sds,tid,start,end)
    }
      end_timing(&record, n);
      MPI_Allreduce(&sum, &global_sum, 1, MPI_DOUBLE,
         MPI_SUM, MPI_COMM_WORLD);
      timestep++;
  }
 endLoop(&lr, (int) (n*fd));
 vSched_finalize(numThrds);
 MPI_Finalize();
}
```

Figure 6.3: Code transformed to use composed scheduler.

Calls to lw-sched

Calls to slack-trace

- `libunwind` to find previous MPI collective, courtesy LLNL's Adagio
- Locality-sensitive Loop Scheduling: github.com/vlkale/lw-sched

Performance Portable Programming with Tunable Low-cost Methods

OpenMP

Code through hand transformation or maybe ROSE/Orio/LLVM.

Github.com/vlkale/lwsRAJA

```
#include "vSched.h"
#define FORALL_BEGIN(strat, s,e, start, end, tid, numThds )
loop_start_ ## strat
(s,e ,&start, &end, tid, numThds); do {
#define FORALL_END(strat, start, end, tid) } while(
loop_next_ ## strat (&start, &end, tid));
void* dotProdFunc(void* arg)
{
  int startInd = (probSize*threadNum)/numThreads; int endInd
=  (probSize*(threadNum+1))/numThreads;
  while(iter < numIters) {
   mySum = 0.0; //reset sum to zero at the beginning of the
product if(threadNum == 0) sum = 0.0;
   if(threadNum == 0) setCDY(static_fraction , constraint.
hunk_size);

#pragma omp parallel
FORALL_BEGIN(statdynstaggered , 0, probSize , startInd,endInd
,threadNum, numThreads)
for (i = startInd ; i < endInd; i++) mySum += a[i]*b[i]
 FORALL_END(statdynstaggered , startInd , endInd,threadNum)
pthread_mutex_lock(&myLock);
  sum += mySum;
  pthread_mutex_unlock(&myLock);
  pthread_barrier_wait(&myBarrier);
  if(threadNum == 0) iter++;
  pthread_barrier_wait(&myBarrier); } // end timestep loop
}
```

MPI+OpenMP code explicitly using lightweight scheduling.

RAJA User Code

```
RAJA::ReduceSum<RAJA::seq_reduce, double> seqdot(0.0);
RAJA::forall<RAJA::omp_lws>(RAJA::RangeSegment(0, N), [=] (int i) {
seqdot += a[i] * b[i]; });
dot = seqdot.get();
std::cout << "\t (a, b) = " << dot << std::endl;
```

RAJA library implementation with policy `omp_lws`

```
#include "vSched.h"
#define FORALL_BEGIN(strat, s,e, start, &end, tid, numThds) do {
#define FORALL_END(strat, start,tid)); start, end, tid, numThds ) loop_start_
## strat (s,e ,&end, tid) } while( loop_next_ ## strat (&start, &end)
template <typename Iterable , typename Func >
RAJA_INLINE void forall_impl(const omp_lws<&, Iterable&& iter, Func&&
loop_body) {
RAJA_EXTRACT_BED_IT(iter);
int startInd , endInd;
int threadNum = omp_get_thread_num();
int numThreads = omp_get_num_threads();
FORALL_BEGIN(statdynstaggered , 0, distance_it , startInd , endInd , threadNum
, numThreads) for (decltype(distance_it) i = startInd; i < endInd; ++i) {
loop_body(begin_it[i]); }
FORALL_END(statdynstaggered , startInd , endInd , threadNum)
}
```

Apply ROSE transformation to *this*

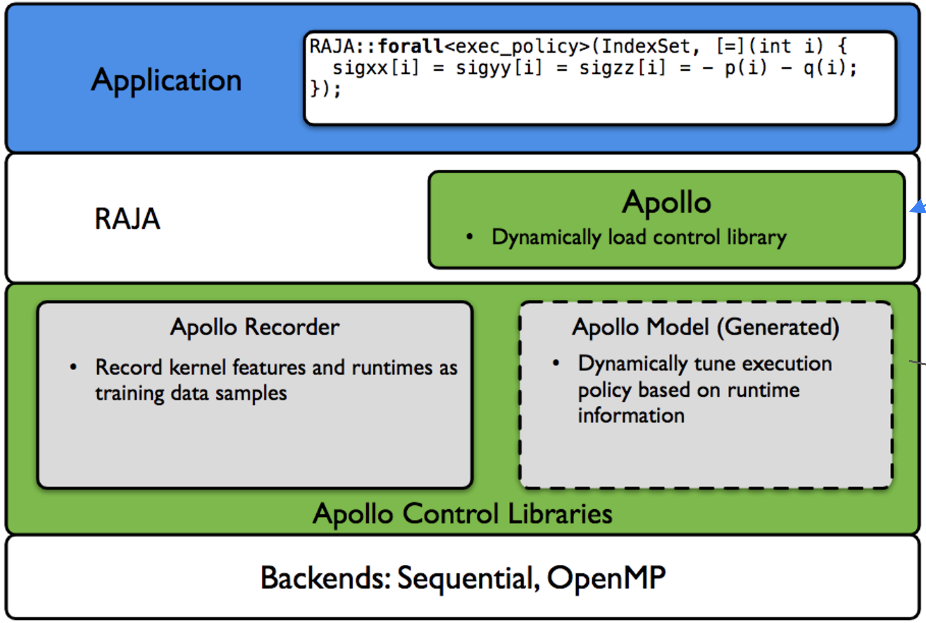
→ Significantly reduces lines of code for application programmer to use strategy: **easy-to-use**

→ Improves ***portability of strategies***.

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Auto-tuning Low-cost Strategies of OpenMP



1. AI can automatically generates new loop schedules through User-defined schedules for Kokkos/RAJA to use *and* tunes parameters of the loop schedules
2. Could combine with user-defined reductions.

Initial value of dynamic fraction f_d

Runtime adjustment of dynamic fraction through slack-trace library.

github.com/vlkale/raja

Conclusions

- HPC Software that tunes a parameter for a dimension **can incur cost** within that dimension or in another dimension.
- **Autotuned Low-cost Algorithmic Strategies** in HPC Software can help. We focus on (LLVM's) OpenMP.
  1. Intra-node support:
     - With multi-cores and with multi-XPUs:
       - Tuned low-cost load balancing strategies with optimizations of affinity and loop transformations.
       - User-defined univ. hierarchical multi-xPU Loop Schedules in OpenMP.
     - With just multi-xPU
       - Tuned low-cost asynchronous offloading in OpenMP
  2. Inter-node support:
     - Multi-level load balancing
       - with Charm++
       - with slack-conscious scheduling.
- **Make these strategies performance portable and intelligently autotuned**
  - Integration of OpenMP UDS in RAJA/C++ integration
  - ROSE source-to-source transformations for injection of runtime support
  - **Ongoing:** AI-enabled auto-tuning with RAJA's Apollo and Caliper.
- **Future Work**
  - Tuning for Dimension of Reproducibliity and Numerical Accuracy
  - Scalability projections through LogOpSim
  - Develop strategies for cloud platform

Appendix

Motivating Example Code Structure

OpenMP

Needs: 1. Scientific Discovery 2. Engineering Innovation 3. service for Industry, AI/ML

Computational *operation*

Computation *region*

Outer iteration, or timestep

Timestepping loop

stencil.c

```
int main(int argc, char** argv)
{
  // read input of problem size n and numSteps
  double *u; // and allocate data array
  double *unew; // and allocate data array
  // initialize unew and u

  for(int step=0; step < numSteps; step++) {
      for(int i = 0; i < n; i++)
          unew[i] = (u[i-1] + u[i] + u[i+1])/3.0;
      std::swap(u,unew);
      // debugging + profiling
   }
   // convergence check
  }
   // print output
   // assess effectiveness and efficiency of program
}
```

cc stencil.c; ./a.out 10000 1000;

Platform: supercomputer, cloud

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

User-defined Multi-xPU Schedule

```
#include <ompFromTheFuture.h>
int main(int argc, char** argv)
{
 struct loop_record_t {double fd; int counter; int chunksz; int increment;};
 void mydyn2_start(int lb, int ub, int incr, int chunksz, loop_record_t * lr) { ... }
 void mydyn2_next(int lb, int ub, int incr, int chunksz, loop_record_t * lr) { ... }
 void mydyn2_fini(loop_record_t * lr) { ... }

 #pragma omp declare spread_schedule(mydyn2)
                                       start(mydyn2_start(omp_lb, omp_ub, omp_incr, omp_chunksz, ptr)) \
                                       next(mydyn2_next(omp_lb, omp_ub, omp_incr, omp_chunksz, ptr)) \
                                       fini(mydyn2_fini(ptr))
 static loop_record_t lr;
 for (tstep =0; tstep < numSteps; tstep++)
 {
  lr->chunksz = 1024; lr->fd = 0.45;
  #pragma omp parallel num_threads(8)
  #pragma omp single
  {
  #pragma omp target spread teams distribute parallel for spread_schedule(mydyn2, &lr) is_device_ptr(u, unew)
     for (int i = 0; i < n; i++) {
         unew[i] = (u[i] + u[i-1] + u[i+1])/3.0;
     }
     swap(u, new);
   }
  }
}
```

spread_schedule to match the target spread

Spread clause is category of worksharing, and spread_schedule the associated schedule

Related Work

- DPLASMA, ParSec
- Legion
- BOLT
- Habanero
- OpenMP Guided Scheduling, Polychronopolous et al.
- Cilk