Composing Low-Overhead Scheduling Strategies for Improving Performance of Scientific Applications on Clusters of SMPs

Vivek Kale[1]

[1]Department of Computer Science
University of Illinois at Urbana-Champaign

October 1, 2015

Sample MPI Code

Common pattern in application codes, e.g., in ones we'll see later.

```
for(ts = 0; ts < 1000; ts++) // 1000 timesteps
  {
    MPI_Irecv(leftGhost,gSz,MPI_DOUBLE,id-1,..,&requests[numRequests++]);
    MPI_Irecv(rightGhost,gSz,MPI_DOUBLE,id+1, ..., &requests[numRequests++]);
    MPI_Isend(leftBoundary,bSz,MPI_DOUBLE,id-1, ..., &requests[numRequests++]);
    MPI_Isend(rightBoundary,bSz,MPI_DOUBLE, id+1,..., &requests[numRequests++]);
    MPI_Waitall(numRequests,requests, MPI_STATUSES_IGNORE);
    for(i = 0; i < n; i++) w[i] = (u[i-1] + u[i+1] + u[i])/3.0;
    temp = w;
    w = u;
    u = temp;
  }
```

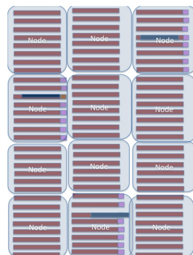
Assuming this code is perfectly load balanced, should be no performance problems.

Transient Load Imbalance and its Mitigation

Infrequent noise → slowdown at scale.

Idea for fix: redistribute within node.

(a) Noise delays almost every iteration.

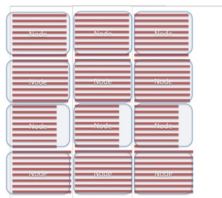


(b) Performance improves, assuming perfect work re-distribution within each node.

Figure 1: Application timeline schematics.

Within-node Persistent Load Imbalance and Mitigation

(a) Application imbalances.



(b) Mitigated by within-node re-distribution.

Indent due to not doing across-node balancing, but still much better than before.

→ Can Dynamic Load Balancing Fix This?

- Dynamic load balancing within a node has potential to mitigate imbalances, *if* it can be done efficiently.
  - Persistent imbalance can also be addressed by across-node load balancing (available in Charm++, Zoltan), but it requires more complex machinery (and is complementary anyway).
- OpenMP supports dynamic balancing.
- Use OpenMP within-node.

Execution Timings Breakdown

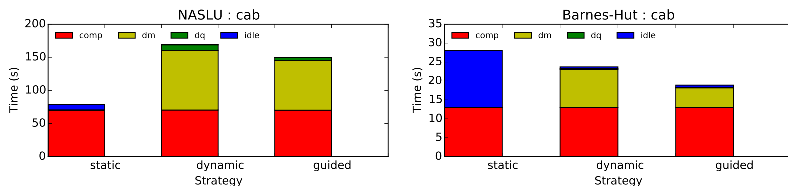


Figure 2: Breakdown of time. Computation in red, idle in blue, synchronization in green.

- Idle time eliminated with dynamic.
- Synchronization overheads for dynamic: small, although significant ($\approx 5\%$).
- Most of the overhead (shown in yellow) must be data movement, i.e., cache performance, locality.

Objective

Objective: Design a set of new scheduling strategies that handles all three causes of the problem, i.e., thread idle time, data movement, and synchronization overhead simultaneously for many applications and platforms, in the context of bulk-synchronous and loosely synchronous MPI applications.

Objective

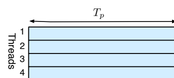
Objective: Design a set of new scheduling strategies that handles all three causes of the problem, i.e., thread idle time, data movement, and synchronization overhead simultaneously for many applications and platforms, in the context of bulk-synchronous and loosely synchronous MPI applications.

.

Key Idea of Solution: Intelligently combine the static and dynamic scheduling schemes to handle these three problems.

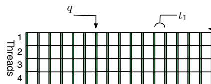
Hybrid Static/Dynamic Scheduling

```
#pragma omp parallel for schedule(static)
   for(int i=0; i<n; i++)
      c[i] += a[i]*b[i];
```



Susceptible to imbalance.

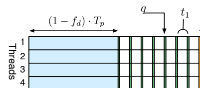
```
#pragma omp parallel
{
#pragma omp for schedule(dynamic)
  for (int i=0; i<n; i++)
    c[i] += a[i]*b[i];
}
```



Scheduler overhead stretches time.

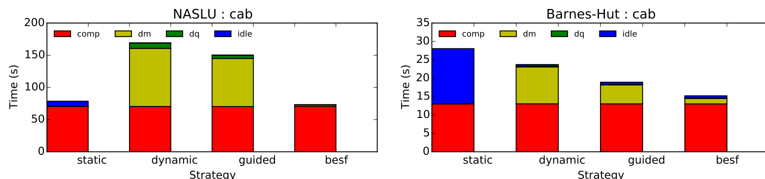
```
double fs = get_env_var(STATIC_FRACTION);
#pragma omp parallel nowait
{
#pragma omp for
  for (int i = 0; i < fs*n; i++)
      c[i] += a[i]*b[i];
}

#pragma omp parallel
{
#pragma omp for schedule(dynamic)
  for (int i = fs*n; i < n; i++)
      c[i] += a[i]*b[i];
}
```



Can reduce imbalance and sched ovhd. simultaneously.

Performance with Tuned Static Fraction



In *besf*, we use the best static fraction. See rightmost bar.

1. Balances the tradeoff between load balance and locality across applications and platforms.
2. Even for NASLU, the *besf* gives performance gains over *static*, i.e., it benefits from the small amount of dynamic load balancing.
3. EuroMPI 2010 : V. Kale and W. Gropp [statdyn] studies this idea in more depth.

Problems with the Basic Hybrid Scheduling Approach and Solutions

1. Search space for tuning the scheduler is large and complex.
2. Spatial locality in dynamic iterations disturbed, and not as good as static scheduling.
3. Previous work solved these problems.

Broader Problem

- Application and platform have many different mentioned factors involved, esp. as we go to exascale.
- Different circumstances require different scheduling tecniques.
- Our infrastructure and techniques make it efficient and easy to combine the desired scheduling strategies.
- We see if we can compose schedulers to handle all factors and circumstances.

Components of Example Composed Scheduler

- **hybSched**: hybrid static/dynamic scheduling with static fraction exposed to user.
- **uSched**: Use model-guided optimization to determine the static fraction, and then experimentally tune around this static fraction.
- **slackSched**: Adds awareness of slack to **uSched** created by MPI's collective operations.
- **vSched**: Enhance spatial locality of uSched by making it likely that each thread will execute spatially contiguous dynamic iterations, i.e., stagger the dynamic iterations.
- **comboSched**: Start with the staggered static/dynamic scheduling scheme defined in **vSched** above, and then do the slack-conscious adjustment described in slack-conscious scheduling section.

Original Code

```
#include "mpi.h"
#include <omp.h>
int main(int argc, char* argv[])
{
  // ...
  MPI_Init(&argc,&argv);
  // ...
  while(timestep < 1000){
#pragma omp parallel 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 3: Original MPI+OpenMP code.

Code Transformation

```
#include <mpi.h>
#include <omp.h>
#include "vSched.h"
int main(int argc, char* argv[]){
  int tid, numThrds, start, end = 0; double fd, fs;
  static LoopTimeRecord *record = NULL;
  MPI_Init(&argc, &argv); //..
  vSched_init(numThrds); //..
  while (timestep < 1000) {
    fd = predict_dynamic_fraction(&record); fs = 1.0 - fd;
#pragma omp parallel
    {
      tid = omp_get_thread_num(); 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 4: Code transformed to use composed scheduling strategy.

Setup

- **SNAP:** Regular mesh computation, implementation done in MPI+OpenMP, used for heat diffusion simulation.
- **miniFE:** Finite element unstructured mesh computation, implementation done in MPI+OpenMP, used for earthquake simulation.
- **Rebound:** N-body computation, implemented done in MPI+OpenMP, used for bio-molecular interaction simulation.

Experimental Setup: Intel Xeon 16-core processor with Fat-tree interconnect. Ran with 1 MPI process per node, and 1 thread per core (16 threads per node).

Scheduling Strategy Composition Results for Rebound

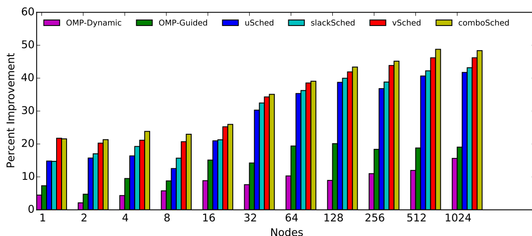


Figure 5: Percent speedup over OpenMP static scheduling for each scheduling strategy for the particle simulation code.

- Combining different techniques seems to add on benefits, i.e., they don't cancel benefit out.
- Reason: strategies based on complementary factors. Spatial locality in the vSched vs. reducing dynamic scheduling in slackSched.
- Small code change: 41,421 loc to 41,982 loc.

Scheduling Strategy Results for *SNAP*

CORAL MPI+OpenMP *SNAP* heat diffusion code performing regular mesh sweep.

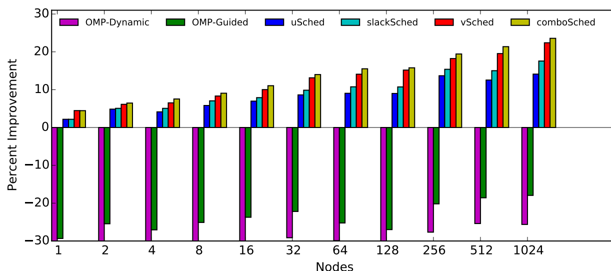


Figure 6: Percent speedup over OpenMP static scheduling for different loop scheduling strategies.

- Even when dynamic strategies worsen performance, the combined strategies show a small benefit.
- Optimizations in composition don't cancel each other out; combo sched gets 7.2% gains over *uSched*.

Related Work

- Locality aware scheduling by Chapman, deSupinski.
- Example of a history-based scheduling strategy by Zhang and Voss.
- Work stealing by Leiserson and group at MIT.

Conclusions

1. Need low-overhead scheduling strategies for improving performance of scientific applications.
2. Many different factors when running an application on a particular architecture. These require multiple low-overhead scheduling strategies to be used at once.
3. Results show that composition is additive in performance.
4. Future work: (a) Integration with OpenMP or other shared memory model; (b) composition with other scheduling strategies (which includes generalized methodology for composition for future scheduling strategies).

Current Work with GPUs

- Working with scientists in physics and mechanical sciences to integrate my techniques in a real-world CFD application.
- Apply work in the context of heterogeneous architectures.
- Use loop scheduling ideas on MIC.

Acknowledgements

- Todd Gamblin: Software
- Franck Cappello: Noise modeling based on fault-tolerance models
- Torsten Hoefler: amplification and slack description
- Laura Grigori: CALU scheduling
- Micheal T. Heath: Suggestions on theoretical analysis and formalization of performance models.

Thanks!