Exploiting Thread-Data Affinity in OpenMP with Data Access Pattern

Andrea Di Biagio Ettore Speziale Giovanni Agosta

Politecnico di Milano

September 2, 2011

Contents

1 Introduction

2 Locality Optimization

3 Conclusion

Contents

1 Introduction

2 Locality Optimization

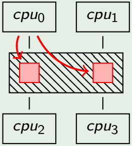
3 Conclusion

Motivating Scenario
Dealing with Non-Uniform Memory Accesses

Current trend in computer designs is towards NUMA architectures.

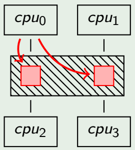
Current trend in computer designs is towards NUMA architectures.

Uniform Accesses

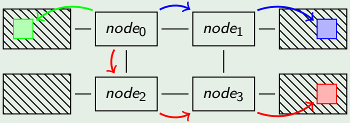


Current trend in computer designs is towards NUMA architectures.

Uniform Accesses



Non-Uniform Accesses

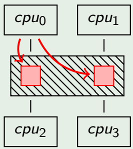


Motivating Scenario

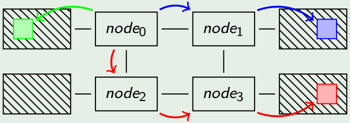
Dealing with Non-Uniform Memory Accesses

Current trend in computer designs is towards NUMA architectures.

Uniform Accesses



Non-Uniform Accesses



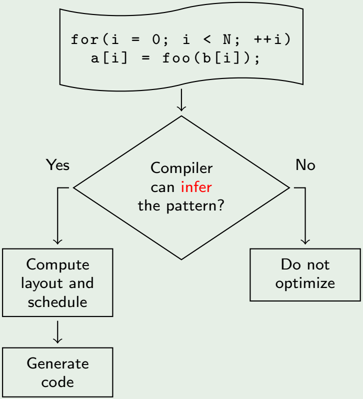
- Memory latency is strongly affected by the network topology
- Remote accesses are slower than local accesses
- Exploiting thread-data locality is necessary to minimize remote accesses

State of the Art Solutions

Static Techniques

Exploiting memory access pattern information is an hard task:

Static compiler analysis [1]

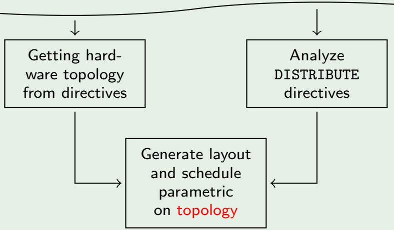


[1] E.g. polyhedral analysis

Some directives can help the compiler:

Explicit memory distribution

```
!HPF$ PROCESSORS LINEAR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (CYCLIC) ONTO LINEAR :: A, B
REAL A(N), B(N)
DO I=1, N
      A(I) = FOO(B(I))
```

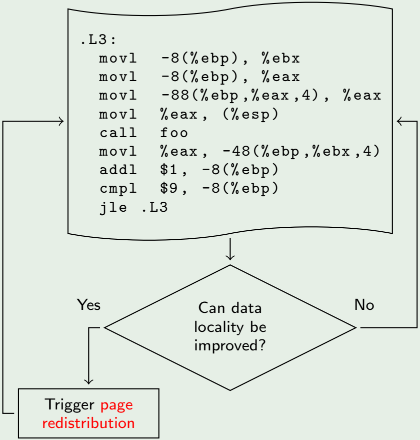


State of the Art Solutions

Dynamic Techniques

Actual access pattern can be spotted at runtime:

Runtime techniques



Contents

1 Introduction

2 Locality Optimization

3 Conclusion

Proposed Locality Optimization

Combine Pattern Hints with Dynamic Scheduling

Key idea:

- Tag data accessed by parallel loops with pattern:
  - Hint on how data will be accessed
- Schedule parallel loops at runtime:
  - Exploit pattern hints to dispatch iterations as near as possible to accessed data

Proposed Locality Optimization
Combine Pattern Hints with Dynamic Scheduling

Key idea:

- Tag data accessed by parallel loops with pattern:
  - Hint on how data will be accessed
- Schedule parallel loops at runtime:
  - Exploit pattern hints to dispatch iterations as near as possible to accessed data

Benefits:

- Can adapt to different workloads
- Minor code modification
- Architecture agnostic

Proposed Locality Optimization
Combine Pattern Hints with Dynamic Scheduling

Key idea:

- Tag data accessed by parallel loops with pattern:
  - Hint on how data will be accessed
- Schedule parallel loops at runtime:
  - Exploit pattern hints to dispatch iterations as near as possible to accessed data

Benefits:

- Can adapt to different workloads
- Minor code modification
- Architecture agnostic

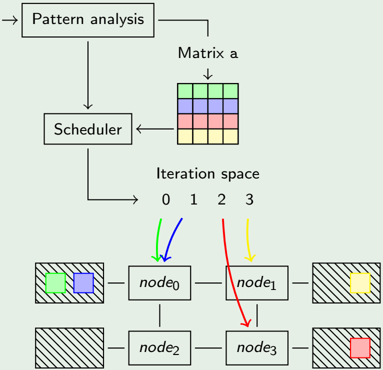
Challenges:

- Efficient scheduler needed

By row partitioning:

Pattern Example

```
#pragma omp for pattern(a[^1, *])
for(i = 0; i < 4; ++i)
  for(j = 0; j < 4; ++j) {
    a[i][j] = foo(a[i][j]);
    if(j < 3)
      a[i][j] += bar(a[i][j + 1]);
    else if(i < 3)
      a[i][j] += baz(a[i + 1][0]);
  }
```



Our dynamic approach does not rely on data distribution:

- iterations are scheduled on those nodes who see data locally

Implementation Details

Runtime Organization

Our dynamic approach does not rely on data distribution:

- iterations are scheduled on those nodes who see data locally

To balance workload, a multi stage scheduling algorithm has been designed:

- threads are split into groups
- each group is composed of threads running on a same node
- each group is assigned to a local queue of iterations

Implementation Details

Address Space Partitioning

Upon entering a parallel loop, the iteration space is partitioned into blocks:

Iteration Grouping

Let I_1, I_2 two adjacent elements of the iterations space I. They belong to the same block if:

- Access the same set of pages, or
- Access to pages mapped on the same node, or
- Access to unmapped pages

To minimize the complexity of such algorithm we can enforce a maximum number of blocks.

The blocks are analyzed in parallel:

- Each block is scored with respect to node locality
- Blocks moved to local queues corresponding to the node with the greatest affinity
- Blocks with no score assigned to a global queue

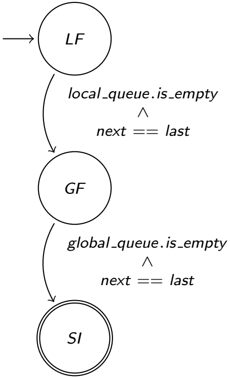
Now loop execution can begin.

Implementation Details

Iteration Dispatching

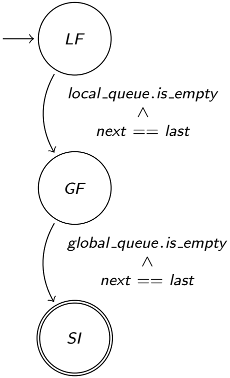
Each thread executes the same scheduling algorithm:

Each thread executes the same scheduling algorithm:



Implementation Details
Iteration Dispatching

Each thread executes the same scheduling algorithm:



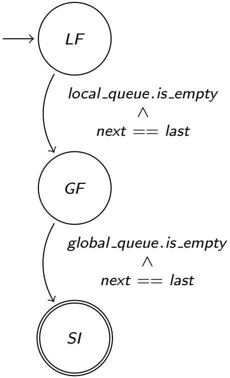
Local Fetching

- Maximize memory locality

Implementation Details

Iteration Dispatching

Each thread executes the same scheduling algorithm:



Local Fetching

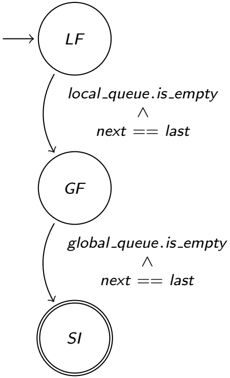
- Maximize memory locality

Global Fetching

- Relies on first-touch policy

Implementation Details
Iteration Dispatching

Each thread executes the same scheduling algorithm:



Local Fetching

- Maximize memory locality

Global Fetching

- Relies on first-touch policy

Steal Iterations

- To further balance workload

Tests performed on a quad-node quad-core Opteron system:

- Interconnect is a square
- Tests performed using 16 threads

Prototype implemented for the *OpenMP* programming model.

The impact of work-stealing strategy has been tested against two different configurations:

Worst uses the furthest neighbour selection policy

Best uses the nearest neighbour selection policy

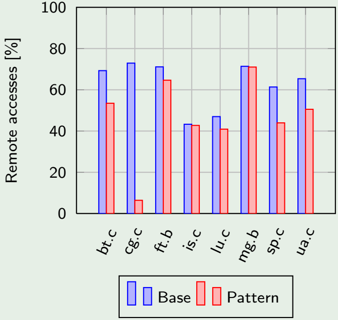
Selection policy based on NUMA distances.

Experimental Evaluation

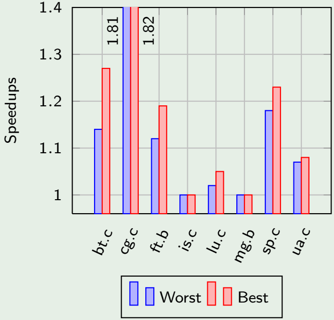
NAS Benchmarks

Baseline is llvm-gcc 4.2 *libgomp*

Remote Accesses



Speedups



Work Distribution

Bench	Blocks fetched from Local [%]	Blocks fetched from Global [%]	Blocks fetched from Steal [%]	Speedup [2] Worst	Speedup [2] Best
bt.c	65.72	0.01	34.27	1.14	1.27
cg.c	99.61	0.03	0.36	1.81	1.82
ft.b	76.40	0.00	23.60	1.12	1.19
is.c	66.67	0.00	33.33	1.00	1.00
lu.c	80.21	0.21	19.58	1.02	1.05
mg.b	35.16	22.26	42.58	1.00	1.00
sp.c	70.03	0.00	29.97	1.18	1.23
ua.c	88.36	0.13	11.51	1.07	1.08

[2] W.r.t. neighbour policy

Contents

1 Introduction

2 Locality Optimization

3 Conclusion

Final Remarks

Patterns:

- enable automatic work distribution
- small syntactic extension to the OpenMP programming model
- programmers do not need to explicitly distribute memory or understanding the system topology

Future directions:

- automatic patterns recognition/filtering
- integration with page-migration [3] techniques

[3] E.g. next-touch

That's All, Folks!

Questions are welcome