Exploiting Fine- and Coarse-Grained Parallelism Using a Directive Based Approach

Arpith C. Jacob[1(✉)], Ravi Nair[1], Alexandre E. Eichenberger[1], Samuel F. Antao[1], Carlo Bertolli[1], Tong Chen[1], Zehra Sura[1], Kevin O'Brien[1], and Michael Wong[2]

[1] IBM T.J. Watson Research Center, 1101 Kitchawan Rd., Yorktown Heights, NY, USA
{acjacob,nair,alexe,cbertol,chentong,zsura,caohmin}@us.ibm.com
[2] IBM Software Group, Toronto, ON, Canada
michaelw@ca.ibm.com

Abstract. Modern high-performance machines are challenging to program because of the availability of a wide array of compute resources that often requires low-level, specialized knowledge to exploit. OpenMP is an effective directive-based approach that can effectively exploit shared-memory multicores. The recently introduced OpenMP 4.0 standard extends the directive-based approach to exploit accelerators. However, programming clusters still requires the use of other specialized languages or libraries.

In this work we propose the use of the target offloading constructs to program nodes distributed in a cluster. We introduce an abstract model of a cluster that defines a clique of distinct shared-memory domains that are manipulated with the target constructs. We have implemented this model in the LLVM compiler with an OpenMP runtime that supports transparent offloading to nodes in a cluster using MPI. Our initial results on HMMER, a widely used Bioinformatics tool, show excellent scaling behavior with a small constant-factor overhead as compared to a baseline MPI implementation. Our work raises the intriguing possibility of a natural progression of a program compiled for serial execution, to parallel execution on a multicore, to offloading onto accelerators, and finally extendible with minimal additional effort onto a cluster.

1 Introduction

Modern computers are difficult to program because of the use of a broad spectrum of compute resources such as SIMD, SIMT, light and heavyweight multicores, accelerators such as GPUs, clusters, and even rented cloud platforms that may all need to be exploited for high performance. It is difficult and time consuming for the average programmer to reason about fine- and coarse-grained parallelism suitable for these resources and then express this parallelism using low-level language extensions or libraries. Many languages have been introduced to address some of these challenges, including X10 [2], Fortress, Chapel [1], Co-array Fortran [12], and UPC [5], but none have gained widespread acceptance.

C. Terboven et al. (Eds.): IWOMP 2015, LNCS 9342, pp. 30–41, 2015.
DOI: 10.1007/978-3-319-24595-9_3

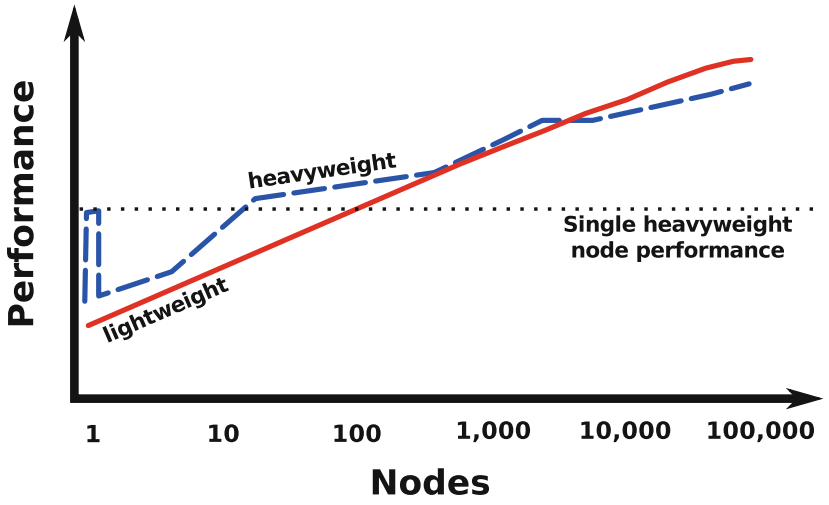

Fig. 1. Comparative performance of light and heavyweight nodes on the Graph500 benchmark. The trend lines are extrapolated from Fig. 12 of Koggee [8].

To illustrate why this is a problem Fig. 1 shows performance of the Graph500 breadth-first search benchmark as a function of node count, illustrated as trend lines extrapolated from Fig. 12 of Kogge [8]. Curves for light and heavyweight nodes are depicted. Heavyweight cores provide maximum compute performance in a single node using 16 or more cores clocked at a high rate, sharing coherent, high-bandwidth memory. They are the easiest to program and suitable for a large number of applications. In contrast, lightweight nodes use simpler cores with lower-bandwidth memory that consume less power. As a consequence, they are better able to scale to thousands of nodes in a system.

Figure 1 shows that the best heavyweight single-node implementation is equivalent in performance to a large multiple of lightweight nodes. It is obviously beneficial to first exploit the compute resources and high-bandwidth communication through shared-memory in a single node and only move to a lightweight cluster when the organization's data requirements explode. However, an actor may unnecessarily commit early on to a cluster solution due to perceived "big data" needs or to simply scale early software development investment.

To program clusters, frameworks such as Hadoop using the MapReduce model have become popular but it includes considerable overheads. This may be required for fault-tolerance in large-scale clusters but is inefficient for most users. Indeed, there is evidence that MapReduce is commonly used for datasets less than 100 GB in size [16], and with several times this size available as RAM in a single node[1], it may be prudent to first exhaust single-node solutions. Consequently, what is desired is a software solution that can exploit compute resources in a single node but seamlessly scale to multiple nodes as data size increases.

OpenMP is a directive-based parallel programming standard for shared-memory multicores. The introduction of offloading constructs in the OpenMP 4.0

[1] For example, the IBM Power® System E880 is configurable up to 16 TB. See http://www-03.ibm.com/systems/power/hardware/e880/.

standard extends directive-based programming to accelerators within a node that may have non-coherent, disjoint memory. The goal of this work is to extend the offloading model to the wider scope of a cluster so as to avoid the use of low-level libraries like MPI. This raises the intriguing possibility of a single program that may be compiled for serial execution, parallel execution on a multicore, offloaded to accelerators, and finally extended with minimal additional effort to a cluster.

The rest of this paper is organized as follows. Section 2 first discusses past efforts that use OpenMP to program clusters. Section 3 summarizes the offloading model of OpenMP 4.0 and in Sect. 4 we describe our model for using these constructs to program a cluster of nodes. Section 5 describes the implementation of our model in the LLVM compiler and we present preliminary results in Sect. 6. Section 7 discusses extensions to the model and implementation that we plan to explore in the future before we conclude in Sect. 8.

2 Related Work

Existing work implements OpenMP 3.1 and prior versions on distributed machines by translating shared-memory programs onto a Software Distributed Shared Memory (SDSM) runtime [7,13] or directly to MPI [11]. An SDSM runtime that transparently keeps memory consistent between nodes was first used in TreadMarks [7] to execute OpenMP programs and was later incorporated into Intel's Cluster OMP [6]. These solutions exploit the relaxed consistency model of OpenMP to cache memory locally and improve performance but problems remain due to memory sharing overheads and frequent global synchronization. Static compiler analyses aim to eliminate barriers and aggressively privatize shared variables but the fine-grained communication pattern of OpenMP 3.1 is a fundamental mismatch for distributed machines.

The distinguishing feature of our work is our abstraction of a machine as a clique of shared-memory domains. Rather than forcing the abstraction of a single shared-memory domain onto a distributed cluster and using compiler analyses to bridge the semantic gap, we rely on an expressive yet simple programming abstraction that is closer to the underlying machine.

Our work is the first to use OpenMP 4.0 offload constructs to program distributed systems. We exploit these constructs to represent our model of disjoint domains of shared memory. Target directives are used to partition a program into distinct regions, each of which can then exploit shared-memory semantics via the standard suite of OpenMP 3.1 constructs. Data mapping clauses allow the user to bridge shared-memory domains via explicit data movement. Our model is easy to comprehend and it allows the user to effectively exploit coarse- and fine-grained parallelism.

3 Background: OpenMP Accelerator Model

The OpenMP 4.0 specification [15] significantly extends the capability of a directive based programming approach to exploit coarse-grained parallelism within a

node. It introduces an execution and data model for an abstract accelerator that enables a programmer to exploit heterogeneous cores using *device* directives. A node is assumed to contain a host device, typically a multicore processor, attached to one or more accelerators termed *target devices*.

The programming model is host-centric and allows offloading of execution control to the device. Code regions to be offloaded, including functions, are explicitly identified using the *target* directive. The target region accepts standard OpenMP directives and uses the fork-join paradigm to exploit cores within the accelerator.

The accelerator memory model defines a data environment for the host and target devices. To support both shared- and distributed-memory systems the standard specifies that the programmer may not make any assumptions regarding data sharing between devices. The user is required to explicitly migrate data between the two environments and the standard provides the *target data*, *declare target*, and *target update* directives for this purpose.

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 }
```

Listing 1.1 shows an OpenMP 4.0 program offloading matrix-matrix multiply onto an accelerator. The *target* directive first establishes a data environment on the device with three arrays A, B, and C before ofloading the computation. Upon completion, the array C is transferred back to the host.

The user is required to guarantee a program free of data races that may otherwise arise due to concurrent execution on devices. These relaxed definitions allow the standard to support a wide spectrum of accelerators including GPUs, FPGAs, DSPs, and PiM devices.

4 An Offloading Model for a Cluster

While not originally defined with node-level parallelism in mind, the flexibility of the OpenMP accelerator model raises the intriguing possibility of exploiting

coarse-grained parallelism across nodes in a cluster. In particular, the assumption in the model that the device data environments may be distinct allows the possibility of more efficient execution on a cluster. We start by first defining an offloading model to program a cluster.

4.1 Definitions

Shared-Memory Domain. An implementation defined logical realm with storage accessible through a global address space by one or more processors within it. Data may be cached by processors but the model assumes that caches are kept coherent.

Host Domain. The shared-memory domain on which a program starts execution.

Target Domain. One or more shared-memory domains other than the Host onto which code and data may be offloaded.

4.2 Execution Model

The execution model defines a *clique* of one or more shared-memory domains laid out in a multi-level tree hierarchy. A program begins with a single thread of execution on some implementation defined domain called the Host. All other domains in the clique are inactive at startup and must be expressly activated by the Host.

When the initial thread on the Host encounters a parallel worksharing construct it may spawn additional threads and distribute work for parallel execution on processors within the same shared-memory domain. When a thread on a domain encounters a target construct (or in general, a target boundary) there is a transfer of data and control from one shared-memory domain to another.

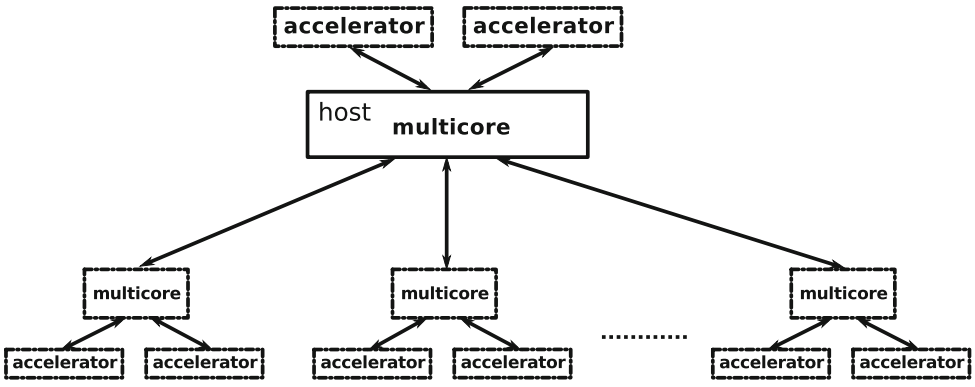

Fig. 2. A clique of shared-memory domains laid out on general-purpose microprocessors and accelerators realized as a multi-level tree hierarchy by the offload model. Domains communicate via messages on a hierarchical network and optionally via files on a shared disk.

The thread offloads code and data to the Target domain and program control is transferred to an initial thread on the Target. The original thread waits at the end of the construct until control is returned from the Target. The Target thread executes in its distinct shared-memory domain and may co-opt other threads for parallel execution on the processors within the domain.

4.3 Memory Model

As mentioned previously, in our memory model every domain has a distinct data environment with storage coherently addressable by all processors contained within it. Since every domain has an independent address and storage space, variables declared on the Host are only addressable by that domain, and those declared on a Target are only addressable by the Target domain. Domain-private variables are shared across processors within the same domain and follow the relaxed consistency model of traditional OpenMP.

Data sharing between domains is explicitly controlled by the programmer. Any Host variable may be mirrored across domains on one or more Targets through a synchronous operation called "mapping". Mapping creates a distinct Corresponding copy on a particular target for an Original variable on the host. Code within the Host and a Target domain may only access and modify the Original and Corresponding values respectively. Original and Corresponding variables may only be synchronized at target boundaries. Outside the explicit movement of data between domains at target boundaries there is no mechanism to address storage across domains.

Mapping is first and foremost a naming operation that assigns a common identifier to distinct storage locations on the Host and one or more Target domains. In addition, it is a data transfer operation that moves data between the Host and one or more Targets (or vice versa).

Mapping points and the direction of the mirroring are explicitly specified by the user. Host local variables are mapped at these well defined interfaces identified through user-specified directives. For example, data is typically transferred to a Target before its invocation and back to the Host upon completion of work. Host global variables that are explicitly mapped to target domains are mirrored at program startup before any user code executes.

The kinds of programs that work well under these assumptions include those that can partition data (such as an array or a set of files) into distinct subsets across target domains.

5 Implementation

We have implemented our offloading model for clusters within LLVM [10], a powerful open-source intermediate representation and optimizer, coupled with Clang [3], a frontend for C/C++ based languages. A recent joint effort by several players has been adding full OpenMP support in Clang [14]. We have extended this implementation to provide offloading support for the accelerator model.

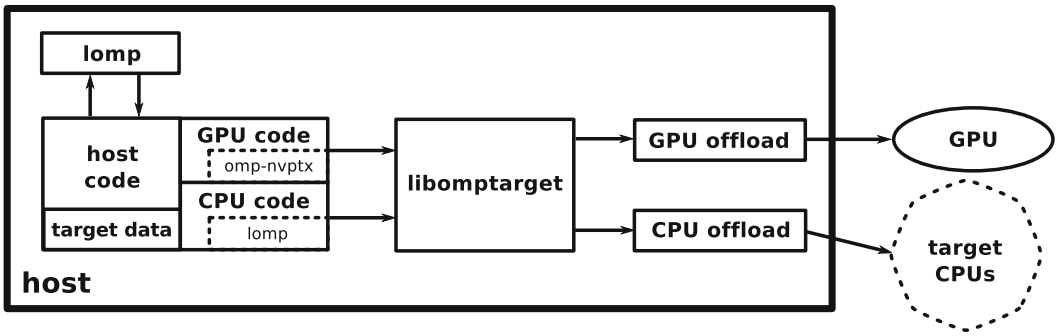

Fig. 3. Compiler generated fat binary containing offloaded code for a GPU and a Cluster (labeled CPU) target. Offloading support is provided by a generic target library and device-specific modules. Non-offloaded OpenMP code is supported by our *lomp* library.

Briefly, support for the offloading directives are added in the parser and semantic analyzer. Each target region is outlined into a standalone function. Next, the driver calls the host and one or more target toolchains to obtain distinct object files. Finally, the linker links the host object file into the executable and embeds the target object "as is", resulting in a fat binary.

Figure 3 illustrates the fat binary with included offloading code generated for the GPU and a cluster node.

5.1 Runtime Support

In order to generate the behavior defined in the OpenMP specification, the compiler interfaces with a series of runtime libraries for host and target regions. The design provides distinct support for offloading and non-offloading OpenMP directives. We use the IBM lightweight OpenMP (*lomp*) runtime [4] for parallelization of the non-offloaded host code. Offloading support is provided by a target agnostic library coupled with low-level device-specific plugins. The former is labeled *libomptarget* in Fig. 3, while the plugins for the two targets shown in the figure are labeled *GPU* and *CPU offload*.

Target Agnostic Offloading Library. The offloading runtime library is completely target agnostic allowing Clang to move the complexity of offloading away from the compiler. This library calls low-level functions in the plugins to allocate memory on a device and transfer data between the host and the target. It also manages the address mapping between Original and Corresponding variables and tracks data references for safe de-allocation. Finally, the library initializes a target device with offloaded code using the plugins, prepares parameters, and initiates execution of the kernel.

Target Specific Plugins. The target agnostic library triggers actions on target devices supported by a set of device-specific plugins. These plugins have a pre-determined interface and are located and loaded at runtime by the target agnostic component. They are used to drive low-level actions on the device.

We have implemented a plugin for clusters on top of MPI. Upon program startup every shared-memory domain is activated and all target domains enter an event loop awaiting communication from the host via MPI messages. The host can direct each domain to allocate, de-allocate, and transfer data via MPI.

Recall that a target regions are outlined in functions and linked together into an elf object embedded in the fat binary. At program startup the host transfers the object to every potential device and is immediately loaded by the event loop on the target domains. When the host encounters a target region it sets up the data environment on the specified device and directs the event loop to execute the function.

Since multiple threads on the host may concurrently offload to distinct targets, the plugin requires an MPI library that supports thread safe execution.

Target Specific Runtime Libraries. The offloaded device code will itself have OpenMP directives and therefore requires runtime library support implemented for the device. Since we are offloading to general-purpose CPUs, standard *lomp* is sufficient for this purpose.

6 Preliminary Results

To illustrate the application of our programming model we have accelerated an important Bioinformatics application, HMMER[2], on a cluster. The application finds homologs of a protein family by comparing its Hidden Markov Model (HMM) representation against a database of protein sequences. A typical search compares tens of thousands of HMMs against tens of millions of sequences and is a compute-intensive task. In this work we offload this search to nodes within a cluster. This benchmark exhibits the typical master-slave programming pattern.

The latest release of HMMER includes code to parallelize the search across nodes within a cluster using traditional MPI. We have summarized the code and illustrated the flow of control initiated via MPI messages in Fig. 4. The master and the worker nodes iterate over each query HMM in lock step. For each query, the database is partitioned into blocks that are offloaded to the next available worker dynamically to avoid load imbalance. Once the entire database is processed the master implements a barrier to synchronize across all workers, before finally requesting and receiving their results. The code is low level and the control flow is fairly involved.

Our implementation of this same search routine is illustrated in Fig. 5. A master thread on the host iterates over each query HMM sequentially. We use the *parallel for* directive to start *num_devices* threads on the host multicore, one for each node in the cluster. Each of these threads operates in parallel, offloading data and execution control to its associated node. Unlike traditional OpenMP where threads are used for compute, we are using them as I/O threads to facilitate parallel offloading. Another approach is to use asynchronous target

[2] HMMER 3.1b2: http://hmmer.org.

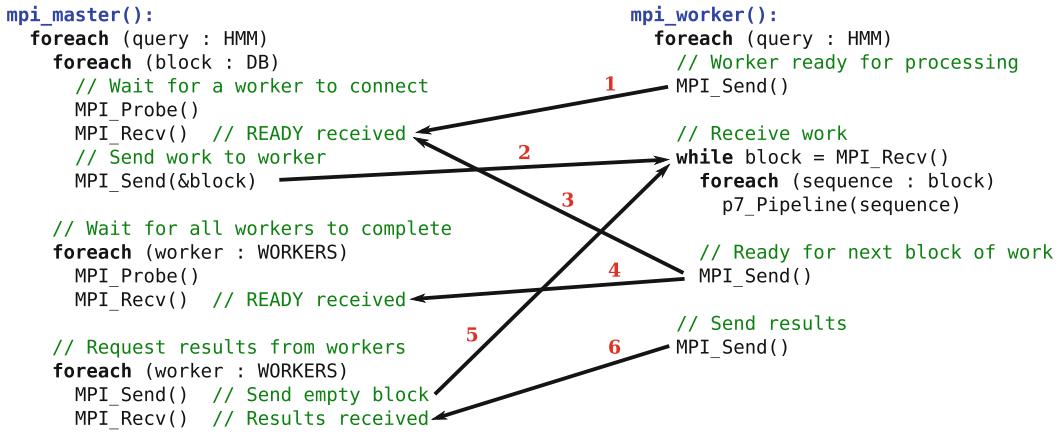

Fig. 4. Pseudocode illustrating a search procedure from HMMER that is currently implemented using traditional MPI.

```
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)
      }
```

Fig. 5. HMMER search pseudocode implemented using our proposed abstraction.

offloading with the *nowait* clause of OpenMP 4.1, in which case only a single host thread is necessary.

The worksharing construct implicitly partitions the database across nodes by distributing iterations of the DB loop across host threads. The OpenMP *dynamic* schedule transparently achieves the desired load balancing. Data movement is achieved using map clauses and implicit barrier after the *parallel* pragma transparently synchronizes across all workers. The combination of the 3.1 parallelization pragma with the cluster enabled offloading construct helps realize a much simpler program.

We note that our current implementation is more involved due to a known limitation of the map clause. The standard does not specify the handling of deep copy of structures with pointers, which is required to map the *query* structure onto the target device. Deep copy is necessary to map the variable in the map clause and all data referenced by fields within the variable (it is being considered for addition to the Accelerator model). One way around this limitation is to manually pack and unpack the structure across the device boundary. Our current implementation instead maps a query identifier to each target, which then reads the query data from a shared disk.

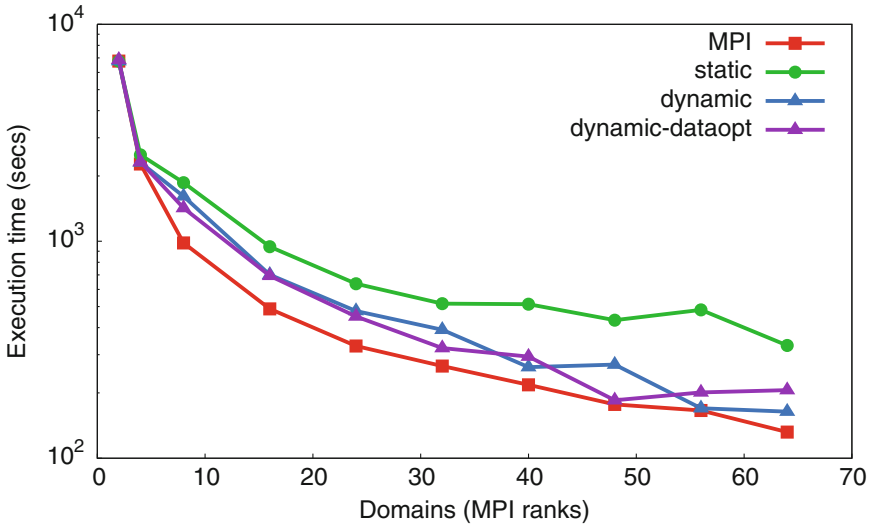

Fig. 6. Comparison of HMMER with stock MPI against the implementation in this work.

To measure performance we run our experiments on a four-node IBM Power 8® S824 cluster. A node has four sockets, each with six cores and eight hardware threads per core running at a maximum clock frequency of 3.3 GHz. We use the Open MPI 1.8.5 library compiled with multithreaded support and the LSF cluster management tool to closely pack MPI ranks, 16 per node.

We compiled HMMER for this cluster, enabling stock MPI support, disabling multithreaded execution, and without SIMD. In our experiment we use HMMER to compare a single query HMM against a database of 3.9 M sequences. We selected an appropriate database block size to ensure repeated invocation of target devices to simulate realistic conditions. In our experiments we vary the number of MPI ranks from 2 to 64. We run each experiment several times and select the minimum execution time.

Figure 6 compares execution time (log scale) using HMMER+MPI against our implementation. We see good scaling behavior that is comparable to the low-level MPI implementation. Using a dynamic instead of a static schedule to offload the database consistently gives superior performance, reducing runtime by over 50 % in some cases. We are able to achieve this performance using simple directives applied to a serial program. The complexity of balanced work scheduling across worker nodes, message passing to transfer data, manage control flow, and synchronize across nodes is completely hidden behind high-level OpenMP abstractions.

We observe a constant factor overhead in our implementation beyond 4 ranks. Currently we have not optimized data communication between the host and target devices. For example, if there are repeated invocations of a target device it may be possible for the runtime to reuse memory on the device that was previously allocated. As shown by the curve labeled *dynamic-dataopt*, we can eliminate some of this overhead by establishing a persistent device data environment using the *target data* directive.

We are investigating the reason for the performance degradation of *dynamic-dataopt* with 56 and 64 ranks. We believe this may be due to unbalanced workload partitioning of coarser database partitions that was selected to reduce offloading overhead.

7 Discussion

Rich and efficient data sharing. The standard accelerator model in OpenMP 4.0 implies that data sharing between the host and a device is specified exclusively via the map clause. While this is likely to be the only method for co-processors like GPUs, in our case we can also exploit a networked filesystem to share data. This also allows the programmer to exploit the bandwidth of a large number of distributed disks.

Exploiting cores in a node. It is possible to naturally extend the implementation in Fig. 5 to exploit multicores in a target node. We may use a nested *parallel for* worksharing directive to dynamically distribute database sequences in the block assigned to a target across its cores.

Nested target regions. The current standard does not define the semantics of nested target regions. These may be useful, however, to model a co-processor such as a GPU within a node in a cluster. Additional semantics, for example, the scope of "global" variables will have to be clarified. Compiling nested target regions is also more challenging, likely requiring recursive calls in the driver and nested containers within the generated fat binary.

Exploiting resources in a cloud. There is no requirement in our model that the clique of shared-memory domains be on the same homogeneous cluster. Our model can be used to offload computation onto rented nodes on one or more cloud platforms to seamlessly scale compute and data requirements from a personal device such as a mobile phone or a laptop, to an organization's cluster, and finally a large-scale cloud platform.

Beyond offloading. Our proposed model uses well-defined offloading semantics that are easy to reason about for a programmer. However, more powerful extensions may be desired. In particular, arbitrary communication between any two target devices may be useful for the expert programmer but this is unlikely to be generally friendly.

One direction we plan to explore is the asynchronous update of variables on the host (and the target) initiated by the device. This will allow the support of the parameter-server model [9], which tolerates asynchronous updates of parameter variables for efficient execution of many machine learning algorithms.

8 Conclusions

In this work we have introduced an abstract model to represent a cluster, and OpenMP 4.0 target directives to implement a simple directive-based approach to programming a distributed machine. We have implemented this idea in the LLVM compiler with a runtime that transparently offloads execution via MPI.

Initial results on a bioinformatics application shows good scaling behavior. Compared to an MPI based approach, high-level OpenMP abstractions in our implementation completely hide the complexity of balanced work scheduling across worker nodes, message passing for data transfer, control flow management, and synchronization across nodes. We have identified a number of possible optimizations to improve performance and directions to extend the presented model.

The application we have considered for acceleration is fully data parallel, though it requires dynamic workload balancing. In the future it would be interesting to study applications that require more frequent communication and synchronization with the host.

References

1. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. J. High Perf. Comput. Appl. **21**(3), 291–312 (2007)
2. Charles, P., et al.: X10: An object-oriented approach to non-uniform cluster computing. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 519–538 (2005)
3. Clang: A C language family frontend for LLVM. http://clang.llvm.org
4. Eichenberger, A.E., O'Brien, K.: Experimenting with low-overhead OpenMP runtime on IBM Blue Gene/Q. IBM J. Res. Dev. **57**(1/2), 8:1–8:8 (2013)
5. El-Ghazawi, T., Smith, L.: UPC: Unified parallel C. In: Supercomputing (2006)
6. Hoeflinger, J.P.: Extending OpenMP to clusters (2006)
7. Hu, Y., Lu, H., Cox, A.L., Zwaenepoel, W.: OpenMP for networks of SMPs. J. Parallel Distrib. Comput. **60**(12), 1512–1530 (2000)
8. Kogge, P.M.: Performance analysis of a large memory application on multiple architectures. In: Conference on Partitioned Global Address Space Programming Models (2013)
9. Li, M., et al.: Scaling distributed machine learning with the parameter server. In: Operating Systems Design and Implementation, pp. 583–598, October 2014
10. The LLVM Compiler Infrastructure. http://llvm.org
11. Millot, D., Muller, A., Parrot, C., Silber-Chaussumier, F.: STEP: a distributed OpenMP for coarse-grain parallelism tool. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 83–99. Springer, Heidelberg (2008)
12. Numrich, R.W., Reid, J.: Co-array fortran for parallel programming. SIGPLAN Fortran Forum **17**(2), 1–31 (1998)
13. Ojima, Y., Sato, M., Harada, H., Ishikawa, Y.: Performance of cluster-enabled OpenMP for the SCASH software distributed shared memory system. In: Cluster Computing and the Grid, pp. 450–456, May 2003
14. OpenMP Application Program Interface. http://www.openmp.org/
15. OpenMP, A.R.B.: OpenMP version 4.0, May 2013
16. Rowstron, A., et al.: Nobody ever got fired for using hadoop on a cluster. In: Workshop on Hot Topics in Cloud Data Processing, pp. 2:1–2:5 (2012)