Exploiting fine- and coarse-grained parallelism using a directive based approach

Arpith C. Jacob, Michael Wong

Ravi Nair, Alexandre E. Eichenberger, Samuel F. Antao, Carlo Bertolli, Tong Chen, Zehra Sura, Kevin O'Brien

IBM

Summary

- Modern HPC clusters are heterogeneous, employing light- and heavy-weight cores with accelerators
- Transparently exploiting heterogeneous clusters is getting increasingly complex
  - No single approach for multicores, accelerators, and clusters
  - Often requires vendor specific languages and toolchains
  - Or unfamiliar languages: X10, Fortress, Chapel, UPC
  - Or high overhead frameworks such as Hadoop
- OpenMP 4.0 introduces an offload model suitable for accelerators with disjoint, non-coherent memory

We present a compiler and runtime that uses OpenMP 4.0 to offload kernels to nodes in a cluster

Related work

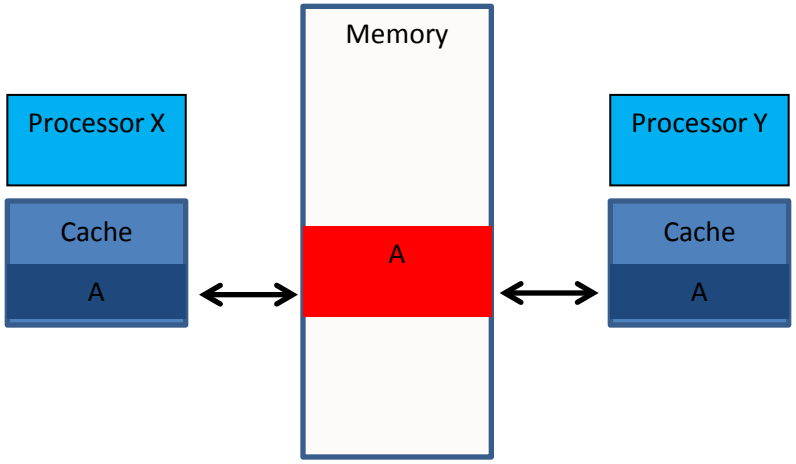
- SDSM
- TreadMarks
- Intel’s Cluster OpenMP

The OpenMP Accelerator Model

- Host-centric model offloads code + data to target devices
- Programmer identifies code to offload using target directive
  - A target region accepts standard OpenMP parallel directives
- Model defines a data environment for the host and target devices that may be disjoint
  - User may not assume that the host and the target devices share an address space
- User migrates data using map clause and target data directive

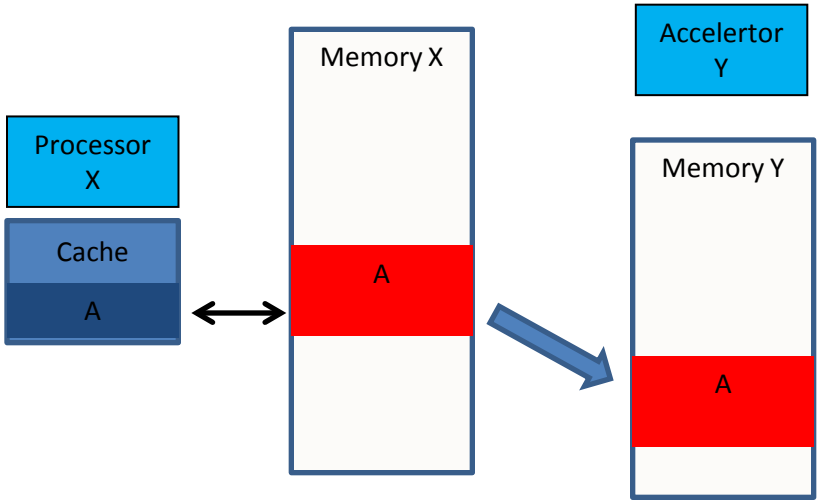
Data mapping: shared or distributed memory (courtesy of Eric, James, Christian, Michael)

Shared memory



- The corresponding variable in the device data environment *may* share storage with the original variable.
- Writes to the corresponding variable may alter the value of the original variable.

Distributed memory



OpenMP Accelerator Example

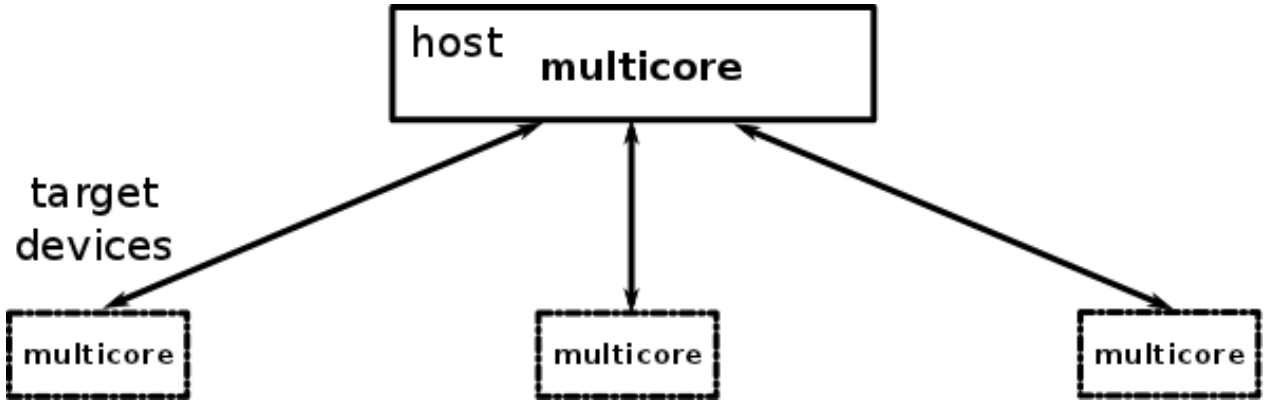
Listing 1.1: Matrix-matrix multiply offloaded to a target device for acceleration.

```
1  double A[P][R], B[R][Q], C[P][Q];
2
3  void main() {
4  // Initialize arrays
5
6  // Offload loop nest for acceleration onto device #1
7  #pragma omp target map(to: A[0:P][0:R], B[0:R][0:Q]) map(tofrom: C[0:P][0:R]) device(1)
8  // Execute iterations of loop i in parallel on 16 accelerator cores
9  #pragma omp parallel for num_threads(16)
10 for (int i=0; i<P; i++)
11   for (int j=0; j<Q; j++)
12     for (int k=0; k<R; k++)
13         C[i][j] += A[i][k] * B[k][j]
14
15 // Computed array C is available on the host
16 }
```

An Offloading Model for a Cluster

- OpenMP 4.0 offload model is designed for accelerators
- Can it be extended to a cluster?

- Master offloads code and data to workers
- Workers co-opt local threads or may offload to their local accelerators



- **A single program could scale to multicores, accelerators and multiple nodes**

Execution Model

- Model defines a clique of shared-memory domains laid out as a tree
- Execution initiates on a host initial thread
- Initial thread on target devices are inactive until activated by a host thread
- Target construct offloads control from host to other shared-memory domains
- Initial thread on host or target may co-opt other threads for parallel execution, for example, using worksharing constructs

Exploiting Multicores

```
#pragma omp parallel for num_threads(64)
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

Exploiting GPUs

```
#pragma omp target
#pragma omp parallel for num_threads(1024)
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

Exploiting Multiple Nodes

```
for (i=0; i<10; i++)
    #pragma omp target device(i)
    #pragma omp parallel for num_threads(64)
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

Definitions

- **Shared-memory domain:** Logical realm of processors with storage accessible through a global address space. Cached data within realm is kept coherent by hardware.
- **Host Domain:** Shared-memory domain on which a program starts execution
- **Target Domain:** One or more shared-memory domains onto which code and data may be offloaded

Memory Model

- Map clause creates corresponding variable on device for every original variable on the host
  - Naming operation identifies distinct host and target storage locations
  - Data transfer operation moves data between the two locations
- Device data are shared across processors within the domain

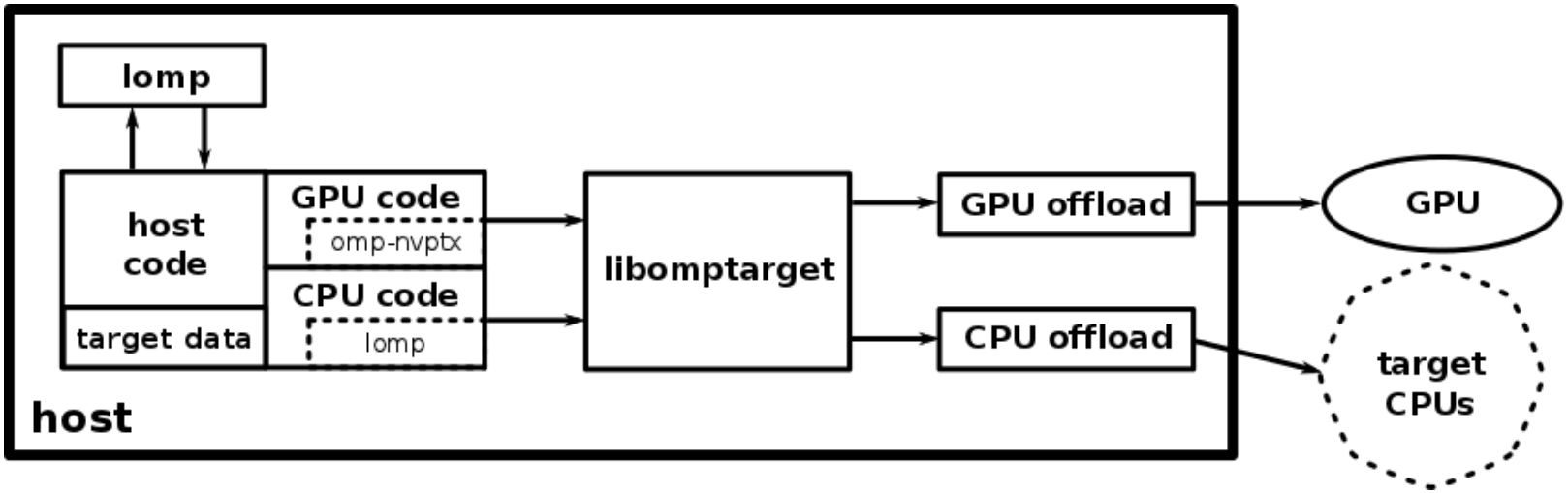
Implementation Details

- We have implemented our offloading model for clusters in Clang/LLVM
- Uses community based implementation for OpenMP in Clang Status:
  - 3.7 has full OpenMP 3.1 support, released in August 2015
  - Now upstreaming OpenMP 4.0 directives to 3.8, planned Feb 2016, Accelerator support of primary interest
  - Have a target–independent interface with LLVM IR
  - See talk at LLVM/clang Dev Con on
    - "OpenMP GPU/Accelerator support Coming of Age in Clang"
- We have added support for offload directives

People involved bringing OpenMP to Clang

- Michael Wong, IBM
- Alexey Bataev, Intel
- Sergey Ostanevich, Intel
- Samuel Antao, IBM
- Andrey Bokhanko, Intel
- Carlo Bertolli, IBM
- Eric Stotzer, TI
- Kelvin Li, IBM
- Hal Finkel, Argonne National Lab
- Arpith Jacob, IBM
- **Ravi Nair, IBM**
- **Tong Chen, IBM**
- **Zehra Sura, IBM**
- Sunita Chandrasekaran, University of Delaware
- Alexandre Eichenberger, IBM
- Kevin O'Brien, IBM
- Guansong Zhang, AMD
- Ravi Narayanaswamy, Intel

Implementation Details



- **Lomp :** OpenMP runtime for multicores on host and target devices
- **Libomptarget:** Manages device data environment (reference counting) and code offloading
- **Target specific plugins:** Implements low-level memory management, data transfer, and code execution commands (MPI for clusters; CUDA for NVIDIA GPUs)

Target Plugin: Cluster Offloading

- Single host binary executes on all MPI ranks
- Workers (rank > 0) enter event loop at program startup and wait on Host (master)
- Master (rank 0) communicates with workers via MPI command messages
- Commands: allocate memory on worker, delete memory, copy to/from worker, load and execute target region
- Uses thread safe MPI library to manage multiple workers simultaneously

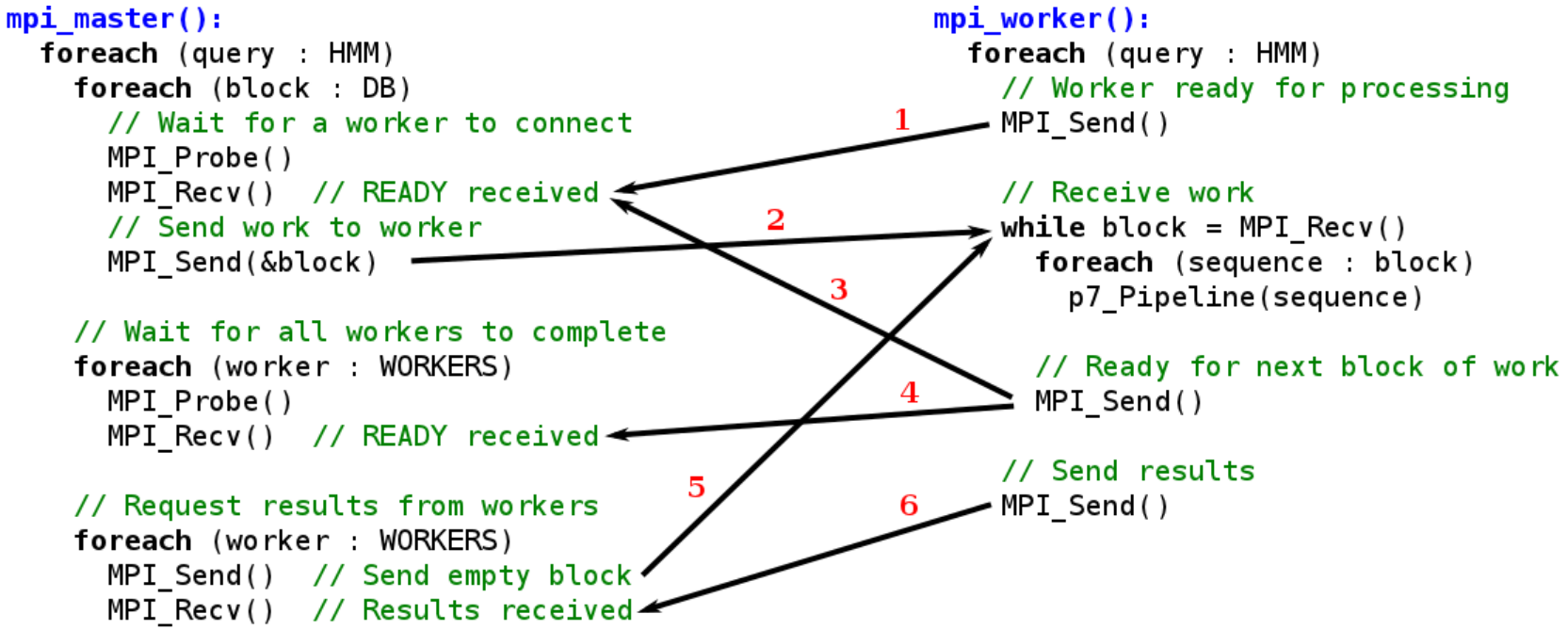
Results: HMM Database Search

- Tested on HMM Search (HMMER 3.1b2)
- Compares Query HMM (Hidden Markov Model of protein family) against database of 3.2 million protein sequences
- We partition database and dynamically offload database search to between 2 to 64 ranks (16 ranks per node)
- Experiments run on 4 node IBM Power 8 cluster (4 sockets, each with 6 cores)
- We use Open MPI library version 1.8.5

- We compare our results against native MPI implementation written by authors of HMMER software

Native MPI Implementation

- **Steps 1-3**: Co-opt workers and dynamically offload database partitions
- **Step 4:** Barrier synchronize all workers for work completion
- **Step 5-6**: Request and receive results from workers



OpenMP Implementation

- Master host thread serially iterates over query HMMs
- ***Parallel for*** starts **num_devices** host I/O threads
- Each I/O thread offloads query+DB partition; initiates execution on device; and collects results on completion

```
target():
  foreach (query : HMM)
    // In parallel, activate a dedicated thread for each device
    // and schedule work dynamically
    #pragma omp parallel for schedule(dynamic) \
                num_threads(num_devices)
    foreach (block : DB)
      device = omp_get_thread_num()
      // Thread sends work to its device and waits for results
      #pragma omp target device(device) map(to:block, query) \
                  map(from: results) {
        foreach (sequence : block)
          p7_Pipeline(sequence)
      }
```

1

2

OpenMP Implementation

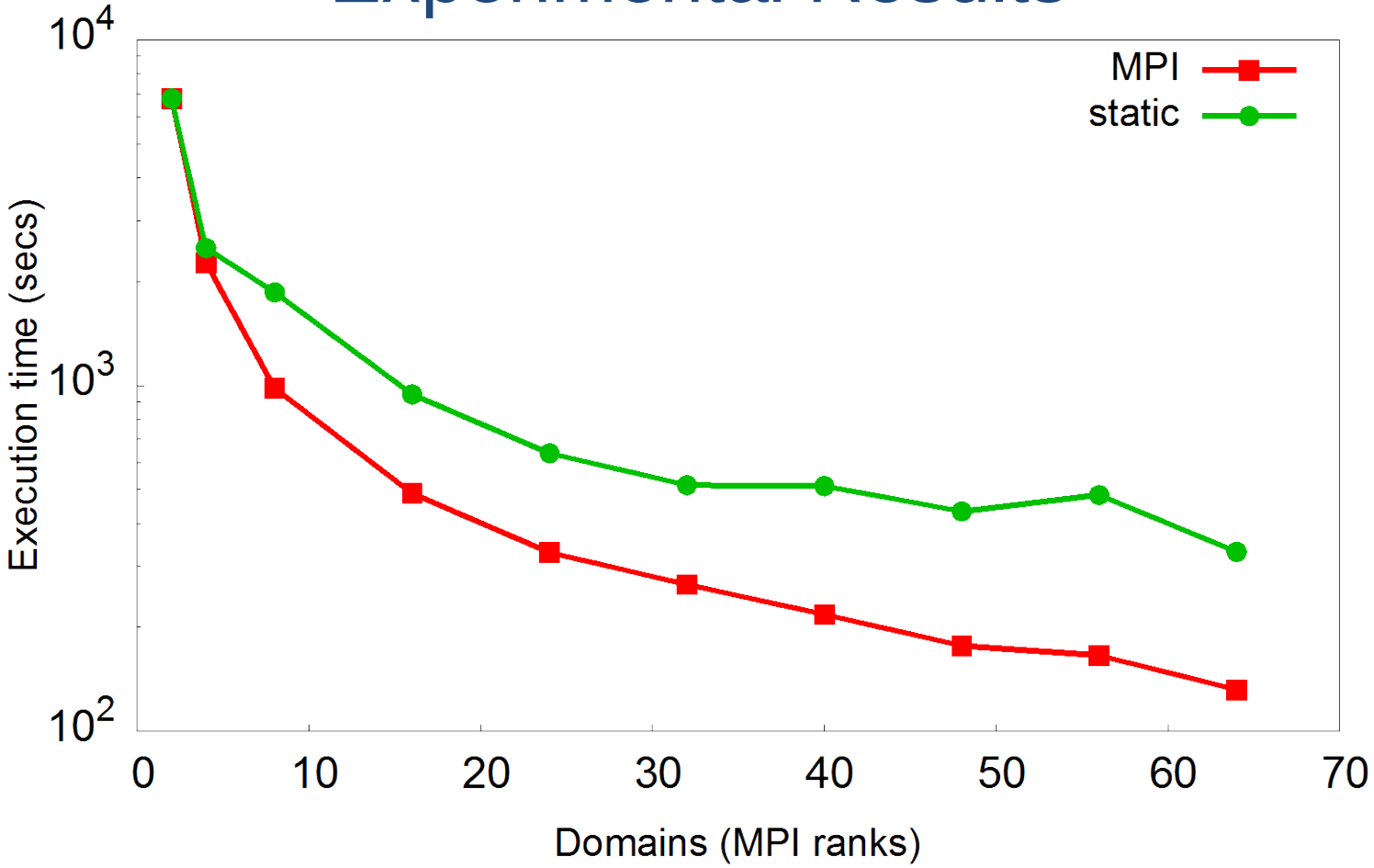
- **Worksharing construct** partitions DB across ranks
- **Dynamic** schedule achieves load balancing
- Query/DB copy and results aggregation via **map** clauses
- **Barrier** after parallel pragma synchronizes workers

```
target():
  foreach (query : HMM)
    // In parallel, activate a dedicated thread for each device
    // and schedule work dynamically
    #pragma omp parallel for schedule(dynamic) \
              num_threads(num_devices)
    foreach (block : DB)
      device = omp_get_thread_num()
      // Thread sends work to its device and waits for results
      #pragma omp target device(device) map(to:block, query) \
                map(from: results) {
        foreach (sequence : block)
          p7_Pipeline(sequence)
      }
```

1

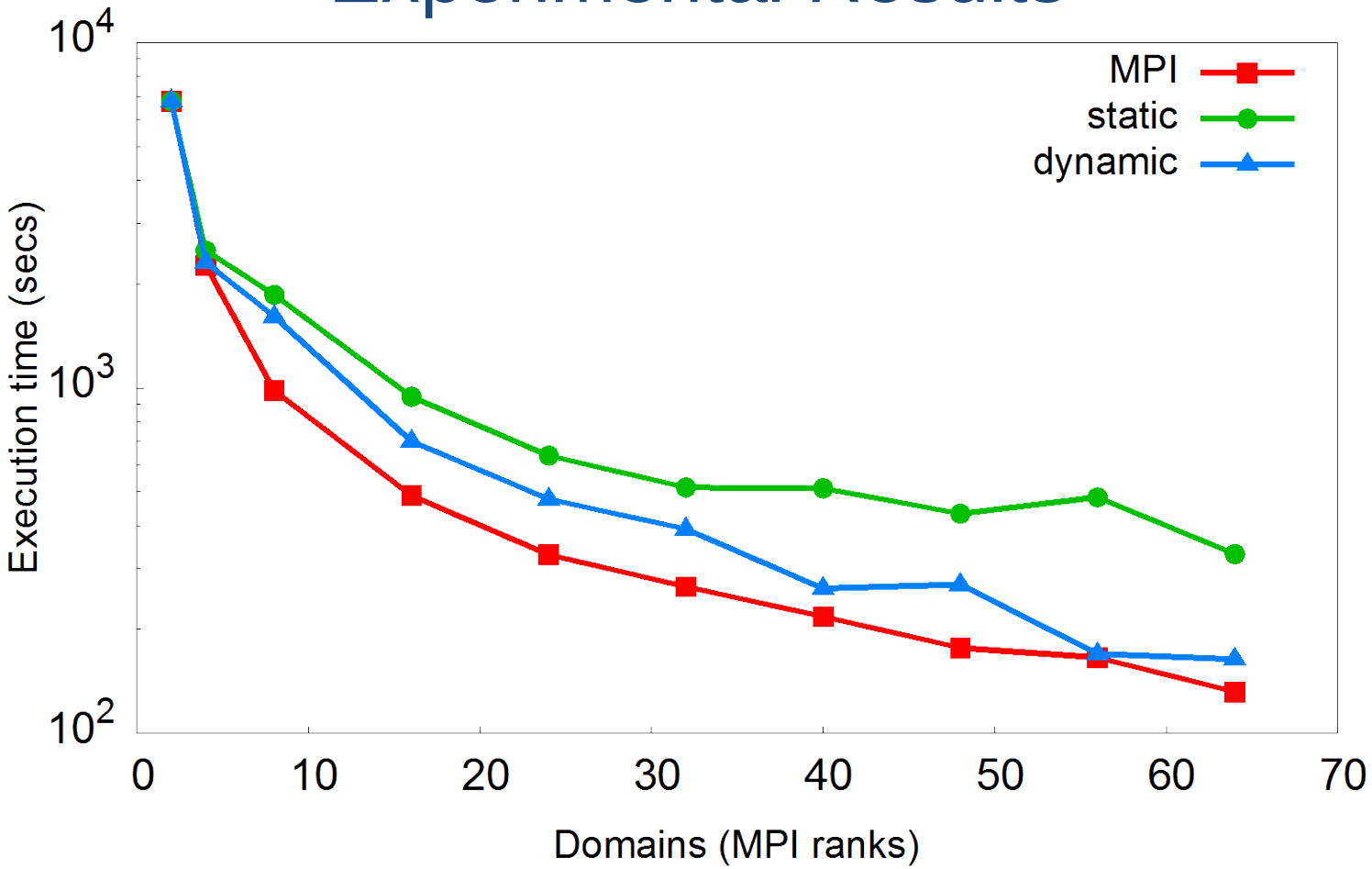
2

Experimental Results



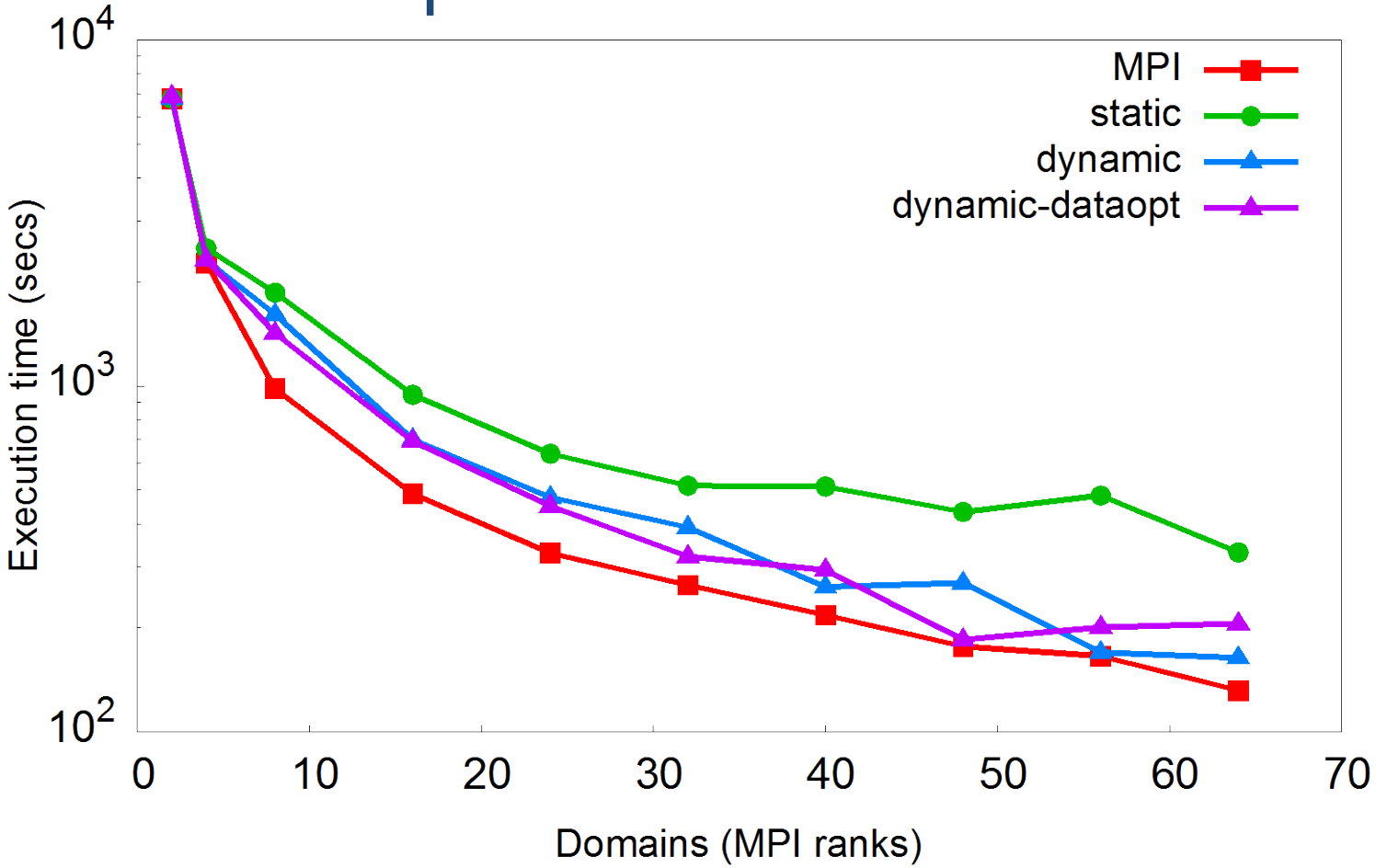
- With **static** schedule we observe scaling behavior comparable to native MPI implementation
- Has considerable overhead

Experimental Results



- **Dynamic** schedule reduces runtime up to 50%
- Load balancing achieved through OpenMP runtime

Experimental Results



- Establish persistent data environment using target data directive to reduce overhead
- Investigating additional techniques to reduce data management overhead (reuse device memory)

Future Work

- Exploiting cores in a node: use worksharing construct to distribute database sequences across device cores
- Beyond offloading: asynchronous updates of host variables by the target; arbitrary communication between devices
- Nested target regions: to exploit accelerators within a node

Conclusions

- We have introduced a model and implementation to program a cluster with OpenMP offload directives
- Results on bioinformatics application shows good scaling
- Compared to native MPI implementation OpenMP abstractions do work scheduling, load balancing, data transfer, and synchronization of nodes